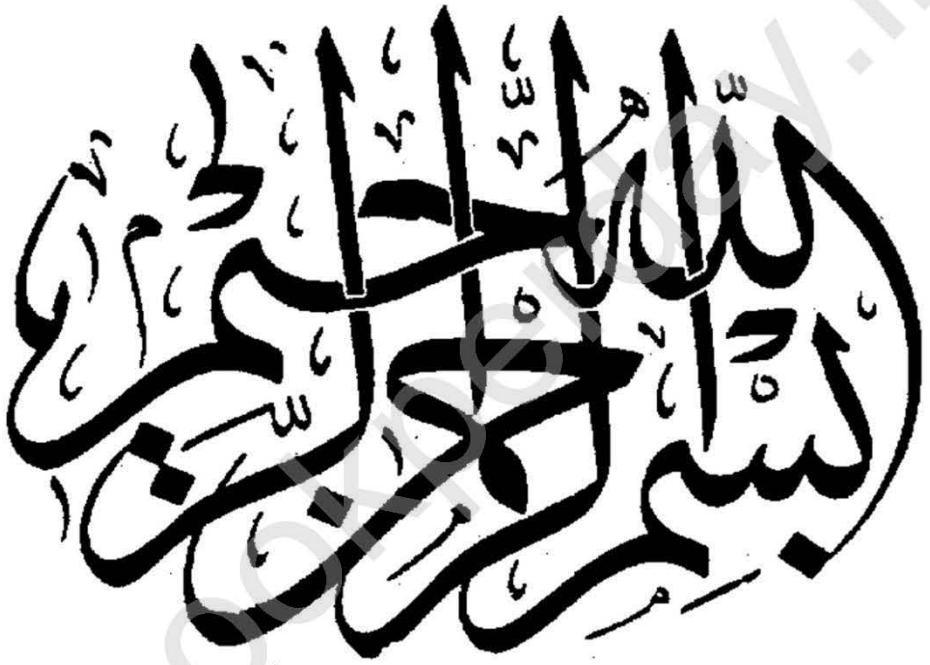
مبانی و مفاهیم
علوم کامپیوتر

مؤلف:
ولادستون فریرا فیلیو

مترجمان:
علی ناصراسدی، علی رهنما



مبانی و مفاهیم

علوم کامپیوتر

مؤلف:

ولادستون فریرا فیلیو

مترجمان:

علی ناصراسدی

عضو هیأت علمی گروه کامپیوتر، مجتمع آموزش عالی زرند

علی رهنما

عضو هیأت علمی گروه کامپیوتر، مجتمع آموزش عالی زرند

سرشناسه	:	فریرا فیلیو، والادستون Ferreira Filho, Wladston
عنوان و نام پدیدآور	:	مبانی و مفاهیم علوم کامپیوتر / مولف ولادستون فریرافیلیو؛ مترجمان علی ناصراسدی، علی رهنما.
مشخصات نشر	:	تهران: موسسه آموزشی تالیفی ارشدان، ۱۴۰۱.
مشخصات ظاهری	:	۱۹۶ص.: مصور (رنگی).
شابک	:	۹۷۸-۶۲۲-۰۸-۵۵۸۰-۴
وضعیت فهرست نویسی	:	فیپا
یادداشت	:	عنوان اصلی: . Computer science distilled : learn the art of solving computational problems, 2017
موضوع		الگوریتم‌های کامپیوتری Computer algorithms
		برنامه‌نویسی Computer programming
		علوم کامپیوتر Computer science
		ساختار داده‌ها Data structures (Computer science)
شناسه افزوده	:	ناصراسدی، علی، ۱۳۶۱ -، مترجم
شناسه افزوده	:	رهنما، علی، ۱۳۵۹-، مترجم
رده بندی کنگره	:	QA۷۶/۹
رده بندی دیویی	:	۰۰۴
شماره کتابشناسی ملی	:	۹۱۰۰۵۷۷
اطلاعات رکورد کتابشناسی	:	فیپا

مؤسسه آموزشی تألیفی ارشدان

■ نام کتاب:	مبانی و مفاهیم علوم کامپیوتر
■ مترجمان:	علی ناصراسدی- علی رهنما
■ ناشر:	آموزشی تألیفی ارشدان
■ ویرایش:	اول
■ نوبت چاپ:	اول ۱۴۰۱
■ حروفچینی و صفحه آرایی:	علی ناصر اسدی- علی رهنما
■ طراح و گرافیست:	علی ناصر اسدی- علی رهنما
■ شابک:	۹۷۸-۶۲۲-۰۸-۵۵۸۰-۴
■ شمارگان:	۱۰۰۰
■ مرکز خرید آنلاین:	www.arshadan.com www.arshadan.net
■ مرکز پخش و توزیع:	۰۲۱۴۷۶۲۵۵۰۰
■ قیمت:	۱۲۰۰۰۰ تومان

سخن مؤلف با خوانندگان ایرانی

خوانندگان ایرانی عزیز،

از دوران باستان، مردم و سرزمین شما اختراعات بزرگی را در زمینه‌ی ریاضیات شاهد بوده‌اند. در حقیقت، موضوع علوم کامپیوتر به وسیله‌ی ریاضی‌دانان ایرانی مطرح شد. محمد بن موسی خوارزمی الگوریتم را ابداع کرد و الگوریتم نامش را از این ریاضی‌دان بزرگ گرفته است. خوشبختانه تمام جهان کار خود را بر مبنای کار خوارزمی به پیش‌برده، و امروز ما یک شگفتی بزرگ جدید داریم: کامپیوتر شخصی.

در این کتاب، من شما را به دنبال کردن مسیر مخترعان جبر و الگوریتم دعوت کرده‌ام. امیدوارم الهام‌بخش شما در استفاده از اکتشافات علوم محاسبات برای ایجاد نرم‌افزارهایی باشم که به بشریت در ساختن جهانی بهتر، جهانی با عشق بیشتر، کمک کنند. و بخش عمده‌ای از شما را که ماجراجو هستید به کاوش و توسعه‌ی علوم محاسبات دعوت کرده‌ام تا فرزندان و نوادگانمان بتوانند کامپیوترهای پیشرفته‌تری برای کار کردن داشته باشند.

مسیر یادگیری شما سرشار از برکت و آرامش باد.

ولادستون فیلیو

Dear Iranian readers,

Since ancient times, your people and your lands have witnessed great inventions in the fields of mathematics. In fact, the subject of computer science was started by Iranian mathematicians. Muḥammad ibn Mūsā al-Khwārizmī invented the algorithm—that word itself has its origin from the great mathematician's name. Fortunately, the entire world has built on top of al-Khwarizm's foundation, and today we have a modern wonder: the personal computer.

In this book, I invite you to follow the path of the invertors of algebra and the algorithm. I hope to inspire you to use the discoveries in the science of computation to create software that helps humanity build a better world, one with more love. And I invite the most adventurous of you to keep exploring and developing the science of computation, so that our children and grandchildren can have even more advanced computers to work with.

May your learning journey be blessed and peaceful.

Wladston Filho

می‌دانم که دو با دو می‌شود چهار؛ اگر می‌توانستم آن را ثابت کنم خوشحال نیز می‌شدم، ولی باید بگویم اگر به هر طریقی می‌توانستم دو با دو را به پنج تبدیل کنم، لذت بیشتری می‌بردم.

– لرد بایرون[1]

در نامه‌ای به زن آینده‌اش آنابلا[2] در سال ۱۸۱۳

دختر ایشان ایدا لاولیس[3] اولین برنامه‌نویس بود.

---

[1] Lord Byron (1788-1824): شاعر و سیاستمدار انگلیسی. م.

[2] Annabella

[3] Ada Lovelace

فهرست مطالب

پیشگفتار مترجمان ۱۳

پیشگفتار ۱۵

فصل ۱: مفاهیم بنیادین ۱۷

۱-۱- ایده‌ها ۱۷

۱-۲- منطق ۲۲

۱-۳- شمارش ۳۱

۱-۴- احتمال ۳۷

نتیجه‌گیری ۴۱

فصل ۲: پیچیدگی ۴۳

۲-۱- شمارش زمان ۴۵

۲-۲- نماد اُی‌بزرگ ۴۸

۲-۳- نمایی‌ها ۵۰

۲-۴- شمارش حافظه ۵۲

نتیجه‌گیری ۵۲

فصل ۳: استراتژی ۵۵
۳-۱- تکرار ۵۶
۳-۲- بازگشت ۵۹
۳-۳- جستجوی فراگیر ۶۱
۳-۴- عقب‌گرد ۶۳
۳-۵- ابتکار ۶۵
۳-۶- تقسیم و حل ۷۰
۳-۷- برنامه‌نویسی پویا ۷۶
۳-۸- شاخه و حد ۸۰
نتیجه‌گیری ۸۴
فصل ٤: داده‌ها ۸۵
۴-۱- انواع داده‌ای مجرد ۸۷
۴-۲- انتزاع‌های رایج ۸۸
۴-۳- ساختارها ۹۳
نتیجه‌گیری ۱۰۵
فصل ۵: الگوریتم‌ها ۱۰۷
۵-۱- مرتب‌سازی ۱۰۸
۵-۲- جستجو ۱۰۹
۵-۳- گراف‌ها ۱۱۱
۵-۴- تحقیق در عملیات ۱۱۸
نتیجه‌گیری ۱۲۳
فصل ٦: پایگاه داده ۱۲۵
۶-۱- رابطه‌ای ۱۲۶
۶-۲- غیررابطه‌ای ۱۳۵

۶-۳- توزیع‌شده ۱۴۰
۶-۴- جغرافیایی ۱۴۵
۶-۵- قالب‌های سریال‌سازی ۱۴۶
نتیجه‌گیری ۱۴۷
فصل ۷: کامپیوترها ۱۴۹
۷-۱- معماری ۱۴۹
۷-۲- کامپایلرها ۱۵۸
۷-۳- سلسله‌مراتب حافظه ۱۶۶
نتیجه‌گیری ۱۷۳
فصل ۸: برنامه‌نویسی ۱۷۵
۸-۱- زبان‌شناسی ۱۷۵
۸-۲- متغیرها ۱۷۸
۸-۳- پارادایم‌ها ۱۸۱
نتیجه‌گیری ۱۸۹
نتیجه‌گیری ۱۹۱
پیوست ۱۹۳

پیشگفتار مترجمان

کامپیوترها به‌طور شگفت‌انگیزی سریع، دقیق و نادان هستند. آدم‌ها به‌طور شگفت‌انگیزی کند، بی‌دقت و باهوش هستند. این دو در کنار هم، قدرتی فراتر از تصور دارند.

– آلبرت اینشتین[1]

کامپیوترها در طی مدت‌زمان نسبتاً اندکی که از اختراع و ورود آن‌ها به بازارها گذشته است، تأثیرات شگرفی بر زندگی انسان‌ها داشته‌اند؛ به‌طوری‌که امروزه تقریباً زندگی کردن بدون آن‌ها، اگر نگوییم غیرممکن ولی بسیار سخت است. این امر، یعنی ورود کامپیوترها به زندگی بشر و تأثیرگذاری بسیار بر آن، موجب شده ما برای تعامل بهتر با این ابزارهای حیاتی، نیازمند درک نحوه‌ی عملکرد آن‌ها شویم. در حقیقت، می‌توان ادعا کرد در دنیای جدید، آشنایی با مبانی و مفاهیم علوم کامپیوتر به‌عنوان زیربنایی برای توسعه و استفاده‌ی بیشتر از کامپیوترها، نه‌تنها برای افراد حاضر در این صنعت اهمیت ویژه‌ای دارد بلکه برای سایر کسانی که قصد به‌کارگیری آن‌ها را دارند نیز حائز اهمیت است.

از سوی دیگر، به دلیل گستردگی و پیچیدگی بسیاری از مطالب اصلی مرتبط با مبانی و مفاهیم علوم کامپیوتر، مطالعه و درک آن‌ها برای افراد تازه‌کار و کاربران عادی دشوار می‌نماید. به همین دلیل، بر آن شدیم کتاب مبانی و مفاهیم علوم کامپیوتر را که در حال حاضر در پیش روی شما قرار دارد، به‌عنوان مرجعی مناسب برای آشنایی با اصلی‌ترین مفاهیم این حوزه ترجمه نموده و در اختیار علاقه‌مندان قرار دهیم.

این کتاب سعی دارد به زبانی بسیار ساده و با بیان مثال‌ها و نکات قابل‌درک و آسان، به واکاوی مهم‌ترین مبانی و مفاهیم علوم کامپیوتر بپردازد. درواقع، هدف اصلی این کتاب ارائه‌ی مطالبی است که ممکن است مورد سؤال بسیاری از افراد بوده ولی به دلیل عدم وجود مرجعی مناسب برای پاسخگویی به آن‌ها، برای ایشان حل‌نشده باقی‌مانده است. به‌این‌ترتیب، این کتاب می‌تواند منبعی مناسب برای دانشجویان جدیدالورود رشته‌ی کامپیوتر، دانش‌آموزان دبیرستانی و هر فرد دیگری که علاقه‌مند به

[1] Albert Einstein (1879-1955): فیزیک‌دان زاده‌ی آلمان و توسعه‌دهنده‌ی نظریه‌ی نسبیت. م.

آشنایی با نحوه‌ی عملکرد و مفاهیم بنیادین کامپیوترها است، باشد. باید توجه داشت که تجارب نویسنده‌ی کتاب به‌عنوان یک برنامه‌نویس موفق، نقش چشمگیری در کیفیت مطالب ارائه‌شده داشته است.

مطالب کتاب حاضر در قالب هشت فصل و یک پیوست ارائه‌شده‌اند. فصل ۱ به بررسی مفاهیم بنیادین علوم کامپیوتر مانند الگوریتم‌ها، فلوچارت‌ها، شبه‌کدها، مدل‌های ریاضی، منطق و احتمال پرداخته است. فصل ۲ مفهوم پیچیدگی را به‌عنوان یکی از اساسی‌ترین مطالب مرتبط با الگوریتم‌ها و کیفیت آن‌ها موردبررسی قرار داده و فصل ۳ مفهوم استراتژی و روش‌های معمول حل مسئله را بابیان مثال‌های ساده ولی جذاب معرفی کرده است. نویسنده فصل ۴ را به مفهوم داده‌ها و اصلی‌ترین ساختارهای آن‌ها در علوم کامپیوتر اختصاص داده است.

در ادامه و در فصل ۵، الگوریتم‌ها به‌عنوان حیاتی‌ترین ابزار کامپیوترها برای حل مسائل معرفی و برخی از معروف‌ترین نمونه‌های آن‌ها ارائه‌شده‌اند. فصل ۶ کتاب به بررسی پایگاه داده، انواع و نقش آن در سیستم‌های کامپیوتری امروزی اختصاص‌یافته است. در فصل ۷ به خود پدیده‌ی کامپیوتر و معماری‌ها، سیستم‌های‌عامل، کامپایلرها و انواع حافظه‌های آن پرداخته‌شده و درنهایت در فصل ۸، برنامه‌نویسی و پارادایم‌های مختلف آن معرفی‌شده است. تنها پیوست کتاب نیز به مبناهای عددی، مجموعه‌ها و برخی از مطالب حاشیه‌ای مرتبط با علوم کامپیوتر اختصاص داده‌شده است.

لازم به تذکر است که متن اصلی کتاب به زبانی بسیار ساده و به‌دوراز تکلف رایج متون علمی نگاشته شده و به همین دلیل ما به‌عنوان مترجمان کتاب نیز به‌منظور رعایت امانت در ترجمه و هم‌چنین برقراری رابطه‌ی بهتر با خواننده، خود را ملزم به رعایت این سبک دانسته‌ایم. هم‌چنین باید اشاره کرد که در راستای حفظ حق مالکیت معنوی کتاب، این ترجمه با اخذ مجوز کتبی از نویسنده و صاحب اثر انجام و منتشر شده است. در انتها، از تمامی خوانندگان عزیز خواهشمندیم نظرات و نکات سازنده‌ی خود را در ارتباط با این کتاب با ما در میان بگذارند.

علی ناصراسدی، علی رهنما
گروه کامپیوتر
مجتمع آموزش عالی زرند
زمستان ۱۴۰۱ هجری خورشیدی
naserasadi@zarand.ac.ir
rahnama@zarand.ac.ir

پیشگفتار

همه در این کشور باید یاد بگیرند که یک کامپیوتر را برنامه‌نویسی کنند، زیرا به آن‌ها نحوه‌ی فکر کردن را می‌آموزد.

– استیو جابز[1]

هم‌زمان با تغییر دنیا با قدرت بی‌سابقه‌ی کامپیوترها، شاخه‌ای جدید از علوم جدید متولد شد: *علوم کامپیوتر*. این شاخه‌ی جدید نشان داد که چگونه می‌توان از کامپیوترها برای حل مسائل استفاده کرد. علوم کامپیوتر ماشین‌ها را به سمت حداکثر قابلیت بالقوه‌ی آن‌ها هدایت کرد؛ و ما به چیزهای جذاب و شگفت‌انگیزی رسیدیم.

علوم کامپیوتر در همه‌جا وجود دارد، ولی همچنان در قالب یک نظریه‌ی خسته‌کننده آموزش داده می‌شود. بسیاری از کدنویسان اصلاً آن را نمی‌خوانند! بااین‌حال، علوم کامپیوتر برای برنامه‌نویسی مؤثر اهمیتی ویژه دارد. برخی از دوستان من حتی نمی‌توانند یک کدنویس خوب برای استخدام پیدا کنند. قدرت محاسبات فراوان است، اما افرادی که بتوانند از آن استفاده کنند کمیاب هستند.

این تلاشی اندک از طرف من برای کمک به جهان است، با هدایت شما به سمت استفاده‌ی مؤثر از کامپیوتر. این کتاب مفاهیم علوم کامپیوتر را به شکلی بسیار ساده بیان می‌کند. سعی شده است از کمترین تشریفات رسمی دانشگاهی استفاده شود. امیدوارم علوم کامپیوتر به ذهن شما کمک کرده و کدهای شما را بهبود بخشد.

آیا این کتاب برای من است؟

اگر می‌خواهید مسائل را با راه‌حل‌های کارآمد حل کنید، این کتاب برای شما مناسب است. تجربه‌ی برنامه‌نویسی کمی موردنیاز است. اگر قبلاً چند خط کد نوشته‌اید و عبارات اصلی برنامه‌نویسی مانند `for` و `while` را می‌شناسید، مشکلی ندارید. اگرنه، این دوره‌های برنامه‌نویسی آنلاین[2] بیش ازآنچه را که

---

[1] Steve Jobs (1955-2011): کارآفرین، مخترع، بنیان‌گذار و مدیر ارشد اجرایی شرکت اپل و یکی از چهره‌های ماندگار صنعت کامپیوتر بود. م.

[2] http://code.energy/coding-courses

لازم دارید پوشش می‌دهند. می‌توانید یک دوره را در یک هفته به‌صورت رایگان بگذرانید. این کتاب خلاصه‌ای عالی برای تثبیت دانش کسانی است که علوم کامپیوتر خوانده‌اند.

شکل ۱: «مشکلات کامپیوتری» (دریافت شده از http://xkcd.com)

ولی مگر علوم کامپیوتر فقط مختص دانشگاهیان نیست؟

این کتاب برای همه است. این کتاب در مورد *تفکر محاسباتی* است. شما یاد خواهید گرفت که مسائل را به سیستم‌های قابل‌محاسبه تغییر دهید. شما از تفکر محاسباتی در مسائل روزمره استفاده خواهید کرد. پیش‌واکشی[۱] و حافظه‌ی نهان[۲] مهیا شدن برای سفرتان را ساده‌تر می‌کنند. موازی‌سازی[۳] سرعت آشپزی شما را افزایش می‌بخشد. به‌علاوه، کد شما عالی‌تر خواهد شد.

باشد که توانا باشید،

ولاد

---

۱ Perfecting

۲ Caching

۳ Parallelism

فصل ۱

مفاهیم بنیادین

> علوم کامپیوتر در مورد ماشین‌ها نیست، همان‌گونه که نجوم در مورد تلسکوپ‌ها نیست. بین ریاضی و علوم کامپیوتر پیوندی الزامی وجود دارد.
>
> – ادسخر دایکسترا[1]

کامپیوترها نیاز دارند مسائل را به قطعات کوچکی برای آن‌ها تقسیم کنیم که بتوانند آن‌ها را هضم و حل کنند. برای انجام این کار، ما به ریاضی نیاز داریم. نترسید! قرار نیست موشک هوا کنید، نوشتن کد خوب به‌ندرت نیازمند معادلات پیچیده است. این فصل فقط یک جعبه‌ابزار برای حل مسئله است. شما یاد خواهید گرفت:

- **ایده‌ها** را در قالب فلوچارت‌ها و شبه‌کدها مدل کنید.
- با **منطق** درست را از غلط تشخیص دهید.
- چیزها را **بشمارید**.
- **احتمالات** را به‌دقت محاسبه کنید.

به‌این‌ترتیب، آنچه را که برای ترجمه‌ی ایده‌هایتان به راه‌حل‌های قابل‌محاسبه نیاز دارید، در اختیار خواهید داشت.

۱-۱- ایده‌ها

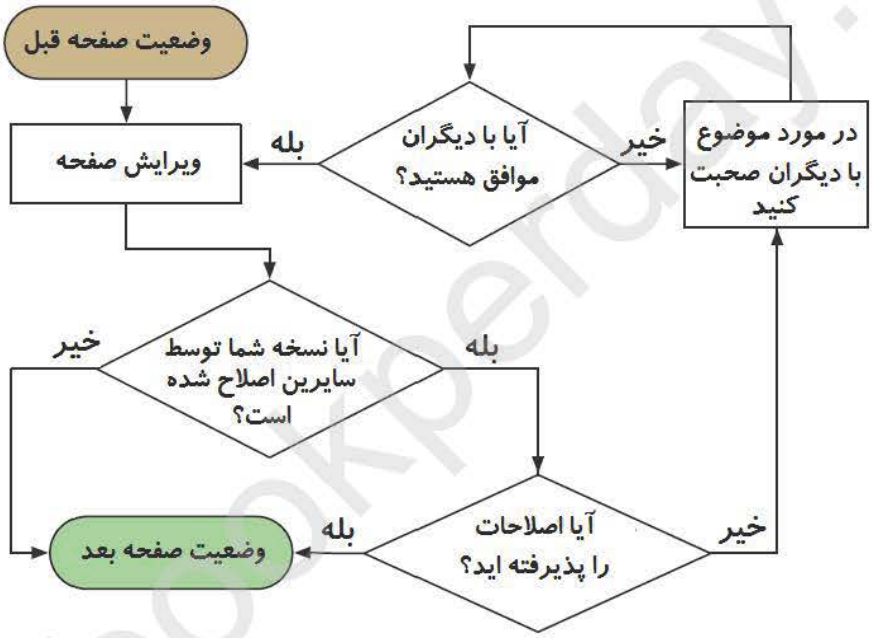
وقتی در حال انجام یک کار پیچیده هستید، مغزتان را در بالاترین سطح خود نگه‌دارید. همه‌ی نکات مهم را روی کاغذ بیاورید. حافظه‌ی کاری مغز ما به‌سادگی با حقایق و ایده‌ها سرریز می‌کند. نوشتن

[1] Edsger Dijkstra (1930-2002): دانشمند هلندی در علوم ریاضی و کامپیوتر و برنده‌ی جایزه‌ی تورینگ ۱۹۷۲.م.

همه‌ی نکات یکی از راه‌های متعدد سازمان‌دهی است. راه‌های بسیار دیگری نیز برای انجام این کار وجود دارد. در ابتدا نگاهی به نحوه‌ی استفاده از فلوچارت‌ها برای نمایش فرایندها خواهیم داشت. سپس یاد خواهیم گرفت که چگونه می‌توان فرایندهای قابل‌برنامه‌ریزی را در قالب شبه‌کد بیان کنیم. هم‌چنین به مدل‌سازی یک مسئله‌ی ساده با استفاده از ریاضی خواهیم پرداخت.

فلوچارت‌ها[1]

زمانی که سازندگان ویکی‌پدیا در مورد فرایندهای همکاری خود بحث می‌کردند، فلوچارتی ساختند که با پیشرفت گفتگوها بهنگام شد. داشتن یک تصویر از آنچه ارائه می‌شد به بحث کمک می‌کرد:



شکل ۱-۱: فرایند ویرایش ویکی (دریافت شده از http://wikipedia.org)

شبیه به فرایند ویرایش فوق، کد کامپیوتری نیز یک فرایند است. برنامه‌نویسان اغلب از فلوچارت‌ها برای نوشتن فرایندهای محاسباتی استفاده می‌کنند. در زمان انجام این کار باید نکات زیر را رعایت کنید تا دیگران فلوچارت شما را بفهمند[2]:

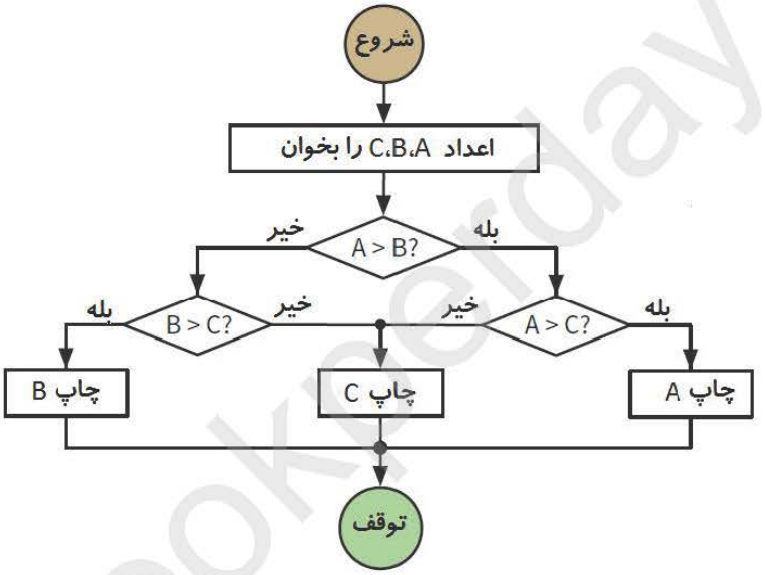
---

[1] Flowchart

[2] حتی یک استاندارد ISO به نام UML وجود دارد که نحوه‌ی ترسیم نمودارهای سیستم‌های نرم‌افزاری را به‌دقت مشخص می‌کند: http://code.energy/UML.

- وضعیت‌ها و گام‌های دستورالعمل را درون مستطیل‌ها بنویسید.
- گام‌های تصمیم‌گیری که ممکن است فرایند مسیرهای متفاوتی را دنبال کند، درون لوزی‌ها بنویسید.
- هیچ‌گاه یک گام دستورالعمل را با یک گام تصمیم‌گیری ترکیب نکنید.
- گام‌های متوالی را با فلش به هم وصل کنید.
- شروع و پایان فرایند را مشخص کنید.

اجازه بدهید ببینیم این روال برای پیدا کردن بزرگ‌ترین عدد از بین سه عدد چگونه انجام می‌شود:



شکل ۱-۲: پیدا کردن مقدار بیشینه بین سه متغیر

شبه‌کد[۱]

درست شبیه فلوچارت‌ها، **شبه‌کد** نیز فرایند محاسبات را شرح می‌دهد. شبه‌کد یک کد انسان‌پسند است که توسط یک ماشین درک نمی‌شود. مثال زیر مشابه شکل ۱-۲ است. یک دقیقه زمان گذاشته و آن را با سه مقدار ساده برای A، B و C آزمایش کنید[۲]:

---

[۱] Pseudocode

[۲] در اینجا علامت ← به معنی عملگر انتساب است: x←1 به این معنی است که مقدار x برابر با ۱ قرار داده می‌شود.

```
function maximum(A, B, C)
    if A > B
        if A > C
            max ← A
        else
            max ← C
    else
        if B > C
            max ← B
        else
            max ← C
    print max
```

توجه کردید که این مثال چگونه به‌طور کامل تمامی قوانین گرامری زبان‌های برنامه‌نویسی را نقض کرد؟ وقتی شما شبه‌کد می‌نویسید، حتی می‌توانید از زبان گفتاری استفاده کنید! همان‌طور که از فلوچارت‌ها برای ترکیب نقشه‌های ذهنی کلی استفاده می‌کنید، اجازه بدهید خلاقیتتان در زمان نوشتن شبه‌کد جریان داشته باشد (شکل ۱-۳).

شکل ۱-۳: شبه‌کد در زندگی واقعی (دریافت شده از http://ctp200.com)

مدل‌های ریاضی

یک **مدل**[1] مجموعه‌ای از مفاهیم است که یک مسئله و مشخصات آن را نمایش می‌دهد. مدل به ما اجازه می‌دهد بهتر استدلال کرده و با مسئله تعامل کنیم. ساخت مدل‌ها آن‌قدر مهم است که در مدرسه

---

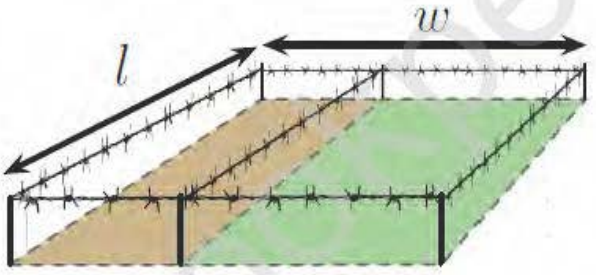
[1] Model

آموزش داده می‌شود. ریاضی دبیرستان به مسائل مدل‌سازی در قالب اعداد و معادلات و به‌کارگیری ابزارها برای رسیدن به یک راه‌حل می‌پردازد (یا باید بپردازد).

مدل‌های تشریح شده‌ی ریاضی یک امتیاز مهم دارند: می‌توان آن‌ها را با استفاده از روش‌های دقیق ریاضی برای فهم توسط کامپیوترها وفق داد. اگر مدل شما دارای گراف‌ها باشد، از نظریه‌ی گراف استفاده کنید. اگر از معادلات استفاده می‌کند، از جبر کمک بگیرید. *بر روی شانه‌های غول‌هایی* که این ابزارها را ساخته‌اند بایستید[1]. این کار جواب خواهد داد. اجازه بدهید این موضوع را به‌صورت عملی و در یک مسئله‌ی معمول دبیرستانی ببینیم:

> حصار مزرعه: فرض کنید مزرعه‌ی شما دو نوع چهارپا دارد. شما ۱۰۰ واحد سیم‌خاردار برای ایجاد یک حصار مستطیل شکل برای حیوانات با یک خط جداکننده‌ی صاف به‌منظور جداسازی آن‌ها، دارید. چگونه حصار را می‌سازید که چراگاه حداکثر مساحت را داشته باشد؟

با آنچه باید تعیین کنیم، یعنی w و l که ابعاد حلفزار هستند و $A = w \times l$ که مساحت آن است، شروع می‌کنیم. پیشینه‌سازی به معنی استفاده از تمام سیم‌خاردار است، بنابراین w و l را به نسبت صد می‌نویسیم:



$$A = w \times l$$
$$100 = 2w + 3l$$

حال باید w و l را به‌گونه‌ای انتخاب کنید که مقدار A پیشینه شود.

با به دست آوردن l از معادله‌ی دوم ($l = \frac{100-2w}{3}$) و قرار دادن آن در معادله‌ی اول، داریم:

$$A = \frac{100}{3}w - \frac{2}{3}w^2$$

این یک معادله‌ی درجه‌ی ۲ است. پیشینه‌ی این معادله به‌سادگی با *فرمول‌های درجه‌ی ۲ دبیرستان* به دست می‌آید. معادلات درجه‌ی ۲ برای شما همانند ارزش یک بخاریز برای یک آشپز، مهم هستند. این معادلات موجب صرفه‌جویی در زمان می‌شوند. معادلات درجه‌ی ۲ به ما کمک می‌کنند مسائل را سریع‌تر

---

[1] اشاره به جمله‌ی معروف نیوتن دارد. اگرچه این عبارت در اصل استعاره‌ای از کشف حقیقت بر پایه‌ی اکتشافات پیشین است و به اساطیر یونان باستان بازمی‌گردد. م.

حل کنیم. به یاد داشته باشید که وظیفه‌ی شما حل مسائل است. یک آشپز وسایل خودش را می‌شناسد، شما هم باید ابزارهای خودتان را بشناسید. شما به مدل‌سازی ریاضی نیاز دارید. هم‌چنین شما به منطق نیز نیاز دارید.

۱-۲- منطق

کدنویسان آن‌قدر از منطق استفاده می‌کنند که کاملاً ذهنشان را پرکرده است. بااین‌حال، بسیاری از کدنویسان هنوز به‌درستی منطق را یاد نگرفته و به‌صورت ناخودآگاه از آن استفاده می‌کنند. با یادگیری منطق صوری[1]، می‌توانیم از آن به‌صورت آگاهانه برای حل مسائل استفاده کنیم.

شکل ۱-٤: منطق برنامه‌نویسان (دریافت شده از http://programmers.life)

بحث در این مورد را با مطرح کردن عبارات منطقی با استفاده از عملگرهای خاص و جبر مخصوص آن شروع می‌کنیم. سپس می‌آموزیم که با استفاده از جدول درستی مسائل را حل کرده و خواهیم دید که کامپیوترها چگونه به منطق تکیه کرده‌اند.

---

[1] Formal Logic

عملگرها[۱]

در ریاضی معمول از متغیرها و عملگرها (+، − و غیره) برای مدل‌سازی مسائل عددی استفاده می‌شود. در منطق ریاضی، متغیرها[۲] و عملگرها اعتبار اشیاء را نشان می‌دهند. این متغیرها عدد را نشان نمی‌دهند بلکه بیانگر مقادیر True (درست) یا False (غلط) هستند. به‌عنوان‌مثال، اعتبار عبارت «اگر استخر گرم باشد، شنا خواهم کرد» بر اساس اعتبار دو چیز تعیین می‌شود که قابل نگاشت به متغیرهای منطقی A و B هستند:

A: استخر گرم است

B: شنا می‌کنم

این متغیرها مقدار True یا False دارند[۳]. اگر داشته باشیم A=True یعنی استخر گرم است و B=False به معنی شنا نکردن است. متغیر B نمی‌تواند نیمه درست باشد، زیرا نمی‌توان به‌صورت نیمه شنا کرد. وابستگی بین متغیرها با استفاده از → نشان داده می‌شود که **عملگر شرطی**[۴] نام دارد. به‌این‌ترتیب، A→B به این معنی است که اگر داشته باشیم A=True باشد آنگاه B=True.

A→B: اگر استخر گرم باشد، آنگاه من شنا خواهم کرد

با عملگرهای بیشتر می‌توان ایده‌های متفاوتی را بیان کرد. برای ایده‌های نقیض از !، **عملگر نقیض**[۵]، استفاده می‌کنیم. به‌این‌ترتیب، A! نقیض A است.

A!: استخر سرد است

B!: شنا نمی‌کنم

عکس نقیض[۶]: با داشتن A→B و دانستن اینکه من شنا نکرده‌ام، در مورد استخر چه می‌توان گفت؟ یک استخر گرم من را مجبور به شنا کردن می‌کرد، بنابراین اگر شنا نکرده‌ام، غیرممکن است که استخر گرم بوده باشد. هر عبارت شرطی دارای یک معادل **عکس نقیض** است. برای هر دو متغیر A و B:

A→B معادل با B!→A! است.

---

۱ Operator

۲ Variable

۳ مقادیر بین این دو در منطق فازی مجاز هستند ولی این منطق در این کتاب پوشش داده نخواهد شد.

۴ Conditional Operator

۵ Negation Operator

۶ Contrapositive

مثال دیگر: *اگر کد خوبی ننویسی، این کتاب را نخوانده‌ای.* عکس نقیض آن به‌این‌ترتیب است که *اگر این کتاب را بخوانی، می‌توانی کد خوبی بنویسی.* هر دو جمله یک معنی‌دارند که به روش‌های مختلفی بیان‌شده است[۱].

دو شرطی[۲]: توجه داشته باشد که گفتن «اگر استخر گرم است، من شنا خواهم کرد» به این معنی نیست که من فقط در آب گرم شنا می‌کنم. این عبارت هیچ حرفی در مورد آب سرد نمی‌زند. به‌عبارت‌دیگر، A→B به معنی B→A نیست. برای بیان هر دو شرط، از **دو شرطی** استفاده می‌کنیم:

A↔B: من شنا خواهم کرد اگر و فقط اگر استخر گرم باشد

در اینجا گرم بودن استخر معادل با شنا کردن است: یعنی اطلاع داشتن در مورد استخر به معنی اطلاع داشتن در مورد شنا کردن است و برعکس. مجدداً، به **خطای معکوس** توجه کنید: هرگز تصور نکنید که A→B از B→A به دست می‌آید.

عطف، فصل و فصل انحصاری[۳]: این عملگرها معروف‌ترین عملگرها هستند و اغلب به‌صورت صریح کدنویسی می‌شوند. عملگر عطف یا AND نشان می‌دهد تمام ایده‌ها درست (True) هستند، عملگر فصل یا OR نشان می‌دهد یکی از ایده‌ها درست است و عملگر فصل انحصاری یا XOR نشان می‌دهد درستی ایده‌ها نقیض یکدیگر است. مثال زیر را در رابطه با یک مهمانی در نظر بگیرید:

A: شما چای نوشیده‌اید

B: شما شیر نوشیده‌اید

A OR B: شما چیزی نوشیده‌اید

A AND B: شما چای و شیر به‌صورت ترکیب‌شده نوشیده‌اید

A XOR B: شما چای یا شیر (و نه ترکیب هر دو) نوشیده‌اید

اطمینان حاصل کنید که نحوه‌ی عملکرد عملگرهایی را که تاکنون دیده‌ایم فهمیده‌اید. جدول زیر تمام ترکیب‌های ممکن از دو متغیر را نشان می‌دهد. توجه داشته باشید که A→B با A OR B! معادل و A XOR B با (A↔B)! معادل است.

---

[۱] به‌هرحال، هر دو جمله درست هستند!

[۲] Biconditional

[۳] AND, OR and EXCLUSIVE OR (XOR)

جدول ۱-۱: عملگرهای منطقی برای چهار مقدار ممکن دو متغیر A و B

A	B	$!A$	$A \rightarrow B$	$A \leftrightarrow B$	A AND B	A OR B	A XOR B
✓	✓	✗	✓	✓	✓	✓	✗
✓	✗	✗	✗	✗	✗	✓	✓
✗	✓	✓	✓	✗	✗	✓	✓
✗	✗	✓	✓	✓	✗	✗	✗

جبر بولی[1]

همان‌گونه که جبر مقدماتی عبارات عددی را ساده می‌کند، **جبر بولی** عبارات منطقی را ساده می‌سازد.

شرکت‌پذیری[2]: پرانتزها تأثیری در دنباله‌ی عملگرهای AND و OR ندارند. همانند دنباله‌ی جمع‌ها یا ضرب‌ها در جبر مقدماتی، می‌توان آن‌ها را با هر ترتیبی محاسبه کرد:

```
A  AND (B  AND  C) = (A  AND  B)  AND  C
A  OR  (B  OR   C) = (A  OR   B)  OR   C
```

توزیع‌پذیری[3]: در جبر مقدماتی می‌توانیم عبارات ضربی را از جمع‌ها فاکتور بگیریم:

$$a \times (b + c) = (a \times b) + (a \times c)$$

به همین ترتیب، در منطق AND پس از OR معادل با OR حاصل از چند AND است و برعکس:

```
A  AND (B  OR  C) = (A  AND B)  OR  (A  AND C)
A  OR  (B  AND C) = (A  OR  B)  AND (A  OR  C)
```

قانون دمورگان[4]: هم‌زمان نمی‌تواند تابستان و زمستان باشد، پس یا تابستان نیست یا زمستان نیست و اینکه الآن تابستان نیست و زمستان هم نیست اگر و فقط اگر زمستان یا تابستان نباشد. با استفاده از این استدلال، AND را می‌توان به OR تبدیل کرد و برعکس:

```
!(A  AND  B) = !A  OR  !B
!A  AND  !B = !(A  OR  B)
```

---

[1] Boolean Algebra: به یاد جورج بول و کتاب وی در سال ۱۸۵۴ شامل ترکیب ریاضی و منطق برای اولین بار.

[2] Associativity

[3] Distributivity

[4] Demorgan's Law: دمورگان دوست بول بود. او به ایدا لاولیس جوان که اولین برنامه‌نویس دریک قرن قبل از ساخته‌شدن اولین کامپیوتر بود، درس می‌داد.

این قوانین موجب تغییر شکل مدل‌های منطقی، آشکار شدن ویژگی‌ها و ساده‌سازی عبارات می‌شوند. بیایید یک مسئله حل کنیم:

> سِرورِ داغ: یک سِرور از کار می‌افتد اگر دمای آن خیلی بالا برود درحالی‌که سیستم تهویه خاموش باشد. هم‌چنین سِرور از کار می‌افتد اگر دمای آن خیلی بالا برود و خنک‌کننده‌ی آن کار نکند. در چه شرایطی سِرور کار می‌کند؟

با مدل‌سازی مسئله در قالب متغیرهای منطقی، شرایط ازکارافتادن سِرور را در قالب یک عبارت می‌توان بیان کرد:

A: دمای سِرور بالا رفته است.
B: تهویه مطبوع خاموش است.
C: خنک‌کننده کار نمی‌کند.
D: سِرور از کار می‌افتد.

```
(A AND B) OR (A AND C) → D
```

با استفاده از توزیع‌پذیری می‌توانیم عبارت را به‌صورت زیر بنویسیم:

```
A AND (B OR C) → D
```

سِرورِ زمانی در شرایط (!D) کار می‌کند. عکس نقیض می‌گوید:

```
!D → !(A AND (B OR C))
```

استفاده از قانون دمورگان برای حذف پرانتزها استفاده می‌کنیم:

```
!D → !A OR !(B OR C)
```

با استفاده‌ی مجدد از قانون دمورگان:

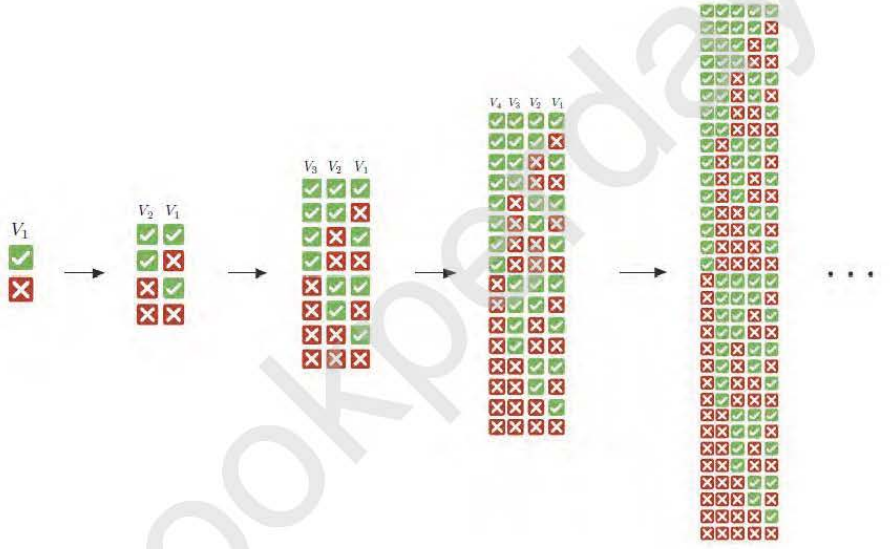
```
!D → !A OR !(B OR C)
!D → !A OR !B AND !C
```

این عبارت به ما می‌گوید زمانی که سِرور کار می‌کند، یا !A (یعنی دمای آن بالا نرفته است) یا !B AND !C (یعنی سیستم تهویه و خنک‌کننده هر دو کار می‌کنند).

جدول‌های درستی[۱]

راه دیگر تحلیل مدل‌های منطقی بررسی تمام حالات ممکن پیکره‌بندی متغیرهای آن است. **جدول درستی** برای هر متغیر یک ستون دارد. سطرها ترکیبات ممکن وضعیت متغیرها را نشان می‌دهند.

یک متغیر به دو سطر نیاز دارد: در یک سطر متغیر مقدار `True` می‌گیرد و در دیگری مقدار `False`. برای اضافه کردن یک متغیر، سطرها را با کپی کردن دو برابر می‌کنیم. مقدار متغیر جدید را در سطرهای اصلی `True` و در سطرهای کپی شده برابر با `False` قرار می‌دهید (شکل ۱-۵). اندازه‌ی جدول درستی به ازای هر متغیر اضافه‌شده دو برابر می‌شود، به همین دلیل این جدول را فقط برای تعداد کمی متغیر می‌توان ساخت[۲].



شکل ۱-۵: جدول‌های شامل پیکره‌بندی‌های ۱ تا ۵ متغیر منطقی

حال اجازه بدهید ببینیم چگونه می‌توان از یک جدول درستی برای تحلیل یک مسئله استفاده کرد.

سیستم شکننده: می‌خواهیم یک سیستم پایگاه داده با نیازمندی‌های زیر بسازیم:

I. اگر پایگاه داده قفل‌شده باشد، می‌توانیم داده‌ها را ذخیره کنیم.

II. نمی‌توان بر روی یک صف پرشده‌ی نوشتن قفل پایگاه داده قرارداد.

---

[۱] Truth Tables

[۲] یک جدول درستی برای ۳۰ متغیر بیش از یک میلیارد سطر دارد.

III. صف نوشتن یا پر است یا حافظه‌ی کَش بارگذاری شده است.
IV. اگر حافظه‌ی کَش بارگذاری شده باشد، نمی‌توان پایگاه داده را قفل کرد.

آیا چنین چیزی ممکن است؟ این سیستم تحت چه شرایطی کار می‌کند؟

ابتدا هر نیازمندی را به یک عبارت منطقی تبدیل می‌کنیم. این پایگاه داده با استفاده از چهار متغیر قابل مدل‌سازی است:

A: پایگاه داده قفل است.
B: می‌توان داده‌ها را ذخیره کرد.
C: صف نوشتن پر است.
D: حافظه‌ی کَش بارگذاری شده است.

```
I: A→B
II: !(A AND C)
III: C OR D
IV: D→!A
```

حال می‌توانیم یک جدول درستی (جدول ۱-۲) با تمام پیکره‌بندی‌های ممکن بسازیم. ستون‌های اضافی برای بررسی نیازمندی‌ها اضافه‌شده‌اند.

تمام نیازمندی‌ها در وضعیت‌های ۹ تا ۱۱ و ۱۳ تا ۱۵ برآورده شده‌اند. در این وضعیت‌ها A=False است یعنی نمی‌توان پایگاه داده را قفل کرد. توجه داشته باشید که فقط در وضعیت‌های ۱۰ و ۱۴ نمی‌توان حافظه‌ی کَش را بارگذاری کرد.

برای آزمایش آنچه آموخته‌اید، معمای گورخر را حل کنید[1]. این یک معمای منطقی معروف است که به‌اشتباه به اینشتین[2] نسبت داده‌شده است. گفته می‌شود فقط ۲٪ از مردم دنیا می‌توانند آن را حل کنند، ولی من در این مورد شک دارم. با استفاده از یک جدول درستی بزرگ و ساده‌سازی و ترکیب صحیح عبارات منطقی، مطمئن هستم می‌توانید آن را بشکنید.

هر زمان با چیزی مواجه شدید که دارای دو حالت ممکن است، به یاد داشته باشید که می‌توان از جدول درستی برای مدل‌سازی آن استفاده کرد. با استفاده از این روش بیان عبارات، ساده‌سازی آن‌ها و نتیجه‌گیری ساده می‌شود. حال اجازه بدهید تأثیرگذارترین کاربرد منطق را ببینیم: طراحی کامپیوترهای الکترونیک.

---

[1] http://code.energy/zebra-puzzle

[2] Albert Einstein (1897-1955): فیزیکدان نظری آلمانی، برنده‌ی جایزه‌ی نوبل و ارائه‌دهنده‌ی نظریه نسبیت. م.

جدول ۱-۲: جدول درستی برای بررسی اعتبار چهار عبارت

State #	A	B	C	D	I	II	III	IV	All four
1	✓	✓	✓	✓	✓	✗	✓	✗	✗
2	✓	✓	✓	✗	✓	✗	✓	✓	✗
3	✓	✓	✗	✓	✓	✓	✓	✗	✗
4	✓	✓	✗	✗	✓	✓	✗	✓	✗
5	✓	✗	✓	✓	✗	✗	✓	✗	✗
6	✓	✗	✓	✗	✗	✗	✓	✓	✗
7	✓	✗	✗	✓	✗	✓	✓	✗	✗
8	✓	✗	✗	✗	✗	✓	✗	✓	✗
9	✗	✓	✓	✓	✓	✓	✓	✓	✓
10	✗	✓	✓	✗	✓	✓	✓	✓	✓
11	✗	✓	✗	✓	✓	✓	✓	✓	✓
12	✗	✓	✗	✗	✓	✓	✗	✓	✗
13	✗	✗	✓	✓	✓	✓	✓	✓	✓
14	✗	✗	✓	✗	✓	✓	✓	✓	✓
15	✗	✗	✗	✓	✓	✓	✓	✓	✓
16	✗	✗	✗	✗	✓	✓	✗	✓	✗

منطق در محاسبات

گروهی از متغیرهای منطقی می‌توانند اعداد را در قالب دودویی[1] نشان دهند[2]. عملگرهای منطقی بر روی ارقام دودویی را می‌توان باهم ترکیب کرد و محاسبات عمومی را انجام داد. **دروازه‌های منطقی**[3] عملگرهای منطقی را بر روی جریان الکتریکی انجام می‌دهند. از این دروازه‌ها در مدارهای الکتریکی انجام‌دهنده‌ی محاسبات با سرعت بسیار زیاد استفاده می‌شود.

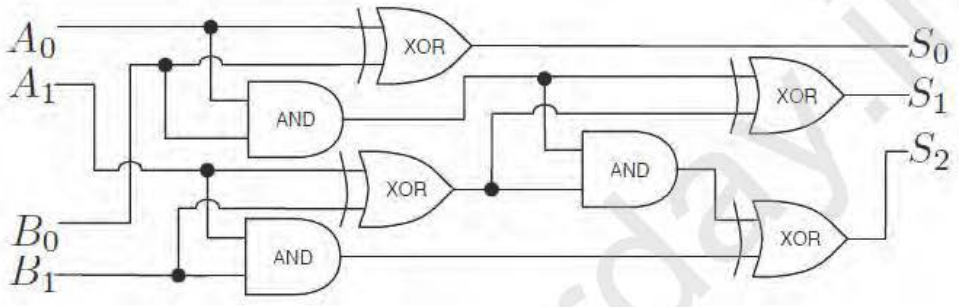
---

[1] Binary Form: استفاده از صفر و یک برای نمایش اعداد. م.

[2] در این حالت True=1 و False=0 است. اگر در رابطه بااینکه ۱۰۱ در قالب دودویی نمایشگر عدد ۵ است، هیچ اطلاعاتی ندارید، پیوست I را که شامل توضیحی بر سیستم‌های عددی است ببینید.

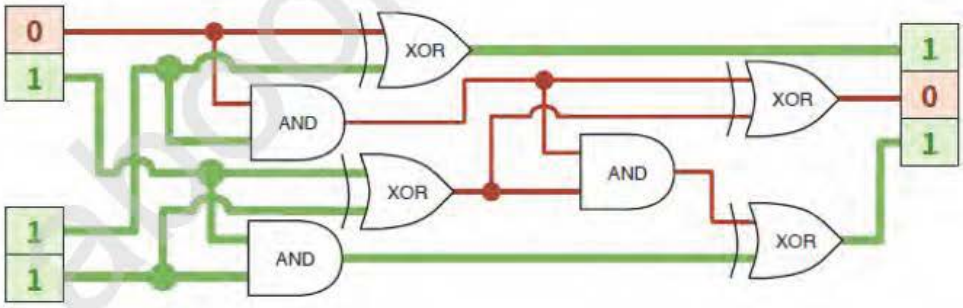
[3] Logic Gate

یک دروازه‌ی منطقی مقادیر را از طریق سیم‌های ورودی دریافت کرده، عملیات خود را انجام داده و نتیجه را بر روی سیم خروجی قرار می‌دهد. دروازه‌های مختلفی مانند دروازه AND، دروازه OR، دروازه XOR و غیره وجود دارند. مقادیر True و False به‌وسیله‌ی جریان الکتریکی با ولتاژ بالا یا پایین نمایش داده می‌شوند. با استفاده از دروازه‌ها می‌توان عبارات منطقی پیچیده را تقریباً بلافاصله محاسبه کرد. به‌عنوان‌مثال، مدار الکتریکی که دو عدد را جمع می‌زند:



شکل ۱-۶: مداری برای جمع زدن اعداد ۲ بیتی دریافت شده از طریق متغیرهای منطقی (A_1A_0 و B_1B_0) در یک عدد ۳ بیتی ($S_2S_1S_0$)

اجازه بدهید ببینیم این مدار چگونه کار می‌کند. یک دقیقه زمان صرف کرده و عملیات انجام‌شده به‌وسیله‌ی مدار را دنبال کنید تا بفهمید جادو چگونه رخ می‌دهد:



شکل ۱-۷: محاسبه‌ی $2 + 3 = 5$ (در دودویی، $10 + 11 = 101$)

برای استفاده از مزایای این محاسبات سریع، ما مسائل عددی را به قالب دودویی/منطقی آن‌ها درمی‌آوریم. جدول‌های درستی در مدل‌سازی و آزمایش مدار کمک می‌کنند. جبر بولی عبارات را ساده می‌کند و درنتیجه موجب ساده شدن مدارها می‌شود.

در ابتدا، مدارها با استفاده از لامپ‌های الکتریکی سنگین، حجیم و گران ساخته می‌شدند. به‌مرور، لامپ‌ها با ترانزیستورها جایگزین شدند و ناگهان دروازه‌های منطقی قابل‌تولید شدند. ما همچنان به دنبال کشف راه‌هایی برای کوچک و کوچک‌تر کردن ترانزیستورها هستیم[1]. قواعد عملکرد پردازنده‌های نوین همچنان بر اساس جبر بولی قرار دارند. یک پردازنده‌ی نوین فقط یک مدار شامل میلیون‌ها سیم و دروازه‌ی منطقی میکروسکوپی است که جریان الکتریکی اطلاعات را پردازش می‌کنند.

۱-۳- شمارش

شمردن صحیح چیزها اهمیت زیادی دارد. شما این کار را بارها و در زمان کار کردن با مسائل محاسباتی انجام داده‌اید[2]. ریاضی در این بخش پیچیده‌تر است، ولی نترسید. برخی مردم فکر می‌کنند نمی‌توانند کدنویسان خوبی باشند زیرا فکر می‌کنند در ریاضی بد هستند. خب، من درس ریاضی دبیرستان را مردود شدم، ولی الآن اینجا هستم. آن نوع ریاضی که موجب می‌شود کدنویس خوبی باشید آن چیزی نیست که در امتحانات معمول ریاضی مدرسه وجود دارد.

بیرون از مدرسه، فرمول‌ها و رویه‌های گام‌به‌گام حفظ نمی‌شوند. در زمان نیاز آن‌ها را در اینترنت جستجو می‌کنند. محاسبات نباید با قلم و کاغذ انجام شود. آن چیزی که یک کدنویس خوب نیاز دارد، درک صحیح است. یادگرفتن مسائل شمارش این درک را تقویت می‌کند. اجازه بدهید به‌صورت گام‌به‌گام با چند ابزار آشنا شویم: حاصل‌ضرب، جایگشت، ترکیب و مجموع.

حاصل‌ضرب

اگر یک پیشامد با n روش مختلف و یک پیشامد دیگر با m روش مختلف رخ دهد، تعداد روش‌هایی که هر دو پیشامد می‌توانند رخ دهند برابر با $n \times m$ است. به‌عنوان‌مثال:

> شکستن کد: یک کد PIN از دو رقم و یک حرف تشکیل‌شده است. آزمایش کردن هر PIN یک ثانیه طول می‌کشد، در بدترین حالت، چقدر زمان برای شکستن کد نیاز است؟

---

[1] در سال ۲۰۱۶، محققان ترانزیستورهای قابل‌استفاده‌ای با اندازه‌ی ۱ نانومتر ساختند. برای مقایسه، باید بدانید که قطر اتم طلا ۰.۱۵ نانومتر است.

[2] **شمارش** و **منطق** به زمینه‌ی مهمی از علم کامپیوتر به نام **ریاضی گسسته** تعلق دارند.

دو رقم را می‌توان به ۱۰۰ راه (۰۰ تا ۹۹) و یک حرف را به ۲۶ راه (A تا Z) انتخاب کرد. ازاین‌رو، $100 \times 26 = 2600$ کد مختلف وجود دارد. در بدترین حالت، باید همه‌ی کدها را امتحان کنیم تا کد صحیح را بیابیم. بعد از ۲۶۰۰ ثانیه (۴۳ دقیقه) می‌توان یک کد را شکست.

> تیم سازی: ۲۳ نامزد برای پیوستن به تیم شما وجود دارند. برای هر نامزد، شما یک سکه را می‌اندازید و اگر شیر آمد او را انتخاب می‌کنید. چند پیکره‌بندی مختلف برای تیم وجود دارد؟

قبل از استخدام، تنها پیکره‌بندی ممکن در تیم، حضور شما به‌تنهایی است. هر پرتاب سکه تعداد حالات پیکره‌بندی را دو برابر می‌کند. این کار ۲۳ بار تکرار می‌شود، بنابراین باید ۲ به توان ۲۳ را حساب کنیم:

$$\overbrace{2 \times 2 \times \ldots \times 2}^{23 \text{ بار}} = 2^{23} = 8{,}388{,}608$$

یعنی بیش از ۸ میلیون پیکره‌بندی مختلف برای تیم شما وجود دارد. توجه داشته باشید که همچنان یکی از این پیکره‌بندی‌ها حضور شما به‌تنهایی در تیم است.

جایگشت[1]

اگر ما n عنصر داشته باشیم، می‌توانیم به n فاکتوریل ($n!$) راه مختلف آن‌ها را قرار دهیم. فاکتوریل به‌صورت انفجاری افزایش می‌یابد و برای مقادیر کوچک n بسیار زیاد می‌شود. اگر با این مفهوم آشنا نیستید:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$$

به‌سادگی می‌توان فهمید که $n!$ برابر با تعداد راه‌های قرار دادن n عنصر است. عنصر اول را به چندراه می‌توانید از بین n عنصر انتخاب کنید؟ بعد از انتخاب عنصر اول، به چندراه می‌توانید عنصر دوم را انتخاب کنید؟ بعدازآن، چند گزینه برای عنصر سوم وجود دارد؟ در مورد این مسئله فکر کنید، سپس به سراغ مسائل پیچیده‌تر بروید[2].

---

[1] Permutation

[2] به‌صورت قراردادی $0! = 1$. می‌گوییم یک‌راه برای قرار دادن صفر عنصر وجود دارد.

فروشنده‌ی دوره‌گرد: شرکت حمل‌ونقل شما باید کالاهایی را به ۱۵ شهر تحویل دهد. شما می‌خواهید بدانید با چه ترتیبی به این شهرها بروید تا حداقل مصرف بنزین را داشته باشد. اگر محاسبه‌ی طول هر مسیر یک میکروثانیه طول بکشد، محاسبه کردن طول همه‌ی مسیرها چقدر طول می‌کشد؟

هر جایگشت از ۱۵ شهر یک مسیر متفاوت است. فاکتوریل، تعداد جایگشت‌های منحصربه‌فرد را به ما می‌دهد، بنابراین تعداد مسیرها عبارت است از:

$$15! = 15 \times 14 \times \ldots \times 1 \approx 13 \times 10^{12}$$

که اگر آن را به میکروثانیه تقسیم کنید، حدود ۱۵ روز خواهد شد. حال اگر ۲۰ شهر داشته باشید، حدود ۷۷ هزار سال طول می‌کشد.

تُن دقیق: یک موسیقیدان در حال مطالعه‌ی یک میزان با ۱۳ نت مختلف است. او می‌خواهد تمام ملودی‌های ممکن شامل فقط ۶ نت را بسازد. هر نت در هر ملودی فقط یک‌بار باید نواخته شود و ملودی شامل ۶ نت باید به مدت ۱ ثانیه نواخته شود. زمان اجرای این صداها برای او چقدر خواهد بود؟

ما باید جایگشت‌های شامل ۶ نت از ۱۳ نت را بشماریم. برای صرف‌نظر کردن از جایگشت‌های نت‌های بلااستفاده، باید توسعه‌ی فاکتوریل بعد از ششمین عامل را متوقف کنیم. به‌صورت رسمی، $n!/(n-m)!$ برابر با تعداد جایگشت‌های m عنصر از n عنصر ممکن است. در مثال ما:

$$\frac{13!}{(13-6)!} = \frac{13 \times 12 \times 11 \times 10 \times 9 \times 8 \times 7!}{7!}$$

$$= 13 \times 12 \times 11 \times 10 \times 9 \times 8 = 1{,}235{,}520$$

پس بیش از ۱/۲ میلیون ملودی یک‌ثانیه‌ای وجود دارد و گوش دادن به همه‌ی آن‌ها ۳۴۳ ساعت طول می‌کشد. بهتر است موسیقیدان را قانع کنیم ملودی مطلوب را به روش دیگری پیدا کند.

جایگشت با عناصر مشابه

اگر عناصر مشابه باشند، فاکتوریل ($n!$) تعداد راه‌های قرار دادن n عنصر را بیش از مقدار واقعی می‌شمارد. عناصر مشابهی که جای خود را عوض می‌کنند نباید به‌عنوان یک جایگشت متفاوت شمرده شوند.

در یک دنباله از n عنصر که 3 عنصر آن مشابه هستند، $3!$ راه برای قرار دادن مجدد عناصر مشابه وجود دارد. بنابراین، $n!$ هر جایگشت واحد را $3!$ بار می‌شمارد. برای به دست آوردن تعداد جایگشت‌های متمایز، باید $n!$ را بر این تعداد شمارش اضافی تقسیم کنیم. به‌عنوان‌مثال، تعداد جایگشت‌های متمایز، حروف عبارت CODE ENERGY برابر با $\frac{10!}{3!}$ است (چون ۳ عنصر مشابه وجود دارد).

> بازی با DNA: یک زیست‌شناس در حال مطالعه‌ی بخش DNA مربوط به یک بیماری ژنتیک است. بخش مربوطه شامل ۲۳ زوج است که ۹ زوج آن A-T و ۱۴ زوج C-G هستند. او می‌خواهد یک برنامه شبیه‌سازی‌شده بر روی هر بخش ممکن DNA شامل این تعداد زوج پایه را اجرا کند. چند شبیه‌سازی مختلف را باید اجرا کند؟

ابتدا باید تمام جایگشت‌های ۲۳ زوج پایه را بشماریم. سپس نتیجه را بر تعداد ۹ زوج A-T مجرا و ۱۴ زوج C-G مجرا تقسیم کنیم:

$$23!/(9! \times 14!) = 871{,}190$$

ولی مسئله هنوز تمام نشده است. نحوه‌ی قرار گرفتن جهت زوج‌های پایه را در نظر بگیرید:

مشابه نیست با

برای هر دنباله از ۲۳ زوج، به تعداد ۲ به توان ۲۳ پیکره‌بندی در دو جهت مختلف وجود دارد، بنابراین تعداد کل حالات عبارت است:

$$817{,}190 \times 2^{23} = 7 \times 10^{12}$$

و این تعداد فقط برای دنباله‌های شامل ۲۳ زوج پایه یا توزیع معلوم است. کوچک‌ترین مدل DNA شناخته‌شده تاکنون مربوط به سیرکوویروس خوکی[۱] کوچک است که ۱۸۰۰ زوج پایه دارد! کدهای DNA و زندگی از منظر فناوری بسیار شگفت‌انگیز هستند. واقعاً دیوانه کننده است: DNA انسان حدود ۳ میلیارد زوج پایه دارد که در هر یک از ۳ تریلیون سلول بدن انسان تکرار شده‌اند.

ترکیب[۲]

یک دسته کارت شامل ۱۳ کارت از یک نوع را در نظر بگیرید. به چندراه می‌توانید ۶ کارت را به هم‌بازی‌تان بدهید؟ دیدیم که $\frac{13!}{(13-6)!}$ جایگشت برای ۶ عنصر از ۱۳ عنصر وجود دارد. ازآنجایی‌که ترتیب ۶ کارت اهمیت ندارد، باید این تعداد را بر $6!$ تقسیم کنیم تا به دست بیاوریم:

$$\frac{13!}{6!\,(13-6)!} = 1{,}716$$

ضریب دوجمله‌ای $\binom{n}{m}$ برابر با تعداد راه‌های انتخاب m عنصر از مجموعه‌ای شامل n عنصر، بدون توجه به ترتیب، است:

$$\binom{n}{m} = \frac{n!}{m!\,(n-m)!}$$

این ضریب به‌صورت «انتخاب m از n» خوانده می‌شود.

> وزیرهای شطرنج: شما یک صفحه‌ی خالی شطرنج و ۸ مهره‌ی وزیر دارید که می‌توانید آن‌ها را هر جای صفحه قرار دهید. به چندراه مختلف می‌توانید وزیرها را در صفحه جا بدهید؟

صفحه‌ی شطرنج ۶۴ خانه در یک محیط ۸ در ۸ دارد. تعداد راه‌های انتخاب ۸ خانه از ۶۴ خانه ممکن به‌صورت زیر است[۳]:

$$\binom{64}{8} \approx 4.4 \times 10^9$$

---

[۱] Porcine Circovirus

[۲] Combination

[۳] راهنمایی: انتخاب ۸ از ۶۴ را با عبارت 64 Choose 8 در گوگل جستجو کنید.

مجموع

محاسبه‌ی مجموع دنباله‌ها معمولاً در زمان شمارش اتفاق می‌افتد. مجموع ترتیبی با استفاده از نماد **سیگمای بزرگ** (Σ) نشان داده می‌شود. این نماد نشان می‌دهد چگونه یک عبارت برای هر مقدار i در قالب جمع محاسبه می‌شود:

$$\sum_{Start\ i}^{Finish\ i} experssion\ of\ i$$

به‌عنوان‌مثال، محاسبه‌ی مجموع ۵ عدد فرد اول به‌صورت زیر نوشته می‌شود:

$$\sum_{i=0}^{4}(2i+1) = 1+3+5+7+9$$

توجه داشته باشید که i باید با هر عدد بین صفر تا ۴ جایگزین شده تا اعداد ۱، ۳، ۵، ۷ و ۹ را بسازد.

محاسبه‌ی مجموع n عدد طبیعی اول به‌صورت زیر است:

$$\sum_{i=1}^{n} i = 1+2+3+\cdots+(n-1)+n$$

زمانی که ریاضیدان بزرگ، گاوس[1]، ده سال داشت و از محاسبه‌ی مجموع اعداد طبیعی به‌صورت تک‌به‌تک خسته شده بود، این حیله‌ی جالب را پیدا کرد:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

آیا می‌توانید حدس بزنید گاوس چگونه به این کشف رسید؟ حیله‌ی مربوطه در پیوست II شرح داده‌شده است. اجازه بدهید ببینیم چگونه می‌توانیم از آن برای حل مسئله استفاده کنیم:

> پرواز ارزان: شما باید در ۳۰ روز آینده پروازی به نیویورک داشته باشید. قیمت بلیت هواپیما به‌صورت غیرقابل‌پیش‌بینی و بر اساس تاریخ رفت‌وبرگشت تغییر می‌کند. چند زوج روز را باید بررسی کنید تا ارزان‌ترین قیمت پرواز را به نیویورک و برگشت ازآنجا در ۳۰ روز آینده به دست آورید؟

هر زوج روز بین امروز (روز ۱) و روز آخر (روز ۳۰)، معتبر است تا زمانی که برگشت روزی یکسان یا بعد از روز رفت باشد. ازاین‌رو، ۳۰ زوج با روز ۱، ۲۹ زوج با روز ۲، ۲۸ زوج با روز ۳ شروع‌شده و

[1] Gauss (1777-1855): ریاضیدان، فیزیکدان و ستاره‌شناس آلمانی که ملقب به شاهزاده‌ی ریاضیدانان است.م.

الی‌آخر. تنها یک زوج نیز با روز ۳۰ شروع می‌شود؛ بنابراین تعداد کل زوج‌هایی که باید بررسی شوند عبارت است از $30 + 29 + 28 + \cdots + 1$؛ که می‌توان آن را به‌صورت زیر نوشت:

$$\sum_{i=1}^{30} i = \frac{30(30+1)}{2} = 465$$

البته این مسئله را با ترکیب نیز می‌توانیم حل کنیم. از ۳۰ روز ممکن، می‌خواهیم ۲ روز را انتخاب کنیم. ترتیب نیز اهمیت ندارد: روز جلوتر روز رفت و روز عقب‌تر روز برگشت است. به‌این‌ترتیب داریم $\binom{30}{2} = 435$؛ اما صبر کنید، باید تعداد حالاتی که در آن‌ها روز رفت‌وبرگشت یکسان است را نیز حساب کنیم. از این نوع ۳۰ حالت وجود دارد، بنابراین $\binom{30}{2} + 30 = 465$.

۱-٤- احتمال

قوانین تصادفی بودن به شما در شرط‌بندی، پیش‌بینی آب‌وهوا یا طراحی یک سیستم پشتیبان با حداقل ریسک خطا کمک می‌کنند. اگرچه این قوانین ساده هستند ولی توسط مردم بد فهمیده شده‌اند.

اجازه بدهید از مهارت شمارش خود برای محاسبه‌ی شانس استفاده کنیم. سپس یاد خواهیم گرفت که از انواع مختلف پیشامدها برای حل مسائل استفاده کنیم. درنهایت، خواهیم دید افرادی که شرط‌بندی می‌کنند چرا همه‌چیز را از دست می‌دهند.

```
int getRandomNumber()
{
    return 4;       //انتخاب شده با پرتاب تاس
                    //تضمین می شود که تصادفی است
}
```

شکل ۱-۸: عدد تصادفی (دریافت شده از http://xkcf.com)

خروجی‌های شمارش

یک تاس دارای شش حالت مختلف است. ازاین‌رو، شانس آمدن عدد چهار برابر با $1/6$ است. شانس آمدن یک عدد فرد چقدر است؟ این اتفاق به سه شکل ممکن است رخ دهد (یک، سه و پنج)،

بنابراین شانس آن برابر با $3/6 = 1/2$ است. به‌صورت رسمی، **احتمال** رخ دادن یک پیشامد به‌صورت زیر است:

$$P\left(\text{پیشامد}\right) = \frac{\text{تعداد حالات رخ دادن پیشامد}}{\text{تعداد کل حالات}}$$

این فرمول جواب می‌دهد، زیرا شانس رخ دادن هر خروجی ممکن، با بقیه یکسان است: تاس متعادل ساخته‌شده و در پرتاب تقلب نمی‌شود.

> تیم سازی مجدد: ۲۳ نامزد برای پیوستن به تیم شما وجود دارند. برای هر نامزد، شما یک سکه را می‌اندازید و اگر شیر آمد او را انتخاب می‌کنید. شانس انتخاب نشدن هیچ‌کدام از نامزدها چقدر است؟

قبلاً دیدم که تعداد $2^{23} = 8{,}388{,}608$ حالت مختلف وجود دارد. تنها راه انتخاب هیچ‌کس این است که در ۲۳ پرتاب متوالی سکه، خط بیاید. احتمال رخ دادن این پیشامد برابر است با

$$P(nobody) = \frac{1}{8{,}388{,}608}$$

برای این‌که بهتر این عدد را درک کنید، بد نیست بدانید که احتمال سقوط یک پرواز تجاری حدود یک در پنج میلیون است.

پیشامدهای مستقل[۱]

اگر یک سکه و یک تاس را بیندازید، شانس آمدن شیر و عدد ۶ برابر است با

$$\frac{1}{2} \times \frac{1}{6} = \frac{1}{12} \approx 0.08$$

که معادل ۸٪ است. زمانی که خروجی یک پیشامد تأثیری بر خروجی پیشامد دیگر نداشته باشد، این پیشامدها **مستقل** هستند. احتمال رخ دادن دو پیشامد مستقل برابر است با حاصل‌ضرب احتمال هر یک از آن‌ها.

---

[۱] Independent Events

پشتیبان گرفتن: شما نیاز دارید که داده‌هایی را برای یک سال ذخیره کنید. احتمال ازکارافتادن یک دیسک یک در یک میلیارد است. هزینه‌ی دیسک دیگر ۲۰٪ هزینه‌ی دیسک اول است ولی احتمال ازکارافتادن آن، یک در دو هزار است. کدام را باید بخرید؟

اگر شما سه دیسک ارزان استفاده کنید، فقط درصورتی‌که هر سه دیسک از کار بیفتند، داده‌ها را از دست خواهید داد. احتمال این پیشامد برابر است با $(\frac{1}{2,000})^3 = \frac{1}{8,000,000,000}$. این افزونگی ریسک از دست دادن داده‌ها را نسبت به دیسک گران کاهش می‌دهد، ضمن این‌که هزینه‌ی آن نیز فقط ۶۰٪ هزینه‌ی دیسک گران است.

پیشامدهای ناسازگار[۱]

با یک‌بار تاس انداختن نمی‌توان هم به عدد ۴ و هم به یک عدد فرد رسید. ازاین‌رو، احتمال آمدن عدد ۴ و یک عدد فرد $\frac{2}{3} = \frac{1}{6} + \frac{1}{2}$ است. زمانی که دو پیشامد نمی‌توانند هم‌زمان رخ دهند، **ناسازگار** هستند. اگر به رخ دادن یکی از دو پیشامد ناسازگار نیاز داشته باشید، احتمال آن برابر با مجموع احتمال هر یک از آن‌ها است.

انتخاب طرح ثبت‌نام: یک وب‌سایت سه طرح برای ثبت‌نام پیشنهاد می‌دهد: رایگان، معمولی و پیشرفته. شما می‌دانید که یک مشتری جدید تصادفی به‌احتمال ۷۰٪ طرح رایگان، به‌احتمال ۲۰٪ طرح معمولی و به‌احتمال ۱۰٪ طرح پیشرفته را انتخاب می‌کند. شانس این‌که یک کاربر جدید برای یک طرح پولی ثبت‌نام کند چقدر است؟

این پیشامدها ناسازگار هستند: یک کاربر نمی‌تواند هم‌زمان هر دو طرح معمولی و پیشرفته را انتخاب کند. احتمال این‌که یک کاربر پرداخت انجام دهد برابر است با $0.2 + 0.1 = 0.3$.

پیشامدهای مکمل[۲]

انداختن یک تاس نمی‌تواند هم‌زمان منجر به یک عدد مضرب ۳ (یعنی ۳ و ۶) و یک عدد غیر بخش‌پذیر بر ۳ شود، بلکه فقط یکی از آن‌ها می‌تواند رخ دهد. احتمال آمدن یک عدد مضرب ۳ برابر با

---

[۱] Mutual Exclusive Events

[۲] Complementary Events

$\frac{1}{3} = \frac{2}{6}$ است، لذا احتمال آمدن یک عدد غیر بخش‌پذیر بر ۳ برابر است با $\frac{2}{3} = 1 - \frac{1}{3}$. زمانی که دو پیشامد ناسازگار تمام خروجی‌های ممکن را پوشش می‌دهند، آن‌ها را **مکمل** می‌نامند. ازاین‌رو، مجموع احتمال تمام پیشامدهای مکمل برابر با ۱۰۰٪ است.

> بازی دفاع از برج: قلعه‌ی شما توسط پنج برج نگهبانی می‌شود. هر برج به‌احتمال ۲۰٪ قادر به از کار انداختن مهاجمان قبل از رسیدن به دروازه است. شانس متوقف کردن مهاجمان چقدر است؟

داریم $0.2 + 0.2 + 0.2 + 0.2 + 0.2 = 1$ یا ۱۰۰٪ شانس زدن دشمن وجود دارد، درست است؟ *اشتباه است!* هیچ‌گاه احتمال پیشامدهای مستقل را باهم جمع نزنید، این‌یک اشتباه معمول است. دومرتبه از پیشامدهای مکمل استفاده کنید:

- ۲۰٪ شانس زدن دشمن مکمل ۸۰٪ شانس نزدن آن است. احتمال این‌که همه‌ی برج‌ها نتوانند دشمن را بزنند برابر است با $0.8^5 \approx 0.33$.
- پیشامد اینکه همه‌ی برج‌ها قادر به زدن دشمن نباشند مکمل این است که حداقل یکی از برج‌ها دشمن را بزند. احتمال متوقف کردن دشمن برابر است با $1 - 0.33 = 0.67$.

مغالطه‌ی شرط‌بندی[1]

اگر شما یک سکه‌ی معمولی را ده بار بیندازید و ده بار شیر بیاید، آنگاه احتمال شیر آمدن در پرتاب یازدهم بیشتر از آمدن خط است؟ یا در زمان بازی‌های شانسی با اعداد ۱ تا ۶، شانس بردن شما نسبت به بازی با اعداد بیشتر، کمتر است؟

قربانی مغالطه‌ی شرط‌بندی نشوید. پیشامدهای گذشته هرگز بر خروجی یک پیشامد مستقل تأثیر نمی‌گذارند. هرگز. هیچ‌وقت. در یک بازی کاملاً شانسی، شانس انتخاب هر عدد خاص دقیقاً معادل با سایر اعداد است. هیچ «قانونی مخفی» در مورد اجبار اعدادی که به‌طور تکراری درگذشته انتخاب‌نشده‌اند به انتخاب شدن در آینده وجود ندارد.

---

[1] The Gambler's Fallacy

احتمالات پیشرفته

جزییات بسیار بیشتری در مورد احتمالات نسبت به آنچه ما در اینجا پوشش دادیم وجود دارد. همیشه به یاد داشته باشید که برای حل مسائل پیچیده به دنبال ابزارهای بیشتری بگردید. به‌عنوان‌مثال:

> تیم سازی مجدد و مجدد: ۲۳ نامزد برای پیوستن به تیم شما وجود دارند. برای هر نامزد، شما یک سکه را می‌اندازید و اگر شیر آمد او را انتخاب می‌کنید. شانس انتخاب هفت نفر یا کمتر چقدر است؟

بله، سخت است. جستجو در این رابطه در گوگل شما را به «توزیع دوجمله‌ای[۱]» می‌رساند. می‌توانید این مسئله را بر روی Wolfram Alpha[۲] از طریق تایپ B(23,1/2)<=7 مصورسازی کنید.

نتیجه‌گیری

در این فصل، چیزهایی را دیدیم که رابطه‌ی نزدیکی با حل مسئله‌دارند، ولی این مطالب واقعاً شامل هیچ‌گونه کدنویسی نیستند. بخش ۱-۱ به بررسی چرایی و چگونگی نوشتن چیزها بر روی کاغذ می‌پردازد. ما مدل‌هایی برای مسائل خود می‌سازیم و از ابزارهای مفهومی بر روی مدل‌های ساخته‌شده استفاده می‌کنیم. بخش ۱-۲ یک جعبه‌ابزار برای مدیریت منطق، با جبر بولی و جدول‌های درستی ارائه می‌کند.

بخش ۱-۳ اهمیت شمارش حالات ممکن و پیکره‌بندی‌های مسائل مختلف را نشان می‌دهد. یک محاسبه‌ی سرانگشتی سریع می‌تواند به شما نشان دهد که آیا یک محاسبه سرراست و بدون پیچیدگی است یا خیر. برنامه‌نویسان تازه‌کار اغلب زمان را برای تحلیل تعداد زیادی سناریو تلف می‌کنند. درنهایت، بخش ۱-۴ قوانین بنیادین محاسبه‌ی شانس را شرح می‌دهد. احتمال در زمان توسعه‌ی راه‌حل‌هایی که باید با جهان فوق‌العاده ولی غیرقطعی ما تعامل داشته باشند، بسیار مفید است.

با این مطالب، ما بسیاری از جنبه‌های مهم آن چیزی را که دانشگاهیان به آن *ریاضی گسسته* می‌گویند نشان دادیم. نظریات جذاب بسیار بیشتری را می‌توانید از مراجع زیر یا با جستجو در ویکی‌پدیا به دست

---

[۱] Binomial Distribution

[۲] http://wolframalpha.com: یک موتور محاسباتی دانش است که با پردازش داده‌ها و اطلاعات قادر به پاسخگویی به سؤالاتی است که سایر موتورها قادر به پاسخگویی به آن‌ها نیستند. یکی از مهم‌ترین و کاربردی‌ترین قسمت‌های این موتور جستجو توانایی آن در حل مسائل ریاضی است. م.

آورید. به‌عنوان‌مثال، می‌توانید از «اصل لانه‌کبوتری[۱]» برای اثبات این‌که حداقل دو نفر در شهر نیویورک دارای تعداد موهای دقیقاً یکسانی هستند استفاده کنید.

برخی از چیزهایی که ما در اینجا آموختیم بسیار مرتبط با فصل بعد که در آن جنبه‌های بسیار مهمی را از علوم کامپیوتر بررسی خواهیم کرد، هستند.

مراجع

- Discrete Mathematics and its Applications, 7th Edition
  - Get it at https://code.energy/rosen
- Prof. Jeannette Wing's slides on computational thinking
  - Get it at https://code.energy/wing

---

۱ Pigeonhole Principle

فصل ۲

پیچیدگی

تقریباً در هر محاسبه‌ای، ترتیب‌های متنوعی برای انجام فرایند وجود دارد. انتخاب ترتیبی که زمان موردنیاز برای محاسبه را کمینه کند، الزامی است.

– ایدا لاولیس[۱]

مرتب کردن ۲۶ کارت به‌هم‌ریخته چقدر زمان می‌برد؟ اگر ۵۲ کارت داشته باشید، آیا زمان دو برابر خواهد شد؟ این زمان برای مرتب کردن هزاران دسته کارت چقدر بیشتر می‌شود؟ پاسخ به **روشی**[۲] که برای مرتب کردن کارت‌ها استفاده می‌شود وابسته است.

یک روش، فهرستی از دستورالعمل‌های غیرمبهم برای رسیدن به یک هدف است. روشی را که همیشه به یک دنباله‌ی متناهی از عملکردها نیاز دارد یک **الگوریتم**[۳] می‌نامند. به‌عنوان نمونه، یک الگوریتم مرتب‌سازی کارت، روشی است که عملیات موردنیاز برای مرتب کردن یک دسته کارت ۲۶-تایی بر اساس عدد و نوعشان را مشخص می‌کند.

عملیات کمتر به توان محاسباتی کمتری نیاز دارد. ما به راه‌حل‌های سریع علاقه‌مندیم، بنابراین بر تعداد عملیات در الگوریتم‌های خود نظارت می‌کنیم. بسیاری از الگوریتم‌ها نیازمند تعداد عملیات با رشد زیاد در زمان افزایش اندازه‌ی ورودی هستند. به‌عنوان‌مثال، الگوریتم مرتب‌سازی کارت ما عملیات اندکی برای مرتب‌سازی ۲۶ کارت انجام می‌دهد، ولی چهار برابر آن را برای مرتب کردن ۵۲ کارت انجام می‌دهد!

برای جلوگیری از شگفت‌زدگی‌های ناخوشایند در زمان افزایش اندازه‌ی مسئله، ما **پیچیدگی زمانی**[۴] الگوریتم خود را به دست می‌آوریم. در این فصل شما یاد خواهید گرفت:

---

[۱] Ada Lovelace (1815-1852): ریاضیدان، نویسنده و نخستین برنامه‌نویس کامپیوتر و همکار چارلز بابیج درزمینه‌ی کامپیوتر همه‌منظوره وی به نام «موتور تحلیلی». م.

[۲] Method

[۳] Algorithm: برگرفته از نام ابوجعفر محمد بن موسی خوارزمی، ریاضیدان بزرگ ایرانی در قرن سوم هجری. م.

[۴] Time Complexity

- پیچیدگی‌های **زمانی** را محاسبه و تفسیر کنید.
- رشد آن‌ها را با مفهوم **اُی‌بزرگ** بیان کنید.
- از الگوریتم‌های **نمایی** دوری کنید.
- اطمینان حاصل کنید که به‌اندازه‌ی کافی **حافظه**ی کامپیوتر دارید.

اما در ابتدا، پیچیدگی زمانی را چگونه تعریف می‌کنیم؟

پیچیدگی زمانی به‌صورت $\mathbb{T}(n)$ نوشته می‌شود. این عبارت تعداد عملیاتی را که الگوریتم در زمان پردازش یک ورودی با اندازه‌ی n انجام می‌دهد بیان می‌کند. ما به $\mathbb{T}(n)$ یک الگوریتم **هزینه‌ی اجرای**[1] آن می‌گوییم. اگر الگوریتم مرتب‌سازی کارت ما دارای $\mathbb{T}(n) = n^2$ باشد، می‌توانیم پیش‌بینی کنیم برای مرتب‌سازی یک دسته کارت در زمان دو برابر شدن تعداد کارت‌ها به چقدر زمان بیشتر نیاز دارد: $\frac{\mathbb{T}(2n)}{\mathbb{T}(n)} = 4$.

امید به بهترین، آماده شدن برای بدترین

آیا مرتب کردن تعدادی کارت که هم‌اکنون مرتب هستند، سریع‌تر نیست؟ اندازه‌ی ورودی تنها مشخصه‌ی تأثیرگذار بر تعداد عملیات موردنیاز یک الگوریتم نیست. وقتی یک الگوریتم می‌تواند مقادیر مختلفی برای $\mathbb{T}(n)$ با مقدار مشابه n داشته باشد، به حالات زیر می‌رسیم:

- بهترین حالت: زمانی که الگوریتم به ازای هراندازه‌ی ورودی، به حداقل تعداد عملیات نیاز داشته باشد. در مرتب‌سازی این پدیده زمانی رخ می‌دهد که ورودی از قبل مرتب باشد.
- بدترین حالت: زمانی که الگوریتم به ازای هراندازه‌ی ورودی، به حداکثر تعداد عملیات نیاز داشته باشد. در بسیاری از الگوریتم‌های مرتب‌سازی این پدیده زمانی رخ می‌دهد که ورودی از قبل به‌صورت معکوس مرتب‌شده باشد.
- حالت میانگین: به میانگین تعداد عملیات موردنیاز به ازای ورودی‌های ممکن با اندازه‌های مشخص اشاره دارد. برای مرتب‌سازی، معمولاً یک ورودی با هر ترتیب تصادفی مدنظر قرار می‌گیرد.

در کل، بدترین حالت مهم‌تر از بقیه است؛ زیرا به کمک آن‌یک معیار تضمین‌شده خواهید داشت که می‌توانید همیشه بر روی آن حساب کنید. زمانی که چیزی در مورد سناریو گفته نمی‌شود، بدترین حالت در نظر گرفته می‌شود. در ادامه خواهیم دید که چگونه بدترین حالت را تحلیل کنیم.

---

[1] Running Cost

شکل ۲-۱: تخمین زدن زمان (دریافت شده از http://xkcd.com)

۲-۱- شمارش زمان

پیچیدگی زمانی یک الگوریتم را با شمارش تعداد عملیات اصلی موردنیاز آن برای یک ورودی فرضی با اندازه‌ی n به دست می‌آوریم. این کار را با **مرتب‌سازی انتخابی**[1]، یک الگوریتم که از حلقه‌های تودرتو استفاده می‌کند، نشان می‌دهیم. یک حلقه‌ی for بیرونی موقعیت جاری در حال مرتب‌سازی را بهنگام‌سازی می‌کند، یک حلقه‌ی for درونی عنصری را که قرار است در موقعیت جاری قرار بگیرد انتخاب می‌کند[2]:

```
function selection_sort(list)
    for current ← 1 … list.length - 1
        smallest ← current
        for i ← current + 1 … list.length
            if list[i] < list[smallest]
                smallest ← i
        list.swap_items(current, smallest)
```

[1] Selection Sort

[2] برای درک کردن یک الگوریتم جدید، آن را بر روی کاغذ با یک ورودی کوچک اجرا کنید.

اجازه بدهید ببینیم برای یک لیست حاوی n عنصر در بدترین حالت چه اتفاقی می‌افتد. حلقه‌ی بیرونی $n-1$ بار تکرار می‌شود و در هر تکرار دو عمل انجام می‌دهد (یک انتساب و یک جابجایی) که در کل $2n-2$ عمل انجام می‌شود. حلقه‌ی درونی ابتدا $n-1$ بار، سپس $n-2$ بار، سپس $n-3$ بار تکرار می‌شود و الی‌آخر. می‌دانیم که این نوع دنباله‌ها را چطور جمع بزنیم[1]:

$$\text{تعداد تکرار حلقه درونی} = \overbrace{\underbrace{n-1}_{\text{اولین تکرار حلقه درونی}} + \underbrace{n-2}_{\text{دومین تکرار حلقه درونی}} + \cdots + 2 + 1}^{n-1 \text{ تکرار حلقه بیرونی}}$$

$$= \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2-n}{2}$$

در بدترین حالت، شرط `if` همیشه برقراراست. این بدان معنی است که حلقه‌ی درونی یک مقایسه و یک انتساب را $\frac{n^2-n}{2}$ بار انجام می‌دهد، ازاین‌رو n^2-n عملیات انجام می‌دهد. در کل، هزینه‌ی الگوریتم برابر است با $2n-2$ عملیات برای حلقه‌ی بیرونی، به‌اضافه‌ی n^2-n عملیات برای حلقه‌ی درونی. بنابراین پیچیدگی زمانی آن را به‌صورت زیر به دست می‌آوریم:

$$\mathbb{T}(n) = n^2 + n - 2$$

حالا چه می‌شود؟ اگر اندازه‌ی لیست ما $n=8$ باشد و آن را دو برابر کنیم، زمان مرتب‌سازی چند برابر می‌شود:

$$\frac{\mathbb{T}(16)}{\mathbb{T}(8)} = \frac{16^2+16-2}{8^2+8-2} \approx 3.86$$

اگر دوباره آن را دو برابر کنیم، زمان ۳/۹۰ برابر خواهد شد. دو برابر کردن چندباره‌ی آن باعث می‌شود زمان به ترتیب ۳/۹۴، ۳/۹۷ و ۳/۹۸ برابر شود. توجه کنید که این اعداد چگونه در حال نزدیک شدن به ۴ هستند. این بدان معنی است که مرتب‌سازی دو میلیون عنصر چهار برابر بیشتر از مرتب‌سازی یک‌میلیون عنصر طول خواهد کشید.

درک کردن میزان رشد

فرض کنید اندازه‌ی ورودی یک الگوریتم خیلی بزرگ باشد و ما همچنان آن را زیادتر کنیم. برای پیش‌بینی نحوه‌ی رشد زمان اجرا نیاز نیست تمام عبارات $\mathbb{T}(n)$ را بدانیم. می‌توانیم $\mathbb{T}(n)$ را با عبارتی از آن‌که بیشترین رشد را دارد و به آن **عبارت غالب**[2] گفته می‌شود، تخمین بزنیم.

---

[1] بر اساس بخش ۱-۳ داریم $\sum_{i=1}^{n} i = n(n+1)/2$.

[2] Dominant Term

کارت‌های فهرست: دیروز شما به یک جعبه از کارت‌های فهرست برخورد کرده و آن را ریخته‌اید. دو ساعت طول کشید تا با استفاده از مرتب‌سازی انتخابی آن‌ها را مجدداً مرتب کنید. امروز، شما ده جعبه را ریختید. به چقدر زمان نیاز دارید تا کارت‌ها را مرتب کنید؟

دیدید که مرتب‌سازی انتخابی به‌صورت $\mathbb{T}(n) = n^2 + n - 2$ است. بیشترین رشد را عبارت n^2 دارد، ازاین‌رو می‌توانیم بنویسیم $\mathbb{T}(n) \approx n^2$. فرض کنید در هر جعبه n کارت وجود دارد، داریم:

$$\frac{\mathbb{T}(10n)}{\mathbb{T}(n)} \approx \frac{(10n)^2}{n^2} = 100$$

بنابراین تقریباً ۱۰۰ برابر بیشتر نسبت به دیروز زمان نیاز دارید، یعنی ۲۰۰ ساعت! اگر از یک روش مرتب‌سازی دیگر استفاده می‌کردیم چه می‌شد؟ به‌عنوان‌مثال، روشی به نام «مرتب‌سازی حبابی[1]» وجود دارد که تقریباً به‌صورت $\mathbb{T}(n) = 0.5n^2 + 0.5n$ است. عبارت دارای بیشترین رشد به‌صورت $\mathbb{T}(n) \approx 0.5n^2$ است، ازاین‌رو:

$$\frac{\mathbb{T}(10n)}{\mathbb{T}(n)} \approx \frac{0.5 \times (10n)^2}{0.5 \times n^2} = 100$$

ضریب ۰/۵ خودش را حذف کرد! ایده‌ی این‌که $n^2 + n - 2$ و $0.5n^2 + 0.5n$ هر دو شبیه به n^2 رشد می‌کنند چندان به‌راحتی قابل‌درک نیست. چطور عبارتی که در یک تابع بیشترین رشد را دارد سایر اعداد را کنار زده و در فرایند رشد غالب می‌شود؟ اجازه بدهید به‌صورت بصری این موضوع را درک کنیم:

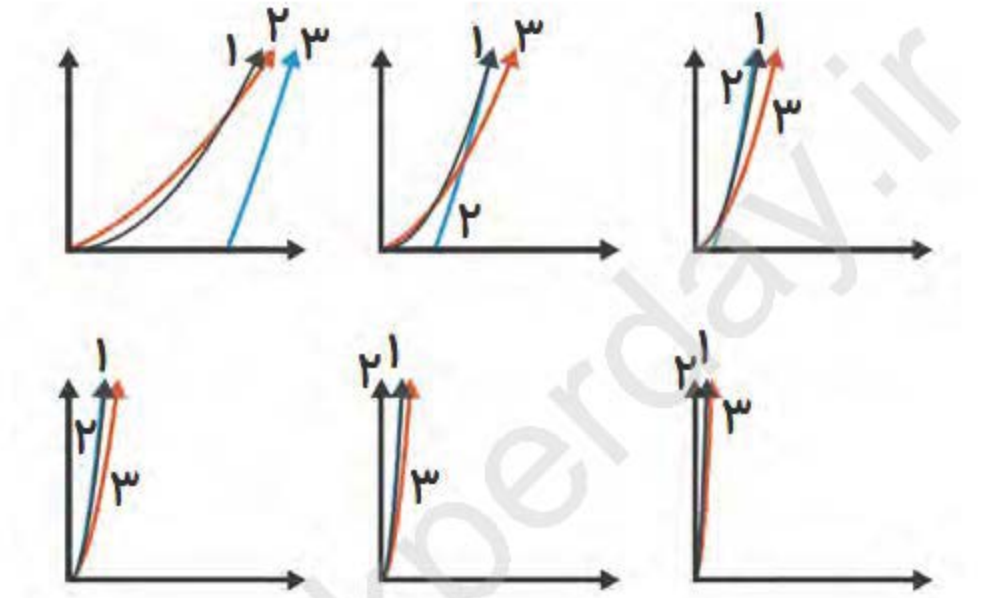
در شکل ۲-۲، دو پیچیدگی زمانی که تاکنون دیده‌ایم با n^2 در درجه‌های بزرگنمایی مختلف مقایسه شده‌اند. با رسم نمودار برای مقادیر بزرگ و بزرگ‌تر n منحنی آن‌ها به یکدیگر نزدیک و نزدیک‌تر می‌شود. در حقیقت، شما می‌توانید در $\mathbb{T}(n) = \blacksquare n^2 + \blacksquare n + \blacksquare$ هر عددی را به‌جای ■ بگذارید، نتیجه‌ی حاصل شبیه به n^2 رشد می‌کند.

به یاد داشته باشید که این اثر نزدیک شدن منحنی‌ها زمانی کار می‌کند که عبارت دارای بیشترین رشد یکسان باشد. نمودار یک تابع با رشد خطی (n) هرگز به نمودار یک تابع درجه ۲ (n^2) نزدیک نخواهد شد. تابع درجه ۲ نیز به تابع درجه ۳ (n^3) نزدیک نخواهد شد.

---

[1] Bubble Sort

این دلیل بدتر عمل کردن الگوریتم‌های با هزینه‌ی رشد درجه ۲ نسبت به الگوریتم‌های دارای هزینه‌ی خطی است. بااین‌حال، این الگوریتم‌ها خیلی بهتر از الگوریتم‌های دارای هزینه‌ی درجه ۳ عمل می‌کنند. اگر این موضوع را فهمیده باشید، بخش بعد آسان خواهد بود؛ نماد حالبی را می‌آموزیم که کدنویسان برای بیان این مطلب از آن استفاده می‌کنند.



شکل ۲-۲: کاهش بزرگ‌نمایی نمودارهای n^2 (شماره ۱)، $n^2 + n - 2$ (شماره ۲) و $0.5n^2 + 0.5n$ (شماره ۳) با افزایش مقدار n

۲-۲- نماد اُی‌بزرگ

یک نماد ویژه برای اشاره به کلاس‌های رشد وجود دارد: **نماد اُی‌بزرگ**[۱]. یک تابع با عبارت دارای بیشترین رشد به‌صورت 2^n یا ضعیف‌تر، از نوع $O(2^n)$ است؛ تابعی با رشد درجه ۲ یا ضعیف‌تر، از نوع $O(n^2)$ است؛ رشد خطی یا کمتر، از نوع $O(n)$ است، و الی‌آخر. این نماد برای بیان عبارت غالب هزینه‌ی الگوریتم در بدترین حالت استفاده می‌شود و روش استاندارد بیان پیچیدگی است[۲].

---

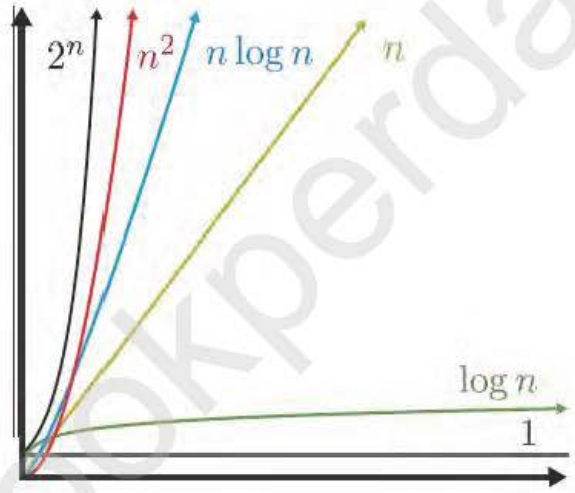
[۱] Big-O Notation

[۲] به‌صورت اُ تلفظ می‌شود، مانند «آن الگوریتم مرتب‌سازی از اُ درجه ۲ است».

هم مرتب‌سازی حبابی و هم مرتب‌سازی انتخابی از نوع $O(n^2)$ هستند، ولی به‌زودی درمی‌یابیم که الگوریتم‌هایی از نوع $O(n\log n)$ نیز وجود دارند که همان کار را انجام می‌دهند. با الگوریتم‌های $O(n^2)$، ۱۰ برابر کردن اندازه‌ی ورودی موجب ۱۰۰ برابر شدن هزینه‌ی اجرا می‌شود. با استفاده از یک الگوریتم $O(n\log n)$ با ۱۰ برابر شدن اندازه‌ی ورودی، هزینه‌ی اجرا تقریباً فقط ۳۴ برابر می‌شود:

$$10\log 10 \approx 34$$

وقتی n برابر با یک‌میلیون باشد، n^2 برابر با یک تریلیون است، درحالی‌که $n\log n$ فقط چند میلیون خواهد بود. سال‌ها اجرای یک الگوریتم درجه ۲ بر روی یک ورودی بزرگ معادل با چند دقیقه اجرا در صورت استفاده از الگوریتم $O(n\log n)$ است. به همین دلیل شما در زمان طراحی سیستم‌هایی برای کار با ورودی‌های خیلی بزرگ به تحلیل پیچیدگی نیاز دارید.



شکل ۲-۳: مرتبه‌های متفاوت رشد که اغلب درون O دیده می‌شوند

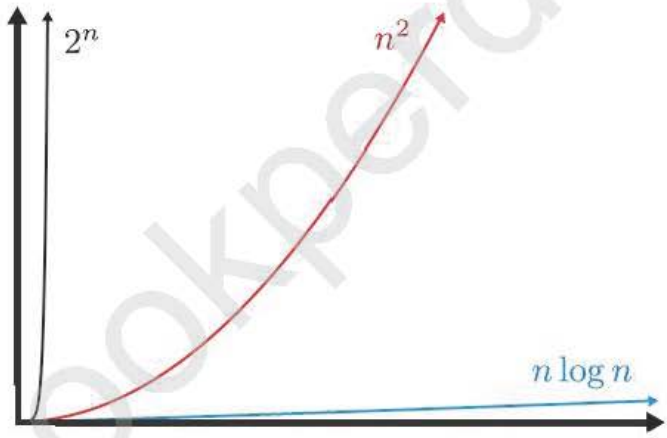
در زمان طراحی یک سیستم محاسباتی، پیش‌بینی پرتکرارترین عملیات اهمیت دارد. در این شرایط باید هزینه‌های اُی‌بزرگ الگوریتم‌های مختلفی که این عملیات را انجام می‌دهند باهم مقایسه کنید[1]. هم‌چنین اغلب الگوریتم‌ها فقط با ساختارهای داده‌ای مشخصی کار می‌کنند. اگر الگوریتمتان را از قبل انتخاب کنید، می‌توانید داده‌های ورودی خود را مطابق با آن ساختاردهی کنید.

[1] برای آشنایی با پیچیدگی اُی‌بزرگ اغلب الگوریتم‌هایی که کارهای معمول را انجام می‌دهند، http://code.enery.bigo را ببینید.

برخی از الگوریتم‌ها فارغ از اندازه‌ی ورودی، در یک دوره‌ی زمانی ثابت اجرا می‌شوند، این الگوریتم‌ها $\mathcal{O}(1)$ هستند. به‌عنوان‌مثال، بررسی فرد یا زوج بودن یک عدد: فقط کافی است ببینیم رقم آخر آن فرد است یا زوج، مسئله به همین راحتی حل می‌شود. اندازه‌ی عدد هیچ تأثیری ندارد. در فصل‌های بعد الگوریتم‌های $\mathcal{O}(1)$ بیشتری خواهیم دید. این الگوریتم‌ها شگفت‌انگیز هستند، ولی ابتدا اجازه بدهید ببینیم کدام الگوریتم‌ها شگفت‌انگیز نیستند!

۲-۳- نمایی‌ها

الگوریتم‌های $\mathcal{O}(2^n)$ را از نوع **زمان نمایی**[1] می‌دانیم. در نمودار رشد مرتبه‌ها (شکل ۲-۳)، به نظر نمی‌رسد که درجه‌ی ۲ یعنی $\mathcal{O}(n^2)$ و نمایی یعنی $\mathcal{O}(2^n)$ چندان متفاوت باشند. با بزرگنمایی نمودار، واضح است رشد نمایی به‌صورت مشهودی بر درجه‌ی ۲ غالب می‌شود:



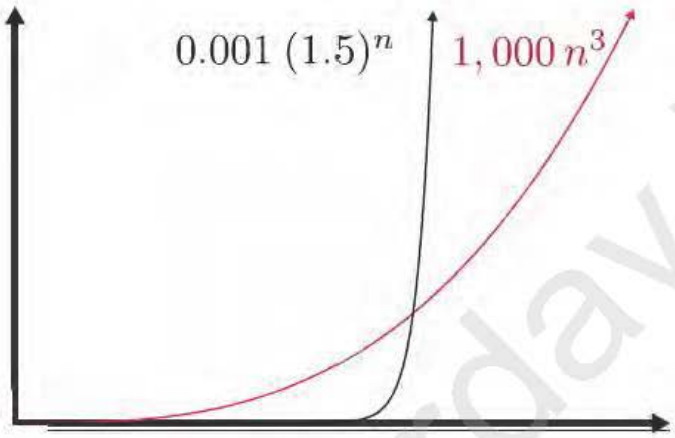
شکل ۲-٤: کاهش بزرگنمایی مرتبه‌های مختلف رشد، منحنی‌های خطی و لگاریتمی آن‌قدر رشد کمی دارند که قابل‌مشاهده نیستند

رشد زمان نمایی آن‌قدر زیاد است که ما این الگوریتم‌ها را «غیرقابل اجرا» می‌دانیم. این الگوریتم‌ها فقط بر روی انواع ورودی معدودی اجرا می‌شوند و در صورت کوچک نبودن ورودی نیازمند حجم زیادی توان محاسباتی هستند. بهینه‌سازی هر جنبه از کد یا استفاده از ابَرکامپیوترها نیز کمکی نمی‌کند. قدرتِ نمایی بر میزان رشد غالب است و موجب می‌شود این الگوریتم‌ها غیرقابل استفاده شوند.

---

[1] Exponential Time

برای نمایش قدرت زیاد رشد نمایی، اجازه بدهید بزرگنمایی نمودار را کمتر کرده و اعداد را عوض کنیم (شکل ۲-۵). پایه‌ی نمایی کم شده (از ۲ به ۱/۵) و رشد آن به هزار تقسیم‌شده است. رشد چندجمله‌ای نیز افزایش‌یافته (از ۲ به ۳) و در هزار نیز ضرب شده است.



شکل ۲-۵: هیچ‌کدام از نمایی‌ها از چندجمله‌ای شکست نمی‌خورد. در این سطح بزرگنمایی حتی منحنی رشد $n\log n$ نیز آن‌قدر کوچک است که دیده نمی‌شود

برخی الگوریتم‌ها حتی از الگوریتم‌های زمان نمایی نیز بدتر هستند. این الگوریتم‌ها از نوع **زمان فاکتوریل** هستند و پیچیدگی آن‌ها $O(n!)$ است. الگوریتم‌های زمانی نمایی و فاکتوریل بسیار وحشتناک هستند ولی ما به آن‌ها برای سخت‌ترین مسائل محاسباتی نیاز داریم: مسائل معروف **NP-کامل**[1]. نمونه‌های مهمی از مسائل NP-کامل را در فصل بعد خواهیم دید. هم‌اکنون به یاد داشته باشید: اولین نفری که یک الگوریتم غیر نمایی برای یک مسئله‌ی NP-کامل پیدا کند یک‌میلیون دلار از موسسه‌ی ریاضی کِلی[2] دریافت می‌کند[3].

---

[1] NP-Complete

[2] Clay Mathematics Institute: یک موسسه غیرانتفاعی که هدف آن گسترش و انتشار ریاضیات است. این موسسه در دبور کلورادو قرار دارد. علیرغم شناخته شدن این موسسه برای این جایزه، فعالیت‌های بسیار دیگری مانند دوره‌های پسادکترا، کنفرانس‌ها، کارگاه‌ها، مدارس تابستانی و غیره نیز توسط این موسسه انجام می‌شود. م.

[3] ثابت‌شده است یک الگوریتم غیر نمایی برای هر یک از مسائل NP-کامل به تمام مسائل NP-کامل قابل‌تعمیم است. به دلیل آن‌که نمی‌دانیم آیا چنین الگوریتمی وجود دارد یا خیر، حتی اگر ثابت کنید یک مسئله‌ی NP-کامل قابل‌حل با الگوریتم‌های غیر نمایی نیست نیز یک‌میلیون دلار دریافت خواهید کرد!

شناسایی کلاس مسئله‌ای که با آن‌طرف هستید، اهمیت دارد. اگر کلاس آن NP-کامل باشد، سعی در پیدا کردن یک راه‌حل بهینه، جنگیدن با غیرممکن است. مگر اینکه آن یک‌میلیون دلار را هدف گرفته باشید.

۲-٤- شمارش حافظه

حتی اگر بتوانیم عملیات را با سرعت نامحدود انجام دهیم، همچنان یک محدودیت برای توان محاسباتی ما وجود دارد. در حین اجرا، الگوریتم‌ها نیاز به حافظه‌ی کاری برای ردیابی محاسبات در حال انجام خود دارند. این امر موجب مصرف **حافظه‌ی کامپیوتر** می‌شود که نامحدود نیست.

سنجش حافظه‌ی کاری موردنیاز یک الگوریتم را **پیچیدگی فضایی**[1] می‌نامند. تحلیل پیچیدگی فضایی شبیه به تحلیل پیچیدگی زمانی است. تفاوت در این است که در این تحلیل، حافظه‌ی کامپیوتر را می‌شماریم، نه عملیات محاسباتی را. مشاهده می‌کنیم که پیچیدگی فضایی، درست همانند پیچیدگی زمانی، چگونه با رشد اندازه‌ی ورودی الگوریتم افزایش می‌یابد.

به‌عنوان‌مثال، مرتب‌سازی انتخابی (بخش ۲-۱) تنها نیاز به حافظه‌ی کاری برای چند متغیر با تعداد ثابت دارد. تعداد متغیرها به‌اندازه‌ی ورودی وابسته نیست. ازاین‌رو، می‌گوییم پیچیدگی فضایی مرتب‌سازی انتخابی از نوع $O(1)$ است: هیچ اهمیتی ندارد اندازه‌ی ورودی چقدر است، این الگوریتم به فضای یکسانی از حافظه‌ی کامپیوتر به‌عنوان حافظه‌ی کاری نیاز دارد.

بااین‌حال، الگوریتم‌های بسیار دیگری نیازمند حافظه‌ای هستند که با رشد اندازه‌ی ورودی، بیشتر می‌شود. برخی اوقات، برآورده کردن نیاز یک الگوریتم به حافظه غیرممکن است. شما نمی‌توانید هیچ الگوریتم مرتب‌سازی بیابید که پیچیدگی زمانی آن $O(n\log n)$ و پیچیدگی فضایی آن $O(1)$ باشد. گاهی اوقات محدودیت حافظه‌ی کامپیوتر موجب نیاز به برقراری تعادل می‌شود. با حافظه‌ی کم، احتمالاً نیاز به یک الگوریتم کند با پیچیدگی زمانی $O(n^2)$ دارید زیرا دارای پیچیدگی فضایی $O(1)$ است. در فصل‌های بعد خواهیم دید چگونه می‌توان با مدیریت هوشمندانه‌ی داده‌ها پیچیدگی فضایی را بهبود داد.

نتیجه‌گیری

در این فصل، یاد گرفتیم که الگوریتم‌ها می‌توانند انواع مختلفی در رابطه با زمان مصرفی محاسبات و استفاده از حافظه‌ی کامپیوتر داشته باشند. هم‌چنین دیدیم که چگونه می‌توانیم مصرف زمان و حافظه‌ی

---

[1] Space Complexity

الگوریتم را با تحلیل پیچیدگی زمانی و فضایی ارزیابی کنیم. روش محاسبه‌ی پیچیدگی زمانی را با به دست آوردن دقیق تابع $\mathbb{T}(n)$، یعنی تعداد عملیات انجام‌شده توسط الگوریتم، آموختیم.

دیدیم که چگونه می‌توانیم پیچیدگی زمانی را با نماد اُی بزرگ (O) بیان کنیم. در طی این کتاب، تحلیل ساده‌ی پیچیدگی زمانی الگوریتم‌ها را با استفاده از این نماد انجام می‌دهیم. بسیاری اوقات، محاسبه‌ی $\mathbb{T}(n)$ برای تعیین پیچیدگی اُی‌بزرگ یک الگوریتم الزامی نیست. راه‌های ساده‌تری را برای محاسبه‌ی پیچیدگی در فصل بعد خواهیم دید.

دیدیم که هزینه‌ی اجرای الگوریتم‌های نمایی آن‌قدر افزایش پیدا می‌کند که این الگوریتم‌ها بر روی ورودی‌های بزرگ قابل‌اجرا نیستند؛ و یاد گرفتیم که چگونه به این سؤالات پاسخ دهیم:

- با داشتن الگوریتم‌های متفاوت، آیا تفاوت چشمگیری درزمینه‌ی عملیات موردنیاز برای اجرای آن‌ها وجود دارد؟
- ضرب کردن اندازه‌ی ورودی در یک عدد ثابت چه تأثیری بر زمان اجرای یک الگوریتم می‌گذارد؟
- آیا یک الگوریتم در صورت افزایش اندازه‌ی ورودی تعدادی منطقی از عملیات را انجام می‌دهد؟
- اگر یک الگوریتم خیلی در اجرا بر روی یک اندازه‌ی ورودی معلوم کند باشد، آیا بهینه‌سازی آن یا استفاده از اَبَرکامپیوترها کمکی می‌کند؟

در فصل بعد، بر موضوع پیدا کردن استراتژی‌های اساسی برای طراحی الگوریتم‌ها بر اساس پیچیدگی زمانی آن‌ها تمرکز خواهیم کرد.

مراجع

- The Art of Computer Programming, Vol. 1, by Knuth
  - Get it at https://code.energy/knuth
- The Computational Complexity Zoo, by hackerdashery
  - Watch it at https://code.energy/pnp

فصل ۳

استراتژی

اگر یک حرکت خوب پیدا کردید، به دنبال یک حرکت بهتر بگردید.

– امانوئل لاسکر[1]

تاریخ ژنرال‌هایی را به یاد دارد که از استراتژی خوب برای رسیدن به نتایج بزرگ استفاده کرده‌اند. برای موفق بودن در حل مسئله باید یک استراتژیست خوب بود. این فصل استراتژی‌های اصلی طراحی الگوریتم را پوشش می‌دهد. شما یاد خواهید گرفت:

- کارهایی که زیاد انجام می‌شوند را با **تکرار** مدیریت کنید.
- تکرار را ظریفانه با استفاده از **بازگشت** انجام دهید.
- اگر تنبل ولی قدرتمند هستید از **جستجوی فراگیر** استفاده کنید.
- گزینه‌های بد را آزمایش کرده و سپس **عقب‌گرد** کنید.
- با **ابتکار** یک راه منطقی برای خروج پیداکرده و در زمان صرفه‌جویی کنید.
- سرسخت‌ترین رقبایتان را با **تقسیم و حل** کنار بزنید.
- مباحث قدیمی را به‌صورت **پویا** شناسایی کرده و انرژی را هدر ندهید.
- مسئله‌ی خودتان را **محدود** کنید تا راه‌حل فرار نکند.

ابزارهای زیادی را اینجا خواهیم دید، ولی نگران نباشید. با مسائل ساده شروع کرده و به‌تدریج و با یافتن روش‌های جدید، راه‌حل‌های بهتری می‌سازیم. به‌زودی شما با راه‌حل‌های درست و روشن بر مسائل محاسباتی خود غلبه خواهید کرد.

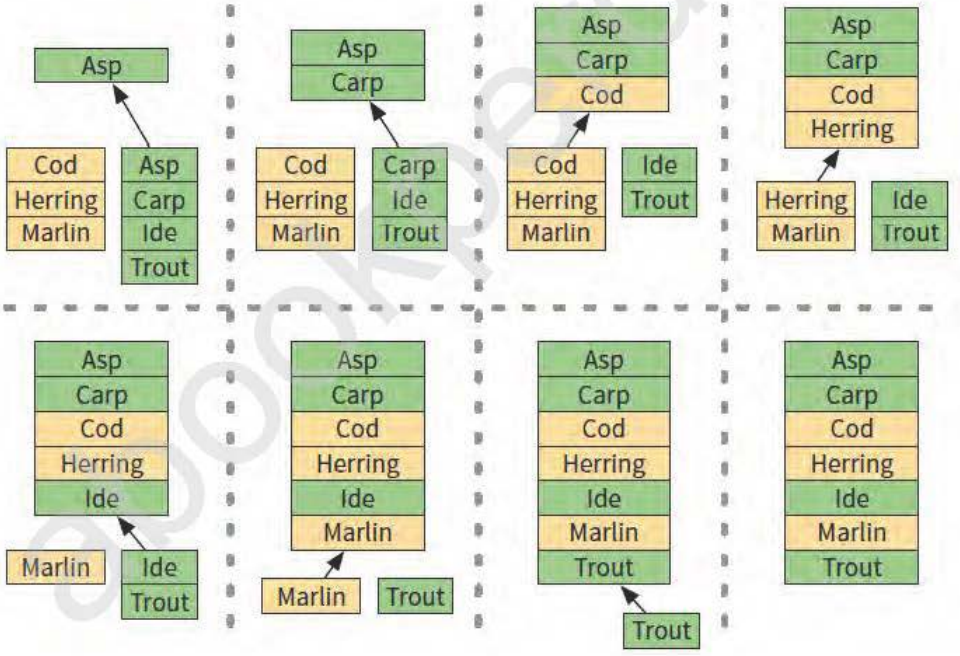
[1] Emanuel Lasker (1868-1941): فیلسوف، ریاضیدان و یکی از قهرمانان شطرنج، اهل آلمان و پدر و بنیان‌گذار روان‌شناسی شطرنج. م.

۳-۱- تکرار

استراتژی تکراری شامل استفاده از حلقه‌ها (مانند for و while) برای تکرار کردن یک فرایند تا زمان برآورده شدن یک شرط است. هر گام در یک حلقه یک **تکرار**[1] نامیده می‌شود. این روش برای اجرا بر روی یک ورودی و اعمال عملیات مشابه بر روی هر بخش آن بسیار عالی است. به‌عنوان‌مثال:

> تجدیددیدار ماهی‌ها: شما لیستی از ماهی‌های آب‌شور و لیستی از ماهی‌های آب‌های آزاد دارید که هر دو به ترتیب الفبا مرتب‌شده‌اند. چگونه می‌توانید یک لیست شامل اسم همه‌ی ماهی‌ها به ترتیب حروف الفبا بسازید؟

می‌توانیم به‌صورت تکراری، عنصر بالایی دو لیست را باهم مقایسه کنیم:



شکل ۳-۱: ادغام دو لیست مرتب‌شده در یک لیست سوم مرتب‌شده

این فرایند را می‌توان با یک حلقه‌ی while تنها نوشت:

---

[1] Iteration

```
function merge(sea, fresh)
    result ← List.new

    while not (sea.empty and fresh.empty)
        if sea.top_item > fresh.top_item
            fish ← sea.remove_top_item
        else
            fish ← fresh.remove_top_item
        result.append(fish)

    return result
```

این کد بر روی تمام اسم‌های ورودی حلقه‌زده و تعداد ثابتی عملیات را بر روی هر اسم انجام می‌دهد[۱]. ازاین‌رو، الگوریتم ادغام (merge) از نوع $O(n)$ است.

حلقه‌های تودرتو[۲] و مجموعه‌ی توانی[۳]

در فصل قبل دیدم مرتب‌سازی انتخاب (selection_sort) چگونه از یک حلقه‌ی تودرتو در درون یک حلقه‌ی دیگر استفاده می‌کند. حال یاد می‌گیریم که از یک حلقه‌ی تودرتو برای محاسبه‌ی مجموعه‌ی توانی استفاده کنیم. با داشتن یک مجموعه از اشیاء مانند S مجموعه‌ی توانی S مجموعه‌ای شامل تمام زیرمجموعه‌های S است[۴].

> کاوش رایحه‌ها: عطرهای گیاهی از طریق ترکیب رایحه‌ی گل‌ها تولید می‌شوند. با داشتن مجموعه‌ای از گل‌ها مانند F چگونه تمام عطرهای قابل ساخت با آن‌ها را فهرست می‌کنید؟

هر عطر از یک زیرمجموعه از F تولید می‌شود، بنابراین مجموعه‌ی توانی آن شامل تمام عطرهای ممکن است. می‌توانیم این مجموعه‌ی توانی را به‌صورت تکراری محاسبه کنیم. برای صفر گل فقط یک

---

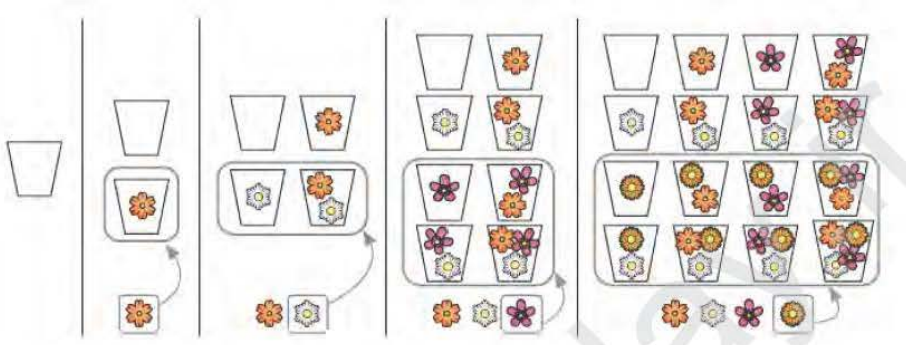
[۱] اندازه‌ی ورودی برابر با تعداد عناصر در دو لیست ترکیب‌شده است. حلقه‌ی while سه عملیات را برای هر یک از این عناصر انجام می‌دهد، ازاین‌رو، $\mathbb{T}(n) = 3n$.

[۲] Nested Loops

[۳] Power Set

[۴] در صورت نیاز به توضیح بهتر در مورد مجموعه‌ها، پیوست III را ببینید.

نوع عطر ممکن است: عطری که حاوی هیچ *رایحه‌ای* نیست. برای اضافه کردن یک گل جدید، عطرهایی که داریم را دو برابر می‌کنیم و عطر جدید را به قسمت تکرار شده اضافه می‌کنیم. درک کردن آنچه رخ می‌دهد به‌صورت تصویری راحت‌تر است:



شکل ۳-۲: فهرست کردن تکراری تمام عطرها با استفاده از چهار گل

این فرایند را می‌توان با استفاده از حلقه‌ها توضیح داد. یک حلقه‌ی بیرونی که گل بعدی را اضافه می‌کند؛ و یک حلقه‌ی درونی که تعداد عطرها را دو برابر کرده و گل جدید را به بخش تکرار شده اضافه می‌کند.

```
function power_set(flowers)
    fragrances ← Set.new
    fragrances.add(Set.new)
    for each flower in flowers
        new_fragrances ← copy(fragrances)
        for each fragrance in new_fragrances
            fragrance.add(flower)
        fragrances ← fragrances + new_fragrances
    return fragrances
```

یک گل اضافه موجب می‌شود اندازه‌ی fragrances دو برابر شود و بیانگر رشد نمایی ($2^{k+1} = 2 \times 2^k$) است. الگوریتم‌هایی که نیاز به دو برابر کردن عملیات در صورت افزایش اندازه‌ی ورودی به‌اندازه‌ی یک واحد دارند، نمایی با پیچیدگی زمانی $O(2^n)$ هستند.

تولید مجموعه‌های توانی معادل با تولید جدول‌های درستی (بخش ۱-۲) است. اگر هر گل را به یک متغیر بولی نگاشت کنیم، هر عطر با این متغیرها با مقادیر True/False قابل‌نمایش است. در جدول درستی این متغیرها هر سطر بیانگر فرمول یک عطر است.

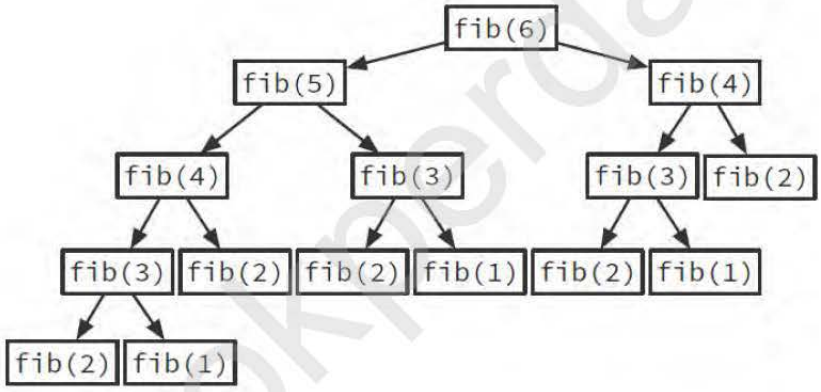
۳-۲- بازگشت

زمانی که یک تابع به‌گونه‌ای عمل می‌کند که خودش را شبیه‌سازی کند، می‌گوییم یک **بازگشت**[۱] رخ‌داده است. زمانی که یک مسئله با عبارتی از خودش تعریف می‌شود، به‌صورت طبیعی یک الگوریتم بازگشتی برای حل آن به ذهن می‌آید. به‌عنوان‌مثال، دنباله‌ی معروف فیبوناچی[۲] را در نظر بگیرید. این دنباله با دو عدد ۱ شروع می‌شود و هر عدد بعدی از طریق مجموع دو عدد قبل از آن به دست می‌آید:

$$1,1,2,3,5,8,13,21,\ldots$$

اگر بخواهید تابعی بنویسید که عدد nام فیبوناچی را برگرداند، کد را چگونه می‌نویسید؟

```
function fib(n)
    if n ≤ 2
        return 1
    return fib(n - 1) + fib(n - 2)
```



شکل ۳-۳: محاسبه‌ی بازگشتی جمله‌ی ششم فیبوناچی

استفاده از بازگشت نیازمند خلاقیت برای درک نحوه‌ی بیان یک مسئله در قالب عباراتی از خودش است. بررسی این‌که یک کلمه متقارن[۳] است معادل است بااینکه بررسی کنیم که آیا آن کلمه در صورت معکوس شدن تغییر می‌کند یا خیر؛ اما می‌توان گفت یک کلمه متقارن است اگر کاراکترهای اول و آخر آن یکسان بوده و زیر کلمه‌ی بین این کاراکترها نیز متقارن باشد:

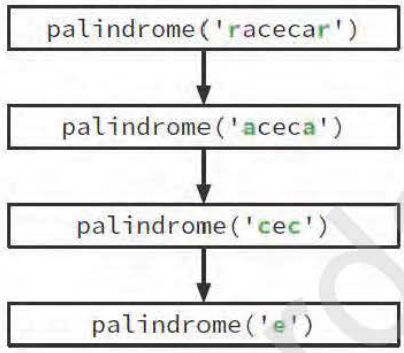
---

[۱] Recursion

[۲] Fibonacci (1170-1250): لئوناردو فیبوناچی نخستین ریاضیدان بزرگ اروپا در قرن سیزدهم میلادی بود. وی متولد شهر پیزا در ایتالیا بود و کارهای ریاضیدانان مسلمان مانند خوارزمی و ابو کامل را دنبال می‌کرد. م.

[۳] کلمات متقارن از هر سمت خوانده شوند یکسان هستند، مانند ada یا racecar.

```
function palindrome(word)
    if word.length ≤ 1
        return True
    if word.first_char ≠ word.last_char
        return False
    w ← word.remove_first_and_last_chars
    return palindrome(w)
```



شکل ۳-٤: بررسی بازگشتی متقارن بودن کلمه‌ی racecar

الگوریتم‌های بازگشتی دارای **حالت‌های پایه**[1] هستند که در آن‌ها ورودی آن‌قدر کوچک است که نمی‌توان آن را بیشتر کاهش داد. حالت‌های پایه برای fib اعداد اول و دوم و برای palindrome کلمات دارای یک یا صفر کاراکتر هستند.

تکرار در مقابل بازگشت

الگوریتم‌های بازگشتی عموماً ساده‌تر و کوتاه‌تر از الگوریتم‌های تکراری هستند. این الگوریتم بازگشتی را با الگوریتم power_set بخش قبل که از بازگشت استفاده نمی‌کرد، مقایسه کنید.

```
function recursive_power_set(items)
    ps ← copy(items)
    for each e in items
        ps ← ps.remove(e)
        ps ← ps + recursive_power_set(ps)
        ps ← ps.add(e)
    return ps
```

[1] Base Cases

این سادگی هزینه هم دارد. الگوریتم‌های بازگشتی مشابه‌های بسیاری از خود را در زمان اجرا می‌سازند و موجب سربارهٔ محاسباتی[۱] می‌شوند. کامپیوتر باید فراخوانی‌های بازگشتی کامل نشده و محاسبات جزئی آن‌ها را ردیابی کند و این کار نیاز به حافظهٔ اضافی دارد. یک چرخهٔ اضافی CPU برای جابجایی بین یک فراخوانی بازگشتی و مورد بعد آن و برگشت مجدد به آن نیاز است.

این مشکل بالقوه را می‌توان در قالب **درخت‌های بازگشت**[۲] دید: نموداری که نشان می‌دهد الگوریتم چگونه با درگیر شدن در عمق محاسبات، فراخوانی‌های بیشتری را انجام می‌دهد. درخت‌های بازگشت را برای محاسبهٔ اعداد فیبوناچی (شکل ۳-۳) و بررسی متقارن بودن کلمه (شکل ۳-۴) دیدیم.

در صورت نیاز به کارایی بیشینه، می‌توانیم از طریق بازنویسی الگوریتم بازگشتی در یک قالب کاملاً تکراری از این سرباره اجتناب کنیم. انجام این کار همیشه ممکن است. این کار ایجاد نوعی تعادل است: کد تکراری به‌صورت کلی زودتر اجرا می‌شود ولی پیچیده‌تر و فهم آن سخت‌تر است.

۳-۳- جستجوی فراگیر

استراتژی جستجوی فراگیر[۳] مسائل را با بررسی تمام راه‌حل‌های کاندیدای ممکن حل می‌کند. این استراتژی که به نام جستجوی همه‌جانبه نیز شناخته می‌شود اغلب خام و غیرماهرانه است: اگر میلیاردها کاندیدا وجود داشته باشد، این استراتژی به قدرت محض کامپیوتر برای بررسی تک‌تک آن‌ها تکیه دارد.

اجازه بدهید ببینیم چگونه می‌توانیم از این استراتژی برای حل مسئله‌ی زیر استفاده کنیم:

> بهترین تجارت: شما قیمت روزانه‌ی طلا را برای یک دوره‌ی زمانی دارید. می‌خواهید دو روز را در این دوره بیابید که اگر در آن‌ها ابتدا طلا بخرید و سپس بفروشید، حداکثر سود را به دست می‌آورید.

خریدن به کمترین قیمت و فروختن به بیشترین قیمت همیشه ممکن نیست: ممکن است کمترین قیمت *بعد* از بیشترین قیمت اتفاق بیفتد و سفر در زمان برای ما ممکن نیست. یک رویکرد جستجوی فراگیر با ارزیابی *تمام زوج‌های ممکن* جواب را پیدا می‌کند. برای هر زوج، این رویکرد سود حاصل از تجارت در آن روزها را محاسبه کرده و آن را با بهترین تجارت تا آن لحظه مقایسه می‌کند. می‌دانیم تعداد زوج روزها در یک بازه به‌صورت درجه ۲ با افزایش طول دوره رشد می‌کند[۴]. بدون نوشتن کد همین‌الان می‌دانیم که این الگوریتم باید $O(n^2)$ باشد.

---

[۱] Computational Overhead

[۲] Recursion Trees

[۳] Brute Force

[۴] از بخش ۱-۳ می‌دانیم به تعداد $n(n+1)/2$ زوج روز در یک دوره‌ی شامل n روز وجود دارد.

توضیح ساده:
حمله جستجوی فراگیر

ملاقات یک همکلاسی قدیمی

شکل ۳-۵: گرفته‌شده از http://geek-and-poke.com

استراتژی‌های دیگر را نیز می‌توان برای حل مسئله‌ی بهترین تجارت با پیچیدگی زمانی بهتر استفاده کرد. به‌زودی آن‌ها را خواهیم دید. ولی در برخی موارد، رویکرد جستجوی فراگیر نیز بهترین پیچیدگی زمانی را ایجاد می‌کند. این مورد در مسئله‌ی بعد رخ می‌دهد:

> کوله‌پشتی: شما یک کوله‌پشتی برای حمل محصولات به‌منظور فروش دارید. این کوله‌پشتی وزن مشخصی را می‌تواند تحمل کند که برای تمام محصولات شما کافی نیست. شما باید انتخاب کنید کدام کالا را حمل کنید. با دانستن وزن و ارزش فروش هر محصول، انتخاب کدام محصولات بیشترین درآمد را به دنبال خواهد داشت؟

مجموعه‌ی توانی[1] محصولات شما شامل تمام انتخاب‌های ممکن محصولات است. یک رویکرد جستجوی فراگیر به‌سادگی تمام این انتخاب‌ها را بررسی می‌کند. به دلیل آن‌که هم‌اکنون می‌دانیم چگونه مجموعه‌ی توانی را محاسبه کنیم، الگوریتم جستجوی فراگیر ساده است:

---

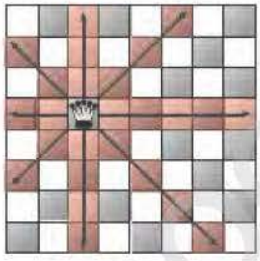
[1] مجدداً برای توضیح بیشتر در مورد مجموعه‌های توانی، پیوست III را ببینید.

```
function knapsack(items, max_weight)
    best_value ← 0
    for each candidate in power_set(items)
        if total_weight(candidate) ≤ max_weight
            if sales_value(candidate) > best_value
                best_value ← sales_value(candidate)
                best_candidate ← candidate
    return best_candidate
```

برای n محصول 2^n انتخاب وجود دارد. برای هر یک از آن‌ها بررسی می‌کنیم وزن کل از ظرفیت کوله‌پشتی بیشتر نشده باشد و این‌که ارزش فروش آن بهتر از بهترین گزینه تاکنون هست یا نه. تعداد ثابتی عملیات برای هر انتخاب از محصول‌ها وجود دارد، یعنی الگوریتم $O(2^n)$ است.

بااین‌حال، تمام انتخاب‌های محصولات نباید بررسی شوند. بسیاری از آن‌ها کوله‌پشتی را تا نصفه پر می‌کنند و این امر نشان می‌دهد رویکردهای بهتری نیز وجود دارد[1]. در ادامه استراتژی‌هایی را برای بهینه‌سازی جستجوی راه‌حل می‌آموزیم که به‌صورت کارا بسیاری از راه‌حل‌های کاندیدای ممکن را دور می‌ریزند.

۳-۴- عقب‌گرد



آیا تاکنون شطرنج بازی کرده‌اید؟ مهره‌های شطرنج بر روی یک صفحه‌ی ۸ در ۸ حرکت و به مهره‌های دشمن حمله می‌کنند. وزیر قوی‌ترین مهره است: این مهره می‌تواند هر مهره‌ای را که در سطر، ستون یا قطرش باشد بزند. استراتژی بعدی درزمینه‌ی یک مسئله‌ی معروف شطرنج بیان می‌شود:

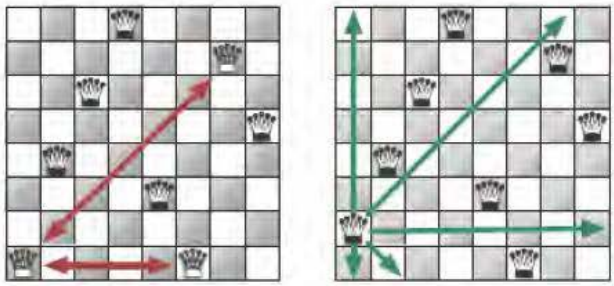
> معمای هشت وزیر: چگونه هشت وزیر را بر روی یک صفحه قرار می‌دهید به‌طوری‌که هیچ وزیری به وزیر دیگر حمله نکند؟

سعی کنید راه‌حل را به‌صورت دستی پیدا کنید: می‌بینید که چندان ساده نیست[2]. شکل ۳-۶ یکی از راه‌های قرار دادن وزیرها به‌صورت صلح‌آمیز را نشان می‌دهد.

---

[1] مسئله‌ی کوله‌پشتی جزئی از کلاس Np-کامل است که در بخش ۲-۳ موردبحث قرار گرفت. فارغ از بحث استراتژی فقط الگوریتم‌های نمایی آن را حل می‌کنند.

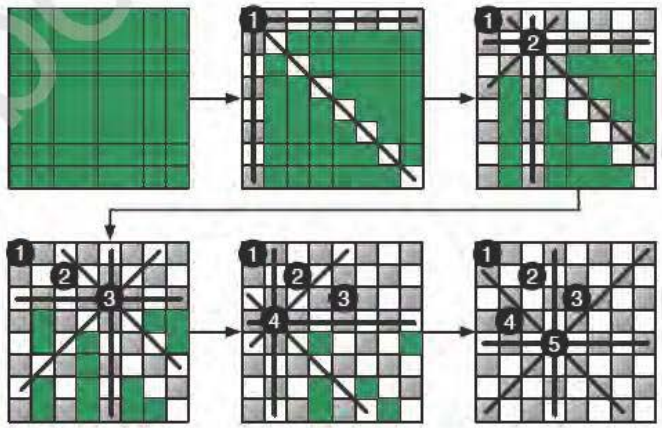
[2] می‌توانید به‌صورت آنلاین امتحان کنید: https://code.energy/8queens



شکل ۳-۶: وزیر سمت چپ به سایر وزیرها حمله می‌کند. با بالا بردن آن هیچ‌کدام از وزیرها تحت حمله‌ی دیگری قرار ندارند

در بخش ۱-۳ دیدیم که هشت وزیر را می‌توان به بیش از *چهار میلیارد راه* در صفحه‌ی شطرنج جا داد. حل این مسئله به روش جستجوی فراگیر نیازمند بررسی تمام این حالات ممکن است. تصور کنید که دو وزیر اول به‌طوری روی صفحه قرار می‌گیرند که به یکدیگر حمله می‌کنند: فارغ از اینکه سایر وزیرها کجا قرار بگیرند، به‌این‌ترتیب ممکن نیست راه‌حلی به دست آید. به‌این‌ترتیب، رویکرد جستجوی فراگیر از طریق بررسی تمام این حالات نامناسب موجب اتلاف زمان می‌شود.

جستجو برای محل‌های قرارگیری *مناسب* بسیار کاراتر است. اولین وزیر را هرجایی می‌توان گذاشت. جای مناسب برای وزیرهای بعدی بر اساس موقعیت وزیرهای روی صفحه محدود می‌شود: یک وزیر را نمی‌توان در محدوده‌ی حمله‌ی وزیر دیگر قرارداد. قرار دادن وزیرها با استفاده از این قانون، ما را به وضعیتی از صفحه می‌رساند که اضافه کردن یک وزیر جدید را، قبل از قرار گرفتن هشت وزیر بر روی صفحه، غیرممکن می‌سازد:



شکل ۳-۷: جا دادن یک وزیر موجب محدود شدن جای مناسب وزیرهای دیگر می‌شود

این بدان معنی است که آخرین وزیر در جای مناسبی قرار نگرفته است؛ بنابراین **عقب‌گرد**[1] می‌کنیم: جای وزیر قبلی را عوض کرده و جستجو را ادامه می‌دهیم. این ماهیت استراتژی عقب‌گرد است: به‌قرار دادن وزیرها درجاهای معتبر ادامه بده. وقتی به مشکل برخورد کردی، جای آخرین وزیر را عوض کن و ادامه بده. این فرایند را با استفاده از بازگشت می‌توان ایجاد کرد:

```
function queens(board)
    if board.has_8_queens
        return board
    for each position in board.unattacked_positions
        board.place_queen(position)
        solution ← queens(board)
        if solution
            return solution
        board.remove_queen(position)
    return False
```

اگر مسئله حل‌نشده باشد، بر روی تمام جاهای مناسب برای وزیر بعدی حلقه می‌زند. این فرایند از بازگشت برای بررسی اینکه آیا قرار دادن یک وزیر در هر موقعیت ممکن راه‌حلی ایجاد می‌کند یا خیر، استفاده می‌کند. نحوه‌ی عملکرد آن به‌صورت شکل ۳-۸ است.

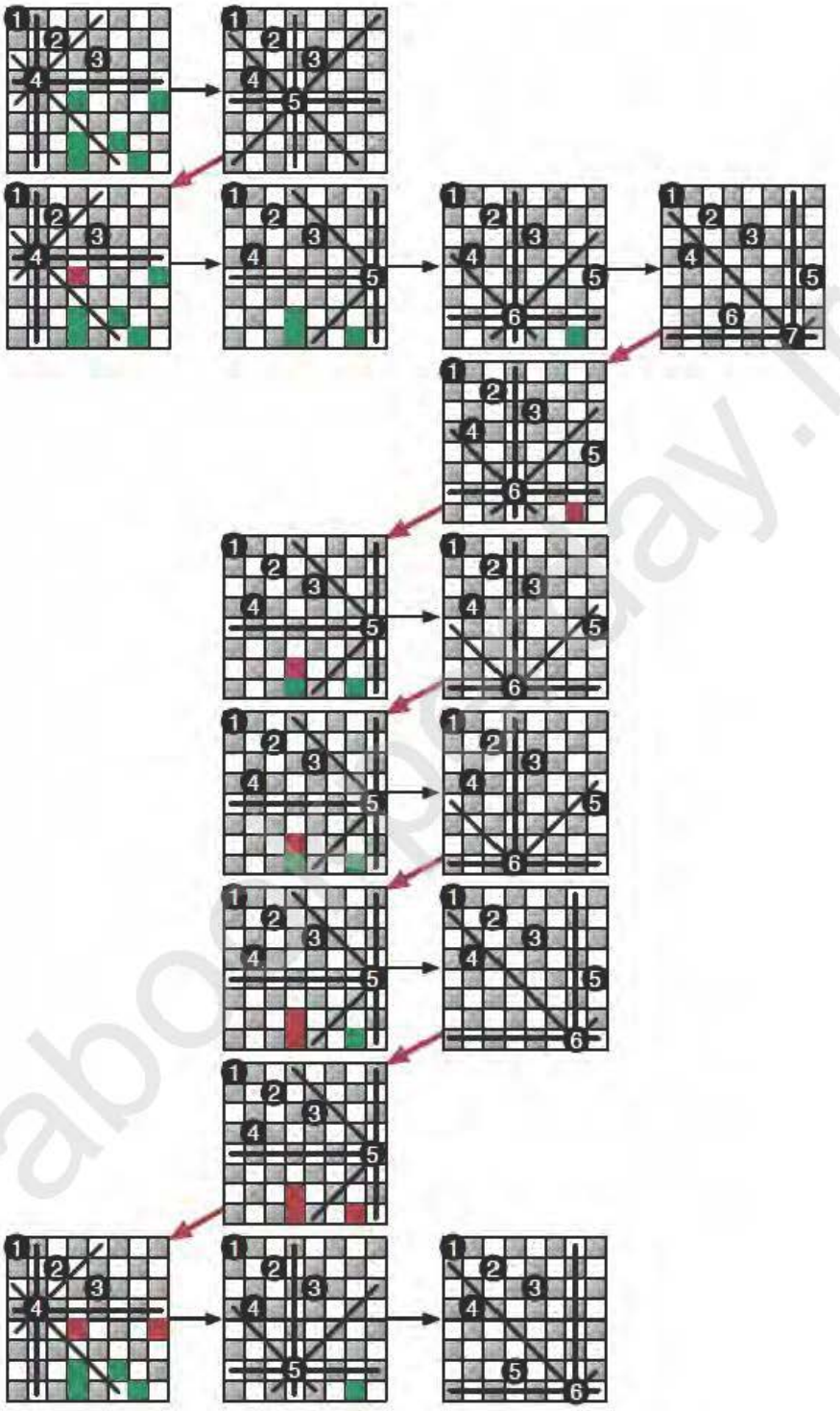
عقب‌گرد در مسائلی که راه‌حل آن‌ها دنباله‌ای از انتخاب‌ها است و انتخاب یک گزینه بر گزینه‌های بعدی تأثیر می‌گذارد، بهترین عملکرد را دارد. این روش در سریع‌ترین زمان ممکن انتخاب‌هایی را که راه‌حل موردنظر را به شما نمی‌دهند، شناسایی می‌کند، بنابراین می‌توانید زودتر عقب‌نشینی کرده و حالت دیگری را امتحان کنید. *از اشتباهات درس بگیرید.*

۳-۵ - ابتکار

در بازی استاندارد شطرنج شما ۳۲ مهره از ۶ نوع و ۶۴ خانه برای جابجایی دارید. بعدازاین که اولین حرکت را انجام دادید، هنوز ۲۸۸ میلیارد موقعیت ممکن وجود دارد. حتی قوی‌ترین بازیکنان جهان نیز نمی‌توانند بهترین حرکت را پیدا کنند. آن‌ها فقط سعی می‌کنند حرکتی را که به‌اندازه‌ی *کافی خوب* باشد پیدا کنند. ما همین کار را با الگوریتم‌ها انجام می‌دهیم. یک **روش ابتکاری**[2]، یا به‌صورت ساده **ابتکار**، روشی است که منجر به راه‌حلی بدون تضمین بهترین یا بهینگی می‌گردد. ابتکار زمانی که سایر روش‌ها مانند جستجوی فراگیر یا عقب‌گرد خیلی کند هستند، مفید خواهد بود. رویکردهای ابتکاری پیچیده‌ای وجود دارند، ولی ما بر ساده‌ترین آن‌ها تمرکز می‌کنیم: عقب‌گرد نکنید.

---

[1] Backtrack
[2] Heuristic



شکل ۳-۸: عقب‌گرد بر روی معماری هشت وزیر

حریصانه

یک رویکرد بسیار معمول ابتکاری، **رویکرد حریصانه**[1] است. این رویکرد شامل عدم برگشت به انتخاب‌های گذشته است. این روش متضاد عقب‌گرد است. سعی کنید در هر مرحله بهترین گزینه را انتخاب کنید و بعداً هیچ سؤالی درباره‌ی آن نپرسید. اجازه بدهید این استراتژی را برای حل مسئله‌ی کوله‌پشتی، اما با کمی پیچش، به کار بگیریم (بخش ۳-۳ را ببینید).

> کوله‌پشتی شیطانی: یک دزد حریص به خانه‌ی شما واردشده و می‌خواهید کالاهایی را که می‌خواهید بفروشید، بدزدد. او تصمیم می‌گیرد از کوله‌پشتی برای حمل کالاهای دزدیده‌شده استفاده کند. کدام کالاها را باید بدزدد؟ به یاد داشته باشید، هرچقدر زمان کمتری در خانه‌ی شما حضورداشته باشد، احتمال دستگیر شدنش نیز کمتر می‌شود.

در کل، در اینجا راه‌حل بهینه دقیقاً مشابه مسئله‌ی کوله‌پشتی است. بااین‌حال، دزد وقت کافی برای محاسبه‌ی تمام ترکیبات ممکن بسته‌بندی ندارد. حتی وقت ندارد از روش عقب‌گرد استفاده کند و کالاهایی را که در کوله‌پشتی قرار داده است، خارج کند. بسته‌بندی حریصانه کالاهای دارای بیشترین ارزش را تا زمانی که کوله‌پشتی جا دارد، درون آن قرار می‌دهد.

```
function greedy_knapsack(items, max_weight)
    bag_weight ← 0
    bag_items ← List.new
    for each item in sort_by_value(items)
        if max_weight ≤ bag_weight + item.weight
            bag_weight ← bag_weight + item.weight
            bag_items.append(item)
    return bag_items
```

در این روش ما هیچ بررسی در مورد تأثیر یک انتخاب بر انتخاب‌های آینده انجام نمی‌دهیم. این رویکرد حریصانه سریع‌تر از روش جستجوی فراگیر مجموعه‌ای از کالاها را پیدا می‌کند. بااین‌حال، هیچ تضمینی وجود ندارد که مجموعه‌ی حاصل بیشترین ارزش ممکن را داشته باشد.

---

[1] Greedy Approach

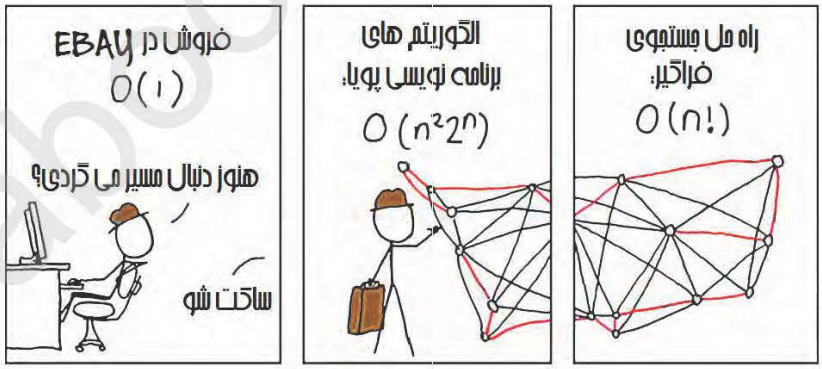
در تفکر محاسباتی، حریص بودن تنها یک گناه برای شیطان به‌حساب نمی‌آید. به‌عنوان یک تاجر قابل‌اعتماد، ممکن است بخواهید از روش حریصانه برای بسته‌بندی کوله‌پشتی استفاده کرده یا به روش حریصانه مسیر سفر را تعیین کنید:

> فروشنده‌ی دوره‌گرد، مجدد: یک فروشنده باید از n شهر مفروض بازدید کرده و در انتها به شهر شروع برگردد. کدام طرح سفر کل مسافت طی شده را کمینه می‌کند؟

در بخش ۱-۳ دیدیم که تعداد جایگشت‌های ممکن شهرها با افزایش کم تعداد شهرها به‌صورت انفجاری افزایش پیدا می‌کند. پیدا کردن راه‌حل بهینه برای مسئله‌ی فروشنده‌ی دوره‌گرد با چند هزار شهر بسیار پرهزینه (یا غیرممکن) است[1]. ولی شما همچنان به یک مسیر نیاز دارید. این‌یک الگوریتم حریصانه ساده برای این مسئله است:

۱. نزدیک‌ترین شهر بازدید نشده را بازدید کن.
۲. تا زمان بازدید شدن تمام شهرها ادامه بده.

آیا می‌توانید به یک راه ابتکاری بهتر از رویکرد حریصانه فکر کنید؟ این‌یک سؤال پژوهشی فعال بین دانشمندان کامپیوتر است.



شکل ۳-۹: مسئله‌ی فروشنده‌ی دوره‌گرد (دریافت شده از http://xkcd.com)

[1] مسئله‌ی فروشنده دوره‌گرد به کلاس NP-کامل که در بخش ۲-۳ بحث شد تعلق دارد. نمی‌توانیم یک راه‌حل بهینه را در زمانی بهتر از نمایی پیدا کنیم.

زمانی که حرص بر قدرت غلبه می‌کند

ترجیح دادن یک روش ابتکاری به یک الگوریتم کلاسیک نوعی برقراری تعادل است. چقدر برای شما قابل‌قبول است اگر از کوله‌پشتی یا مسیر مسافرت بهینه فاصله بگیرید؟ مورد به مورد انتخاب کنید.

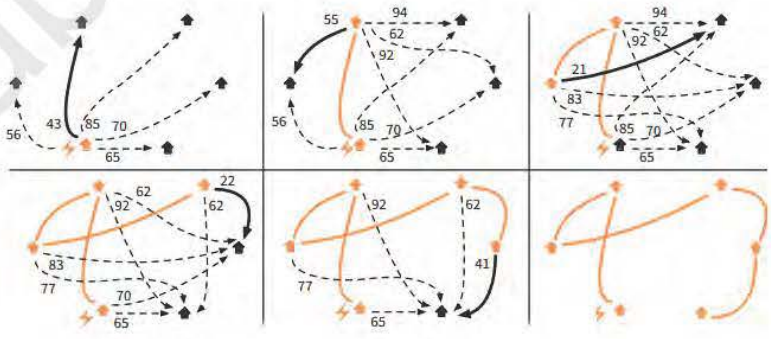
بااین‌حال، حتی وقتی نیاز به راه‌حل بهینه هم دارید، به‌طورکلی از حریصانه صرف‌نظر نکنید. راه حال ابتکاری یک مسئله ممکن است گاهی اوقات منجر به بهترین راه‌حل شود. به‌عنوان‌مثال، ممکن است یک الگوریتم حریصانه توسعه دهید که به‌صورت منظم راه‌حلی مشابه با یک حمله‌ی جستجوی فراگیر قدرتمند پیدا کند. اجازه بدهید ببینیم این امر چگونه ممکن است رخ دهد:

> شبکه‌ی برق: شهرک‌های یک منطقه‌ی دورافتاده برق ندارند، اما یکی از شهرک‌ها در حال ساخت نیروگاه است. برق را می‌توان از یک شهرک به شهرک بعدی که از طریق خطوط متصل شده‌اند، توزیع کرد. چگونه همه شهرک‌ها را با کمترین سیم به یک شبکه برق متصل می‌کنید؟

این مسئله به‌سادگی حل می‌شود:

۱. از بین شهرک‌هایی که برق ندارد، شهرکی را که به یک شهرک دارای برق نزدیک‌تر است انتخاب و آن‌ها را به هم وصل کن.
۲. تا زمان برق‌دار شدن همه‌ی شهرک‌ها ادامه بده.

در هر مرحله، با این فکر که می‌خواهیم بهترین گزینه را در لحظه‌ی جاری پیدا کنیم، یک زوج شهرک را برای اتصال انتخاب می‌کنیم. حتی بدون بررسی تأثیر یک انتخاب بر انتخاب‌های آینده، متصل کردن نزدیک‌ترین شهرک‌ها به یکدیگر همیشه بهترین گزینه است. ما اینجا خوش‌شانس هستیم: ساختار این مسئله کاملاً مناسب حل شدن با یک الگوریتم حریصانه است. در بخش بعد، ساختارهایی را برای مسائل خواهیم دید که مناسب یک استراتژی بسیار عمومی‌تر هستند.



شکل ۳-۱۰: حل مسئله‌ی شبکه‌ی برق با انتخاب‌های حریصانه

۳-۶ - تقسیم و حل

وقتی‌که دشمن به قسمت‌های کوچک‌تر تقسیم می‌شود، غلبه بر آن بسیار ساده‌تر است. سزار و ناپلئون با تقسیم و حل کردن دشمنانشان بر اروپا حکومت کردند. می‌توانید با همین استراتژی مسائل را حل کنید، به‌ویژه مسائلی که دارای **زیرساختارهای بهینه**[1] هستند. مسائل دارای زیرساختارهای بهینه را می‌توان به زیرمسئله‌های مشابه ولی با اندازه‌ی کوچک‌تر تقسیم کرد. می‌توان آن‌ها را تا زمان رسیدن به زیرمسئله‌های ساده تقسیم کرد. سپس راه‌حل زیرمسئله‌ها باهم ترکیب می‌شوند تا راه‌حل مسئله‌ی اصلی به دست آید.

تقسیم و مرتب‌سازی

اگر لیست بزرگی برای مرتب‌سازی داشته باشیم، می‌توانیم آن را به دونیمه تقسیم کنیم: هر نیم لیست به یک زیرمسئله‌ی مرتب‌سازی تبدیل می‌شود. برای مرتب‌سازی لیست بزرگ، راه‌حل زیرمسئله‌ها (یعنی نیمه‌های مرتب‌شده) را می‌توان در یک لیست واحد با استفاده از الگوریتم merge باهم ادغام کرد[2]. ولی دو زیرمسئله را چطور مرتب کنیم؟ آن‌ها را نیز می‌توان با تقسیم به زیرمسئله‌هایی، مرتب‌سازی و ادغام کرد. زیرمسئله‌های جدید نیز مجدداً تقسیم، مرتب‌سازی و ادغام می‌شوند. عمل تقسیم تا زمان رسیدن به حالت پایه ادامه پیدا می‌کند: یک لیست تک عنصری. یک لیست تک عنصری خودبه‌خود مرتب است!

```
function merge_sort(list)
    if list.length = 1
        return list
    left ← list.first_half
    right ← list.last_half
    return merge(merge_sort(left),
                 merge_sort(right))
```

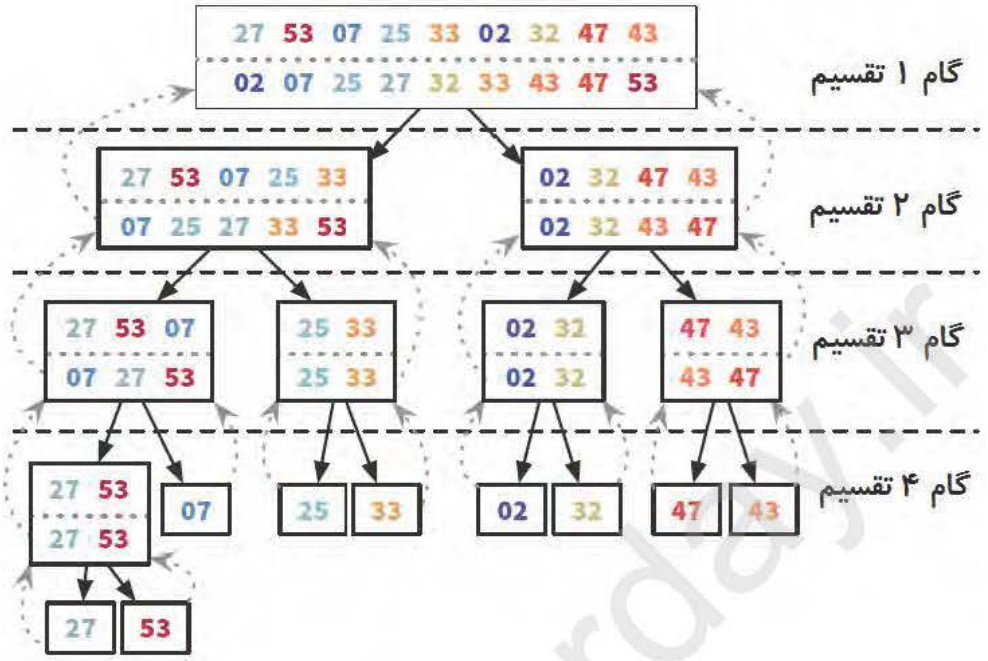
این الگوریتم بازگشتی زیبا را **مرتب‌سازی ادغامی**[3] می‌نامند. درست همانند دنباله‌ی فیبوناچی (بخش ۳-۲)، یک درخت بازگشت به ما کمک می‌کند بفهمیم تابع merge_sort چند بار خودش را فراخوانی می‌کند:

---

[1] Optimal Substructure

[2] اولین الگوریتمی که در این فصل دیدیم (بخش ۳-۱).

[3] Merge Sort



شکل ۳-۱۱: اجرای ساده‌ی مرتب‌سازی ادغامی. هر مستطیل یک فراخوانی تابع `merge_sort` را نشان می‌دهد. ورودی‌ها در بالا و خروجی‌ها در پایین قرار دارند

حال اجازه بدهید پیچیدگی زمانی مرتب‌سازی ادغامی را بیابیم. برای این کار، ابتدا باید تعداد عملیات ایجادشده در هر گام تقسیم را بشماریم. سپس، تعداد کل گام‌های تقسیم را می‌شماریم.

شمارش عملیات: فرض کنید لیست بزرگی به‌اندازه‌ی n داریم. تابع `merge_sort` در زمان فراخوانی عملیات زیر را انجام می‌دهد:

- تقسیم لیست به دونیمه که ربطی به‌اندازه‌ی لیست ندارد: $O(1)$.
- یک `merge`: از بخش ۳-۱ به یاد داریم که تابع `merge` از نوع $O(n)$ است.
- دو فراخوانی بازگشتی `merge_sort` که شمرده نمی‌شوند[1].

به دلیل اینکه قوی‌ترین عبارت را در نظر گرفته و فراخوانی‌های بازگشتی را نمی‌شماریم، پیچیدگی زمانی تابع $O(n)$ است. حال اجازه بدهید پیچیدگی زمانی هر گام تقسیم را بشماریم:

گام ۱ تقسیم: تابع `merge_sort` برای لیستی حاوی n عنصر فراخوانی می‌شود. پیچیدگی زمانی این گام $O(n)$ است.

[1] عملیات انجام‌شده به‌وسیله‌ی فراخوانی‌های بازگشتی در گام بعدی تقسیم شمرده می‌شوند.

گام ۲ تقسیم: تابع merge_sort دو بار و هر بار با $n/2$ عنصر فراخوانی می‌شود. درمی‌یابیم $2 \times O(n/2) = O(n)$ است.

گام ۳ تقسیم: تابع merge_sort چهار بار و هر بار با $n/4$ عنصر فراخوانی می‌شود. درمی‌یابیم $4 \times O(n/4) = O(n)$ است.

.

.

.

گام x تقسیم: تابع merge_sort به تعداد 2^x و هر بار برای لیستی با $n/2^x$ عنصر فراخوانی می‌شود. درمی‌یابیم $2^x \times O(n/2^x) = O(n)$ است.

گام‌های تقسیم همگی دارای پیچیدگی مشابه $O(n)$ هستند. بنابراین پیچیدگی زمانی مرتب‌سازی ادغامی به‌صورت $x \times O(n)$ است، به‌طوری که x تعداد گام‌های تقسیم موردنیاز برای اجرای کامل آن است[1].

شمارش گام‌ها: چطور می‌توانیم مقدار x را اندازه بگیریم؟ می‌دانیم توابع بازگشتی در زمان رسیدن به حالت پایه فراخوانی خود را متوقف می‌کنند. حالت پایه‌ی ما لیست تک عنصری است. هم‌چنین دیدیم که گام x تقسیم بر روی لیست‌هایی با $n/2^x$ عنصر کار می‌کند، از این رو

$$\frac{n}{2^x} = 1 \quad \rightarrow \quad 2^x = n \quad \rightarrow \quad x = \log_2 n$$

اگر با تابع $\log_2$ آشنا نیستید، نترسید! $x = \log_2 n$ روش دیگری برای نوشتن $2^x = n$ است. کدنویسان به رشد لگاریتمی علاقه دارند. در جدول ۳-۱ ببینید که تعداد گام‌های تقسیم موردنیاز با افزایش تعداد کل عناصری که باید مرتب شوند، چقدر کند افزایش می‌یابند[2].

ازاین‌رو، پیچیدگی زمانی الگوریتم مرتب‌سازی ادغامی $\log_2 n \times O(n) = O(n \log n)$ است. که بهبود *بسیار زیادی* نسبت به $O(n^2)$ مرتب‌سازی انتخابی پیداکرده است. شکاف کارایی بین الگوریتم‌های خطی‌لگاریتمی و درجه ۲ را که در شکل ۲-۴ دیدیم، به یاد می‌آورید؟ حتی اگر یک

---

[1] نمی‌توانیم از x صرف‌نظر کنیم، چون یک مقدار ثابت نیست. اگر اندازه‌ی لیست یعنی n دو برابر شود، به یک گام اضافه‌ی تقسیم نیاز خواهید داشت. اگر n چهار برابر شود، به دو گام اضافه‌ی تقسیم نیاز خواهید داشت.

[2] هر فرایندی که یک ورودی را گام‌به‌گام و با تقسیم آن به یک عامل ثابت در هر گام کوچک می‌کند، به تعدادی لگاریتمی گام برای کاهش کامل ورودی نیاز دارد.

کامپیوتر سریع‌تر الگوریتم $O(n^2)$ را اجرا کند، دیرتر از کامپیوتری که $O(n \log n)$ را اجرا می‌کند به پایان می‌رسد (جدول ۳-۲).

جدول ۳-۱: تعداد گام‌های تقسیم موردنیاز برای ورودی با اندازه‌های مختلف

اندازه ورودی (n)	$\log_2 n$	تعداد گام‌های تقسیم موردنیاز
۱۰	۳/۳۲	۴
۱۰۰	۶/۶۴	۷
۱،۰۲٤	۱۰/۰۰	۱۰
۱،۰۰۰،۰۰۰	۱۹/۹۳	۲۰
۱،۰۰۰،۰۰۰،۰۰۰	۲۹/۸۹	۳۰

جدول ۳-۲: برای ورودی‌های بزرگ الگوریتم‌های $O(n \log n)$ در کامپیوترهای کند بسیار سریع‌تر از الگوریتم‌های $O(n^2)$ در کامپیوترهایی که هزار بار سریع‌تر هستند کار می‌کنند.

اندازه ورودی	درجه ۲	خطی‌لگاریتمی
۱۹۶ (کشورهای دنیا)	۳۸ میلی‌ثانیه	۲ ثانیه
۴۴ هزار (فرودگاه در دنیا)	۳۲ دقیقه	۱۲ دقیقه
۱۷۱ هزار (کلمات فرهنگ لغت انگلیسی)	۸ ساعت	۵۱ دقیقه
۱ میلیون (ساکنان هاوایی)	۱۲ روز	۶ ساعت
۱۹ میلیون (ساکنان فلوریدا)	۱۱ سال	۶ روز
۱۳۰ میلیون (کل کتاب‌های منتشرشده)	۵۰۰ سال	۴۱ روز
۷/۵ میلیارد (صفحات وب روی اینترنت)	۷۰۰ هزار سال	۵ سال

خودتان ببینید: یک الگوریتم مرتب‌سازی خطی‌لگاریتمی و یک الگوریتم مرتب‌سازی درجه ۲ بنویسید، اجرای آن‌ها را بر روی لیست‌های تصادفی با اندازه‌های متفاوت مقایسه کنید. برای ورودی‌های بزرگ، چنین بهبودهایی در پیچیدگی بسیار حیاتی هستند. حال اجازه بدهید به تقسیم و حل مسائلی بپردازیم که پیش‌تر سعی کردیم با جستجوی فراگیر آن‌ها را حل کنیم.

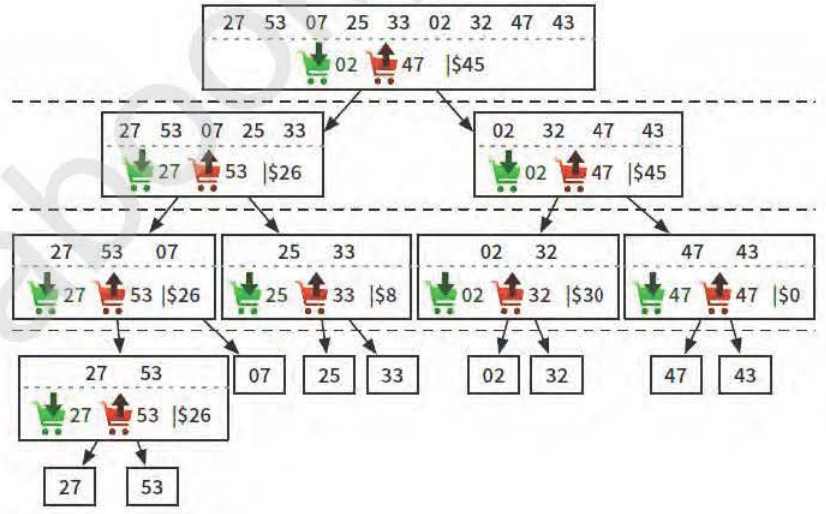
تقسیم و تجارت

تقسیم و حل رویکرد بهتری برای مسئله‌ی بهترین تجارت (بخش ۳-۳) نسبت به جستجوی فراگیر است. تقسیم تاریخچه‌ی قیمت‌ها به دونیم، منجر به دو زیرمسئله می‌شود: یافتن بهترین تجارت در بخش اول و سپس در بخش دوم. بهترین تجارت در کل دوره نیز به‌صورت زیر حساب می‌شود:

۱. بهترین تجارتی که خرید‌وفروش آن در بخش اول انجام می‌شود.
۲. بهترین تجارتی که خرید‌وفروش آن در بخش دوم انجام می‌شود.
۳. بهترین تجارتی که خرید آن در بخش اول فروش آن در بخش دوم انجام می‌شود.

دو حالت اول، راه‌حل زیرمسئله‌ها هستند. حالت سوم نیز به‌سادگی پیدا می‌شود: خرید با کمترین قیمت در بخش اول و فروش به بالاترین قیمت در بخش دوم. برای ورودی‌هایی که فقط شامل یک روز هستند، تنها راه‌حل ممکن خرید‌وفروش در همان روز است که منجر به سود برابر با صفر می‌شود.

```
function trade(prices)
    if prices.length = 1
        return 0
    former ← prices.first_half
    latter ← prices.last_half
    case3 ← max(latter) - min(former)
    return max(trade(former), trade(latter), case3)
```



شکل ۳-۱۲: اجرای ساده تابع trade. مستطیل‌ها فراخوانی‌های trade با ورودی‌ها و خروجی‌های آن‌ها هستند

زمانی که trade فراخوانی می‌شود، حالت ساده را بررسی کرده و سپس به عمل تقسیم و یافتن بیشینه و کمینه در دو بخش ورودی می‌پردازد. پیدا کردن بیشینه یا کمینه‌ی n عنصر نیازمند بررسی تمام n عنصر است، لذا هزینه‌ی یک فراخوانی مجزای trade از نوع $O(n)$ است.

می‌بینید که درخت بازگشت trade (شکل ۳-۱۲) بسیار شبیه به درخت بازگشت مرتب‌سازی ادغامی (شکل ۳-۱۱) است. این درخت نیز $\log_2 n$ گام تقسیم دارد که هر گام دارای هزینه‌ی $O(n)$ است. ازاین‌رو، پیچیدگی trade نیز $O(n \log n)$ است و به نسبت الگوریتم قبلی مبتنی بر رویکرد جستجوی فراگیر که $O(n^2)$ بود بهبود بسیاری یافته است.

تقسیم و بسته‌بندی

تقسیم و حل را بر روی مسئله‌ی کوله‌پشتی (بخش ۳-۳) می‌توان انجام داد. به یاد دارید، ما n محصول برای انتخاب داریم. ویژگی‌های هر محصول به‌صورت زیر هستند:

- مقدار w_i وزن کالای iام است.
- مقدار v_i ارزش کالای iام است.

اندیس هر عنصر کالا یعنی i می‌تواند یک عدد بین ۱ تا n باشد. مقدار بیشینه‌ی سود برای یک کوله‌پشتی با ظرفیت c از طریق انتخاب از بین n کالا با $k(n, c)$ نشان داده می‌شود. اگر یک کالای اضافه‌ی $i = n + 1$ در نظر گرفته شود، ممکن است مقدار بیشینه‌ی ممکن سود را افزایش بدهد یا ندهد. این مقدار برابر است با

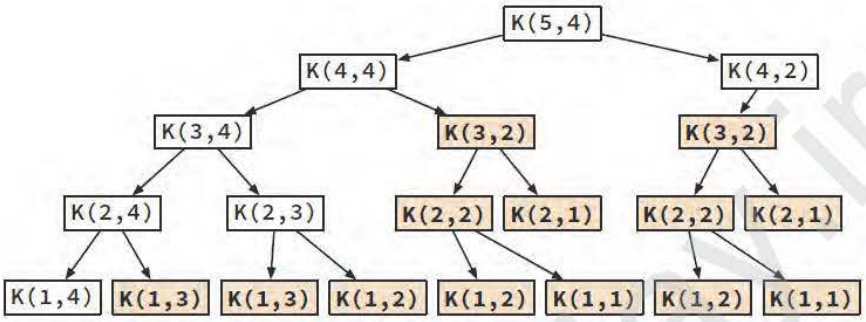
۱. مقدار $k(n, c)$ اگر کالای اضافه انتخاب نشود.
۲. مقدار $k(n, c - w_{n+1}) + v_{n+1}$ اگر کالای اضافه انتخاب شود.

حالت ۱، کالای اضافه را کنار می‌گذارد. حالت ۲ شامل کالای اضافه و انتخاب از بین کالاهای اولیه ضمن توجه به وجود فضای کافی برای آن‌ها است. این بدان معنی است که می‌توانیم راه‌حل برای n کالا را به‌صورت بیشینه مقدار زیرراه‌حل‌های $n - 1$ کالا تعریف کنیم:

$$K(n, c) = \max(K(n-1, c), K(n-1, c - w_n) + v_n)$$

حال تبدیل کردن این فرمول بازگشتی به یک الگوریتم بازگشتی ساده است. شکل ۳-۱۳ نحوه‌ی حل یک مسئله‌ی نمونه را به‌وسیله‌ی فرایند بازگشتی نشان می‌دهد. مستطیل‌هایی که بیش از یک‌بار

ظاهرشده‌اند، به‌صورت متمایزی نشان داده‌شده‌اند، زیرا نمایانگر زیر مسائل مشابهی هستند که بیش از یک‌بار در این فرایند محاسبه می‌شوند. در ادامه یاد می‌گیریم چگونه با اجتناب از چنین محاسبات تکراری، کارایی را افزایش دهیم.



شکل ۳-۱۳: حل یک مسئله‌ی کوله‌پشتی با ۵ کالا و ظرفیت ٤ برای کوله‌پشتی. کالاهای شماره ٤ و ۵ وزن ۲ و سایر کالاها وزن ۱ دارند

۳-۷- برنامه‌نویسی پویا

گاهی اوقات محاسبات مشابه چندین بار در حین حل یک مسئله انجام می‌شوند[1]. **برنامه‌نویسی پویا**[2] زیرمسئله‌های مشابه را به‌منظور محاسبه‌ی یک‌باره‌ی آن‌ها شناسایی می‌کند. یک‌راه معمول برای این کار روشی مشابه با به خاطرسپردن است، با املایی شبیه همان[3].

به‌خاطرسپاری فیبوناچی

الگوریتم محاسبه‌ی اعداد فیبوناچی را به یاد می‌آورید؟ درخت بازگشت آن (شکل ۳-۳) نشان می‌دهد که `fib(3)` چندین بار تکرار شده است. می‌توانیم این مشکل را با ذخیره‌سازی محاسبات `fib` در حین انجام آن‌ها حل کرده و فقط فراخوانی‌هایی از `fib` را محاسبه کنیم که تاکنون ذخیره نشده‌اند. این حیله برای استفاده‌ی مجدد از محاسبات را **به‌خاطرسپاری**[4] می‌نامند. این عمل موجب بهبود چشمگیر کارایی `fib` می‌شود.

---

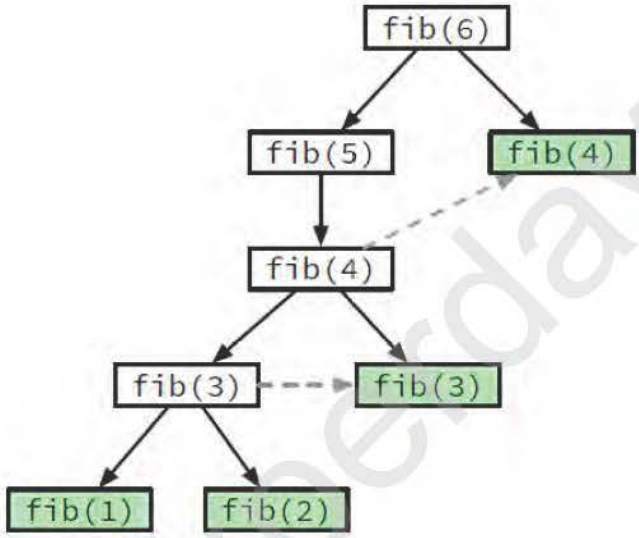
[1] مسائلی که در آن‌ها چنین پدیده‌ای رخ می‌دهد را زیرمسئله‌های هم‌پوشا می‌نامند.

[2] Dynamic Programming

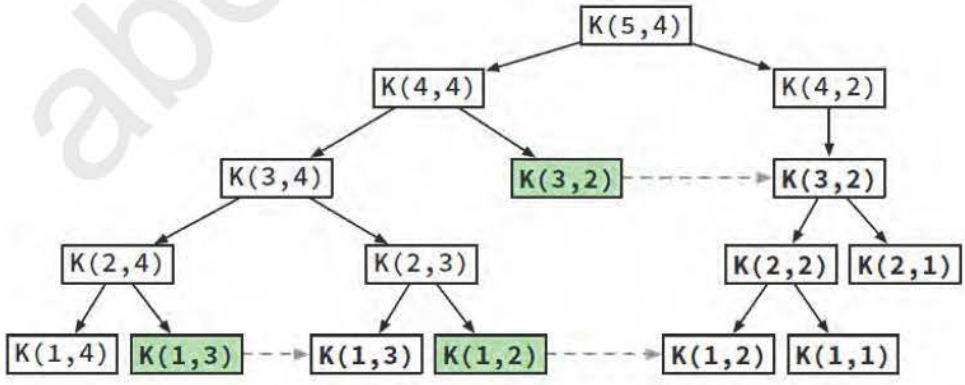
[3] در متن اصلی از کلمه‌ی Memorizing استفاده‌شده است که ازلحاظ املایی به کلمه‌ی Memoizing شبیه است. م.

[4] Memoization

```
M ← [1 ⇒ 1; 2 ⇒ 2]
function dfib(n)
    if n not in M
        M[n] ← dfib(n-1) + dfib(n-2)
    return M[n]
```



شکل ۳-۱۴: درخت بازگشت dfib. مستطیل‌های رنگی فراخوانی‌هایی را نشان می‌دهند که مجدداً محاسبه نشده‌اند



شکل ۳-۱۵: حل کوله‌پشتی بازگشتی با به‌خاطرسپاری

به‌خاطرسپاری کوله‌پشتی

بدیهی است که چندین فراخوانی تکراری در درخت بازگشت کوله‌پشتی (شکل ۳-۱۳) وجود دارند. با استفاده از روشی مشابه با روش مورداستفاده در تابع فیبوناچی، می‌توان از این محاسبات مجدد اجتناب کرد. این امر منجر به کاهش تعداد محاسبات می‌گردد.

برنامه‌نویسی پویا می‌تواند برنامه‌های بسیار کند را به کدهایی با سرعت قابل‌قبول تبدیل کند. الگوریتم خود را به‌دقت تحلیل کنید تا مطمئن شوید هیچ محاسبه‌ی تکراری در آن انجام نمی‌شود. در بخش بعد، خواهیم دید که گاهی اوقات پیدا کردن زیرمسئله‌های هم‌پوشا به‌سادگی ممکن نیست.

بهترین تجارت پایین به بالا

درخت بازگشت برای trade (شکل ۳-۱۲) هیچ فراخوانی تکراری ندارد، بااین‌حال محاسبات تکراری انجام می‌دهد. این الگوریتم، ورودی را برای یافتن مقادیر بیشینه و کمینه بررسی می‌کند. بعدازاین که ورودی به دو بخش تقسیم شد، فراخوانی‌های بازگشتی مجدداً ورودی را برای یافتن بیشینه و کمینه در این دو بخش بررسی می‌کنند[1]. به رویکرد دیگری برای اجتناب از این بررسی‌های تکراری نیاز داریم.

تاکنون ما از یک رویکرد **بالا به پایین**[2] استفاده می‌کردیم که در آن ورودی کاهش می‌یافت تا زمانی که به حالت پایه برسد. ولی می‌توانیم به‌صورت **پایین به بالا**[3] عمل کنیم: ابتدا حالت‌های پایه را بررسی کنیم، آن‌ها را باهم ترکیب کرده و همین کار را بارها تا زمان رسیدن به راه‌حل کلی تکرار کنیم. اجازه بدهید مسئله‌ی بهترین تجارت (بخش ۳-۳) را به این روش حل کنیم.

بیایید $P(n)$ را قیمت در روز nام بنامیم. هم‌چنین بیایید $B(n)$ را بهترین روز برای خرید در صورت فروش در روز nام بنامیم. اگر در روز اول بفروشیم، فقط در روز اول می‌توانیم بخریم، ازاین‌رو $B(1) = 1$. ولی اگر در روز دوم بفروشید، $B(2)$ می‌تواند ۱ یا ۲ باشد:

- اگر $P(2) < P(1)$ آنگاه $B(2) = 2$ (خریدوفروش در روز دوم)
- اگر $P(2) \geq P(1)$ آنگاه $B(2) = 1$ (خرید در روز اول و فروش در روز دوم)

---

[1] فرض کنید می‌خواهید بلندقدترین مرد، بلندقدترین زن و بلندقدترین فرد در یک اتاق را پیدا کنید. آیا هر کس را برای شناسایی بلندقدترین فرد اندازه‌گیری کرده، سپس هر زن و هر مرد را مجدداً برای شناسایی بلندقدترین زن و مرد اندازه‌گیری می‌کنید؟

[2] Top-Down

[3] Bottom-Up

روز دارای کمترین قیمت قبل از روز سوم (بدون احتساب روز سوم) برابر ست با $B(2)$ پس برای $B(3)$:

- $P(3) < P(B(2))$ قیمت روز $\rightarrow$ $B(3) = 3$
- $P(3) \geq P(B(2))$ قیمت روز $\rightarrow$ $B(3) = B(2)$

توجه کنید روز دارای کمترین قیمت قبل از روز چهارم $B(3)$ است. در حقیقت، برای هر n مقدار $B(n-1)$ برابر با روزی قبل از روز n است که کمترین قیمت را دارد. با استفاده از این روش می‌توانیم $B(n)$ را در قالب $B(n-1)$ بیان کنیم:

$$B(n) = \begin{cases} n & if \quad P(n) < P(B(n-1)) \\ B(n-1) & \text{در غیر اینصورت} \end{cases}$$

با داشتن تمام زوج‌های $[n, B(n)]$ برای هر روز n در ورودی، راه‌حل زوجی است که بیشترین سود را دارد. این الگوریتم مسئله را با محاسبه‌ی تمام مقادیر B از پایین به بالا حل می‌کند.

```
function trade_dp(P)
    B[1] ← 1
    sell_day ← 1
    best_profit ← 0

    for each n from 2 to P.length
        if P[n] < P[B[n-1]]
            B[n] ← n
        else
            B[n] ← B[n-1]

        profit ← P[n] - P[B[n]]
        if profit > best_profit
            sell_day ← n
            best_profit ← profit

    return (sell_day, B[sell_day])
```

این الگوریتم تعداد ثابتی عملیات را به ازای هر عنصر در لیست ورودی انجام می‌دهد، بنابراین $O(n)$ است. به‌این‌ترتیب کارایی نسبت به الگوریتم $O(n \log n)$ قبلی بسیار بهبود می‌یابد. به‌علاوه، این الگوریتم به‌هیچ‌عنوان با الگوریتم $O(n^2)$ جستجوی فراگیر قابل‌مقایسه نیست. پیچیدگی فضایی این

الگوریتم نیز $O(n)$ است زیرا تعداد عناصر بردار کمکی B برابر با تعداد عناصر ورود است. در پیوست IV، می‌توانید ببینید چگونه در حافظه‌ی کامپیوتر با تبدیل یک الگوریتم به $O(1)$ در بعد پیچیدگی فضایی صرفه‌جویی کنید.

۳-۸- شاخه و حد

بسیاری مسائل شامل کمینه‌سازی یا بیشینه‌سازی یک مقدار هدف هستند: پیدا کردن کوتاه‌ترین مسیر، به دست آوردن بیشترین سود و غیره. این مسائل را **مسائل بهینه‌سازی**[۱] می‌نامند. وقتی راه‌حل دنباله‌ای از انتخاب‌ها است، معمولاً از استراتژی به نام **شاخه و حد**[۲] استفاده می‌کنیم. هدف از این استراتژی حفظ زمان از طریق تشخیص و حذف سریع گزینه‌های بد است. برای درک این‌که گزینه‌های بد چگونه شناسایی می‌شوند، باید ابتدا مفاهیم حد بالا و پایین را یاد بگیریم.

حدهای بالا و پایین

منظور از حد محدوده‌ای از مقادیر است. یک **حد بالا**[۳] محدودیتی بر روی بیشترین مقدار ممکن تعریف می‌کند. یک **حد پایین**[۴] حداقل مقدار مورد انتظار است: این حد تضمین می‌کند مقدار موردنظر برابر یا بیشتر از عدد تعیین‌شده است.

ما اغلب به‌راحتی می‌توانیم به راه‌حل‌های زیربهینه برسیم: یک مسیر کوتاه، ولی نه الزاماً کوتاه‌ترین مسیر؛ یک سود زیاد، ولی نه الزاماً بیشترین سود. این راه‌حل‌ها حدهایی را برای راه‌حل بهینه فراهم می‌کنند. به‌عنوان نمونه، کوتاه‌ترین مسیر بین دو مکان هیچ‌گاه کوتاه‌تر از فاصله‌ی مستقیم بین آن‌ها نخواهد بود. ازاین‌رو، فاصله‌ی مستقیم یک حد پایین برای کمترین مسافت رانندگی است.

در مسئله‌ی کوله‌پشتی شیطانی (بخش ۳-۵)، سود حاصل از greedy_knapsack یک حد پایین برای سود بهینه است (این مقدار ممکن است نزدیک به سود بهینه باشد یا نباشد). حال نسخه‌ای از مسئله‌ی کوله‌پشتی را تصور کنید که در آن همه‌ی کالاها از جنس پودر هستند، بنابراین می‌توانیم کسری از کالاها را درون کوله‌پشتی قرار بدهیم. این نسخه‌ی مسئله را می‌توان به یک روش ساده‌ی حریصانه حل کرد: کالاهایی را زودتر در کوله‌پشتی جا می‌دهیم که نسبت ارزش به وزن بیشتری دارند:

---

۱ Optimization Problems
۲ Branch and Bound
۳ Upper Bound
۴ Lower Bound

```
function powdered_knapsack(items, max_weight)
    bag_weight ← 0
    bag_items ← List.new
    items ← sort_by_value_weight_ratio(items)
    for each i in items
        weight ← min(max_weight - bag_weight,
                     i.weight)
        bag_weight ← bag_weight + weight
        value ← weight * i.value_weight_ratio
        bagged_value ← bagged_value + value
        bag_items.append(item, weight)
    return bag_items, bag_value
```

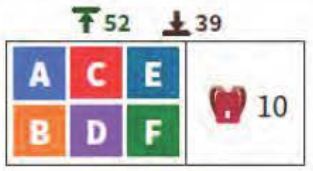
اضافه کردن محدودیتی که موجب تقسیم‌ناپذیر شدن کالاها می‌شود، فقط موجب کاهش بیشترین سود ممکن می‌گردد زیرا باید آخرین کالای اضافه‌شده به کوله‌پشتی را با یک کالای ارزان‌تر جایگزین کنیم. این بدان معنی است که powdered_knapsack یک حد بالا برای سود بهینه با عناصر تقسیم‌ناپذیر ارائه می‌دهد[1].

شاخه و حد در مسئله‌ی کوله‌پشتی

پیش‌تر دیدیم که پیدا کردن سود بهینه در مسئله‌ی کوله‌پشتی نیازمند محاسبات پرهزینه‌ی $\mathcal{O}(2^n)$ است. بااین‌حال، می‌توانیم حد بالا و پایین سود بهینه را به‌راحتی با استفاده از powdered_knapsack و greedy_knapsack به دست بیاوریم. بیایید این کار را بر روی یک مسئله‌ی نمونه‌ی کوله‌پشتی امتحان کنیم:

کالا	ارزش	وزن	نسبت ارزش به وزن	حداکثر ظرفیت
A	۲۰	۵	۴/۰۰	۱۰
B	۱۹	۴	۴/۷۵	
C	۱۶	۲	۸/۰۰	
D	۱۴	۵	۲/۸۰	
E	۱۳	۳	۴/۳۳	
F	۹	۲	۴/۵۰	

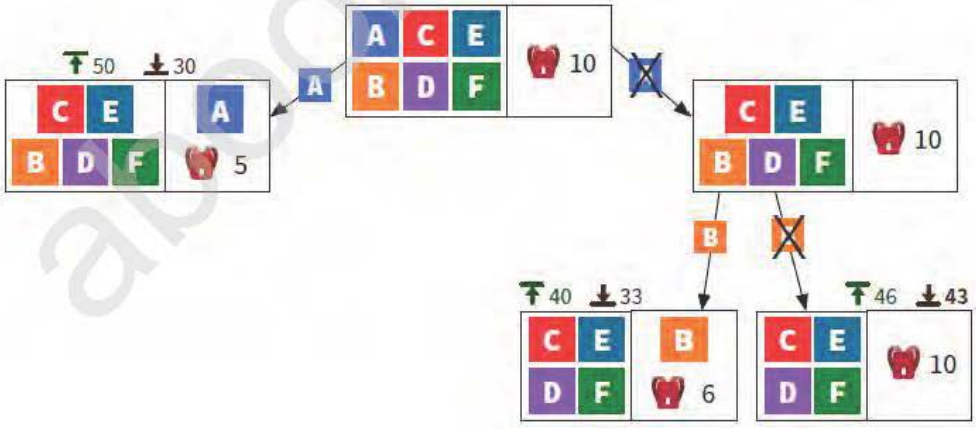
[1] روش حذف محدودیت‌های مسئله را سست‌سازی (Relaxation) می‌نامند. این روش اغلب برای محاسبه‌ی حدها در مسائل بهینه‌سازی استفاده می‌شود.



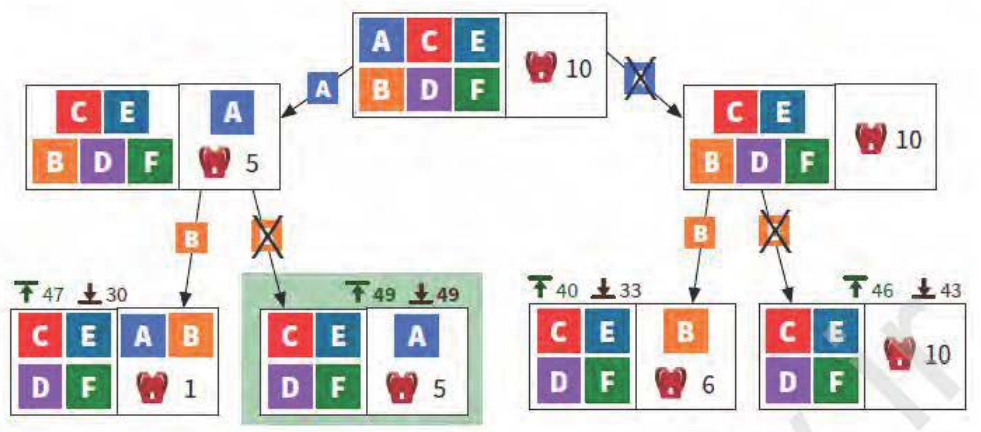
تصویر سمت چپ وضعیت قبل از شروع به بسته‌بندی را نمایش می‌دهد. کادر اول کالاهای بسته‌بندی نشده که باید بررسی شوند را نشان می‌دهد. کادر دوم ظرفیت موجود کوله‌پشتی و کالاهای درون آن را نشان می‌دهد.

اجرای greedy_knapsack سودی معادل ۳۹ و اجرای powdered_knapsack سودی معادل ۵۲/۶۶ را ایجاد می‌کند. این بدان معنی است که سود بهینه بین ۳۹ و ۵۲/۶۶ است. بخش ۳-۶ به ما یاد داد که این مسئله با n کالا را می‌توان به دو زیرمسئله با $1 - n$ کالا تقسیم کرد. اولین زیرمسئله قرار گرفتن کالای A در کوله‌پشتی و زیرمسئله‌ی دوم قرار نگرفتن این کالا در کوله‌پشتی را نشان می‌دهند:

حد بالا و پایین دو زیرمسئله را به دست می‌آوریم. یکی دارای حد پایین ۴۸ است: حال می‌دانیم سود بهینه بین ۴۸ و ۵۲ است. اجازه بدهید زیرمسئله‌ی سمت راست را بررسی کنیم، زیرا حدهای جذاب‌تری دارد:



حال اولین زیرمسئله از سمت چپ موردی است که امیدبخش‌ترین حد بالا را دارد. اجازه بدهید بررسی خود را با تقسیم این زیرمسئله ادامه بدهیم:



حال می‌توانیم یک نتیجه‌ی مهم بگیریم. زیرمسئله‌ی مشخص‌شده دارای حد پایین ۴۹ و حد بالایی با همین مقدار است. این بدان معنی است که سود بهینه از این زیرمسئله باید *دقیقاً* ۴۹ باشد. به‌علاوه، توجه کنید که ۴۹ از حد بالای تمام زیرمسئله‌های شاخه‌های دیگر نیز بیشتر است. این امر موجب توقف بررسی می‌شود. هیچ زیرمسئله‌ی دیگری در سایر شاخه‌ها سودی بهتر از ۴۹ نمی‌دهد، یعنی می‌توانیم این شاخه‌ها را از جستجوی خود حذف کنیم.

استفاده‌ی عاقلانه از حدهای بالا و پایین به ما اجازه می‌دهد سود بهینه را با حداقل تلاش محاسباتی به دست بیاوریم. به‌این‌ترتیب با بررسی احتمالات ممکن فضای جستجوی خود را به‌صورت پویا وفق می‌دهیم. به‌طور خلاصه می‌توان گفت شاخه و حد به روش زیر عمل می‌کند:

۱. مسئله را به زیرمسئله‌ها تقسیم کن.
۲. حد بالا و پایین هر زیرمسئله را پیدا کن.
۳. حد زیرمسئله‌ها در تمام شاخه‌ها را بررسی کن.
۴. به گام ۱ برگرد و با امیدبخش‌ترین زیرمسئله ادامه بده.

احتمالاً به یاد می‌آورید که استراتژی عقب‌گرد (بخش ۳-۴) نیز بدون بررسی هر راه‌حل کاندیدای ممکن به جواب منجر شد. در عقب‌گرد تا جایی که می‌توانیم، بعد از بررسی مسیرها آن‌ها را حذف و زمانی که جواب به‌دست‌آمده موردقبول بود فرایند را متوقف می‌کنیم. در شاخه و حد، پیش‌بینی می‌کنیم که کدام مسیرها بدترین هستند و به‌منظور اجتناب از هدر رفتن انرژی با بررسی آن‌ها، آن‌ها را حذف می‌کنیم.

نتیجه‌گیری

حل مسئله عبارت از بررسی فضای راه‌حل‌های ممکن آن برای یافتن مورد صحیح است. راه‌های متعددی را برای این کار آموختیم. ساده‌ترین آن‌ها جستجوی فراگیر است: بررسی تمام عناصر در فضای جستجو به‌صورت تک‌به‌تک.

دیدیم که چگونه می‌توانیم به‌صورت منظم مسائل را به نمونه‌های کوچک‌تر تقسیم کرده و موجب بهبود کارایی شویم. تقسیم مسائل به‌صورت تکراری اغلب شامل مواجهه با زیرمسئله‌های مشابه است. در این حالات، استفاده از برنامه‌نویسی پویا برای اجتناب از تکرار محاسبات مشابه اهمیت دارد.

دیدیم که عقب‌گرد چگونه می‌تواند برخی از انواع جستجوی فراگیر را بهینه کند. برای مسائلی که حد بالا و پایین آن‌ها را می‌توان تخمین زد، دیدیم که چگونه می‌توانیم با شاخه و حد سریع‌تر به جواب برسیم؛ و ابتکار زمانی استفاده می‌شود که هزینه‌ی محاسبه‌ی راه‌حل بهینه قابل‌قبول نباشد.

تمام استراتژی‌هایی که دیدیم برای کار با داده‌ها طراحی‌شده‌اند. در ادامه معمول‌ترین راه‌های سازمان‌دهی داده‌ها در حافظه‌ی کامپیوتر و چگونگی تأثیر این فرایند را بر کارایی معمول‌ترین عملیات قابل انجام بر روی‌داده‌ها یاد خواهیم گرفت.

مراجع

- Algorithm Design, 1st Edition, by Kleinberg
  - Get it at https://code.energy/kleinberg
- Choosing Algorithm Design Strategy, by Shailendra Nigam
  - Get it at https://code.energy/nigam
- Dynamic programming, by Umesh V. Vazirani
  - Get it at https://code.energy/vazirani

فصل ۴

داده‌ها

برنامه‌نویسان خوب به ساختارهای داده و روابط بین
آن‌ها توجه می‌کنند.

– لینوس توروالدز[۱]

کنترل داده‌ها در علوم کامپیوتر اهمیت زیادی دارد: فرایندهای محاسباتی از این عملیات بر روی داده‌ها ساخته می‌شوند. ولی الگوریتم‌ها معمولاً *نحوه‌ی* انجام عملیاتی را که بر روی داده‌ها انجام می‌دهند مشخص نمی‌کنند. به‌عنوان نمونه، `merge` (بخش ۳-۱) بر یک کد بیرونی نامشخص برای ایجاد لیست اعداد، بررسی خالی بودن لیست و اضافه کردن عناصر به انتهای لیست‌ها تکیه دارد. الگوریتم `queens` (بخش ۳-۴) همین کار را می‌کند: این الگوریتم اهمیتی به نحوه‌ی انجام عملیات بر روی صفحه‌ی شطرنج نمی‌دهد، حتی به نحوه‌ی ذخیره‌سازی موقعیت‌ها در حافظه نیز اهمیت نمی‌دهد. این جزئیات در پشت مفهومی که به آن **انتزاع**[۲] می‌گوییم، پنهان هستند. در این فصل می‌آموزیم:

- ✨ **انواع داده‌ای مجرد** چگونه کد شما را تمیز نگه می‌دارند.
- 🛠️ **انتزاع‌های معمولی** که در جعبه‌ابزارتان نیاز دارید.
- 🏗️ راه‌های مختلف **ساختاردهی داده‌ها** در حافظه.

ولی قبل از شیرجه زدن در این مطالب، اجازه بدهید ابتدا معنی عبارات «انتزاع» و «نوع داده‌ای» را بفهمیم.

انتزاع

انتزاع به ما اجازه‌ی حذف جزئیات را می‌دهد. این روش ابزاری برای ساده‌سازی عملکرد اشیاء پیچیده است. به‌عنوان نمونه، در خودروها جزئیات پیچیده‌ی مکانیکی در پشت یک مجموعه‌ای از ابزارهای

---

[۱] Linus Torvalds (-1969): مهندس نرم‌افزار فنلاندی- آمریکایی که از بنیان‌گذاران هسته‌ی لینوکس و هم‌چنین نرم‌افزار گیت به شمار می‌رود. او هم‌اکنون مسئولیت هماهنگی پروژه هسته‌ی لینوکس را بر عهده دارد. م.

[۲] Abstractions

رانندگی مخفی‌شده‌اند، به‌طوری‌که هرکسی بدون درک مباحث مهندسی مربوطه، می‌تواند به‌سادگی رانندگی کند.

در نرم‌افزار، **انتزاع روالی**[۱] پیچیدگی‌های یک فرایند را در پشت فراخوانی یک رویه مخفی می‌کند. در الگوریتم trade (بخش ۳-۶) رویه‌های min و max چگونگی پیدا کردن مقادیر کمینه و بیشینه را مخفی کرده و موجب ساده‌تر شدن الگوریتم می‌شوند. با انتزاع بر روی سایر انتزاع‌ها، می‌توانیم پیمانه‌ها[۲] را بسازیم که به ما امکان انجام کارهای پیچیده را با رویه‌های واحد می‌دهند. مانند:

```
html ← fetch_source("https://code.energy")
```

در یک خط کد، کد منبع یک وب‌سایت را واکشی [دریافت] می‌کنیم، هرچند که عملیات درونی این کار بسیار پیچیده است[۳].

انتزاع‌های داده‌ای[۴] موضوع اصلی این فصل خواهند بود. این انتزاع‌ها جزئیات فرایندهای مدیریت داده‌ها را مخفی می‌کنند. ولی قبل از این‌که بتوانیم نحوه‌ی کار انتزاع داده‌ای را درک کنیم، باید دانسته‌های خود درباره‌ی انواع داده‌ای را تقویت کنیم.

انواع داده‌ای[۵]

ما انواع مختلفی از اتصالات را می‌شناسیم (مانند پیچ، مهره و میخ) و بر اساس کاری که می‌توانیم روی آن‌ها انجام دهیم از ابزار مناسب استفاده می‌کنیم (مانند آچار، چکش، پیچ‌گوشتی). به شکل مشابه، بین **انواع داده‌ای** بر اساس عملیاتی که بر روی داده‌ها انجام می‌دهند، تمایز قائل می‌شویم.

به‌عنوان نمونه، یک متغیر داده‌ای را که قابل‌تقسیم بر اساس مکان کاراکترها بوده، و می‌توان آن را به حروف بزرگ یا کوچک تبدیل کرد یا کاراکترهایی به انتهای آن افزود، از نوع رشته[۶] می‌دانیم. رشته‌ها متن‌ها را نمایش می‌دهند. یک متغیر داده‌ای که قابل معکوس سازی است و می‌توان عملگرهای XOR، OR و AND را بر روی آن اعمال کرد از نوع بولی است. متغیرهای بولی می‌توانند True یا False باشند. متغیرهایی که می‌توان آن‌ها را جمع، تقسیم و تفریق کرد از نوع عددی هستند.

---

۱ Procedural Abstraction

۲ یک پیمانه (Module) یا کتابخانه (Library) قطعه‌ای نرم‌افزار است که رویه‌های عمومی محاسباتی را ارائه می‌کند. این رویه‌ها را می‌توان به‌تناسب نیاز به سایر قطعات نرم‌افزار اضافه کرد.

۳ این کار نیازمند شناسایی نام دامنه، ایجاد یک سوکت شبکه TCP، انجام دست‌دهی (Handshaking) رمزنگاری SSL و بسیاری کار دیگر است.

۴ Data Abstractions

۵ Data Type

۶ String

هر نوع داده‌ای مرتبط با یک مجموعه‌ی خاص از رویه‌ها است. رویه‌هایی که بر روی متغیرهای ذخیره‌ی لیست کار می‌کنند متفاوت از رویه‌هایی هستند که بر روی متغیرهای ذخیره‌ی مجموعه‌ها کار می‌کنند. این رویه‌ها با رویه‌هایی که بر روی اعداد کار می‌کنند نیز متفاوت هستند.

٤-١- انواع داده‌ای مجرد

یک نوع داده‌ای مجرد[1] (ADT) شرحی از یک گروه از عملیات است که بر روی یک نوع داده‌ای مفروض عمل می‌کنند. این مفهوم یک واسط برای کار کردن با متغیرهای نگه‌دارنده‌ی نوع داده‌ای موردنظر تعریف کرده و تمام جزئیات چگونگی ذخیره‌سازی داده‌ها و انجام عملیات در حافظه را مخفی می‌کند.

زمانی که الگوریتم‌های ما نیاز به کار بر روی داده‌ها داشته باشند، به‌طور مستقیم به حافظه‌ی کامپیوتر دستور خواندن یا نوشتن نمی‌دهیم. ما از پیمانه‌های مدیریت داده‌ی بیرونی که رویه‌های تعریف‌شده در انواع ADT را ارائه می‌کنند، استفاده می‌کنیم.

به‌عنوان‌مثال، برای کار کردن با متغیرهایی که لیست‌ها را ذخیره می‌کنند به رویه‌هایی برای ایجاد و حذف لیست‌ها، رویه‌هایی برای دسترسی و حذف عنصر mام لیست، و یک رویه برای اضافه کردن یک عنصر جدید به لیست نیاز داریم. تعریف این رویه‌ها (نام آن‌ها و کاری که انجام می‌دهند) یک ADT لیست به وجود می‌آورد. ما می‌توانیم با تکیه‌بر این رویه‌ها با لیست‌ها کارکنیم. به این روش، هیچ‌گاه به‌صورت مستقیم حافظه‌ی کامپیوتر را دست‌کاری نمی‌کنیم.

امتیازات استفاده از ADTها

سادگی: ADTها موجب ساده‌تر شدن درک و اصلاح کد می‌شوند. با صرف‌نظر از جزئیات رویه‌های مدیریت داده‌ها، فقط بر تصویر اصلی تمرکز می‌کنید: فرایند حل مسئله‌ی الگوریتم.

انعطاف: راه‌های زیادی برای ساختاردهی داده‌ها در حافظه وجود دارند که منجر به پیمانه‌های مختلف مدیریت داده برای یک نوع داده‌ای مشابه می‌شوند. ما باید بهترین مورد را برای وضعیت موجود انتخاب کنیم. پیمانه‌هایی که یک ADT را پیاده‌سازی می‌کنند رویه‌های مشابهی ارائه می‌دهند. یعنی می‌توانیم روش ذخیره‌سازی داده‌ها و دست‌کاری آن‌ها را فقط با استفاده از یک پیمانه‌ی مدیریت داده‌ی متفاوت عوض کنیم. شبیه به خودروها: خودروهای الکتریکی و خودروهای گازسوز همگی واسط رانندگی یکسانی دارند. کسی که بتواند یکی را براند، به‌راحتی می‌تواند خودروی دیگری را نیز براند.

---

[1] Abstract Date Type

قابلیت استفاده‌ی مجدد: می‌توانیم از پیمانه‌های مدیریت داده‌ی یکسان در پروژه‌های نیازمند مدیریت داده با نوع داده‌ای مشابه استفاده کنیم. به‌عنوان نمونه، هر دو تابع power_set و recursive_power_set در فصل قبل با متغیرهای نمایشگر مجموعه‌ها کار می‌کنند. یعنی می‌توانیم از پیمانه‌ی Set در هر دو الگوریتم استفاده کنیم.

سازمان‌دهی: معمولاً باید با چندین نوع داده کارکنیم: اعداد، متن، مختصات جغرافیایی، تصاویر و غیره. برای سازمان‌دهی بهتر کدهایمان، پیمانه‌های مجزایی ایجاد می‌کنیم که هر یک مختص به یک نوع داده است. این امر **تفکیک دغدغه‌ها**[1] نامیده می‌شود: بخش‌هایی از کد که با جنبه‌ی منطقی یکسانی سروکار دارند باید در پیمانه‌ی جداگانه‌ی خود گروه‌بندی شوند. هنگامی‌که این بخش‌ها با سایر عملکردها ترکیب می‌شوند، حاصل را *کد اسپاگتی*[2] می‌نامیم.

راحتی: ما می‌توانیم یک پیمانه‌ی مدیریت داده را که توسط شخص دیگری کدنویسی شده است، دریافت کرده و یاد بگیریم که از رویه‌های تعریف‌شده توسط ADT آن استفاده کنیم. سپس می‌توانیم از این رویه‌ها برای کار با متغیرهایی از نوع داده‌ای جدید استفاده کنیم. درک نحوه‌ی عملکرد پیمانه‌ی مدیریت داده‌ها لازم نیست.

رفع خطاها: اگر از یک پیمانه‌ی مدیریت داده‌ی بدون خطا استفاده می‌کنید، کد شما عاری از خطاهای مربوط به مدیریت داده خواهد بود. اگر در یک پیمانه‌ی مدیریت داده خطایی پیدا کردید، یک‌بار رفع آن به معنی رفع فوری خطاهای بخش‌های تأثیر پذیرفته از آن خطا در کد است.

٤-٢- انتزاع‌های رایج

برای حل یک مسئله‌ی محاسباتی، درک نوع داده‌ای که روی آن کار می‌کنید و عملیاتی که نیاز دارید بر روی آن انجام دهید بسیار مهم است. تصمیم‌گیری در مورد ADT مورداستفاده به همان اندازه اهمیت دارد. در ادامه، انواع رایج داده‌های انتزاعی را که باید با آن‌ها آشنا باشید، معرفی می‌کنیم. این داده‌های انتزاعی در الگوریتم‌های بی‌شماری مورداستفاده قرار می‌گیرند، و حتی در بسیاری از زبان‌های برنامه‌نویسی به‌صورت درونی آورده شده‌اند.

---

[1] Separation of Concerns

[2] Spaghetti Code

انواع داده‌ای اولیه

انواع داده‌ای اولیه[۱] مواردی هستند که در زبان برنامه‌نویسی مورداستفاده‌ی شما پشتیبانی داخلی دارند. این انواع داده‌ای بدون پیمانه‌های خارجی کار می‌کنند و همیشه شامل اعداد صحیح، ممیز شناور[۲] و عملیات عمومی مرتبط با آن‌ها (جمع، تفریق، تقسیم) هستند. اغلب زبان‌ها دارای پشتیبانی داخلی برای ذخیره‌ی متن، مقادیر بولی و دیگر انواع داده‌ای ساده نیز در متغیرهای خود هستند.

پشته

یک توده‌ی کاغذ بر روی‌هم را تصور کنید. می‌توانید یک ورق را در بالای این توده قرار دهید یا ورق بالایی را بردارید. اولین برگه‌ای که اضافه می‌شود همیشه آخرین برگه‌ای است که حذف می‌شود. **پشته**[۳] زمانی استفاده می‌شود که انبوهی از عناصر داشته باشیم و فقط می‌خواهیم با عنصر بالایی آن‌ها کارکنیم. عنصر بالای توده همیشه جدیدترین موردی است که به آن اضافه‌شده است. یک پیاده‌سازی پشته باید حداقل این دو عملیات را ارائه دهد:

- **عملیات** `push(e)`: اضافه کردن عنصر e به بالای پشته.
- **عملیات** `pop()`: بازیابی و خارج کردن عنصر بالایی پشته.

پشته‌های «پیشرفته‌تر» ممکن است عملیات بیشتری را ارائه دهند: بررسی خالی بودن پشته، یا دریافت تعداد عناصر موجود در پشته در لحظه‌ی جاری.

پردازش داده‌ها به این روش را `LIFO`[۴] می‌نامند؛ ما فقط عناصر را از بالای پشته حذف می‌کنیم که این امر همیشه جدیدترین عنصر درج‌شده در پشته را شامل می‌شود. پشته نوع داده‌ای مهمی است که در بسیاری از الگوریتم‌ها استفاده می‌شود. برای پیاده‌سازی ویژگی Undo در ویرایشگر متن، هر تغییری باید وارد یک پشته شود. وقتی‌که می‌خواهید از این ویژگی استفاده کنید، ویرایشگر متن یک تغییر را از بالای پشته خارج و آن را معکوس می‌کند.

برای پیاده‌سازی عقب‌گرد (بخش ۳-۴) بدون الگوریتم‌های بازگشتی، باید دنباله‌ی گزینه‌های انتخاب‌شده را درون یک پشته نگهداری کنید. در زمان بررسی یک گره جدید، مرجعی از آن گره را به پشته می‌فرستیم. برای برگشتن، به‌سادگی و با استفاده از `pop()` مرجع موردنظر را از پشته خارج می‌کنیم.

---

[۱] `Primitive Data Types`

[۲] ممیزهای شناور راهی معمول برای نمایش اعداد دارای بخش اعشاری هستند.

[۳] `Stack`

[۴] `Last In First Out`

صف

صف[1] متضاد پشته است. از صف نیز برای ذخیره‌سازی و بازیابی عناصر استفاده می‌شود، ولی عنصر بازیابی شده همیشه موردی است که در *جلوی* صف قرار دارد، یعنی موردی که بیشتر از همه در صف مانده است. گیج نشوید، صف درست شبیه به صف‌های مردم منتظر در رستوران در زندگی واقعی است! عملیات ضروری صف عبارت‌اند از:

- **عملیات enqueue(e)**: اضافه کردن عنصر e به انتهای صف.
- **عملیات dequeue()**: خارج کردن عنصر از ابتدای صف.

صف از طریق سازمان‌دهی داده‌ها به‌صورت **FIFO**[2] عمل می‌کند، زیرا اولین (قدیمی‌ترین) عنصری که وارد صف شده است همیشه زودتر از بقیه خارج می‌شود.

صف‌ها در بسیاری از سناریوهای محاسباتی استفاده می‌شوند. اگر در حال پیاده‌سازی یک سرویس برخط فروش پیتزا هستید، باید سفارش‌های پیتزاها را در یک صف ذخیره کنید. به‌عنوان یک تمرین فکری، به این فکر کنید که اگر رستوران پیتزای شما طوری طراحی‌شده باشد که سفارش‌ها را با استفاده از پشته به‌جای صف ارائه دهد، چه چیزی متفاوت خواهد بود.

صف اولویت

صف اولویت[3] شبیه به صف است، با این تفاوت که عناصر درون صف دارای یک *اولویت* مشخص هستند. مردمی که در یک بیمارستان منتظر دریافت خدمات پزشکی هستند، مثالی از دنیای واقعی از یک صف اولویت هستند. موارد اضطراری اولویت بیشتری دارند و مستقیماً به ابتدای صف فرستاده می‌شوند، درحالی‌که موارد خفیف‌تر به انتهای صف اضافه می‌شوند. این موارد عملیات یک صف اولویت هستند:

- **عملیات enqueue(e,p)**: اضافه کردن عنصر e به صف بر اساس سطح اولویت p.
- **عملیات dequeue()**: خارج کردن عنصر از ابتدای صف و برگرداندن آن.

در یک کامپیوتر به‌طورمعمول پردازش‌های بسیاری در حال انجام هستند ولی فقط یک (یا چند) CPU برای اجرای آن‌ها وجود دارد. سیستم‌عامل این فرایندها را به‌صورت منتظر در یک صف اولویت سازمان‌دهی می‌کند. هر فرایند منتظر در صف دارای یک سطح اولویت است. سیستم‌عامل یک فرایند را

---

[1] Queue
[2] First In First Out
[3] Priority Queue

از صف خارج کرده و مدت کوتاهی به آن اجازه‌ی اجراشدن می‌دهد. بعدازآن، اگر فرایند به پایان نرسیده باشد، مجدداً وارد صف می‌شود. سیستم‌عامل به‌طور مداوم این کار را تکرار می‌کند.

برخی فرایندها حساسیت بیشتری به زمان داشته و بلافاصله زمان CPU را در اختیار می‌گیرند، بقیه در صف برای مدت طولانی‌تری منتظر می‌مانند. فرایندهایی که از صفحه‌کلید ورودی می‌گیرند معمولاً بیشترین اولویت را دارند. اگر صفحه‌کلید از کار بیافتد، ممکن است کاربر گمان کند کامپیوتر ازکارافتاده و اقدام به راه‌اندازی مجدد آن بنماید، که *اصلاً* خوب نیست.

لیست

در زمان مرتب‌سازی تعدادی عنصر، گاهی اوقات ممکن است به انعطاف بیشتری نیاز داشته باشید. به‌عنوان نمونه، ممکن است بخواهید آزادانه جای عناصر را عوض کنید، یا نیاز به دسترسی، درج و حذف عناصر در هر موقعیتی داشته باشد. در این موارد، **لیست**[1] به کار می‌آید. عملیاتی که به‌طورمعمول در ADT لیست تعریف می‌شوند عبارت‌اند از:

- **عملیات `insert(n,e)`**: درج عنصر e در موقعیت n.
- **عملیات `remove(n)`**: خارج کردن عنصر واقع در موقعیت n.
- **عملیات `get(n)`**: دریافت (خواندن) عنصر واقع در موقعیت n.
- **عملیات `sort()`**: مرتب‌سازی عناصر لیست.
- **عملیات `slice(start,end)`**: برگرداندن یک زیرلیست که با عناصری که از موقعیت `start` شروع‌شده و تا موقعیت `end` ادامه‌دارند.
- **عملیات `reverse()`**: معکوس کردن ترتیب عناصر لیست.

لیست‌ها یکی از پرکاربردترین انواع ADT هستند. به‌عنوان نمونه، اگر نیاز به ذخیره‌سازی لینک‌های فایل‌های با دسترسی زیاد در یک سیستم داشته باشید، یک لیست بسیار مناسب است: می‌توانید لینک‌ها را برای نمایش مرتب کرده و لینک‌های مربوط به فایل‌هایی را که اخیراً زیاد مورداستفاده قرار نگرفته‌اند حذف کنید.

اگر به انعطاف لیست نیاز نباشد، پشته یا صف باید ترجیح داده شوند. استفاده از یک ADT ساده‌تر این اطمینان را به وجود می‌آورد که داده‌ها دقیق‌تر و مطمئن‌تر مدیریت شوند. این کار هم‌چنین باعث ساده‌تر

---

[1] `List`

شدن درک کد می‌شود: این‌که بدانیم یک متغیر از نوع پشته است به ما کمک می‌کند نحوه‌ی جریان ورود و خروج داده‌ها را بهتر بفهمیم.

لیست مرتب‌شده

یک **لیست مرتب‌شده**[۱] زمانی مفید است که نیاز به نگهداری لیستی همیشه مرتب از عناصر داشته باشید. در این موارد، به‌جای پیدا کردن موقعیت درست قبل از درج عنصر در لیست (و مرتب‌سازی دستی آن به‌صورت دوره‌ای)، از یک لیست مرتب‌شده استفاده می‌کنیم. فرایند درج این نوع لیست باعث می‌شود که لیست همیشه مرتب باشد. هیچ‌کدام از عملیات این لیست اجازه‌ی جابجایی عناصر آن را نمی‌دهند؛ تضمین می‌شود که لیست همیشه مرتب است. تعداد عملیات لیست مرتب‌شده کمتر از لیست است:

- **عملیات `insert(e)`**: درج عنصر e در موقعیت مناسب در لیست.
- **عملیات `remove(n)`**: خارج کردن عنصر واقع در موقعیت n.
- **عملیات `get(n)`**: دریافت (خواندن) عنصر واقع در موقعیت n.

نگاشت

نگاشت[۲] (یا **واژه‌نامه**[۳]) برای ذخیره‌سازی نگاشت بین دو شیء استفاده می‌شود: یک شیء **کلید** و یک شیء **مقدار**. شما می‌توانید با دادن کلید در نگاشت پرس‌وجو کرده و مقدار متناظر آن را دریافت کنید. به‌عنوان نمونه، می‌توانید از یک نگاشت برای ذخیره کردن شماره شناسایی کاربران به‌عنوان کلید و نام کامل آن‌ها به‌عنوان مقدار استفاده کنید. به‌این‌ترتیب، با دادن شماره شناسایی یک کاربر، نگاشت نام مرتبط با وی را برمی‌گرداند. عملیات نگاشت عبارت‌اند از:

- **عملیات `set(key, value)`**: اضافه کردن یک نگاشت کلید- مقدار.
- **عملیات `delete(key)`**: حذف کلید key و مقدار متناظر با آن.
- **عملیات `get(key)`**: بازیابی مقدار متناظر با کلید key.

---

[۱] Sorted List
[۲] Map
[۳] Dictionary

مجموعه

مجموعه[۱] نمایشگر گروهی از عناصر متمایز بدون ترتیب است، شبیه به مجموعه‌های ریاضی که در پیوست III مطرح‌شده‌اند. زمانی که ترتیب عناصر مورداستفاده اهمیتی ندارد از مجموعه‌ها استفاده می‌شود، یا زمانی که می‌خواهید مطمئن شوید یک عنصر در گروه تکرار نمی‌شود. عملیات معمول مجموعه‌ها عبارت‌اند از:

- **عملیات add(e)**: اضافه کردن یک عنصر به مجموعه یا ایجاد یک خطا در صورت تکراری بودن عنصر [وجود آن در مجموعه].
- **عملیات list()**: فهرست کردن عناصر درون مجموعه.
- **عملیات delete(e)**: حذف یک عنصر از مجموعه.

شما به‌عنوان یک کدنویس با این ADTها می‌توانید با داده‌ها تعامل کنید، شبیه به یک راننده که از داشبورد خودرو استفاده می‌کند. حال اجازه بدهید سعی کنیم بفهمیم سیم‌ها چطور در پشت داشبورد ساختاردهی می‌شوند.

٤-۳- ساختارها

یک نوع داده‌ای مجرد فقط نحوه‌ی عملکرد یک نوع داده‌ای مفروض را شرح می‌دهد. این پدیده، فهرستی از عملیات را ارائه می‌دهد، ولی توضیح نمی‌دهد این عملیات *چگونه* انجام می‌شوند. به‌صورت متقابل، **ساختار داده**[۲] چگونگی سازمان‌دهی داده‌ها و دسترسی به آن‌ها را در حافظه‌ی کامپیوتر شرح می‌دهد. این مفاهیم راه‌هایی برای پیاده‌سازی ADTها در پیمانه‌های مدیریت داده‌ها فراهم می‌کنند.

راه‌های مختلفی برای پیاده‌سازی ADTها وجود دارند، زیرا ساختارهای داده‌ی متفاوتی وجود دارند. انتخاب یک پیاده‌سازی ADT که از بهترین ساختارهای داده بر اساس نیاز شما استفاده می‌کند، برای ساخت برنامه‌های کامپیوتری کارا اهمیت دارد. در ادامه، معمول‌ترین ساختارهای داده را بررسی کرده و نقاط قوت و ضعف آن‌ها را می‌آموزیم.

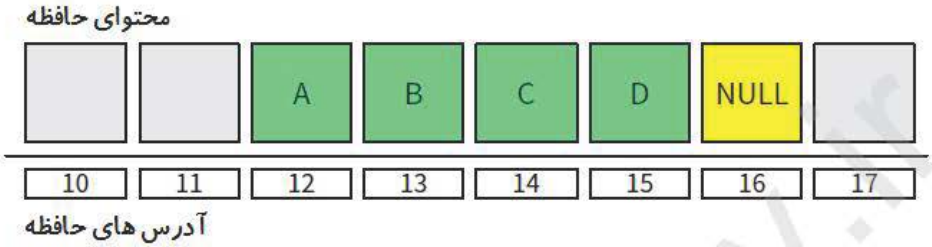
آرایه

آرایه[۳] ساده‌ترین راه ذخیره‌سازی تعدادی عنصر در حافظه‌ی کامپیوتر است. آرایه شامل تخصیص یک فضای پیوسته در حافظه‌ی کامپیوتر و نوشتن عناصرتان به‌صورت ترتیبی در آن فضا است. انتهای این دنباله با یک نشانه‌ی خاص NULL مشخص می‌شود.

---

۱ Set
۲ Data Structure
۳ Array

هر شیء در یک آرایه فضای یکسانی را از حافظه اشغال می‌کند. آرایه‌ای را تصور کنید که از خانه‌ی s حافظه شروع‌شده و هر عنصر آن b بایت اشغال می‌کند. عنصرnام در آرایه از طریق واکشی b بایت از خانه‌ی $s + (b \times n)$ در حافظه به دست می‌آید.



شکل ٤-١: یک آرایه در حافظه‌ی کامپیوتر

این راه به ما امکان می‌دهد به‌صورت آنی به هر عنصر آرایه دسترسی پیدا کنیم. آرایه به‌صورت ویژه برای پیاده‌سازی پشته مناسب است، ولی برای پیاده‌سازی لیست‌ها و صف‌ها نیز قابل‌استفاده است. کدنویسی آرایه‌ها ساده است و دارای امتیاز دسترسی آنی هستند. ولی معایبی نیز دارند.

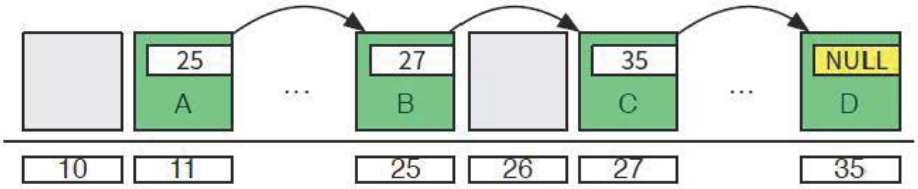
تخصیص اندازه‌های بزرگ حافظه‌ی متوالی در حافظه ممکن است غیرعملی باشد. اگر نیاز به بزرگ‌تر کردن آرایه داشته باشید، ممکن است فضای کافی خالی در مجاورت آن در حافظه وجود نداشته باشد. حذف یک عنصر از وسط آرایه می‌تواند مشکل‌ساز شود: باید *تمام* عناصر بعدی را یک گام به عقب منتقل کرده، یا عنصر حذف‌شده را در حافظه کامپیوتر به‌عنوان «مُرده[1]» علامت‌گذاری کنید. هیچ‌کدام از دو گزینه مطلوب نیستند. به شکل مشابه، اضافه کردن یک عنصر شما را مجبور می‌کند تمام عناصر بعداز آن را یک گام به سمت جلو منتقل کنید.

لیست پیوندی

در **لیست‌های پیوندی**[2] عناصر در زنجیره‌ای از سلول‌ها که قرار نیست در آدرس‌های متوالی حافظه قرار داشته باشند، ذخیره می‌شوند. حافظه‌ی هر سلول در زمان نیاز تخصیص می‌یابد. هر سلول یک اشاره‌گر[3] دارد که آدرس سلول بعدی را در زنجیره نشان می‌دهد. یک سلول با اشاره‌گر تهی نشان‌دهنده‌ی پایان زنجیره است.

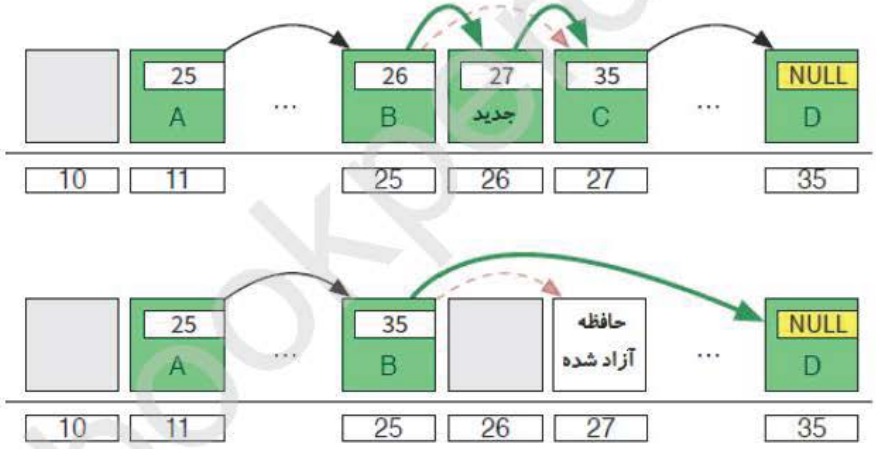
---

[1] Dead
[2] Link Lists
[3] Pointer



شکل ٤-۲: یک لیست پیوندی در حافظه‌ی کامپیوتر

از لیست‌های پیوندی می‌توان برای پیاده‌سازی پشته‌ها، لیست‌ها، و صف‌ها استفاده کرد. هیچ مشکلی در رابطه با رشد لیست وجود ندارد: هر سلول در هر خانه‌ای از حافظه قابل نگهداری است. می‌توانیم تا جایی که حافظه‌ی خالی وجود دارد، یک لیست بزرگ بسازیم. هم‌چنین درج کردن یک عنصر در وسط لیست یا حذف هر یک از عناصر با تغییر اشاره‌گر سلول بسیار ساده است:

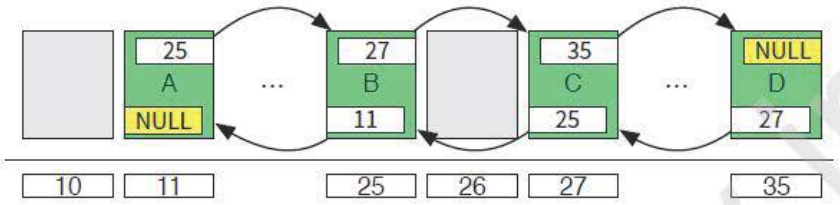


شکل ٤-۳: اضافه کردن یک عنصر بین B و C؛ حذف C

لیست‌های پیوندی نیز مشکلات خود را دارند: به‌صورت آنی نمی‌توانیم عنصر mام را بازیابی کنیم. زیرا باید جستجو را از سلول اول شروع کرده و از آن برای به دست آوردن آدرس سلول دوم استفاده کنیم، سپس به سلول دوم برویم و از اشاره‌گر آن برای رفتن به سلول بعدی استفاده کنیم و الی‌آخر تا اینکه به سلول mام برسیم. هم‌چنین اگر فقط آدرس یک سلول به ما داده شود، حذف آن یا حرکت رو به عقب ساده نیست. بدون هیچ اطلاعات دیگری، نمی‌توانیم آدرس سلول قبلی زنجیره را به دست آوریم.

لیست پیوندی دوطرفه

لیست پیوندی دوطرفه[1] یک لیست پیوندی با یک بخش اضافی است: سلول‌ها دو اشاره‌گر دارند: یکی به سلولی که قبل از آن آمده است، و دیگری به سلولی که بعداز آن می‌آید.



شکل ٤-٤: یک لیست پیوندی دوطرفه در حافظه‌ی کامپیوتر

این نوع لیست پیوندی فوایدی همانند لیست پیوندی معمولی دارد: نیاز به حجم زیادی از حافظه در تخصیص اولیه نیست، زیرا حافظه‌ی موردنیاز برای سلول‌های جدید بر اساس نیاز قابل تخصیص است. و یک اشاره‌گر اضافه به ما اجازه‌ی حرکت روبه‌جلو و رو به عقب بین سلول‌ها را می‌دهد. و اگر فقط آدرس یک سلول واحد را به ما بدهند، می‌توانیم آن را حذف کنیم.

همچنان هیچ راهی برای دسترسی آنی به عنصر mام نیست. به‌علاوه، ذخیره‌سازی دو اشاره‌گر در هر سلول موجب افزایش پیچیدگی کد و نیاز به حافظه‌ی بیشتر برای ذخیره‌سازی داده‌های ما می‌شود.

آرایه‌ها در مقایسه با لیست‌های پیوندی

زبان‌های برنامه‌نویسیِ غنی ازلحاظ ویژگی‌ها، اغلب با پیاده‌سازی‌هایی برای لیست، صف، پشته و سایر ADTها همراه هستند. این پیاده‌سازی‌ها معمولاً به یک ساختار داده‌ی پیش‌فرض تبدیل می‌شوند. برخی از این پیاده‌سازی‌ها حتی می‌توانند در زمان اجرا به‌صورت خودکار و بر اساس نحوه‌ی دسترسی به داده‌ها، بین ساختارهای داده جابجا شوند.

زمانی که کارایی اهمیت نداشته باشد، می‌توانیم به این پیاده‌سازی‌های عمومی ADT تکیه کرده و نگران ساختارهای داده نباشیم. ولی زمانی که قرار است کارایی بهینه باشد، یا در زمان کار با یک‌زبان سطح پایین‌تر که چنین ویژگی‌هایی ندارد، شما باید تصمیم بگیرید که از کدام ساختار داده استفاده کنید. عملیاتی که باید برروی داده‌ها انجام شود را تحلیل کرده و نوعی از پیاده‌سازی را انتخاب کنید که از یک ساختار داده‌ی مناسب استفاده می‌کند. لیست‌های پیوندی در شرایط زیر نسبت به آرایه‌ها ارجحیت دارند:

---

[1] Double Linked List

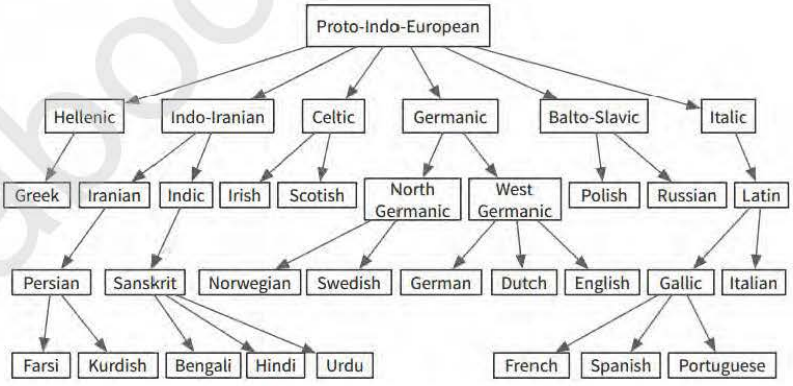
- به درج/حذف بسیار سریع در لیست نیاز داشته باشید.
- به دستیابی تصادفی و بدون ترتیب به عناصر نیاز نداشته باشید.
- عناصر را در وسط یک لیست درج یا حذف می‌کنید.
- نمی‌توانید اندازه‌ی دقیق لیست را تعیین کنید (لیست در زمان اجرا نیاز به افزایش یا کاهش اندازه داشته باشد).

آرایه‌ها در شرایط زیر نسبت لیست‌های پیوندی به ارجحیت دارند:

- به‌صورت مکرر به دستیابی تصادفی و بدون ترتیب به عناصر نیاز داشته باشید.
- به کارایی بسیار بالا برای دستیابی به عناصر نیاز داشته باشید.
- تعداد عناصر در زمان اجرا تغییر نمی‌کند، به‌طوری‌که می‌توانید به‌سادگی فضایی متوالی در حافظه‌ی کامپیوتر تخصیص دهید.

درخت

شبیه به لیست‌های پیوندی، **درخت**[1] از سلول‌های حافظه‌ای که الزاماً در حافظه‌ی فیزیکی متوالی نیستند برای ذخیره‌سازی استفاده می‌کند. سلول‌ها دارای اشاره‌گرهایی به سایر سلول‌ها هستند. برخلاف لیست‌های پیوندی، سلول‌ها و اشاره‌گرهای آن‌ها در یک زنجیره‌ی خطی سازمان‌دهی نشده‌اند، بلکه ساختاری شبیه به یک درخت‌دارند. درخت‌ها به‌صورت خاص برای داده‌های سلسله‌مراتبی مانند ساختار پوشه‌بندی فایل[2]، یا زنجیره‌ی دستورات در یک ارتش مناسب هستند.



شکل ٤-٥: درختی برای ریشه‌های زبان‌های هندو-اروپایی

---

[1] Tree

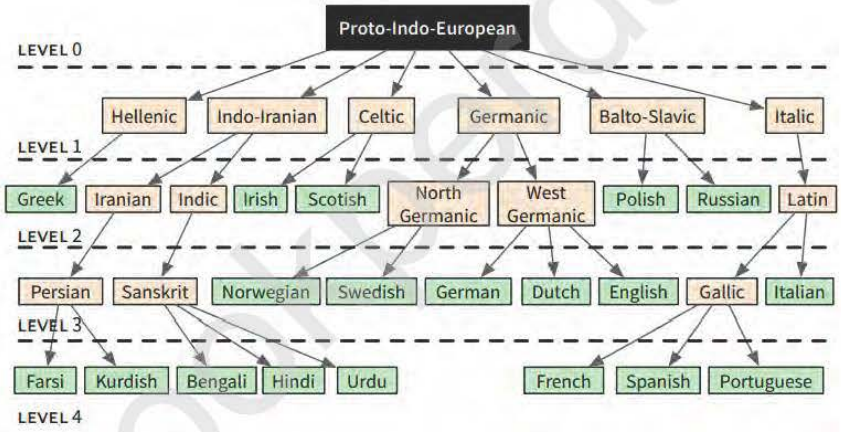
[2] File Directory

در مجموعه اصطلاحات درخت، یک سلول را **گره**[۱] و یک اشاره‌گر از یک سلول به سلول دیگر را یک **یال**[۲] می‌نامند. بالاترین گره یک درخت **گره ریشه**[۳] است: تنها گرهی که والد ندارد. به‌غیراز گره ریشه، گره‌های یک درخت باید دقیقاً یک والد داشته باشند[۴].

دو گره که دارای والد یکسان باشند همزاد[۵] هستند. والد، پدربزرگ و جد یک گره (و سایر گره‌ها در مسیر رو به بالا تا گره ریشه) نیاکان[۶] آن گره را می‌سازند. به همین ترتیب، فرزند، نوه و نتیجه‌ی یک گره (و سایر گره‌ها در مسیر رو به پایین در درخت) زادگان[۷] آن گره هستند.

گره‌هایی که فرزند ندارند را **گره برگ**[۸] می‌نامند (به برگ درختان در دنیای واقعی فکر کنید). و یک **مسیر**[۹] بین دو گره مجموعه‌ای از گره‌ها و یال‌ها است که از یک گره به گره دیگر می‌رسند.

عمق یا **سطح**[۱۰] یک گره برابر با اندازه‌ی مسیر آن تا گره ریشه است. **ارتفاع**[۱۱] یک درخت برابر با سطح عمیق‌ترین گره آن است. و درنهایت، مجموعه‌ای از درخت‌ها را یک **جنگل**[۱۲] می‌نامند.



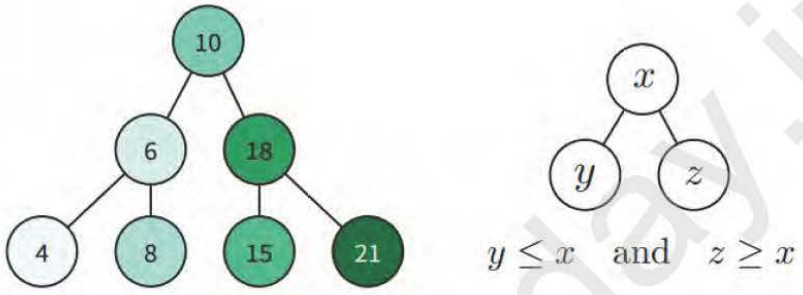
شکل ٤-٦: برگ‌های این درخت زبان‌های روزمره هستند

---

[۱] Node
[۲] Edge
[۳] Root Node
[۴] اگر یک گره از این قانون تخطی کند، بسیاری از الگوریتم‌های جستجو در درخت، به‌درستی کار نخواهند کرد.
[۵] Siblings
[۶] Ancestors
[۷] Descendants
[۸] Leaf Node
[۹] Path
[۱۰] Level
[۱۱] Height
[۱۲] Forest

درخت جستجوی دودویی

یک **درخت جستجوی دودویی**[1] یک نوع خاص درخت است که به‌صورت کارا می‌توان در آن آن جستجو انجام داد. گره‌ها در یک درخت جستجوی دودویی حداکثر دو فرزند دارند. و موقعیت گره‌ها بر اساس مقدار/ کلید آن‌ها تعیین می‌شود. گره‌های فرزند در سمت چپ والد باید کوچک‌تر از والد، گره‌های فرزند سمت راست باید بزرگ‌تر باشند.



شکل ٤-۷: یک درخت جستجوی دودویی نمونه

اگر این ویژگی در درخت رعایت شود، جستجوی یک گره با کلید / مقدار مفروض در درخت ساده است:

```
function find_node(binary_tree, value)
    node ← binary_tree.root_node
    while node:
        if node.value = value
            return node
        if value > node.value
            node ← node.right
        else
            node ← node.left
    return "NOT FOUND"
```

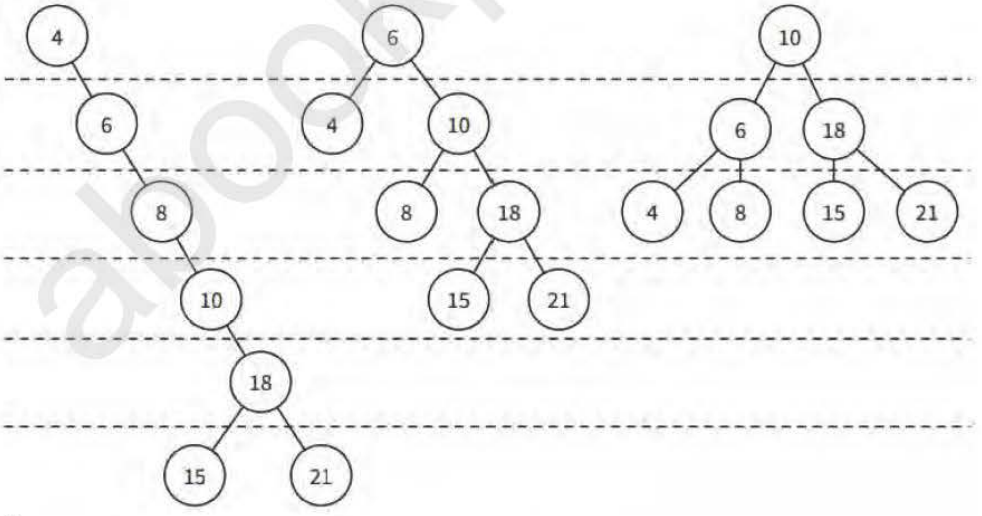
برای اضافه کردن یک عنصر، مقداری را که می‌خواهیم در درخت وارد کنیم جستجو می‌کنیم. آخرین گرهی را که در این جستجو به‌دست‌آمده است در نظر گرفته و اشاره‌گر سمت راست یا چپ آن را به گره جدید اشاره می‌دهیم.

---

[1] Binary Search Tree

```
function insert_node(binary_tree, new_node)
    node ← binary_tree.root_node
    while node:
        last_node ← node
        if new_node.value > node.value
            node ← node.right
        else
            node ← node.left
    if new_node.value > last_node.value
        last_node.right ← new_node
    else
        last_node.left ← new_node
```

متوازن ساختن درخت: اگر تعداد زیادی گره به یک درخت جستجوی دودویی وارد کنیم، ممکن است درنهایت به یک درخت با عمق زیاد برسیم، که بسیاری از گره‌های آن فقط یک فرزند دارند. به‌عنوان‌مثال، اگر گره‌های دارای مقدار/ کلید بزرگ‌تر را همیشه به سمت راست گره قبلی اضافه کنیم، در انتها به چیزی می‌رسیم که شبیه به یک لیست پیوندی است. ولی می‌توانیم گره‌ها را در درخت سازمان‌دهی مجدد کنیم به‌گونه‌ای که عمق آن کاهش یابد. یک درخت کاملاً متوازن حداقل عمق ممکن را دارد.



شکل ٤-٨: یک درخت جستجوی دودویی واحد با یک قالب بسیار نامتوازن، قالب نسبتاً متوازن و قالب کاملاً متوازن

اغلب عملیاتی که بر روی درخت‌ها انجام می‌شود، نیازمند دنبال کردن لینک‌های بین گره‌ها تا رسیدن به یک گره خاص هستند. هر چه عمق درخت بیشتر باشد، متوسط اندازه‌ی مسیر بین گره‌ها افزایش می‌یابد و زمان بیشتری برای دسترسی به حافظه نیاز خواهد بود. ازاین‌رو، کاهش عمق درخت بسیار مهم است. ایجاد یک درخت جستجوی دودویی کاملاً متوازن از یک لیست مرتب‌شده از گره‌ها به‌صورت زیر انجام می‌شود:

```
function build_balanced(nodes)
    if nodes is empty
        return NULL
    middle ← nodes.length/2
    left ← nodes.slice(0, middle - 1)
    right ← nodes.slice(middle + 1, nodes.length)
    balanced ← BinaryTree.new(root=nodes[middle])
    balanced.left ← build_balanced(left)
    balanced.right ← build_balanced(right)
    return balanced
```

یک درخت جستجوی دودویی با n گره را در نظر بگیرید. حداکثر عمق آن n است، که در این حالت کاملاً شبیه به یک لیست پیوندی است. حداقل عمق یک درخت کاملاً متوازن $\log_2 n$ است. پیچیدگی جستجوی یک عنصر در یک درخت جستجوی دودویی با عمق آن نسبت مستقیم دارد. در بدترین حالت، جستجو باید تا پایین‌ترین عمق جلو برود و تمام‌مسیر تا برگ‌های درخت را برای پیدا کردن عنصر موردنظر بررسی کند. ازاین‌رو، جستجو در یک درخت متوازن با n عنصر از نوع $O(\log n)$ است. به همین دلیل این ساختارداده اغلب برای پیاده‌سازی مجموعه‌ها (که نیازمند یافتن عناصر در صورت وجود آن‌ها هستند) و نگاشت‌ها (که نیازمند یافتن زوج‌های کلید – مقدار هستند) مورداستفاده قرار می‌گیرد.

بااین‌حال، متوازن ساختن درخت فرایند پرهزینه‌ای است، زیرا نیازمند مرتب‌سازی تمام گره‌ها است. متوازن ساختن مجدد درخت بعد از هر درج یا حذف می‌تواند سرعت این عملیات را به طرز چشمگیری کاهش دهد. درخت‌ها اغلب بعد از چند درج یا حذف متوازن‌سازی می‌شوند. ولی متوازن ساختن درخت به‌صورت گاه‌به‌گاه فقط برای درخت‌هایی که به‌ندرت تغییر می‌کنند یک استراتژی منطقی به شمار می‌رود.

برای مدیریت کارای درخت‌های دودویی با میزان تغییرات زیاد، **درخت‌های دودویی خودمتوازن**[۱] به وجود آمدند. رویه‌های این درخت‌ها برای درج یا حذف عناصر موجب ایجاد اطمینان از متوازن ماندن درخت می‌شوند. یک **درخت سرخ- سیاه**[۲] مثال معروفی از درخت‌های خودمتوازن است که گره‌ها را بر اساس استراتژی ایجاد توازن خود به‌صورت «سرخ» یا «سیاه» رنگ‌آمیزی می‌کند[۳]. درخت‌های سرخ – سیاه اغلب برای پیاده‌سازی نگاشت‌ها استفاده می‌شوند: بر روی نگاشت می‌توان به روشی بسیار کارا ویرایش سنگین انجام داد و یافتن هر کلید مفروض در نگاشت بسیار سریع خواهد بود که دلیل این امر خودمتوازن بودن درخت است.

درخت AVL نوع دیگری از درخت‌های خودمتوازن است. این نوع درخت به کمی زمان بیشتر برای درج و حذف عناصر نسبت به درخت سرخ – سیاه نیاز داشته ولی توازن بهتری دارد. این بدان معنی است که این درخت‌ها در بازیابی عناصر سریع‌تر از درخت‌های سرخ – سیاه هستند. درخت‌های AVL اغلب برای بهینه‌سازی کارایی در سناریوهای نیازمند اجرای زیاد عملیات خواندن مورداستفاده قرار می‌گیرند.

داده‌ها به‌صورت سنتی در دیسک‌های مغناطیسی ذخیره می‌شوند که داده‌ها را در قالب قطعات بزرگ می‌خوانند. در این موارد، **درخت B** که نمونه‌ای تعمیم‌یافته از درخت دودویی است، استفاده می‌شود. در درخت‌های B گره‌ها می‌توانند بیش از یک عنصر را ذخیره کرده و بیش از دو فرزند داشته باشند. این امر موجب کارا شدن آن‌ها در زمان کار با قطعات بزرگ داده‌ها می‌شود. همان‌گونه که به‌زودی خواهیم دید، درخت‌های B به‌صورت معمول در پایگاه‌های داده استفاده می‌شوند.

هرم دودویی

هرم دودویی[۴] نوع خاصی از درخت جستجوی دودویی است که در آن می‌توانیم به‌صورت آنی کوچک‌ترین (یا بزرگ‌ترین) عنصر را پیدا کنیم. این ساختار داده به‌طور خاص در پیاده‌سازی صف‌های اولویت کاربرد دارد. در هرم، دستیابی به مقدار کمینه (یا بیشینه) با هزینه‌ی $O(1)$ انجام می‌شود زیرا این مقدار همیشه ریشه‌ی درخت است. جستجو یا درج گره‌ها همچنان با هزینه‌ی $O(\log n)$ انجام می‌شود. قوانین تعیین موقعیت گره‌ها در هرم شبیه به درخت جستجوی دودویی است، به‌علاوه‌ی یک محدودیت اضافه: گره والد باید بزرگ‌تر (یا کوچک‌تر) از هر دو فرزند باشد.

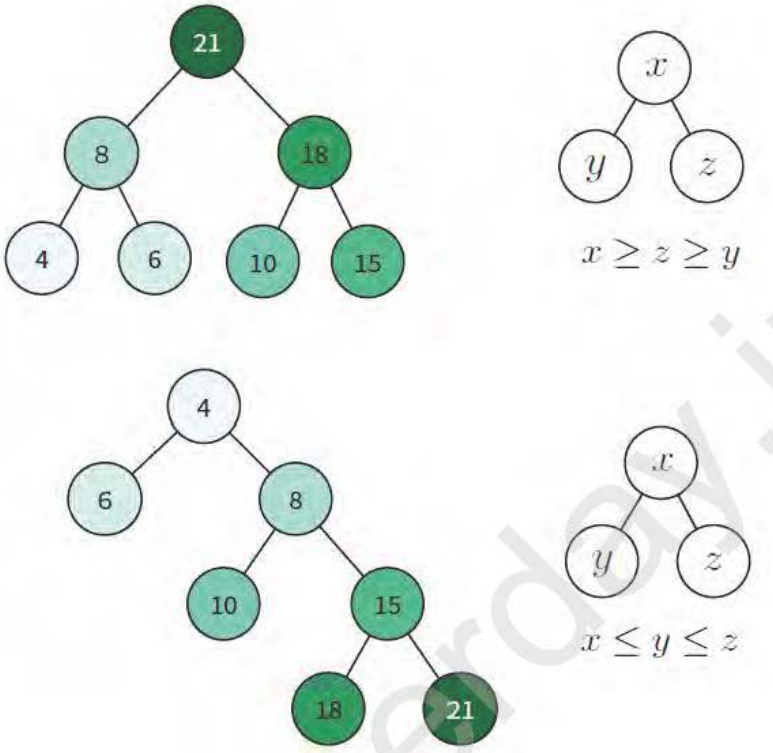
---

[۱] Self-Balancing Binary Tree

[۲] Red-Black Tree

[۳] استراتژی‌های خودمتوازن خارج از محدوده‌ی این کتاب هستند. اگر در این رابطه کنجکاو هستید، ویدیوهایی در مورد نحوه‌ی کار آن‌ها در اینترنت موجود است.

[۴] Binary Heap



شکل ٤-۹: گره‌های سازماندهی‌شده در قالب یک هرم دودویی بیشینه (بالا) و هرم کمینه (پایین)

به یاد داشته باشید، هر زمان که باید به‌صورت مکرر با عنصر بیشینه (یا کمینه) در یک مجموعه کار کنید، از هرم استفاده نمایید.

گراف

گراف[1] شبیه به درخت است. تفاوت در این است که هیچ گره فرزند یا والدی وجود ندارد، و ازاین‌رو، هیچ گره ریشه‌ای نیز وجود ندارد. داده‌ها به‌صورت آزادانه در قالب گره‌ها و یال‌ها سازمان‌دهی می‌شوند و هر گره می‌تواند چندین یال ورودی و خروجی داشته باشد.

این ساختار داده منعطف‌ترین نوع ساختار داده است و می‌توان از آن برای نمایش هر نوع داده‌ای استفاده کرد. به‌عنوان‌مثال، گراف‌ها برای نمایش یک شبکه‌ی اجتماعی بسیار مناسب هستند که در آن‌ها گره‌ها مردم و یال‌ها روابط دوستی را نشان می‌دهند.

---

[1] Graph

جدول درهم‌ریزی

جدول درهم‌ریزی[۱] یک ساختار داده است که اجازه‌ی پیدا کردن عناصر در زمان $O(1)$ را می‌دهد. جستجوی یک عنصر زمان ثابتی به طول می‌انجامد، خواه شما در حال جستجو در بین ۱۰ میلیون یا فقط ۱۰ عنصر باشید.

شبیه به آرایه، درهم‌ریزی نیازمند تخصیص پیشاپیش بخش بزرگی از حافظه به‌صورت پیوسته برای ذخیره‌سازی داده‌ها است. ولی برخلاف آرایه، عناصر به ترتیب ذخیره نمی‌شوند. جایی که یک عنصر اشغال می‌کند به‌صورت جادویی و به‌وسیله‌ی **تابع درهم‌ریزی** تعیین می‌شود. این تابع، یک تابع خاص است که داده‌ی موردنظر شما برای ذخیره‌سازی را به‌عنوان ورودی گرفته و عددی را که به‌ظاهر تصادفی است به‌عنوان خروجی برمی‌گرداند. این عدد جایی را در حافظه که عنصر ذخیره می‌شود تعیین می‌کند.

این کار به ما اجازه می‌دهد عناصر را به‌صورت آنی بازیابی کنیم. یک مقدار مفروض ابتدا به تابع درهم‌ریزی داده می‌شود. تابع محل دقیق ذخیره‌سازی آن را مشخص می‌کند. آن بخش از حافظه واکشی می‌شود. اگر عنصر در آن مکان ذخیره‌شده باشد، آن را خواهید یافت.

جدول‌های درهم‌ریزی یک مشکل‌دارند: گاهی اوقات تابع درهم‌ریزی یک مکان واحد حافظه را برای دو ورودی مختلف برمی‌گرداند. این اتفاق را **تصادم درهم‌ریزی**[۲] می‌نامند. زمانی که این اتفاق رخ بدهد، هر دو عنصر باید در یک آدرس واحد حافظه ذخیره شوند (به‌عنوان‌مثال، با یک لیست پیوندی که از همان آدرس شروع می‌شود). تصادم‌های درهم‌ریزی یک سربارِه‌ی اضافی برای CPU و حافظه به وجود می‌آورند، بنابراین سعی می‌کنیم از آن‌ها اجتناب کنیم.

یک تابع درهم‌ریزی مناسب برای ورودی‌های مختلف اعدادی را برمی‌گرداند که به‌ظاهر تصادفی هستند. ازاین‌رو، هر چه بازه‌ی مقادیر مجاز خروجی تابع درهم‌ریزی بزرگ‌تر باشد، جای بیشتری برای ذخیره‌سازی داده‌ها وجود دارد و احتمال وقوع تصادم درهم‌ریزی کمتر خواهد بود. بنابراین باید اطمینان حاصل کنیم که حداقل ۵۰٪ فضای در دسترس جدول درهم‌ریزی آزاد باشد. در غیر این صورت، تصادم به‌صورت مکرر رخ‌داده و موجب کاهش چشمگیر کارایی جدول درهم‌ریزی می‌شود.

جدول‌های درهم‌ریزی اغلب برای پیاده‌سازی نگاشت‌ها و مجموعه‌ها استفاده می‌شوند. این جدول‌ها اجازه‌ی درج و حذف سریع‌تر نسبت به ساختارهای داده‌ی مبتنی بر درخت می‌دهند. بااین‌حال، این جدول‌ها برای داشتن عملکرد مناسب نیاز به حجم زیادی از حافظه‌ی متوالی دارند.

---

۱ Hash Table

۲ Hash Collision

نتیجه‌گیری

یاد گرفتیم ساختارهای داده راه‌هایی متمرکز برای سازمان‌دهی داده‌ها در حافظه‌ی کامپیوتر ارائه می‌دهند. ساختارهای داده‌ی متفاوت نیازمند عملیات مختلف برای ذخیره‌سازی، حذف، جستجو و اجرا بر روی داده‌های ذخیره‌شده هستند. هیچ راهکار واحدی وجود ندارد: شما باید تصمیم بگیرید بر اساس وضعیتی که پیش رو دارید، از کدام ساختارداده استفاده کنید.

آموختیم که به‌جای استفاده‌ی مستقیم از ساختارهای داده در کد خود، بهتر است از انواع داده‌ای مجرد استفاده کنیم. این کار موجب مجزا سازی کد شما از جزئیات کار با داده‌ها می‌شود و به شما اجازه می‌دهد به‌راحتی و بدون نیاز به تغییر کد، ساختارهای داده‌ی موجود در برنامه خود را تغییر دهید.

سعی نکنید با ساخت ساختارهای داده‌ی ابتدایی و انواع داده‌ای مجرد از پایه، چرخ را دوباره اختراع کنید. مگر این‌که بخواهید این کار را برای تفریح، یا برای پژوهش انجام دهید. از کتابخانه‌های مدیریت داده‌ی توسعه‌یافته توسط افراد دیگر که به‌خوبی آزمایش و بررسی‌شده‌اند، استفاده کنید. بسیاری از زبان‌ها به‌صورت درونی از این ساختارها پشتیبانی می‌کنند.

مراجع

- Balancing a Binary Search Tree, by Stoimen
  - See it at https://code.energy/stoimen
- Cornell Lecture on Abstract Data Types and Data Structures,
  - See it at https://code.energy/cornell-adt
- IITKGP nodes on Abstract Data Types
  - See it at https://code.energy/iitkgp
- Search Tree Implementation by “Interactive Python”
  - See it at https://code.energy/python-tree

فصل ۵

الگوریتم‌ها

[کدنویسی] جذاب است، نه‌فقط به این دلیل که می‌تواند نتایج خوب اقتصادی یا علمی داشته باشد، بلکه چون این کار یک تجربه‌ی هنری شبیه به سرودن شعر یا ساختن یک قطعه‌ی موسیقی است.

– دانلد کنوث[1]

بشر به دنبال راه‌حل‌هایی برای مسائل سخت روزافزون است. بسیاری از اوقات شما با مسئله‌ای روبرو می‌شوید که افراد بسیار دیگری بر روی مسئله‌ای مشابه با آن کارکرده‌اند. شانس در این است آن‌ها الگوریتم‌های کارایی یافته باشند که شما بتوانید از آن‌ها به‌صورت آماده استفاده کنید. جستجوی الگوریتم‌های موجود باید همیشه اولین گام شما در حل مسائل باشد[2]. در این فصل الگوریتم‌های معروفی را بررسی خواهیم کرد که:

- لیست‌های بسیار بسیار بزرگ را به‌صورت کارا **مرتب** می‌کنند.
- به‌سرعت عنصر موردنیاز شما را **جستجو** می‌کنند.
- با **گراف‌ها** کار می‌کنند.
- از **تحقیق در عملیات** جنگ جهانی دوم برای بهینه‌سازی عملیات استفاده می‌کنند.

یاد خواهید گرفت مسائلی را که می‌توانید این راه‌حل‌ها را بر روی آن‌ها اعمال کنید شناسایی نمایید. این مسائل از انواع مختلفی هستند: مرتب‌سازی داده‌ها، جستجوی الگوها، مسیریابی و غیره. بسیاری از انواع الگوریتم‌ها مخصوص زمینه‌های خاص مطالعاتی هستند: پردازش تصویر، رمزنگاری، هوش

---

[1] Donald Knuth (-1938): دانشمند علوم کامپیوتر و استاد افتخاری دانشگاه استنفورد آمریکا. شهرت او به دلیل نگارش مجموعه کتاب‌های هنر برنامه‌نویسی کامپیوتر است. وی پایه‌گذار مبحث تحلیل الگوریتم‌ها است و سیستم حروف‌چینی تِک و سیستم متافونت یا فراقلم ازجمله کارهای وی به شمار می‌روند. م.

[2] پیدا کردن یک مسئله‌ی جدید که تاکنون بررسی نشده باشد به‌ندرت رخ می‌دهد. زمانی که پژوهشگران یک مسئله‌ی جدید پیدا می‌کنند، یک مقاله‌ی علمی در مورد آن می‌نویسند.

مصنوعی و غیره که ما نمی‌توانیم همه‌ی آن‌ها را در این کتاب پوشش دهیم[۱]. هم‌چنین، برخی از مهم‌ترین الگوریتم‌هایی را که هر کدنویس خوب باید با آن‌ها آشنا باشند نیز یاد خواهیم گرفت.

۵-۱- مرتب‌سازی

قبل از کامپیوترها، مرتب‌سازی داده‌ها مشکل بسیار بزرگی بود که انجام دستی آن زمان زیادی می‌گرفت. زمانی که شرکت ماشین جدول‌ساز[۲] (که بعدها به IBM تبدیل شد)، عملیات مرتب‌سازی را در دهه‌ی ۱۸۹۰ میلادی خودکارسازی کرد، جمع‌آوری داده‌های سرشماری آمریکا را سرعت بخشید.

الگوریتم‌های مرتب‌سازی زیادی وجود دارند. نمونه‌های ساده‌ی آن از نوع $O(n^2)$ هستند. **مرتب‌سازی انتخابی** (بخش ۲-۱) یکی از این الگوریتم‌ها است. این الگوریتمی است که مردم از آن برای مرتب‌سازی یک دسته کارت فیزیکی استفاده می‌کنند. مرتب‌سازی انتخابی به گروهی بزرگ از الگوریتم‌های دارای پیچیدگی درجه ۲ تعلق دارد. ما نوعاً از آن‌ها برای مرتب‌سازی پایگاه داده‌های کوچک شامل کمتر از هزار عنصر استفاده می‌کنیم. یک الگوریتم مرتب‌سازی قابل‌توجه درجه ۲، **مرتب‌سازی درجی**[۳] است. این الگوریتم در مرتب‌سازی مجموعه داده‌های تقریباً مرتب، حتی در صورت بزرگ بودن آن، بسیار کارا است:

```
function insertion_sort(list)
    for i ← 2 … list.length
        j ← i
        while j and list[j-1] > list[j]
            list.swap_items(j, j-1)
            j ← j - 1
```

این الگوریتم را با کاغذ و قلم و بر روی یک لیست تقریباً مرتب از اعداد اجرا کنید. برای ورودی‌هایی که تعداد کمی از عناصر آن‌ها مرتب نیستند، insertion_sort از نوع $O(n)$ است. در این حالت، این الگوریتم عملیات کمتری نسبت به سایر الگوریتم‌های مرتب‌سازی انجام می‌دهد.

برای مجموعه داده‌های بزرگ که مرتب نیستند، الگوریتم‌های $O(n^2)$ بسیار کند هستند. معروف‌ترین الگوریتم‌های مرتب‌سازی کارا **مرتب‌سازی ادغامی**[۴] (بخش ۳-۶) و **مرتب‌سازی**

---

[۱] در اینجا یک فهرست کامل‌تر وجود دارد: https://code.energy/algo-list

[۲] Tabulating Machine Company

[۳] Insertion Sort

[۴] Merge Sort

سریع[۱] به شمار می‌روند که هر دو از نوع $O(n \log n)$ هستند. نحوه‌ی عملکرد مرتب‌سازی سریع برای مرتب کردن یک دسته کارت به‌صورت زیر است:

۱. اگر دسته کمتر از ۴ کارت داشت، آن‌ها را به ترتیب صحیح قرار بده و فرایند به پایان رسیده است. در غیر این صورت به گام ۲ برو.
۲. به‌صورت تصادفی یکی از کارت‌های دسته کارت را انتخاب کن و به آن **محور**[۲] بگو.
۳. کارت‌های *بزرگ‌تر* از محور به یک دسته کارت جدید در سمت *راست* آن می‌روند؛ کارت‌های *کوچک‌تر* از محور به یک دسته کارت جدید در سمت چپ آن می‌روند.
۴. این رویه را برای هر یک از دودسته کارت جدید تکرار کن.
۵. دسته‌ی سمت چپ، محور و دسته‌ی سمت راست را به هم وصل کن تا دسته کارت مرتب‌شده به دست آید.

قاتی کردن یک دسته کارت و دنبال کردن این گام‌ها راه بسیار خوبی برای یادگیری مرتب‌سازی سریع است. این کار موجب تقویت درک شما از مفهوم بازگشت نیز خواهد شد.

حال شما آماده‌ی مواجهه با اغلب مسائلی هستید که شامل فرایند مرتب‌سازی هستند. ما در اینجا همه‌ی الگوریتم‌های مرتب‌سازی را پوشش نداده‌ایم، بنابراین به یاد داشته باشید که موارد بسیار دیگری نیز وجود دارند که هر یک مناسب سناریوهای مرتب‌سازی خاص هستند.

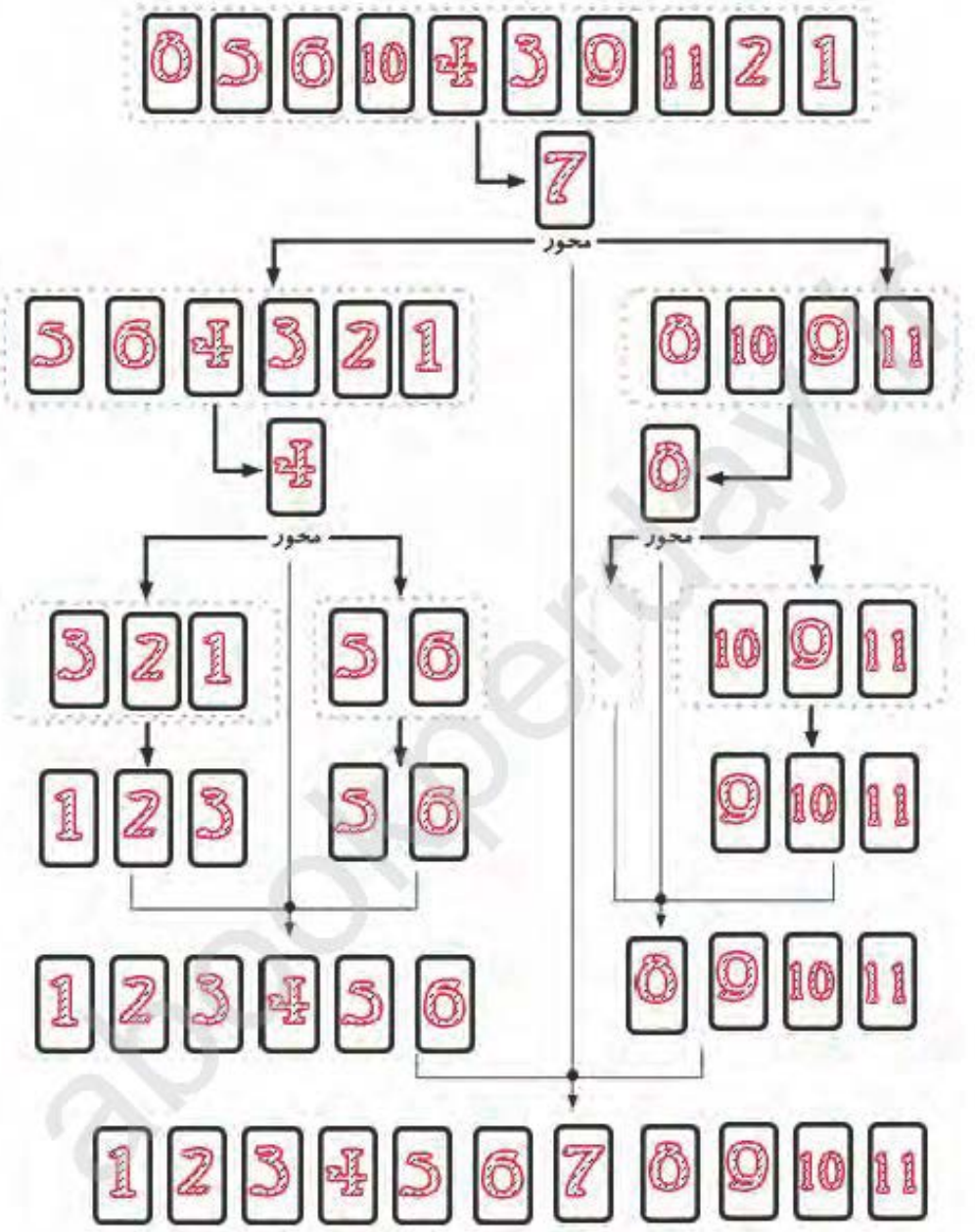
۵-۲- جستجو

یکی از انواع عملیات کلیدی در محاسبات، گشتن به دنبال اطلاعاتی خاص در حافظه‌ی کامپیوتر است. داشتن دانشی مناسب در مورد الگوریتم‌های جستجو الزامی است. ساده‌ترین الگوریتم جستجو، **جستجوی ترتیبی**[۳] است: تمام عناصر را یکی پس از دیگری بررسی کن تا زمانی که عنصر موردنظر را پیدا کنی؛ یا همه‌ی عناصر را بررسی کنی و بفهمی عنصر موردنظر آنجا نیست.

فهمیدن این‌که مرتب‌سازی ترتیبی $O(n)$ است، ساده است؛ به‌طوری‌که n تعداد کل عناصر در فضای جستجو است. ولی وقتی عناصری که در حال جستجو در آن‌ها هستیم، ساختار مناسب داشته باشند، راه‌های کاراتری نیز برای جستجو وجود دارند. در بخش ۴-۳ دیدیم که هزینه جستجو در داده‌های ساختاردهی شده در یک درخت جستجوی دودویی فقط $O(\log n)$ است.

---

۱ Quick Sort
۲ The Pivot
۳ Sequential Search



شکل ۵-۱: یک اجرای نمونه از مرتب‌سازی سریع

اگر عناصر شما در قالب یک آرایه‌ی مرتب‌شده ساختاردهی شده باشند، می‌توانیم جستجو در آن‌ها را نیز با استفاده از **جستجوی دودویی**[۱] و در زمان $O(\log n)$ انجام دهیم. این فرایند جستجو در هر گام نصف فضای جستجو را کنار می‌گذارد:

```
function binary_search(items, key):
    if not items
        return NULL
    i ← items.length / 2
    if key = items[i]
        return items[i]
    if key > items[i]
        sliced ← items.slice(i+1, items.length)
    else
        sliced ← items.slice(0, i-1)
    return binary_search(sliced, key)
```

هر گام binary_search تعداد ثابتی عملیات انجام می‌دهد و نصف ورودی را کنار می‌گذارد. این بدان معنی است که برای n عنصر، $\log_2 n$ گام کل ورودی را کاهش می‌دهد. به دلیل آن‌که هر گام شامل تعداد ثابتی عملیات است، بنابراین الگوریتم از نوع $O(\log n)$ است. شما می‌توانید در یک‌میلیون یا یک‌تریلیون عنصر با کارایی بالا جستجو کنید.

از این نیز می‌توان کاراتر عمل کرد. با مرتب‌سازی عناصر در یک جدول درهم‌ریزی (بخش ۴-۳) فقط کافی است مقدار درهم‌ریزی کلید موردنظر برای جستجو را حساب کنید. این مقدار، آدرس عنصر مربوط به کلید را به شما می‌دهد! زمان صرف شده برای پیدا کردن یک عنصر در صورت بزرگ شدن فضای جستجو، *افزایش نخواهد یافت*. اصلاً اهمیتی ندارد که جستجو را در بین یک‌میلیون، میلیارد یا تریلیون عنصر انجام بدهید، تعداد عملیات ثابت است، یعنی فرایند ازلحاظ زمان $O(1)$ است. کاملاً ثابت.

۵-۳- گراف‌ها

پیش‌تر دیدیم که گراف‌ها ساختارهای داده‌ای منعطفی هستند که از گره‌ها و یال‌ها برای ذخیره‌سازی اطلاعات استفاده می‌کنند. گراف‌ها به‌صورت گسترده برای نمایش داده‌هایی مانند شبکه‌های اجتماعی[۲] (گره‌ها افراد و یال‌ها روابط دوستی بین آن‌ها هستند)، شبکه‌های تلفن (گره‌ها تلفن‌ها و ایستگاه‌ها و یال‌ها ارتباطات بین آن‌ها هستند) و غیره استفاده می‌شوند.

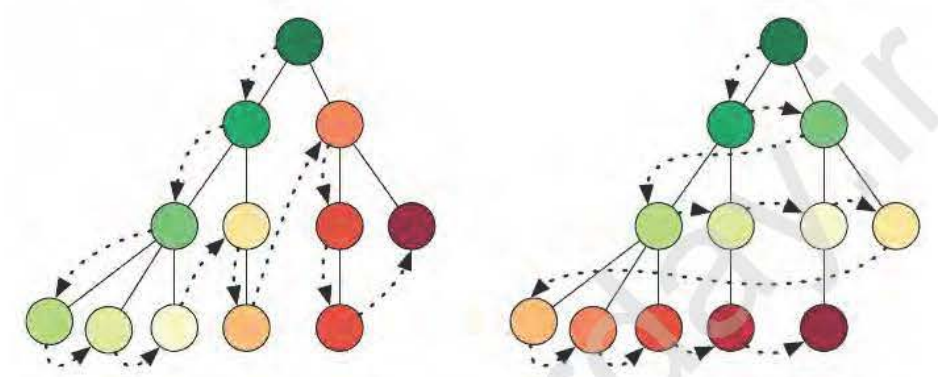
---

۱ Binary Search

۲ Social Networks

جستجو در گراف‌ها

یک گره در یک گراف را چطور پیدا می‌کنید؟ اگر ساختار گراف هیچ کمکی در جابجایی به شما نکند، باید تمام گره‌های گراف را تا زمان پیدا کردن گره موردنظر، بازدید کنید. برای این کار دو رویکرد وجود دارد: اول‌عمق[۱] و اول‌سطح[۲].



شکل ۵-۲: پیمایش گراف به‌صورت اول‌عمق (سمت چپ) و اول‌سطح (سمت راست)

در جستجوی گراف با استفاده از جستجوی اول‌عمق[۳]، امتداد یال‌ها را دنبال کرده و در عمق گراف حرکت می‌کنیم. وقتی به گره‌ای رسیدیم که هیچ یالی به یک گره جدید نداشت، به گره قبلی برگشته و فرایند را ادامه می‌دهیم. برای نگهداری مسیر پیمایش از یک پشته استفاده می‌کنیم و بعد از بازدید یک گره آن را وارد پشته کرده و در زمان نیاز به برگشت، آن را از پشته خارج می‌کنیم. استراتژی عقب‌گرد (بخش ۳-۴) در اجرای این راه‌حل مورداستفاده قرار می‌گیرد.

اگر رفتن به عمق گراف ایده‌ی خوبی نباشد، می‌توانید جستجوی اول‌سطح[۴] را امتحان کنید. این نوع جستجو گراف را سطح به سطح پیمایش می‌کند: ابتدا گره‌های مجاور گره شروع، سپس گره‌های مجاور مجاور آن و الی‌آخر. برای ردیابی گره‌هایی بازدید شده، از یک صف استفاده می‌کنیم. وقتی‌که یک گره را بازدید کردیم، فرزندان آن را وارد صف می‌کنیم، سپس گره بعدی را برای بازدید و ادامه دادن فرایند پیمایش، از صف خارج می‌کنیم.

---

۱ Depth-First

۲ Breadth-First

۳ Depth-First Search (DFS)

۴ Breadth-First Search (BFS)

```
function DFS(start_node, key)
    next_nodes ← Stack.new()
    seen_nodes ← Set.new()

    next_nodes.push(start_node)
    seen_nodes.add(start_node)

    while not next_nodes.empty
        node ← next_nodes.pop()
        if node.key = key:
            return node
        for n in node.connected_nodes
            if not n in seen_nodes
                next_nodes.push(n)
                seen_nodes.add(n)
    return NULL
```

```
function BFS(start_node, key)
    next_nodes ← Queue.new()
    seen_nodes ← Set.new()

    next_nodes.enqueue(start_node)
    seen_nodes.add(start_node)

    while not next_nodes.empty
        node ← next_nodes.dequeue()
        if node.key = key:
            return node
        for n in node.connected_nodes
            if not n in seen_nodes
                next_nodes.enqueue(n)
                seen_nodes.add(n)
    return NULL
```

توجه داشته باشید که BFS و DFS تنها در نحوه‌ی ذخیره‌سازی گره‌ی بعدی جستجو باهم تفاوت دارند: یکی از صف و دیگری از پشته استفاده می‌کند.

بنابراین کدام رویکرد را باید استفاده کنیم؟ پیاده‌سازی DFS ساده‌تر است و حافظه‌ی کمتری مصرف می‌کند: فقط کافی است گره‌ی والد گره‌ی جاری را ذخیره کنید. در BFS باید تمام گره‌های پیش روی فرایند جستجو را ذخیره کنید. اگر گرافی با یک‌میلیون گره داشته باشید، این روش عملی نیست.

زمانی که گمان می‌کنید سطح گره موردنظر در جستجو فاصله‌ی چندانی با گره شروع ندارد، معمولاً پرداختن هزینه‌ی اضافی BFS ارزشمند است زیرا به‌احتمال‌زیاد گره را زودتر پیدا می‌کنید. زمانی که نیاز به پیمایش همه‌ی گره‌های یک گراف داشته باشید، معمولاً بهتر است به دلیل سادگی پیاده‌سازی و حافظه‌ی مورداستفاده‌ی کمتر، از DFS استفاده شود.

شکل ۵-۳: جستجوی اول‌عمق یا DFS (گرفته‌شده از http://xkcd.com)

می‌توانید در شکل ۵-۳ ببینید که انتخاب یک روش پیمایش غلط می‌تواند نتایج وخیمی به دنبال داشته باشد.

رنگ‌آمیزی گراف

مسائل **رنگ‌آمیزی گراف**[1] زمانی به وجود می‌آیند که شما تعدادی ثابت «رنگ» (هر مجموعه‌ی دیگری از برچسب‌ها) دارید و باید به هر گره در گراف یک رنگ نسبت دهید. گره‌هایی که به‌وسیله‌ی

[1] Graph Coloring

یک یال به هم وصل شده‌اند، نباید دارای رنگ یکسان باشند. به‌عنوان نمونه، مسئله‌ی زیر را در نظر بگیرید:

> تداخل: به شما لیستی از برج‌های تلفن‌های همراه و نواحی تحت پوشش آن‌ها داده‌اند. برج‌های واقع در مجاورت هم باید به‌منظور اجتناب از تداخل، از فرکانس‌های مختلف استفاده کنند. در کل چهار فرکانس برای انتخاب وجود دارند. به هر برج کدام فرکانس را نسبت می‌دهید؟

اولین گام، مدل‌سازی این مسئله به‌وسیله‌ی یک گراف است. برج‌ها گره‌های گراف هستند. اگر دو برج به‌اندازه‌ی کافی به هم نزدیک هستند که تداخل ایجاد کنند، آن‌ها را با یک یال به هم وصل می‌کنیم. هر فرکانس نیز یک رنگ است.

چطور یک مجموعه انتساب فرکانس مناسب پیدا می‌کنید؟ آیا پیدا کردن یک راه‌حل با سه رنگ ممکن است؟ دو رنگ؟ پیدا کردن کمترین تعداد رنگ برای یک انتساب معتبر در حقیقت یک مسئله‌ی NP-کامل است و فقط الگوریتم‌های نمایی برای آن وجود دارند.

برای این مسئله نمی‌خواهیم یک الگوریتم ارائه کنیم. شما باید ازآنچه که تاکنون آموخته‌اید استفاده کرده و سعی کنید این مسئله را خودتان حل کنید. می‌توانید این کار را در UVA، یک داور برخط که راه‌حل پیشنهادی شما را آزمایش می‌کند، انجام دهید[1]. این برنامه کد شما را اجرا کرده و اگر درست باشد به شما اعلام خواهد کرد. در این صورت، کد را ازلحاظ زمان اجرا در مقایسه با کد سایر افراد نیز رتبه‌بندی می‌کند. شروع کنید! در مورد الگوریتم‌ها و استراتژی‌های حل این مسئله تحقیق کرده و آن‌ها را امتحان کنید. شما تاکنون فقط یک کتاب را خوانده‌اید. ارسال کد برای یک داور برخط به شما تجربه‌ی عملی موردنیاز را برای تبدیل‌شدن به یک کدنویس بزرگ می‌دهد.

مسیریابی

پیدا کردن کوتاه‌ترین مسیر بین گره‌ها معروف‌ترین مسئله‌ی گراف است. سیستم‌های ناوبری GPS در یک گراف از خیابان‌ها و چهارراه‌ها جستجو می‌کنند تا مسیر سفر شما را محاسبه کنند. برخی از آن‌ها حتی از داده‌های ترافیک برای افزایش وزن یال‌های نشان‌دهنده‌ی خیابان‌های شلوغ نیز استفاده می‌کنند.

برای پیدا کردن کوتاه‌ترین مسیر، استراتژی‌های BFS و DFS قابل‌استفاده ولی بد هستند. یک‌راه مشهور و خیلی کارا برای یافتن کوتاه‌ترین مسیر، **الگوریتم دایکسترا** است. همان‌گونه که BFS از یک

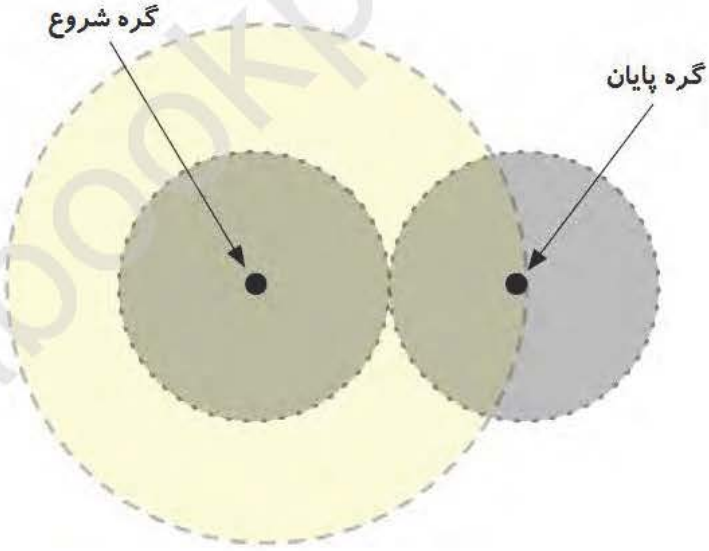
---

[1] مسئله‌ی رنگ‌آمیزی گراف UVA در: https://code.energy/uva-graph-coloring

صف کمکی برای نگهداری گره‌های موردنیاز برای پیمایش استفاده می‌کند، الگوریتم دایکسترا از یک صف اولویت استفاده می‌کند. اولویت یک گره برابر با وزن یال‌هایی است که آن را به گره شروع وصل می‌کنند. به این روش گره بعدی برای بازدید، همیشه نزدیک‌ترین گره به محل شروع است.

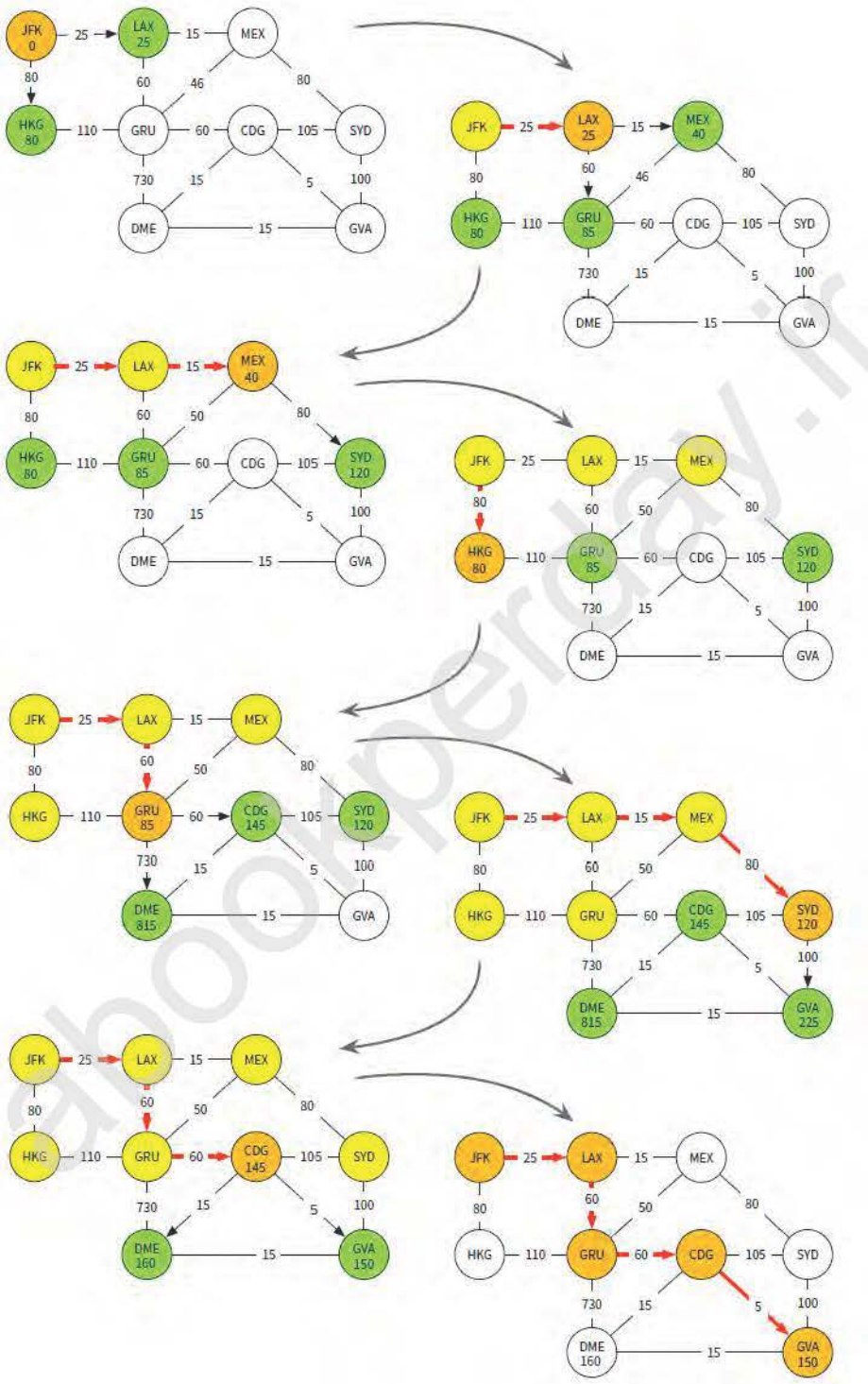
مواردی وجود دارند که الگوریتم دایکسترا بدون یافتن گره مقصد برای همیشه در یک چرخه می‌افتد. یک چرخه‌ی منفی می‌تواند فرآیند جستجو را فریب دهد تا به صورتی بی‌پایان آن را پیمایش کند. چرخه‌ی منفی مسیری در نمودار است که از یک گره شروع و در همان گره به پایان می‌رسد و مجموع وزن یال‌های مسیر، یک مقدار منفی است. اگر به دنبال یک مسیر کمینه در گرافی هستید که در آن یال‌هایی با وزن منفی وجود دارند، مراقب باشید.

اگر گراف موردنظر برای جستجو بزرگ باشد، چه؟ برای افزایش سرعت جستجو می‌توان از **جستجوی دوطرفه**[1] استفاده کرد. دو فرآیند جستجو به‌طور هم‌زمان اجرا می‌شوند: یکی از گره شروع، دیگری از گره مقصد. هنگامی که گره‌ای از یک ناحیه‌ی جستجو در ناحیه‌ی دیگر نیز وجود داشته باشد، فرایند متوقف می‌شود و مسیر به دست می‌آید. ناحیه‌ی جستجوی در جستجوی دوطرفه نصف جستجوی یک‌طرفه است. ببینید که ناحیه خاکستری (رنگ تیره) از ناحیه زرد (رنگ روشن) کوچک‌تر است:



شکل ۵-٤: ناحیه‌ی جستجو در جستجوی یک‌طرفه در مقایسه با جستجوی دوطرفه

[1] Bidirectional Search



شکل ۵-۵: پیدا کردن کوتاه‌ترین مسیر از JFK به GVA با دایکسترا

رتبه‌بندی صفحه

آیا تابه‌حال فکر کرده‌اید که گوگل چگونه می‌تواند میلیاردها صفحه‌ی وب را تجزیه‌وتحلیل کند و مرتبط‌ترین آن‌ها را به شما ارائه دهد؟ الگوریتم‌های زیادی در این کار دخیل هستند، اما مهم‌ترین آن‌ها **الگوریتم رتبه‌بندی صفحه**[۱] است.

سرگئی برین[۲] و لری پیج[۳] قبل از تأسیس گوگل، در رشته‌ی علوم کامپیوتر در دانشگاه استنفورد مشغول به تحصیل بودند و در مورد الگوریتم‌های گراف تحقیق می‌کردند. آن‌ها وب را در قالب یک گراف مدل کردند: صفحات وب گره‌ها و پیوندهای بین صفحات وب یال‌ها هستند.

آن‌ها متوجه شدند اگر یک صفحه‌ی وب لینک‌های زیادی را از صفحات مهم دیگر دریافت کند، خود نیز باید مهم باشد. آن‌ها پس‌ازاین ایده الگوریتم PageRank را ایجاد کردند. الگوریتم در چندین دور اجرا می‌شود. هر صفحه‌ی وب در گراف با تعداد مساوی «نقطه» شروع می‌شود. پس از هر دور، هر صفحه امتیاز خود را در صفحاتی که به آن‌ها لینک دارد، توزیع می‌کند. این روند تا زمانی که تمام امتیازات ثابت شوند تکرار می‌شود. امتیاز تثبیت‌شده‌ی هر صفحه، PageRank آن نامیده می‌شود. با استفاده از الگوریتم PageRank برای تعیین اهمیت صفحه‌ی وب، گوگل به‌سرعت بر سایر موتورهای جستجو برتری یافت. الگوریتم PageRank را می‌توان بر روی انواع دیگر گراف‌ها نیز اعمال کرد. به‌عنوان‌مثال، می‌توانیم کاربران شبکه توییتر[۴] را در یک گراف مدل‌سازی کنیم و سپس PageRank هر کاربر را محاسبه کنیم. به نظر شما کاربرانی که رتبه صفحه‌ی بالاتری دارند احتمالاً افراد مهمی هستند؟

۵-٤- تحقیق در عملیات

در طول جنگ جهانی دوم[۵]، ارتش بریتانیا باید بهترین تصمیمات استراتژیک را برای بهینه‌سازی تأثیر عملیات خود اتخاذ می‌کرد. آن‌ها ابزارهای تحلیلی بسیاری برای کشف بهترین راه برای هماهنگ کردن عملیات نظامی خود ایجاد کردند.

---

[۱] PageRank Algorithm

[۲] Sergey Brin (- 1973): دانشمند کامپیوتر و کارآفرین آمریکایی با اصالت روس که یکی از هم‌بنیان‌گذارن گوگل به شمار می‌رود. وی دانش‌آموخته‌ی دانشگاه‌های مریلند و استنفورد است. م.

[۳] Larry Page (- 1973): کارآفرین و مخترع آمریکایی و یکی از هم‌بنیان‌گذاران گوگل است. وی دانش‌آموخته‌ی دانشگاه‌های میشیگان و استنفورد است. م.

[۴] Twitter

[۵] World War II: جنگ جهانی یا بین‌الملل دوم که از سال ۱۹۳۹ میلادی تا ۱۹۴۵ میلادی به طول انجامید و به کشته شدن بیش از ۸۰ میلیون نفر منجر شد. م.

این تجربیات، **تحقیق در عملیات**[۱] نام‌گذاری شد. این سیستم رادار هشدار اولیه‌ی بریتانیا را بهبود بخشید و به بریتانیا کمک کرد تا نیروی انسانی و منابع خود را بهتر مدیریت کند. در طول جنگ، صدها انگلیسی در تحقیق در عملیات شرکت داشتند. پس‌ازآن، این ایده‌های جدید برای بهینه‌سازی فرایندها در مشاغل و صنایع به کار گرفته شدند. تحقیق در عملیات شامل تعریف یک هدف برای *بیشینه‌سازی* یا *کمینه‌سازی* است. این پدیده می‌تواند در بیشینه‌سازی اهدافی مانند بازده، سود یا عملکرد و کمینه‌سازی اهدافی مانند میزان ضرر، ریسک یا هزینه کمک کند.

به‌عنوان‌مثال، تحقیق در عملیات توسط شرکت‌های هواپیمایی برای بهینه‌سازی برنامه‌های پرواز استفاده می‌شود. دقت در برنامه‌ریزی نیروی کار و تجهیزات می‌تواند میلیون‌ها دلار صرفه‌جویی ایجاد کند. نمونه دیگر، استفاده از تحقیق در عملیات در پالایشگاه‌های نفت است که یافتن نسبت‌های بهینه‌ی مواد خام در یک ترکیب را می‌توان به‌عنوان یک مسئله‌ی تحقیق در عملیات در نظر گرفت.

مسائل بهینه‌سازی خطی

مسائلی را که در آن‌ها بتوان اهداف و قیدها را با استفاده از معادلات خطی مدل‌سازی کرد، **مسائل بهینه‌سازی خطی**[۲] می‌نامند[۳].

> دکوراسیون هوشمند: دفتر شما نیاز به کابینت دارد. هزینه‌ی کابینت از نوع X برابر با ۱۰ دلار است و ۶ مترمربع مساحت مقطع داشته و ۸ مترمکعب فضا اشغال می‌کند. هزینه‌ی کابینت از نوع Y برابر با ۲۰ دلار است و ۸ مترمربع مساحت مقطع داشته و ۱۲ مترمکعب فضا اشغال می‌کند. شما ۱۴۰ دلار بودجه دارید تا ۷۲ مترمربع در دفتر خود را کابینت کنید. کدام نوع کابینت را باید بخرید تا حداکثر فضای ذخیره‌سازی را داشته باشید؟

ابتدا، متغیرهای مسئله را شناسایی می‌کنیم. ما به دنبال تعداد کابینت‌ها از هر نوع، برای خرید هستیم؛ بنابراین:

---

[۱] Operations Research

[۲] Linear Optimization Problems

[۳] به‌صورت رسمی، چندجمله‌ای‌های با درجه‌ی ۱ را معادلات خطی می‌نامند. این معادلات هیچ نمونه‌ای از توان ۲ (یا هر توان دیگری) ندارند و متغیرهای آن‌ها تنها در ضرایب ثابت ضرب می‌شوند.

- متغیر x: تعداد کابینت‌های خریداری‌شده از نوع X.
- متغیر y: تعداد کابینت‌های خریداری‌شده از نوع Y.

ما می‌خواهیم فضای ذخیره‌سازی را بیشینه کنیم. اجازه بدهید فضای ذخیره‌سازی را z نامیده و آن را در قالب تابعی از x و y مدل‌سازی کنیم:

- $z = 8x + 12y$

حال باید مقادیری را برای x و y انتخاب کنیم که حداکثر مقدار ممکن z را ایجاد کنند. این مقادیر باید به‌گونه‌ای انتخاب شوند که قیدهای مرتبط با بودجه (کمتر از ۱۴۰ دلار) و فضا (کمتر از ۷۲ مترمربع) را برآورده کنند. اجازه بدهید این قیدها را مدل‌سازی کنیم:

- قید بودجه: $10x + 20y \leq 140$.
- قید فضا: $6x + 8y \leq 72$.
- چون نمی‌توانیم تعداد منفی کابینت خریداری کنیم: $x \geq 0$ و $y \geq 0$.

چگونه این مسئله را حل می‌کنید؟ صرف خرید هر چه بیشتر کابینت با بهترین نسبت فضای ذخیره‌سازی به فضا راه‌حل این مسئله نیست، زیرا فضای محدودی در دفتر برای جا دادن کابینت‌ها وجود دارد. شاید با جستجوی فراگیر عمل کنید: برنامه‌ای بنویسید که z را برای تمام مقادیر ممکن x و y محاسبه کند، و زوجی که بهترین z را تولید می‌کنند به دست آورد. این روال برای مسائل ساده جواب می‌دهد، اما انجام آن با تعداد زیادی متغیر غیرممکن است.

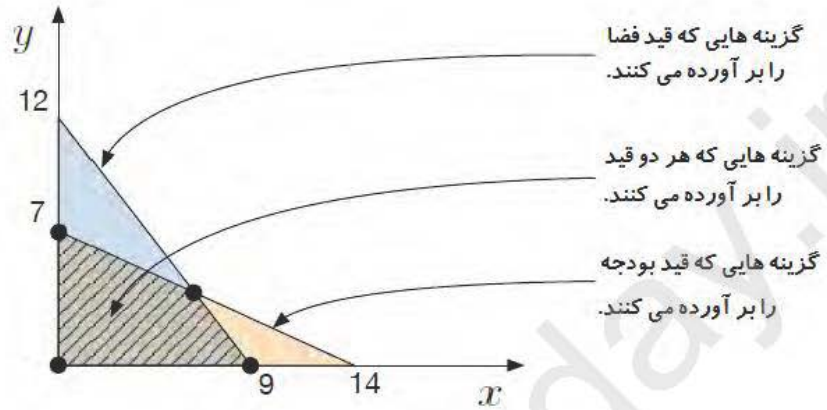
به نظر می‌رسد که برای حل مسائل بهینه‌سازی خطی مانند این مسئله، نیازی به کدنویسی نیست. شما فقط باید از ابزار مناسب برای این کار استفاده کنید: **روش غیرمرکب**[1]. روش غیرمرکب مسائل بهینه‌سازی خطی را به‌صورت بسیار کارآمد حل می‌کند. این روش از دهه‌ی ۱۹۶۰ به صنایع کمک می‌کند تا مسائل پیچیده را حل کنند. هنگامی‌که باید یک مسئله‌ی بهینه‌سازی خطی را حل کنید، چرخ را دوباره اختراع نکنید: یک حل‌کننده غیرمرکب آماده برای استفاده انتخاب کنید.

حل‌کننده‌های غیرمرکب فقط از شما می‌خواهند تابعی را که باید بیشینه (کمینه) شود، به همراه معادلاتی که قیدهای شما را مدل می‌کنند، وارد کنید. حل‌کننده بقیه کارها را انجام می‌دهد. در این مسئله، انتخابی برای x و y که z را بیشینه می‌کند $x = 8$ و $y = 3$ است.

---

[1] Simplex Method

روش غیرمرکب از طریق کاوش هوشمند در فضای راه‌حل‌های قابل‌قبول کار می‌کند. برای درک نحوه عملکرد روش غیرمرکب، اجازه بدهید تمام مقادیر ممکن برای x و y را در یک صفحه‌ی دوبعدی نشان دهیم. بودجه و قید فضای دفتر به‌صورت خطوط نشان داده می‌شوند:



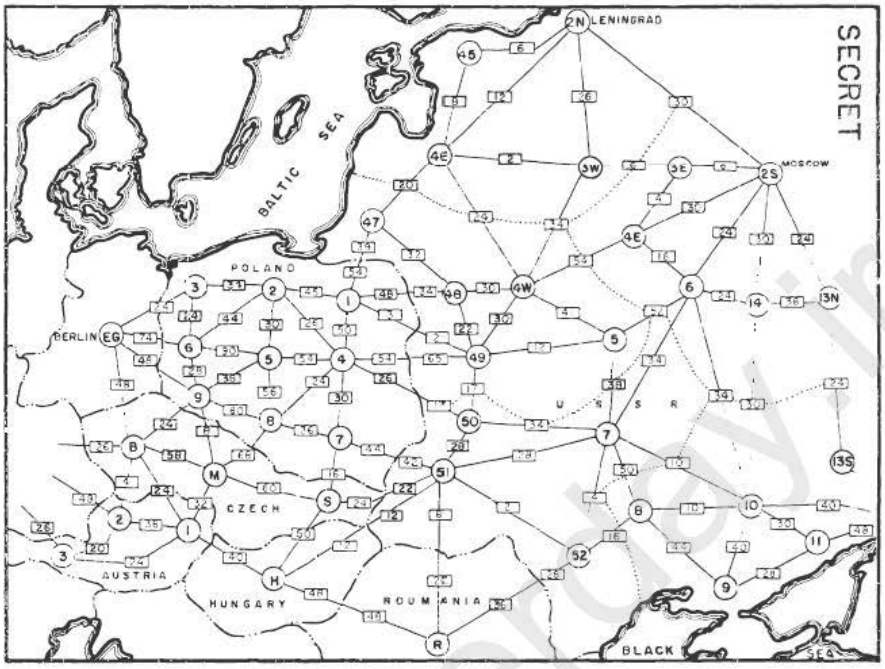
شکل ۵-۶: مقادیر x و y که قیدهای مسئله را برآورده می‌کنند

توجه داشته باشید که فضای شامل تمام راه‌حل‌های ممکن، یک ناحیه‌ی بسته در نمودار است. ثابت‌شده است که راه‌حل بهینه برای یک مسئله‌ی خطی باید گوشه‌ای از این ناحیه‌ی بسته باشد، یعنی نقطه‌ای که در آن خطوط نشان‌دهنده‌ی قیدها باهم تلاقی می‌کنند. روش غیرمرکب این نقاط گوشه را بررسی کرده و موردی را انتخاب می‌کند که z را بهینه می‌کند. مصورسازی این فرایند در مسائل بهینه‌سازی خطی که بیش از دو متغیر دارند آسان نیست، اما اصل ریاضی به همین صورت عمل می‌کند.

مسائل جریان شبکه

بسیاری از مسائل مربوط به شبکه‌ها و جریان‌ها را می‌توان برحسب معادلات خطی فرموله کرد و بنابراین این مسائل به‌راحتی با روش غیرمرکب قابل‌حل هستند. به‌عنوان نمونه، در طول جنگ سرد، ارتش ایالات‌متحده مسیرهای بالقوه تأمین ریلی را که شوروی می‌توانست در اروپای شرقی استفاده کند، ترسیم کرد (شکل ۵-۷).

> شبکه‌ی تأمین: یک شبکه‌ی ریلی با خطوطی که شهرها را به هم متصل می‌کنند نشان داده می‌شود. هر خط دارای حداکثر ظرفیت مشخصی است: بزرگ‌ترین جریان قابل‌حمل کالا در آن خط به‌صورت روزانه. چه مقدار از منابع را می‌توان از یک شهر تولیدکننده‌ی مفروض به یک شهر مصرف‌کننده معین منتقل کرد؟



شکل ۵-۷: گزارش‌های نظامی از طبقه‌بندی خارج‌شده از شبکه‌ی ریلی شوروی از سال ۱۹۵۵ که نشان‌دهنده‌ی ظرفیت حمل‌ونقل خطوط ریلی آن هستند

برای مدل‌سازی مسئله با معادلات خطی، هر خط ریلی به متغیری تبدیل می‌شود که نشان‌دهنده مقدار کالایی است که باید در خط جریان داشته باشد. قیدها عبارت‌اند از: هیچ خط ریلی نمی‌تواند بیش از ظرفیت خود را انتقال دهد، جریان ورودی کالا باید با جریان خروجی کالا در همه‌ی شهرها به‌جز شهرهای تولیدکننده و مصرف‌کننده برابر باشد. سپس مقادیری را برای متغیرهای خود انتخاب می‌کنیم که کالاهای ورودی را در شهر دریافت‌کننده بیشینه می‌سازد.

ما قرار نیست تبدیل به شکل خطی را با جزئیات توضیح دهیم. هدف ما در اینجا فقط این است به شما اطلاع دهیم که بسیاری از مشکلات بهینه‌سازی مربوط به گراف‌ها، هزینه و جریان‌ها را می‌توان به‌راحتی با پیاده‌سازی‌های غیرمرکب موجود حل کرد. بسیاری از اسناد مفید برخط در این رابطه را می‌توان یافت. چشمان خود را باز نگه‌دارید و به یاد داشته باشید: زمان را برای اختراع مجدد چرخ از دست ندهید.

نتیجه‌گیری

نشان دادیم چندین الگوریتم و روش شناخته‌شده برای حل انواع مسائل وجود دارند. اولین قدمی که هنگام حل یک مسئله باید بردارید، جستجوی همیشگی برای یافتن الگوریتم‌ها و روش‌های موجود است.

الگوریتم‌های مهم بسیاری وجود دارند که ما اشاره‌ای به آن‌ها نکردیم. به‌عنوان‌مثال، الگوریتم‌های جستجوی پیشرفته‌تری نسبت به دایکسترا (مانند A*)، الگوریتم‌هایی برای تخمین شباهت دو کلمه (Levenshtein Edit Distance)، الگوریتم‌های یادگیری ماشین و بسیاری موارد دیگر داریم.

مراجع

- Introduction to Algorithms, by Cormen
  - Get it at https://code.energy/cormen
- Algorithms, by Sedgewick
  - Get it at https://code.energy/sedgewick
- Simple Linear Programming Model,by Katie Pease
  - Get it at https://code.energy/katie

فصل ۶

پایگاه داده

اگرچه من برای کارهایم در مورد پایگاه داده معروف شده‌ام، مهارت‌های اصلی من همانند مهارت‌های یک معمار هستند: تحلیل نیازمندی‌ها و ساختن راه‌حل‌های ساده ولی زیبا.

– چارلز بکمن[۱]

مدیریت مجموعه‌های بزرگ داده‌ها در سیستم‌های کامپیوتری سخت، ولی اغلب حیاتی است. زیست‌شناسان به ذخیره‌سازی و بازیابی دنباله‌های DNA مرتبط با ساختارهای پروتئینی می‌پردازند. فیس‌بوک محتوای تولیدشده توسط میلیاردها انسان را مدیریت می‌کند. آمازون اطلاعات مربوط به فروش، انبارداری و حمل‌ونقل را نگهداری می‌کند.

شما چطور این مجموعه‌های بزرگ و در حال تغییر داده‌ها را بر روی دیسک‌ها ذخیره می‌کنید. چطور به عامل‌های مختلف اجازه‌ی بازیابی، ویرایش و اضافه کردن داده‌ها را به‌صورت هم‌زمان می‌دهید؟ به‌جای این‌که چنین عملکردهایی را خودمان پیاده‌سازی کنیم، ما از سیستم‌های مدیریت پایگاه داده[۲] استفاده می‌کنیم: یک نرم‌افزار خاص برای مدیریت پایگاه‌های داده. این سیستم به سازمان‌دهی و ذخیره‌سازی داده‌ها می‌پردازد و به‌عنوان واسطه‌ای برای دسترسی و تغییر پایگاه داده عمل می‌کند. در این فصل یاد خواهید گرفت:

- مدل **رابطه‌ای** اغلب پایگاه‌های داده را درک کنید.
- منعطف باشید و از سیستم‌های پایگاه داده‌ی **غیررابطه‌ای** استفاده کنید.
- کامپیوترها را هماهنگ کرده و داده‌های خود را **توزیع** کنید.
- با استفاده از سیستم‌های پایگاه داده **جغرافیایی** مسائل را بهتر نگاشت کنید.
- داده‌ها را بین سیستم‌ها با استفاده از **سریال‌سازی** داده‌ها به اشتراک بگذارید.

---

[۱] Charles Bachman (1924-): دانشمند علوم کامپیوتر اهل آمریکا که پژوهش‌های زیادی در مورد پایگاه داده‌ها انجام داده و یکی از برندگان جایزه‌ی تورینگ است. م.

[۲] DataBase Management System (DBMS)

سیستم‌های پایگاه داده‌ی رابطه‌ای[1] گستردگی بیشتری دارند و غالب هستند، اما سیستم‌های پایگاه داده‌ی غیررابطه‌ای[2] اغلب می‌توانند ساده‌تر و کاراتر باشند. سیستم‌های پایگاه داده بسیار متنوع هستند، و انتخاب یکی از آن‌ها می‌تواند سخت باشد. این فصل مروری کلی بر انواع مختلف سیستم‌های پایگاه داده‌ی موجود ارائه می‌کند.

چون داده‌ها از طریق یک سیستم پایگاه داده به‌سادگی در دسترس هستند، می‌توان از آن به‌خوبی استفاده کرد. یک معدنچی می‌تواند مواد معدنی ارزشمند و فلزات را از یک زمین سنگلاخی ارزان‌قیمت استخراج کند. به همین روش، ما اغلب می‌توانیم اطلاعات ارزشمندی را از داده‌ها استخراج کنیم. به این کار **داده‌کاوی**[3] می‌گویند.

به‌عنوان‌مثال، یک خواربارفروشی بزرگ زنجیره‌ای داده‌های تراکنش‌های فروش محصول خود را تجزیه‌وتحلیل کرد و متوجه شد که مشتریانی که بیشترین خرید را انجام می‌دهند، اغلب نوعی پنیر را خریداری می‌کنند که رتبه زیر ۲۰۰ را در فروش دارد. این فروشگاه‌ها معمولاً فروش محصولات با میزان فروش کم را متوقف می‌کردند. داده‌کاوی نه‌تنها موجب الهام گرفتن مدیران برای حفظ آن نوع پنیر شد، بلکه به ایشان گوشزد کرد که آن را در محل‌هایی با دید بهتر قرار دهند. این کار باعث خوشحال شدن بهترین مشتریان آن‌ها شد و موجب شد که آن‌ها دوباره برای خرید به فروشگاه برگردند. خواربارفروشی زنجیره‌ای برای این‌که بتواند چنین حرکت هوشمندانه‌ای انجام دهد، می‌بایست داده‌های خود را به‌خوبی در یک سیستم پایگاه داده سازمان‌دهی می‌کرد.

۶-۱- رابطه‌ای

ظهور مدل رابطه‌ای در اواخر دهه‌ی ۱۹۶۰ موفقیت بزرگی برای مدیریت اطلاعات بود. پایگاه داده‌ی رابطه‌ای پیشگیری از اطلاعات تکراری و ناسازگاری داده‌ها را ساده می‌کند. بخش عمده‌ای از پایگاه‌های داده‌ای که امروزه استفاده می‌شوند، رابطه‌ای هستند.

در مدل رابطه‌ای داده‌ها در **جدول‌های** مختلفی تقسیم می‌شوند. یک جدول شبیه به یک ماتریس یا یک صفحه‌ی گسترده عمل می‌کند. هر بخش از داده‌ها یک **سطر** در جدول است. ستون‌ها ویژگی‌های متفاوتی هستند که داده‌ها می‌توانند داشته باشند. ستون‌ها معمولاً نوع داده‌ای را که می‌توانند شامل شوند مشخص می‌کنند. هم‌چنین، ستون‌ها می‌توانند سایر محدودیت‌ها را نیز تعیین کنند: این‌که برای هر سطر

---

[1] Relational Database Systems

[2] Non-Relational Database Systems

[3] Data Mining

داشتن مقدار در آن ستون اجباری است یا خیر، یا مقدار موجود در هر ستون باید به ازای تمام سطرهای جدول منحصربه‌فرد باشد یا خیر، و الی آخر.

معمولاً به هر ستون یک **فیلد**[1] گفته می‌شود. اگر یک ستون فقط اجازه داشته باشد که اعداد صحیح را نشان دهد، آن را یک **فیلد اعداد صحیح**[2] می‌نامیم. جدول‌های مختلف از فیلدهای متفاوتی استفاده می‌کنند. سازمان‌دهی یک جدول توسط فیلدهای آن و محدودیت‌هایی که باید رعایت کنند انجام می‌شود. این ترکیب فیلدها و محدودیت‌ها را **طرح‌واره‌ی** (یا **اسکیمای**) **جدول**[3] می‌نامند.

داده‌های ورودی یک جدول سطرها هستند، و سیستم پایگاه داده درصورتی‌که یک سطر از محدودیت‌های یک جدول تخطی کند، آن را نمی‌پذیرد. این یک محدودیت بزرگ برای مدل رابطه‌ای است. زمانی که ویژگی‌های داده‌ها بسیار زیاد باشند، قرار دادن داده‌ها در یک طرح‌واره‌ی ثابت می‌تواند مشکل‌ساز باشد. ولی اگر در حال کار با داده‌هایی با ساختار متجانس هستید، یک طرح‌واره‌ی ثابت می‌تواند به شما در کسب اطمینان از معتبر بودن داده‌ها کمک کند.

روابط

پایگاه داده‌ای از صورت‌حساب‌ها شامل یک جدول واحد را در نظر بگیرید. برای هر صورت‌حساب، باید اطلاعاتی را در مورد سفارش و مشتری ثبت کنیم. وقتی بیش از یک صورت‌حساب برای یک مشتری ذخیره می‌شود، اطلاعات تکراری می‌شوند:

Date	Customer Name	Customer Phone Number	Order Total
2017-02-17	Bobby Tables	997-1009	$93.37
2017-02-18	Elaine Roberts	101-9973	$77.57
2017-02-20	Bobby Tables	997-1009	$99.73
2017-02-22	Bobby Tables	991-1009	$12.01

شکل ۶-۱: داده‌های صورت‌حساب ذخیره‌شده در یک جدول

مدیریت و به‌هنگام‌سازی اطلاعات تکراری سخت است. برای اجتناب از این امر، مدل رابطه‌ای اطلاعات مرتبط را در جدول‌های مختلف تقسیم می‌کند. به‌عنوان نمونه، ما داده‌های صورت‌حساب‌های

---

[1] Field
[2] Integer Field
[3] Table's Schema

خود را به دو جدول تقسیم می‌کنیم: «سفارشات» و «مشتریان». می‌توانیم هر سطر در جدول «سفارشات» را به یک سطر در جدول «مشتریان» مرتبط سازیم:

سفارشات
orders

ID	Date	Customer	Amount
1	2017-02-17	37	$93.37
2	2017-02-18	73	$77.57
3	2017-02-20	37	$99.73
4	2017-02-22	37	$12.01

مشتریان
customers

ID	Name	Phone
37	Bobby Tables	997-1009
73	Elaine Roberts	101-9973

شکل ۶-۲: روابط بین سطرها موجب حذف داده‌های تکراری می‌شوند

با مرتبط سازی داده‌های جدول‌های مختلف، یک مشتری می‌تواند در سفارشات متعددی بدون تکرار داده‌ها حضور داشته باشد. برای پشتیبانی از روابط، هر جدول یک فیلد شناسایی ویژه یا ID دارد. از مقادیر ID برای ارجاع به یک سطر خاص در یک جدول استفاده می‌کنیم. این مقادیر باید منحصربه‌فرد باشند: دو سطر با مقدار ID یکسان نمی‌توانند وجود داشته باشند. فیلد ID در یک جدول به‌عنوان **کلید اصلی**[1] نیز شناخته می‌شود. فیلدی را که ارجاعات به ID سایر سطرها را نگه می‌دارد **کلید خارجی**[2] می‌نامند.

با استفاده از کلیدهای اصلی و کلیدهای خارجی، می‌توانیم روابط پیچیده‌ای بین مجموعه داده‌های مجزا به وجود آوریم. به‌عنوان نمونه، جدول‌های شکل ۶-۳ اطلاعاتی را در مورد برندگان جایزه‌ی تورینگ ذخیره می‌کنند[3].

رابطه‌ی بین دانشمندان کامپیوتر و جوایز به شفافیت رابطه‌ی بین مشتریان و سفارشات نیست. یک جایزه ممکن است بین دو دانشمند کامپیوتر تقسیم شود، و هیچ جا گفته نشده است که یک دانشمند کامپیوتر فقط یک‌بار باید برنده‌ی جایزه شود. به همین دلیل، می‌توانیم از یک جدول «برندگان» برای ذخیره‌سازی روابط بین دانشمندان کامپیوتر و جوایز استفاده کنیم.

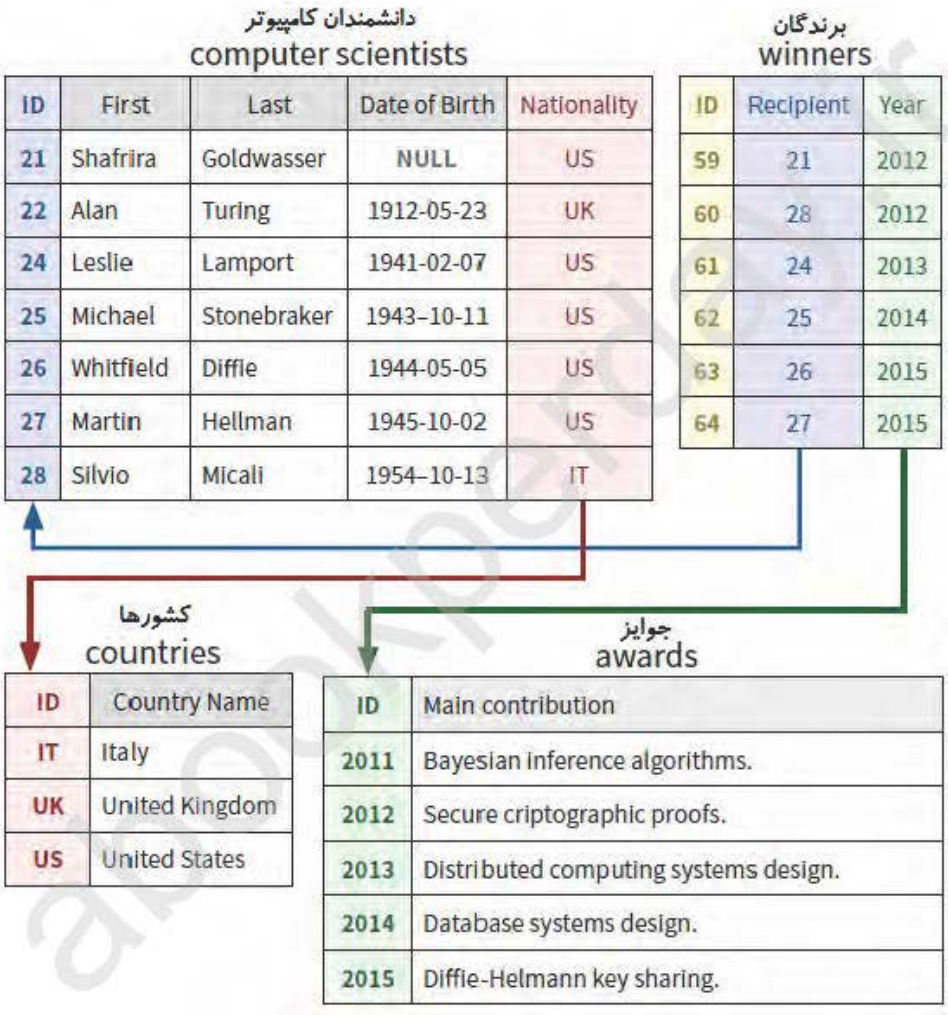
---

[1] Primary Key

[2] Foreign Key

[3] جایزه‌ی تورینگ شبیه به جایزه‌ی نوبل است، ولی به دانشمندان کامپیوتر اهدا می‌شود. ارزش این جایزه یک‌میلیون دلار است.

وقتی یک پایگاه داده طوری سازمان‌دهی می‌شود که کاملاً خالی از اطلاعات تکراری باشد، می‌گوییم **نرمال‌شده**[1] شده است. فرایند تغییر شکل یک پایگاه داده با داده‌های تکراری به یک پایگاه داده‌ی بدون داده‌های تکراری را **نرمال‌سازی**[2] می‌گویند.



ID	First	Last	Date of Birth	Nationality
21	Shafrira	Goldwasser	NULL	US
22	Alan	Turing	1912-05-23	UK
24	Leslie	Lamport	1941-02-07	US
25	Michael	Stonebraker	1943-10-11	US
26	Whitfield	Diffie	1944-05-05	US
27	Martin	Hellman	1945-10-02	US
28	Silvio	Micali	1954-10-13	IT

ID	Recipient	Year
59	21	2012
60	28	2012
61	24	2013
62	25	2014
63	26	2015
64	27	2015

ID	Country Name
IT	Italy
UK	United Kingdom
US	United States

ID	Main contribution
2011	Bayesian inference algorithms.
2012	Secure criptographic proofs.
2013	Distributed computing systems design.
2014	Database systems design.
2015	Diffie-Helmann key sharing.

شکل ٦-٣: دانشمندان کامپیوتر و جوایز تورینگ

---

۱ Normalized
۲ Normalization

مهاجرت طرح‌واره

با رشد یک برنامه و اضافه شدن ویژگی‌های جدید، بعید است که ساختار پایگاه داده‌ی آن (طرح‌واره‌ی تمام جداول آن) ثابت بماند. هنگامی که نیاز به تغییر ساختار پایگاه داده داریم، یک اسکریپت **مهاجرت طرح‌واره**[1] ایجاد می‌کنیم. این اسکریپت به‌طور خودکار طرح‌واره را ارتقا و داده‌های موجود را بر این اساس تغییر می‌دهد. به‌طور معمول، این اسکریپت‌ها می‌توانند تغییرات خود را لغو کنند. این امر اجازه می‌دهد تا به‌راحتی ساختار پایگاه داده را برای مطابقت با نسخه‌ی قبلی نرم‌افزار بازیابی کنید.

در اکثر سیستم‌های مدیریت پایگاه داده ابزارهای آماده‌ای برای انتقال طرح‌واره وجود دارد. این ابزارها به شما در ایجاد، اعمال و برگرداندن اسکریپت‌های مهاجرت طرح‌واره کمک می‌کنند. برخی از سیستم‌های بزرگ سالانه صدها تغییر طرح‌واره را انجام می‌دهند، بنابراین این ابزارها اجتناب‌ناپذیر هستند. بدون ایجاد مهاجرت‌های طرح‌واره، بازگرداندن تغییرات پایگاه داده‌ی «دستی» شما به یک نسخه‌ی کاری خاص دشوار خواهد بود. تضمین سازگاری بین پایگاه داده‌های محلی توسعه‌دهندگان نرم‌افزارهای مختلف دشوار خواهد بود. این مشکلات اغلب در پروژه‌های نرم‌افزاری بزرگی که در طراحی پایگاه داده دقت نمی‌کنند رخ می‌دهند.

SQL

تقریباً تمام سیستم‌های مدیریت پایگاه داده‌ی رابطه‌ای با یک زبان پرس‌وجو به نام SQL کار می‌کنند[2]. آموزش کامل SQL در این کتاب مدنظر نیست، ولی یک ایده‌ی کلی در مورد چگونگی عملکرد آن به شما داده می‌شود. داشتن کمی آشنایی با SQL، حتی اگر به‌صورت مستقیم با آن کار نمی‌کنید، لطف دارد. یک پرس‌وجوی SQL عبارتی است که نشان می‌دهد چه داده‌هایی باید بازیابی شوند:

```
SELECT <field name> [, <field name>, <field name>,…]
FROM <table name>
WHERE <condition>;
```

بخش‌هایی که بعد از SELECT می‌آیند، فیلدهایی هستند که می‌خواهید آن‌ها را به دست آورید. برای به دست آوردن همه‌ی فیلدها می‌توانید بنویسید SELECT *. ممکن است در یک پایگاه داده چندین

---

[1] Schema Migration Script

[2] این زبان اغلب به‌صورت سیکوئل (sequel) تلفظ می‌شود ولی تلفظ اس‌کیوال نیز نادرست نیست.

جدول وجود داشته باشد، بنابراین بخش FROM مشخص می‌کند از کدام جدول‌ها می‌خواهید پرس‌وجو کنید. بعد از دستور WHERE شرایط موردنظر را برای انتخاب سطرها اعلام می‌کنید. پرس‌وجوی زیر تمام فیلدهای جدول «مشتریان» را به دست آورده و سطرها را نیز بر اساس نام (name) و سن (age) پالایش می‌کند:

```
SELECT * FROM customers
WHERE age > 21 AND name = "John";
```

می‌توانید پرس‌وجو را به‌صورت SELECT * FROM customers و بدون مشخص کردن عبارت WHERE بنویسید. این امر موجب می‌شود، اطلاعات تمام مشتریان برگردانده شود. عملگرهای پرس‌وجوی دیگری نیز وجود دارند که باید آن‌ها را بشناسید: عملگر ORDER BY نتایج پرس‌وجو را بر اساس یک یا چند فیلد مشخص‌شده مرتب می‌کند؛ عملگر GROUP BY نیز در زمان نیاز به گروه‌بندی نتایج در قالب موارد مشخص و برگرداندن نتایج هر گروه به‌صورت تجمیع‌شده استفاده می‌شود. به‌عنوان نمونه، اگر جدولی از مشتریان داشته باشید که دارای یک فیلد به نام «کشور» (country) و یک فیلد به نام «سن» (age) باشد، می‌توانید پرس‌وجویی به‌صورت زیر بنویسید:

```
SELECT country, AVG(age)
FROM customers
GROUP BY country
ORDER BY country;
```

این پرس‌وجو یک لیست مرتب‌شده از کشورهایی که مشتریان شما در آن‌ها زندگی می‌کنند به همراه میانگین سن مشتریان در هر کشور برمی‌گرداند. در SQL توابع تجمعی دیگری نیز وجود دارند. به‌عنوان نمونه، با جایگزین کردن MAX(age) به‌جای AVG(age) سن مسن‌ترین مشتری در هر کشور را به دست می‌آورید.

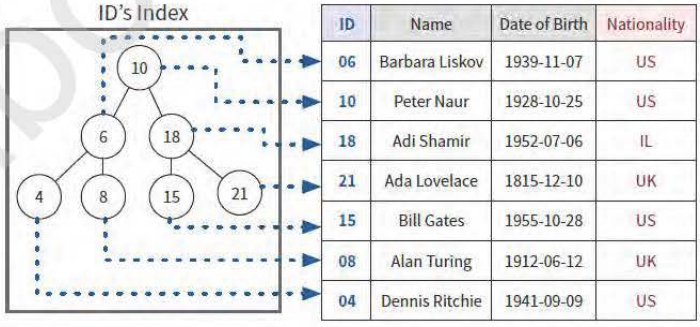
گاهی اوقات نیاز دارید اطلاعات را از یک سطر و سطرهای مرتبط با آن به دست بیاورید. فرض کنید یک جدول برای ذخیره کردن سفارشات و یک جدول برای ذخیره کردن مشتریان دارید. جدول سفارشات (orders) یک کلید خارجی برای ارجاع به مشتریان (customers) دارد (شکل ۶-۲). پیدا کردن اطلاعاتی در مورد مشتریانی که سفارشات گران داشته‌اند، نیازمند واکشی اطلاعات از هر دو جدول است. ولی نیاز نیست هر دو جدول را به‌صورت مجزا جست‌وجو کرده و خودتان نتایج را تطبیق دهید. برای این کار یک دستور SQL وجود دارد:

```
SELECT DISTINCT customers.name, customers.phone
FROM customers
JOIN orders ON orders.customer = customers.id
WHERE orders.amount > 100.00;
```

این پرس‌وجو نام و تلفن مشتریانی را که سفارشات بالای ۱۰۰ دلار داشته‌اند برمی‌گرداند. عبارت SELECT DISTINCT باعث می‌شود اطلاعات هر مشتری فقط یک‌بار در خروجی آورده شود. دستور پیوند یا JOIN امکان پرس‌وجوهای منعطف را به وجود می‌آورد[۱]، ولی این کار هزینه‌هایی نیز دارد. محاسبه‌ی پیوندها هزینه‌بر است. آن‌ها ممکن است همه‌ی ترکیبات ممکن سطرهای جدول‌های مورداستفاده را بررسی کنند. یک مدیر پایگاه داده باید همیشه به تعداد سطرهای جدول‌های مورد پیوند توجه داشته باشد. برای جدول‌های خیلی بزرگ، پیوندها غیرممکن می‌شوند. دستور JOIN مهم‌ترین توانایی و درعین‌حال، بزرگ‌ترین نقطه‌ی ضعف پایگاه‌های داده‌ی رابطه‌ای است.

شاخص‌گذاری

برای این‌که کلید اصلی یک جدول مفید باشد، باید بتوانیم در زمان ارائه شدن مقدار ID سریعاً داده‌ها را بازیابی کنیم. برای برآورده کردن این هدف، سیستم مدیریت پایگاه داده یک **شاخص**[۲] کمکی به وجود می‌آورد و به کمک آن ID سطرها را به آدرس حافظه‌ی مربوط به آن نگاشت می‌کند. یک شاخص الزاماً یک درخت جستجوی دودویی خودمتوازن (بخش ۴-۳) است. هر سطر در جدول به یک گره در درخت مرتبط است.



ID	Name	Date of Birth	Nationality
06	Barbara Liskov	1939-11-07	US
10	Peter Naur	1928-10-25	US
18	Adi Shamir	1952-07-06	IL
21	Ada Lovelace	1815-12-10	UK
15	Bill Gates	1955-10-28	US
08	Alan Turing	1912-06-12	UK
04	Dennis Ritchie	1941-09-09	US

شکل ٦-٤: نگاشت مقادیر ID به مکان سطرها توسط شاخص

[۱] راه‌هایی مختلفی برای JOIN وجود دارد، برای کسب اطلاعات بیشتر https://code.energy/joins را ببینید.

[۲] Index

کلیدهای گره‌ها همان مقادیر فیلد موردنظر برای شاخص‌گذاری هستند. برای یافتن یک سطر با مقدار مفروض، آن مقدار را در درخت جستجو می‌کنیم. زمانی که گره را پیدا کردیم، آدرس محل ذخیره‌سازی را به دست آورده و از آن برای واکشی سطر استفاده می‌کنیم. جستجو در درخت جستجوی دودویی $O(\log n)$ است، بنابراین پیدا کردن سطرها در جدول‌های بزرگ سریع انجام می‌شود.

معمولاً یک شاخص به‌وسیله‌ی سیستم مدیریت پایگاه داده برای هر کلید اصلی در پایگاه داده ساخته می‌شود. اگر اغلب از سایر فیلدها برای پیدا کردن سطرها استفاده می‌کنیم (به‌عنوان نمونه، اگر مشتریان را با نام جستجو می‌کنیم)، می‌توانیم به سیستم مدیریت پایگاه داده دستور بدهیم که شاخص‌های اضافی برای این فیلدها نیز بسازد.

قیدهایی یکتایی[1]: شاخص‌ها اغلب به‌صورت خودکار برای فیلدهایی که دارای قید یکتایی هستند ساخته می‌شوند. در زمان درج یک سطر جدید، سیستم مدیریت پایگاه داده باید کل جدول را برای کسب اطمینان از نقض نشدن قید یکتایی جستجو کند. فهمیدن این‌که آیا یک مقدار در یک فیلد وجود دارد یا خیر، بدون استفاده از یک شاخص، به معنی بررسی کردن تمام سطرهای جدول است. با یک شاخص می‌توانیم سریعاً مقدار موردنظر برای درج را جستجو کرده و وجود آن را بررسی کنیم. برای درج سریع عناصر، شاخص‌گذاری فیلدهایی که دارای قید یکتایی هستند الزامی است.

مرتب‌سازی: شاخص‌ها در بازیابی سطرها به‌صورت مرتب‌شده بر اساس فیلدهای شاخص‌گذاری شده کمک می‌کنند. به‌عنوان نمونه، اگر یک شاخص برای فیلد «نام» (name) داشته باشیم، می‌توانیم سطرها را به‌صورت مرتب‌شده بر اساس نام و بدون محاسبه‌ی اضافی به دست بیاوریم. وقتی از دستور ORDER BY در مورد یک فیلد بدون شاخص استفاده می‌کنید، سیستم مدیریت پایگاه داده باید داده‌ها را در حافظه و قبل از ارائه‌ی نتایج پرس‌وجو مرتب کند. بسیاری از سیستم‌های مدیریت پایگاه داده ممکن است در صورت زیاد بودن تعداد سطرهای درگیر، پرس‌وجوهایی را که نیاز به مرتب‌سازی بر اساس یک فیلد بدون شاخص دارند نپذیرند.

اگر باید سطرها را بر اساس کشور و بعداز آن بر اساس سن مرتب کنید، داشتن یک شاخص بر روی فیلد «سن» یا «کشور» چندان کمکی نمی‌کند. یک شاخص بر روی «کشور» به شما اجازه‌ی واکشی سطرها را به‌صورت مرتب‌شده بر اساس کشور می‌دهد، ولی همچنان نیاز دارید عناصر دارای کشور مشابه را به‌صورت دستی بر اساس سن مرتب کنید. در زمان نیاز به دو فیلد، از **شاخص‌های مشترک**[2] استفاده

---

[1] Uniqueness Constraints

[2] Joint Indexes

می‌شود. این شاخص‌ها چندین فیلد را شاخص‌گذاری کرده و کمکی در یافتن سریع عناصر نمی‌کنند، ولی داده‌های مرتب‌شده بر اساس چند فیلد را سریعاً برمی‌گردانند.

کارایی: در کل می‌توان گفت شاخص‌ها بسیار مفید هستند: آن‌ها امکان پرس‌وجوی فوق‌سریع و دسترسی آنی به داده‌های ذخیره‌شده را فراهم می‌آورند. پس چرا نباید برای همه‌ی فیلدهای جداول شاخص داشته باشیم؟ مشکل این است که در زمان درج یا حذف یک سطر در یک جدول، به‌منظور نمایش این تغییرات تمام شاخص‌های آن باید بهنگام‌سازی شوند. اگر تعداد شاخص‌ها زیاد باشد بهنگام‌سازی، درج یا حذف سطرها ازلحاظ محاسباتی بسیار پرهزینه خواهد بود (مبحث متوازن ساختن درخت را به یاد داشته باشید). به‌علاوه، شاخص‌ها فضای دیسک را به‌عنوان یک منبع محدود اشغال می‌کنند.

شما باید بر نحوه‌ی استفاده‌ی برنامه‌هایتان از پایگاه داده نظارت داشته باشید. معمولاً ابزارهایی در سیستم‌های مدیریت پایگاه داده برای کمک به شما در این امر وجود دارند. این ابزارها می‌توانند به تشریح پرس‌وجوها، گزارش‌دهی در مورد شاخص‌های مورداستفاده‌ی یک پرس‌وجو و تعداد سطرهایی که باید برای انجام پرس‌وجو به‌صورت ترتیبی بررسی شوند، بپردازند. اگر پرس‌وجوهای شما زمان زیادی را برای بررسی ترتیبی داده‌های یک فیلد هدر بدهند، یک شاخص برای آن فیلد اضافه کرده و ببینید چه کمکی در این راه می‌کند. به‌عنوان‌مثال، اگر به‌صورت مکرر از پایگاه داده در مورد افرادی با سن مفروض پرس‌وجو می‌کنید، تعریف یک شاخص بر روی فیلد سن به سیستم مدیریت پایگاه داده امکان می‌دهد به‌صورت مستقیم سطرهای مرتبط با یک سن خاص را به دست آورد. با این روش و با اجتناب از بررسی ترتیبی برای پالایش سطرهایی که با سن موردنظر تطابق ندارند، در وقت صرفه‌جویی می‌کنید.

برای تطبیق دادن پایگاه داده و کسب کارایی بالاتر، مهم است بدانید که کدام شاخص‌ها را نگه‌داشته و کدام‌یک را دور بیندازید. اگر یک پایگاه داده اغلب برای خواندن استفاده‌شده و به‌ندرت بهنگام‌سازی می‌شود، نگه‌داشتن شاخص‌های بیشتر منطقی است. شاخص‌گذاری ضعیف یکی از علل اصلی کند بودن سیستم‌های تجاری است. مدیران بی‌دقت سیستم اغلب به نحوه‌ی اجرای پرس‌وجوهای معمول توجه نداشته و فقط برخی از فیلدها را به‌صورت تصادفی و آن‌هم بر اساس این‌که «حس» می‌کنند به کارایی کمک خواهد کرد، شاخص‌گذاری می‌کنند. این کار را نکنید! از ابزارهای «تشریح» برای بررسی پرس‌وجوها و اضافه کردن شاخص‌ها، فقط زمانی که تفاوت ایجاد می‌کنند، استفاده کنید.

تراکنش‌ها

تصور کنید یک بانک مخفی سوئیسی هیچ رکوردی از انتقالات پول نگهداری نمی‌کند: پایگاه داده‌ی آن فقط موجودی حساب را ذخیره می‌کند. فرض کنید یک نفر می‌خواهد پولی را از حساب خودش به‌حساب دوستش در همان بانک منتقل کند. دو عمل باید بر روی پایگاه داده‌ی بانک انجام شوند: کسر پول از یک موجودی، و اضافه کردن آن به موجودی دیگر.

یک میرور پایگاه داده معمولاً اجازه می‌دهد چندین مشتری به‌صورت هم‌زمان به خواندن و نوشتن بپردازند؛ اجرای دستورات به‌صورت متوالی باعث می‌شود سیستم مدیریت پایگاه داده بسیار کند عمل کند. حال مشکل اینجاست: اگر یک نفر مجموع موجودی تمام حساب‌ها را *بعد از* انجام عملیات کسر و *قبل از* انجام عملیات جمع مربوطه درخواست کند، بخشی از پول گم می‌شود. یا بدتر از آن: اگر برق سیستم بین دو عملیات قطع شود چه رخ می‌دهد؟ زمانی که سیستم مجدداً در دسترس قرار می‌گیرد، یافتن علت ناسازگاری داده‌ها مشکل خواهد بود.

ما به راهی نیاز داریم که بر اساس آن سیستم پایگاه داده یا *تمام* تغییرات ناشی از یک عملیات چندبخشی را اعمال، یا داده‌ها را بدون تغییر حفظ کند. به این منظور، سیستم‌های پایگاه داده عملکردی را ارائه می‌دهند که **تراکنش**[1] نام دارد. یک تراکنش لیستی از عملیات پایگاه داده است که باید به‌صورت **اتمیک** اجرا شوند[2]. این امر موجب آسان‌تر شدن زندگی برنامه‌نویسان می‌شود: سیستم پایگاه داده مسئول حفظ سازگاری پایگاه داده است. تمام کاری که برنامه‌نویس باید انجام دهد این است که عملیات مرتبط باهم را در کنار هم قرار دهد:

```
START TRANSACTION;
UPDATE vault SET balance = balance + 50 WHERE id=2;
UPDATE vault SET balance = balance - 50 WHERE id=1;
COMMIT;
```

به یاد داشته باشید، بهنگام‌سازی‌های چندمرحله‌ای بدون تراکنش‌ها به‌تدریج منجر به ناسازگاری‌های پیچیده، پیش‌بینی‌نشده و پنهان در داده‌های شما می‌شوند.

۶-۲- غیر رابطه‌ای

پایگاه‌های داده‌ی رابطه‌ای عالی هستند ولی محدودیت‌هایی نیز دارند. با پیچیده‌تر شدن یک برنامه، پایگاه داده‌ی رابطه‌ای آن جدول‌های بیشتر و بیشتری خواهد داشت. پرس‌وجوها بزرگ‌تر و درک آن‌ها

---

[1] Transaction

[2] عملیات اتمیک (Atomic) در یک گام واحد انجام می‌شوند: این عملیات را نمی‌توان به‌صورت نصفه انجام داد.

سخت‌تر خواهد شد. و هر پرس‌وجو به پیوندهای بیشتر و بیشتری نیاز خواهد داشت که موجب افزایش هزینه‌ی محاسبات و به وجود آمدن تنگناهای جدی خواهد شد.

مدل غیررابطه‌ای[1] روابط جدولی را حذف می‌کند. این مدل به‌ندرت ما را ملزم به ترکیب اطلاعات از چندین داده‌ی ورودی می‌کند. از آنجایی که سیستم‌های پایگاه داده‌ی غیررابطه‌ای از زبان‌های پرس‌وجویی به‌غیراز SQL استفاده می‌کنند، به آن‌ها **پایگاه داده‌ی NoSQL** نیز می‌گویند.

شکل ٦-٥: دریافت شده از http://geek-and-poke.com

مخزن اسناد

معروف‌ترین نوع پایگاه داده‌ی NoSQL **مخزن اسناد**[2] است. در مخازن اسناد، داده‌های ورودی دقیقاً به شکلی که موردنیاز برنامه هستند ذخیره می‌شوند. شکل ۶-۶ روش جدول و روش سند برای ذخیره‌سازی پست‌ها در یک بلاگ را مقایسه می‌کند.

توجه کنید تمام داده‌های مربوط به یک پست چگونه در سطر مربوط به آن قرارگرفته‌اند. مدل رابطه‌ای از ما *توقع* دارد اطلاعات را در هر مکان مرتبط تکرار کنیم. بهنگام‌سازی و سازگار نگه‌داشتن داده‌های تکراری کاری مشکل است. در عوض، با گروه‌بندی داده‌های مرتبط، مخازن سند می‌توانند انعطاف‌پذیری بیشتری ارائه دهند:

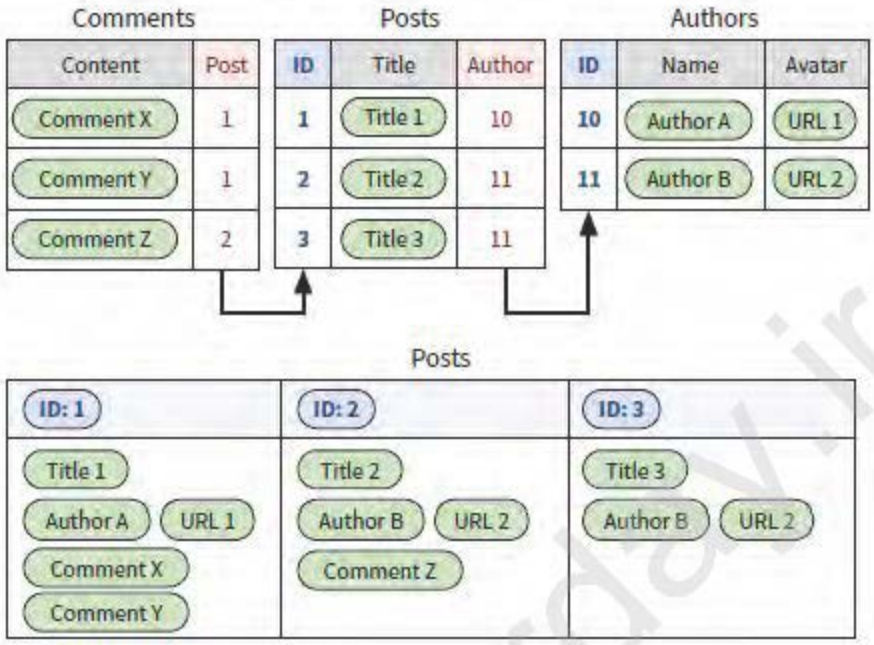
- نیاز به پیوند سطرها ندارید.
- به طرح‌واره‌های ثابت نیاز ندارید.
- هر داده‌ی ورودی می‌تواند پیکره‌بندی فیلدهای خاص خود را داشته باشد.

این یعنی هیچ جدول و سطری در مخازن اسناد وجود ندارد. در عوض، هر داده‌ی ورودی یک **سند** نامیده می‌شود. اسناد مرتبط باهم در یک **مجموعه** گروه‌بندی می‌شوند.

---

[1] Non-Relational Model

[2] Document Store



شکل ٦-٦: داده‌ها در مدل رابطه‌ای (بالا) در مقایسه با NoSQL (پایین)

اسناد دارای یک فیلد کلید اصلی هستند، بنابراین ایجاد رابطه بین اسناد ممکن است. ولی پیوندها بر روی مخازن اسناد بهینه نیستند. گاهی اوقات حتی پیاده‌سازی نیز نمی‌شوند، بنابراین باید خودتان روابط بین اسناد را دنبال کنید. هر دو روش در صورت به اشتراک‌گذاری داده‌ها بین اسناد نامناسب هستند و باید این داده‌ها را در اسناد تکرار کرد.

شبیه به پایگاه‌های داده‌ی رابطه‌ای، پایگاه‌های داده‌ی NoSQL شاخص‌هایی برای فیلدهای کلید اصلی می‌سازند. شما نیز می‌توانید شاخص‌های اضافی برای فیلدهایی که اغلب در پرس‌وجوها یا مرتب‌سازی استفاده می‌کنید، تعریف کنید.

مخزن کلید- مقدار

مخزن کلید - مقدار[1] ساده‌ترین نوع روش ذخیره‌سازی سازمان‌یافته و پایدار داده‌ها است. این روش عمدتاً برای ذخیره‌سازی در حافظه‌ی نهان مورداستفاده قرار می‌گیرد. به‌عنوان‌مثال، زمانی که یک کاربر یک صفحه‌ی وب خاص را از یک سرور درخواست می‌کند، سرور باید داده‌های صفحه‌ی وب را

[1] Key-Value Store

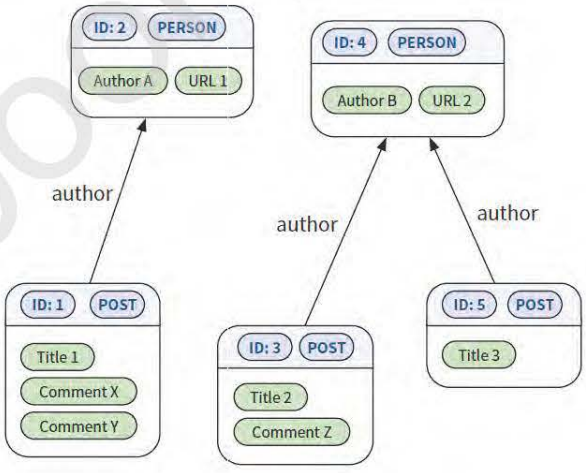
از پایگاه داده واکشی کرده و از آن‌ها برای تولید کد HTML ارسالی برای کاربر استفاده کند. در وب سایت‌های با ترافیک زیاد و با هزاران دسترسی هم‌زمان، انجام این کار غیرممکن است.

برای حل این مشکل، ما از یک مخزن کلید – مقدار به‌عنوان یک مکانیزم ذخیره‌سازی در حافظه‌ی نهان استفاده می‌کنیم. در این مخزن کلید همان URL درخواست شده و مقدار کد نهایی HTML مرتبط با صفحه‌ی وب است. اگر فرد دیگری همان URL را درخواست کند، فقط همان کد آماده‌ی HTML از مخزن کلید – مقدار و با استفاده از URL به‌عنوان کلید بازیابی می‌شود.

اگر شما یک عملیات کند را که همواره خروجی مشابهی تولید می‌کند تکرار می‌کنید، بهتر است به ذخیره‌سازی آن در حافظه‌ی نهان فکر کنید. البته الزاماً نیاز نیست از مخزن کلید – مقدار استفاده نمایید، بلکه می‌توانید از هر پایگاه داده‌ی دیگری نیز برای ذخیره‌سازی در حافظه‌ی نهان استفاده کنید. فقط در مواردی که داده‌های ذخیره‌شده در حافظه‌ی نهان به‌صورت مکرر مورد دسترسی قرار بگیرند، سیستم مخزن کلید – مقدار اهمیت پیدا خواهد کرد.

پایگاه‌های داده گراف

در **پایگاه‌های داده‌ی گراف**[1]، داده‌های ورودی در قالب گره‌ها و روابط در قالب یال‌ها ذخیره می‌شوند. گره‌ها محدود به یک طرح‌واره‌ی ثابت نیستند و می‌توانند داده‌ها را به شکلی منعطف ذخیره نمایند. ساختار گراف کار کردن با داده‌های ورودی را بر اساس روابط بین آن‌ها کارا می‌سازد. شکل زیر نحوه‌ی نمایش اطلاعات مربوط به شکل ۶-۶ را در یک گراف نشان می‌دهد:



شکل ٦-٧: اطلاعات بلاگ ذخیره‌شده در یک پایگاه داده‌ی گراف

[1] Graph Databases

این منعطف‌ترین نوع پایگاه داده است. با حذف جدول‌ها و مجموعه‌ها، می‌توانید داده‌های شبکه‌ای را به روش‌هایی خلاقانه ذخیره کنید. اگر بخواهید نقشه‌ی ایستگاه‌های مترو و اتوبوس عمومی یک شهر را بر روی یک تخته‌ی سفید رسم کنید، از داده‌های جدولی استفاده نمی‌کنید؛ بلکه از دایره‌ها، کادرها و فلش‌ها استفاده می‌کنید. پایگاه‌های داده‌ی گراف به شما اجازه می‌دهند داده‌ها را این‌گونه ذخیره کنید.

اگر داده‌های شما شبیه به یک شبکه باشند، به استفاده از پایگاه داده‌ی گراف فکر کنید. این پایگاه‌های داده در زمان وجود روابط مهم متعدد بین قطعات داده‌ها بسیار مفید هستند. پایگاه‌های داده‌ی گراف امکان انجام انواع مختلف پرس‌وجوهای مبتنی بر گراف را نیز به وجود می‌آورند. به‌عنوان‌مثال، با ذخیره کردن داده‌های حمل‌ونقل عمومی در یک گراف، می‌توانید به‌صورت مستقیم بهترین مسیرهای یک‌طرفه و دوطرفه‌ی بین دو ایستگاه اتوبوس مفروض را پرس‌وجو کنید.

کلان‌داده

کلیدواژه‌ی **کلان‌داده**[1] به تشریح وضعیت‌هایی از پردازش داده‌ها می‌پردازد که به دلیل حجم، سرعت پردازش یا تنوع شدیداً چالش‌برانگیز هستند[2]. منظور از حجم در کلان‌داده این است که شما صدها هزار ترابایت را باید مدیریت کنید، مانند نمونه‌ی LHC[3]. سرعت پردازش کلان‌داده به این معنی است که شما باید میلیون‌ها دستور نوشتن را در ثانیه و بدون توقف انجام داده، یا حتی میلیاردها دستور خواندن را با سرعت انجام دهید. تنوع داده‌ها به این معنی است که داده‌ها دارای ساختار قدرتمندی نیستند، بنابراین مدیریت کردن آن‌ها با پایگاه‌های داده‌ی رابطه‌ای سنتی سخت است.

هر زمان که به دلیل حجم، سرعت پردازش یا تنوع به یک رویکرد مدیریت داده‌ی غیراستاندارد نیاز داشته باشید، می‌توان آن را یک برنامه‌ی کلان‌داده بنامید. به‌منظور اجرای برخی از موفق‌ترین آزمایش‌های علمی (مانند نمونه‌های موجود در LHC یا SKA)[4]، دانشمندان کامپیوتر در حال تحقیق در مورد موضوعی به نام **ابر کلان‌داده** هستند: ذخیره‌سازی و تحلیل میلیون‌ها ترابایت از داده‌ها.

---

[1] Big Data

[2] این سه ویژگی را ویژگی‌های 3V (Volume، Velocity و Variety) می‌نامند. برخی افراد با اضافه کردن دو مورد ناپایستگی (Variability) و درستی (Veracity) آن‌ها را به‌صورت 5V می‌شناسند. (البته برخی افراد مورد ششم یعنی ارزشمندی (Value) را نیز اضافه کرده‌اند. م.)

[3] برخورددهنده‌ی هادرونی بزرگ (The Large Hadron Collider یا LHC) بزرگ‌ترین شتاب‌دهنده‌ی ذرات در دنیا است. در طی یک آزمایش، حسگرهای آن ۱۰۰۰ ترابایت داده را در هر ثانیه تولید کردند.

[4] آرایه‌ی کیلومترمربعی (The Square Kilometer Array یا SKA) مجموعه‌ای از تلسکوپ‌ها هستند که از سال ۲۰۲۰ مورداستفاده قرار می‌گیرند. این مجموعه هر روز میلیون‌ها ترابایت داده تولید می‌کند.

کلان‌داده‌ها اغلب به دلیل انعطاف زیاد آن‌ها، با پایگاه‌های داده‌ی غیررابطه‌ای کار می‌کنند. پیاده‌سازی انواع متعدد برنامه‌های کلان‌داده‌ها با پایگاه‌های داده‌ی رابطه‌ای عملی نیست.

مقایسه‌ی SQL و NoSQL

پایگاه‌های داده‌ی رابطه‌ای داده‌محور هستند: این پایگاه‌های داده فارغ از نحوه‌ی استفاده از داده‌ها، ساختاردهی داده‌ها را بیشینه کرده و از تکرار آن‌ها ممانعت می‌کنند. پایگاه‌های داده‌ی غیررابطه‌ای برنامه‌محور هستند: این پایگاه‌های داده دسترسی و استفاده را بر اساس نیازهای شما تسهیل می‌کنند.

دیدیم که پایگاه‌های داده‌ی NoSQL به ما اجازه می‌دهند داده‌های حجیم، با میزان تغییر بسیار و غیرساخت‌یافته را به‌صورت سریع و کارا ذخیره کنیم. بدون نگرانی در مورد طرح‌واره‌های ثابت و مهاجرت طرح‌واره، می‌توانید راه‌حل‌های خود را سریع‌تر بسازید. پایگاه‌های داده‌ی غیررابطه‌ای اغلب طبیعی‌تر بوده و کدنویسی آن‌ها ساده‌تر است.

پایگاه داده‌ی غیررابطه‌ای شما قدرتمند خواهد بود، ولی *شما* مسئول بهنگام‌سازی و تکرار اطلاعات در اسناد و مجموعه‌ها هستید. *شما* باید روش‌های لازم را برای حفظ سازگاری آن فراهم آورید. به یاد داشته باشید، قدرت بیشتر مسئولیت بیشتری نیز به دنبال خواهد داشت.

۶-۳- توزیع‌شده

وضعیت‌های متعددی وجود دارند که در آن‌ها نه یک، بلکه چندین کامپیوتر باید باهم برای ارائه‌ی سیستم پایگاه داده همکاری کنند:

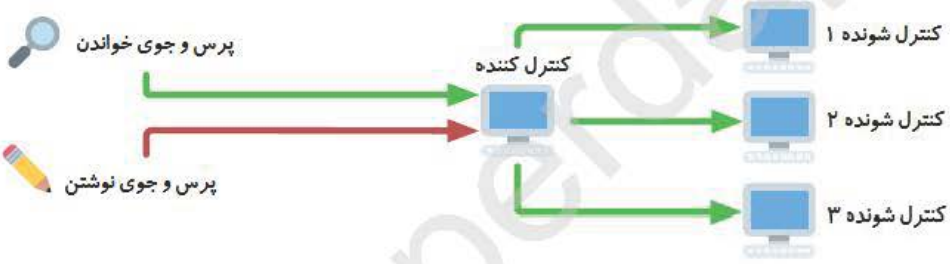
- پایگاه‌های داده با صدها ترابایت داده. پیدا کردن یک کامپیوتر واحد با چنین فضای ذخیره‌سازی غیرعملی است.
- سیستم‌های پایگاه داده چندین هزار پرس‌وجو را در ثانیه پردازش می‌کنند[1]. هیچ کامپیوتر واحدی قدرت شبکه یا پردازش کافی برای مدیریت چنین بار کاری ندارد.
- پایگاه‌های داده‌ی دارای مأموریت‌های بحرانی، مانند مواردی که ارتفاع و سرعت هواپیماهای حاضر در یک فضای هوایی خاص را ثبت می‌کنند. تکیه کردن به یک کامپیوتر واحد بسیار خطرناک خواهد بود، اگر این کامپیوتر از کار بیفتد، پایگاه داده از دسترس خارج خواهد شد.

[1] درست بعد از بازی فینال جام جهانی ۲۰۱۴، توییتر مقدار بار بیشینه‌ای را برابر با ۱۰،۰۰۰ توییت جدید در ثانیه تجربه کرد.

برای این سناریوها، سیستم‌های مدیریت پایگاه داده‌ای وجود دارند که می‌توانند بر روی چندین کامپیوتر هماهنگ عمل کرده و یک سیستم **پایگاه داده‌ی توزیع‌شده**[1] به وجود آورند. حال اجازه بدهید معمول‌ترین راه‌های ایجاد پایگاه‌های داده‌ی توزیع‌شده را بررسی کنیم.

تکرار تک‌کنترل‌کننده[2]

یک کامپیوتر نقش **کنترل‌کننده** را داشته و تمام پرس‌وجوهای پایگاه داده را دریافت می‌کند. این کامپیوتر به چندین کامپیوتر **کنترل‌شونده** متصل است. هر کنترل‌شونده یک کپی از پایگاه داده را دارد. زمانی که کنترل‌کننده پرس‌وجوهای نوشتن را دریافت می‌کند، آن‌ها را به کنترل‌شونده‌ها فرستاده و آن‌ها را هماهنگ نگه می‌دارد:



شکل ۶-۸: پایگاه داده‌ی توزیع‌شده‌ی تک‌کنترل‌کننده

با این تنظیمات، کنترل‌کننده می‌تواند پرس‌وجوهای خواندن بیشتری را انجام دهد، زیرا می‌تواند آن‌ها را به کنترل‌شونده‌ای تفویض کند. و سیستم قابل‌اعتمادتر می‌شود: اگر کامپیوتر کنترل‌کننده خاموش شود، ماشین‌های کنترل‌شونده می‌توانند باهم همکاری کرده و یک کنترل‌کننده‌ی جدید را به‌صورت خودکار انتخاب کنند. با این روش، سیستم دیگر متوقف نخواهد شد.

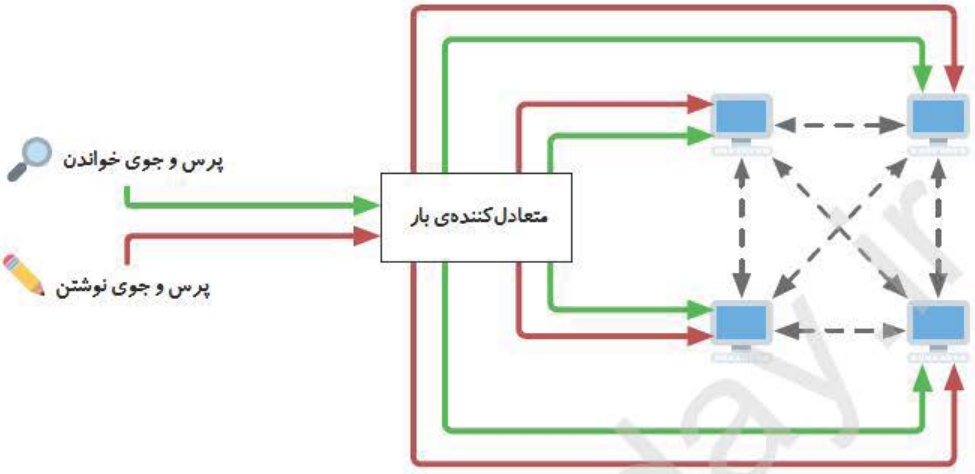
تکرار چندکنترل‌کننده[3]

اگر پایگاه داده‌ی شما باید حجم زیادی از پرس‌وجوهای نوشتن را به‌صورت هم‌زمان پشتیبانی کند، مدل تک‌کنترل‌کننده نمی‌تواند آن حجم بار را مدیریت کند. در این حالت، همه‌ی کامپیوترها در یک

---

[1] Distributed Database
[2] Single-Master Replication
[3] Multi-Master Replication

خوشه به کنترل‌کننده تبدیل می‌شوند. یک متعادل‌کننده‌ی بار نیز برای توزیع پرس‌وجوهای ورودی خواندن و نوشتن به‌صورت یکنواخت بین تمام ماشین‌ها در خوشه استفاده می‌شود.



شکل ٦-٩: پایگاه داده‌ی توزیع‌شده‌ی چندکنترل‌کننده

هر کامپیوتر به تمام کامپیوترهای دیگر خوشه متصل است. این کامپیوترها پرس‌وجوهای نوشتن را بین خود به اشتراک می‌گذارند، بنابراین همه‌ی آن‌ها باهم هماهنگ خواهند بود. هر یک از این کامپیوترها یک نسخه از کل پایگاه داده را در اختیار دارد.

بخش‌بندی کردن[1]

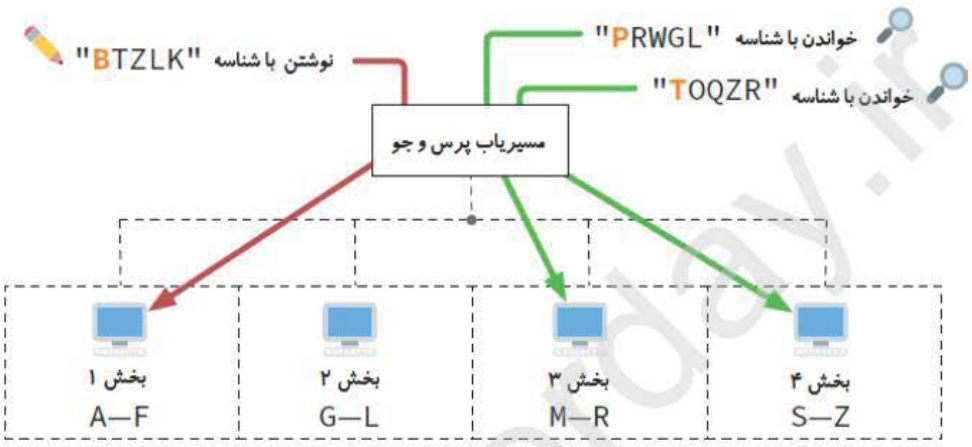
اگر پایگاه داده‌ی شما تعداد زیادی از پرس‌وجوهای نوشتن برای حجم زیادی از داده‌ها دریافت می‌کند، هماهنگ کردن پایگاه‌های داده در تمام خوشه مشکل است. برخی کامپیوترها ممکن است فضای کافی برای جا دادن کل محتوا نداشته باشند. یک راه‌حل، تقسیم کردن پایگاه داده بین کامپیوترها است. چون هر ماشین بخشی از پایگاه داده را دارد، از یک مسیریاب پرس‌وجو برای ارسال پرس‌وجوها به ماشین مربوطه استفاده می‌شود (شکل ۶-۱۰).

با این تنظیمات می‌توان پرس‌وجوهای خواندن و نوشتن بسیاری را برای پایگاه‌های داده‌ی بسیار بزرگ انجام داد. ولی یک مشکل وجود دارد: اگر یک ماشین در خوشه از کار بیفتد، آن بخش از داده‌ها که مسئول آن‌ها بوده است از دسترس خارج خواهند شد. برای مقابله با این خطر، می‌توان عملیات بخش‌بندی را به‌صورت تکراری انجام داد (شکل ۶-۱۱).

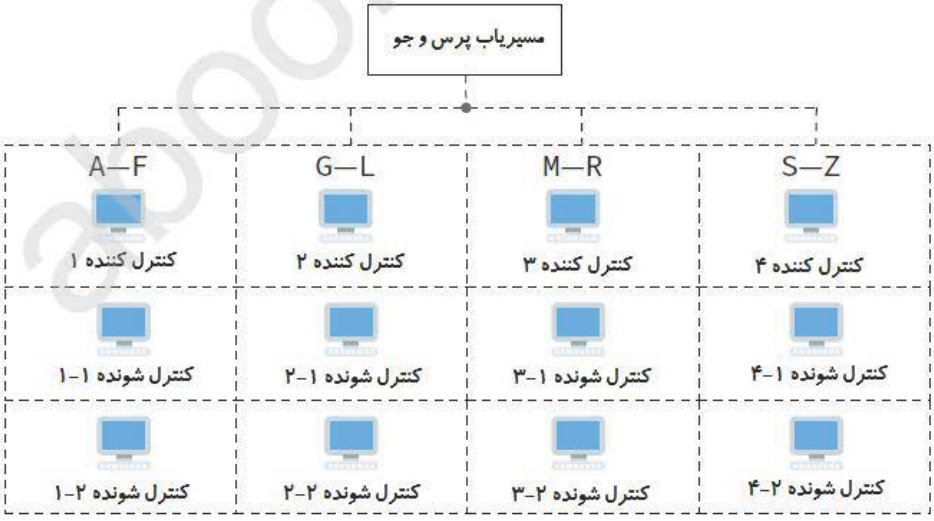
---

[1] Sharding

با این تنظیمات، هر بخش به‌وسیله‌ی یک خوشه‌ی کنترل‌کننده-کنترل‌شونده مدیریت می‌شود. این امر موجب افزایش ظرفیت پایگاه داده برای انجام پرس‌وجوهای خواندن می‌گردد. و اگر یکی از سرورهای اصلی دریکی از بخش‌ها از کار بیفتد، یک کنترل‌شونده به‌صورت خودکار جای آن را می‌گیرد و اطمینان حاصل می‌شود که سیستم از کار نیفتاده یا داده‌ای از دست نمی‌رود.



شکل ٦-١٠: تنظیمات ساده برای بخش‌بندی. پرس‌وجوها بر اساس حرف اول در شناسه‌ی پرس‌وجو مسیردهی می‌شوند



شکل ٦-١١: تنظیمات بخش‌بندی با سه تکرار در هر بخش

سازگاری داده‌ها

در پایگاه‌های داده‌ی توزیع‌شده با تکرار، بهنگام‌سازی‌های انجام‌شده بر روی یک ماشین به‌صورت آنی در تمام تکرارها منتشر نمی‌شوند. کمی زمان می‌برد تا تمام ماشین‌ها در یک خوشه باهم هماهنگ شوند. این امر می‌تواند سازگاری داده‌ها[1] را تخریب کند.

در نظر بگیرید در حال فروش بلیت سینما در یک وب‌سایت هستید. این وب‌سایت ترافیک بسیار زیادی دارد، به همین دلیل پایگاه داده‌ی آن بر روی دو سِرور توزیع‌شده است. آلیس یک بلیت را از سِرور A می‌خرد. باب در حال استفاده از سِرور B است و همان بلیت را آزاد می‌بیند. قبل از این‌که خرید آلیس در سِرور B منتشر شود، باب نیز همان بلیت را می‌خرد. حال دو سِرور دچار **ناسازگاری داده‌ها**[2] شده‌اند. برای حل این مشکل، باید یکی از خریدها را برگردانید و از آلیس یا باب عصبانی عذرخواهی کنید.

سیستم‌های پایگاه داده ابزارهایی برای کنترل ناسازگاری داده‌ها ارائه می‌کنند. به‌عنوان نمونه، برخی از این سیستم‌ها به شما اجازه می‌دهند پرس‌وجوهایی را صادر کنید که موجب سازگار شدن داده‌ها در کل خوشه می‌شوند. بااین‌حال، ایجاد سازگاری داده‌ها به این روش موجب کاهش کارایی سیستم پایگاه داده می‌گردد. تراکنش‌ها نیز به‌صورت خاص می‌توانند موجب به وجود آمدن مشکلات مربوط به کارایی در پایگاه‌های داده‌ی توزیع‌شده شوند، زیرا همه‌ی ماشین‌های خوشه را مجبور به همکاری برای قرنطینه کردن بخش عمده‌ای از داده‌ها می‌کنند.

بین سازگاری و کارایی باید تعادل به وجود آورد. اگر پرس‌وجوهای پایگاه داده‌ی شما نیاز به سازگاری داده‌ها از نوع قوی ندارند، اصطلاحاً تحت **سازگاری مشروط**[3] کار می‌کنند. در این حالت تضمین می‌شود که داده‌ها درنهایت و بعد از گذشت کمی زمان، سازگار خواهند شد. این بدان معنی است که برخی پرس‌وجوهای نوشتن نباید اعمال شوند، و برخی پرس‌وجوهای خواندن اطلاعاتی منقضی شده برمی‌گردانند.

در بسیاری حالات، کار کردن با سازگاری مشروط مشکلی به وجود نمی‌آورد. به‌عنوان نمونه، اگر در بخش مربوط به یک محصول که به‌صورت آنلاین فروخته می‌شود، تعداد نظرات کاربران به‌جای ۲۸۵ مورد، ۲۸۴ مورد نشان داده شود مشکلی به وجود نخواهد آمد؛ زیرا یکی از نظرات به‌تازگی ثبت‌شده است.

---

[1] Data Consistency
[2] Data Inconsistency
[3] Eventual Consistency

٦-٤- جغرافیایی

بسیاری از پایگاه‌های داده اطلاعات جغرافیایی را مانند مکان شهرها، یا چندضلعی‌های تعیین‌کننده‌ی مرزهای ایالات ذخیره می‌کنند. برنامه‌های حمل‌ونقل نیاز دارند از وضعیت اتصال جاده‌ها، ریل‌ها و ایستگاه‌ها به یکدیگر مطلع باشند. اداره آمار باید شکل کارتوگرافی هزاران قطعه‌ی سرشماری را همراه با داده‌های سرشماری جمع‌آوری‌شده در هر بخش ذخیره کند.

در این پایگاه‌های داده پرس‌وجوی اطلاعات فضایی استفاده‌ی زیادی دارد. به‌عنوان نمونه، اگر شما مسئول یک سرویس خدمات اضطراری پزشکی هستید، به یک پایگاه داده از محل بیمارستان‌ها در ناحیه نیاز دارید. سیستم پایگاه داده‌ی شما باید بتواند سریعاً به سؤال نزدیک‌ترین بیمارستان به یک مکان مفروض کدام است؟ جواب دهد.

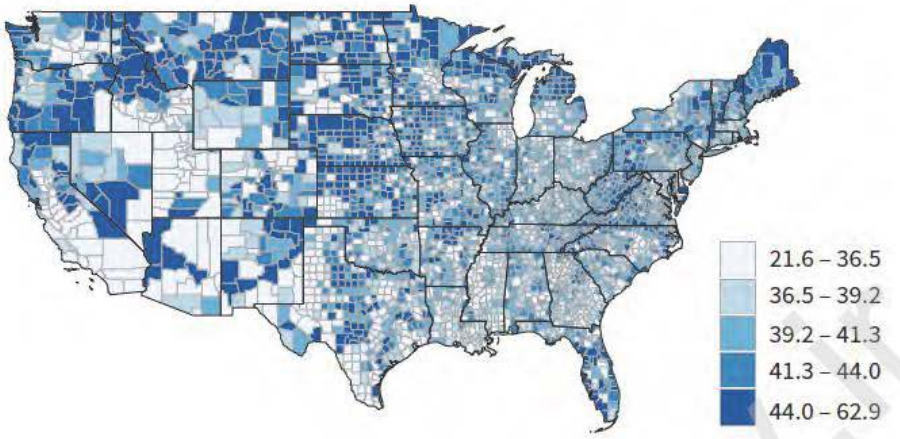
این برنامه‌ها موجب بروز سیستم‌های پایگاه داده‌ی خاصی شده‌اند که به نام سیستم‌های اطلاعاتی جغرافیایی[1] معروف هستند. این سیستم‌ها فیلدهای خاصی را که برای داده‌های جغرافیایی طراحی‌شده‌اند ارائه می‌کنند: فیلد مکان، فیلد خط، فیلد چندضلعی و غیره. هم‌چنین این سیستم‌ها می‌توانند پرس‌وجوهای فضایی را بر روی این فیلدها انجام دهند. در یک پایگاه داده‌ی GIS از رودخانه‌ها و شهرها می‌توانید پرس‌وجوهایی را مانند «شهرهایی که در فاصله‌ی ۱۰ مایلی رودخانه‌ی می‌سی‌سی‌پی هستند را به‌صورت مرتب‌شده بر اساس جمعیت فهرست کن» به‌صورت مستقیم انجام دهید. سیستم‌های GIS می‌توانند از شاخص‌های فضایی نیز استفاده کنند، بنابراین جستجو بر اساس تقریب فضایی بسیار کارا است.

این سیستم‌ها حتی به شما اجازه می‌دهند قیدهای فضایی را نیز تعریف کنید. به‌عنوان نمونه، در جدولی که قطعات زمین ذخیره‌شده‌اند می‌توانید این قید را مطرح کنید که هیچ دو قطعه‌ی زمینی نباید باهم همپوشانی داشته و یک بخش مشترک از زمین را اشغال کنند. این کار می‌تواند سازمان‌های مرتبط با ثبت زمین را از دردسرهای بزرگی خلاص کند.

بسیاری از سیستم‌های مدیریت پایگاه داده‌ی همه‌منظوره الحاقاتی برای GIS نیز دارند. وقتی در حال کار با داده‌های جغرافیایی هستید، مطمئن شوید از موتور پایگاه داده‌ای استفاده کنید که از GIS پشتیبانی می‌کند، و از ویژگی‌های آن برای انجام پرس‌وجوهای هوشمندانه‌تر بهره بگیرید. برنامه‌های GIS اغلب در زندگی روزمره استفاده می‌شوند، به‌عنوان نمونه می‌توان از برنامه‌های ناوبری GPS مانند Google Map یا Waze نام برد.

---

[1] Geographical Information Systems (GIS)



شکل ٦-١٢: میانه‌ی سن در ایالات‌متحده آمریکا (داده‌ها از census.gov)

٦-٥- قالب‌های سریال‌سازی

چگونه می‌توانیم داده‌ها را در بیرون از پایگاه و در قالبی که در سیستم‌های مختلف قابل‌تبادل و استفاده باشد ذخیره کنیم؟ به‌عنوان نمونه، ممکن است بخواهیم از داده‌ها پشتیبان گرفته، یا آن‌ها را وارد سیستم دیگری بنماییم. برای انجام این کار، داده‌ها باید وارد فرایندی به نام **سریال‌سازی**[1] شوند، که در آن بر اساس یک قالب کدگذاری تغییر شکل داده می‌شوند. فایل حاصل به‌وسیله‌ی هر سیستمی که از آن قالب کدگذاری پشتیبانی می‌کند، قابل‌درک است. اجازه بدهید برخی از قالب‌های کدگذاری معمول مورداستفاده در سریال‌سازی را بررسی کنیم.

قالب SQL: این قالب معمول‌ترین قالب سریال‌سازی برای پایگاه‌های داده‌ی رابطه‌ای است. در این رابطه، دنباله‌ای از دستورات SQL می‌نویسیم که یک نسخه از پایگاه داده و تمام جزئیات آن را ارائه می‌دهند. اغلب سیستم‌های پایگاه داده‌ی رابطه‌ای دارای یک دستور dump برای ایجاد یک فایل سریال‌سازی‌شده SQL از پایگاه داده‌ی شما هستند. این سیستم‌ها هم‌چنین شامل دستوری به نام restore برای بارگذاری فایل حاصل از dump در سیستم پایگاه داده هستند.

قالب XML: این قالب راه دیگری برای نمایش داده‌های ساخت‌یافته است، ولی به مدل رابطه‌ای یا یک پیاده‌سازی از سیستم‌های پایگاه داده وابسته نیست. قالب XML برای ایجاد همکاری بین سیستم‌های محاسباتی مختلف، و تشریح ساختار و پیچیدگی داده‌ها ایجادشده است. برخی مردم می‌گویند XML به‌وسیله‌ی افراد دانشگاهی که نمی‌دانستند ساخته‌ی دست آن‌ها کاربردی نیست، تولیدشده است.

---

[1] Serialization

قالب JSON: این قالب یک قالب سریال‌سازی است که بیشتر دنیا به آن تمایل دارند. این قالب می‌تواند داده‌های رابطه‌ای و غیررابطه‌ای را به روشی قابل‌درک برای کدنویسان نمایش دهد. الحاقات بسیاری برای JSON وجود دارند: BSON (مدل دودویی JSON) که بیشترین کارایی را برای JSON در پردازش داده‌ها به همراه دارد؛ JSON-LD که قدرت ساختار XML را به JSON هدیه می‌کند.

قالب CSV یا مقادیر جداشده با کاما[1]: این قالب ساده‌ترین قالب برای تبادل داده‌ها است. داده‌ها در این قالب به شکل متنی ذخیره می‌شوند، به‌طوری‌که هر عنصر داده‌ای در یک خط آورده می‌شود. ویژگی‌های هر عنصر داده‌ای با کاما، یا هر کاراکتر دیگری که در داده‌ها وجود ندارد، از هم جدا می‌شوند. قالب CSV برای ایجاد نسخه از داده‌های ساده بسیار مفید، ولی برای نمایش داده‌های پیچیده استفاده از آن سخت است.

نتیجه‌گیری

در این فصل آموختیم که ساختاردهی اطلاعات در یک پایگاه داده برای استفاده‌ی مفید از آن‌ها اهمیت دارد. راه‌های مختلفی را برای این کار یاد گرفتیم. دیدیم که مدل رابطه‌ای چگونه داده‌ها را به جدول‌هایی تقسیم می‌کند و چگونه به کمک روابط مجدداً بین آن‌ها پیوند به وجود می‌آورد.

اغلب کدنویسان تنها کار با پایگاه‌های داده‌ی رابطه‌ای را یاد گرفته‌اند، ولی ما از آن فراتر رفتیم. راه‌های جایگزین غیررابطه‌ای را برای ساختاردهی داده‌ها دیدیم. در مورد مشکلات سازگاری داده‌ها و چگونگی اجتناب از آن‌ها به کمک تراکنش‌ها بحث کردیم. در رابطه با نحوه‌ی مقیاس‌دهی به سیستم‌های پایگاه داده برای مدیریت بارهای زیاد با استفاده از پایگاه‌های داده‌ی توزیع‌شده صحبت کردیم. سیستم‌های GIS و ویژگی‌های ارائه‌شده به‌وسیله‌ی آن‌ها را برای کار با داده‌های جغرافیایی ارائه کردیم. و راه‌های معمول برای تبادل داده‌ها بین برنامه‌های مختلف را نشان دادیم.

درنهایت، سعی کنید یک سیستم مدیریت پایگاه داده را که به‌صورت گسترده مورداستفاده قرار می‌گیرد انتخاب کنید، مگر این‌که در حال کسب تجربه باشید. چنین سیستمی خطاهای کمتر و کارایی بهتری دارد. هیچ نسخه‌ی شفابخش واحدی برای انتخاب سیستم پایگاه داده وجود ندارد. هیچ سیستم مدیریت پایگاه داده‌ی خاصی وجود ندارد که برای هر سناریویی بهترین گزینه باشد. با خواندن این فصل، باید با انواع مختلف سیستم‌های مدیریت پایگاه داده و ویژگی‌های آن‌ها آشنا شده باشید، به‌گونه‌ای که بتوانید به‌صورت آگاهانه از بین آن‌ها انتخاب کنید.

---

[1] Comma Separated Values (CSV)

مراجع

- Database System Concepts, by Silberschatz
    - Get it at https://code.energy/silber
- NoSQL Distilled, by Sadalage
    - Get it at https://code.energy/sadalage
- Principles of Distributed Database Systems
    - by Özsu, Get it at https://code.energy/ozsu

فصل ۷

کامپیوترها

هر فناوری که به‌اندازه‌ی کافی پیشرفت کرده باشد،
متمایز از جادو است.
- آرتور سی. کلارک[۱]

ماشین‌های متفاوت بسیاری برای حل مسائل اختراع‌شده‌اند. انواع متعددی از کامپیوترها وجود دارند، از کامپیوترهای تعبیه‌شده در ربات‌های جستجوگر مریخ تا کامپیوترهای کنترل‌کننده‌ی سیستم‌های ناوبری زیردریایی‌های هسته‌ای. تقریباً همه‌ی کامپیوترها، شامل لپ‌تاپ‌ها و تلفن‌های ما، قوانین عملکردی مشابهی با قوانین اولین مدل اختراع‌شده به‌وسیله‌ی فون‌نیومان[۲] در سال ۱۹۴۵ دارند. آیا می‌دانید کامپیوترها در زیر پوششی که دارند چگونه کار می‌کنند؟ در این فصل یاد می‌گیرد:

- مفاهیم بنیادین **معماری** کامپیوترها را درک کنید.
- یک **کامپایلر** برای ترجمه‌ی کد خودتان به زبان کامپیوترها انتخاب کنید.
- با **سلسله‌مراتب حافظه** سرعت را در قبال فضای ذخیره‌سازی به دست بیاورید.

در کل، کدنویسی برای افراد غیر کدنویس شبیه به جادو است، نه ما.

۷-۱- معماری

کامپیوتر ماشینی است که دستورات را برای کار با داده‌ها دنبال می‌کند. این ماشین دو مؤلفه‌ی اصلی دارد: پردازنده و حافظه. حافظه یا RAM[۳] جایی است که در آن دستورات را می‌نویسیم. این بخش هم‌چنین

---

[۱] Arthur C. Clarke (1917-2008): نویسنده، مخترع و دانشمند بریتانیایی که رمان معروف علمی تخیلی اودیسه‌ی فضایی ۲۰۰۱ از آثار اوست. وی دارنده‌ی ۷ مدرک دکترای افتخاری در رشته‌های مختلف بود.م.

[۲] John von Neumann (1903-1957): ریاضیدان و دانشمند آمریکایی که سهم بزرگی در توسعه‌ی رشته‌های متعددی مانند ریاضیات، فیزیک، علوم کامپیوتر و آمار داشت. وی از کسانی بود که در طراحی و ساخت نخستین کامپیوتر به نام انیاک نقش داشتند. م.

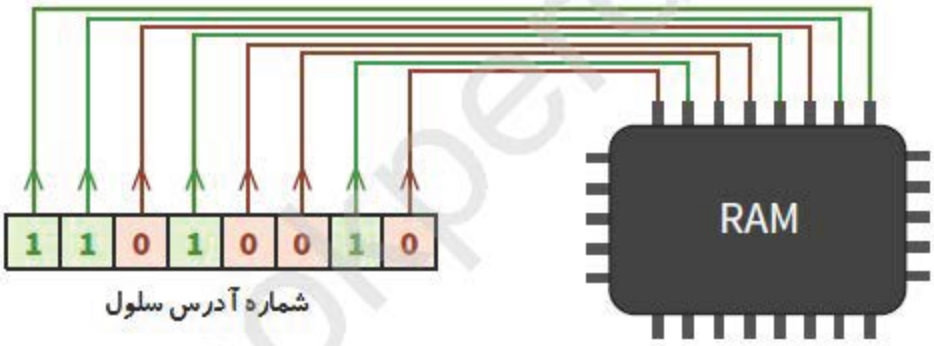
[۳] Random Access Memory (RAM)

داده‌ها را برای انجام عملیات ذخیره می‌کند. پردازنده یا CPU[1] دستورات و داده‌ها را از حافظه دریافت کرده و محاسبات مرتبط با آن‌ها را انجام می‌دهد. اجازه بدهید ببینیم این دو مؤلفه چگونه کار می‌کنند.

حافظه

حافظه به سلول‌های متعددی تقسیم می‌شود. هر سلول حجم کوچکی از داده‌ها را ذخیره کرده و یک آدرس عددی دارد. خواندن یا نوشتن داده‌ها در حافظه از طریق عملیاتی که در هرلحظه بر روی یک سلول تأثیر می‌گذارند، انجام می‌گیرد. برای خواندن یا نوشتن یک سلول خاص، باید آدرس عددی آن را ارائه دهیم.

به دلیل آن‌که حافظه یک مؤلفه‌ی الکتریکی است، آدرس سلول را از طریق سیم‌ها و اعداد دودویی منتقل می‌کنیم[2]. هر سیم یک رقم دودویی را منتقل می‌کند. سیم‌هایی که دارای ولتاژ بالاتری باشند، نماد «یک» و ولتاژ پایین‌تر نماد «صفر» را منتقل می‌کنند.



شکل ۷-۱: اطلاع دادن به RAM برای کار کردن با سلول شماره ۲۱۰ (11010010)

حافظه با گرفتن آدرس یک سلول دو کار می‌تواند انجام دهد: مقدار آن را برگرداند، یا یک مقدار جدید ذخیره کند. حافظه یک سیم ورودی ویژه برای تعیین نوع عملیات دارد (شکل ۷-۲).

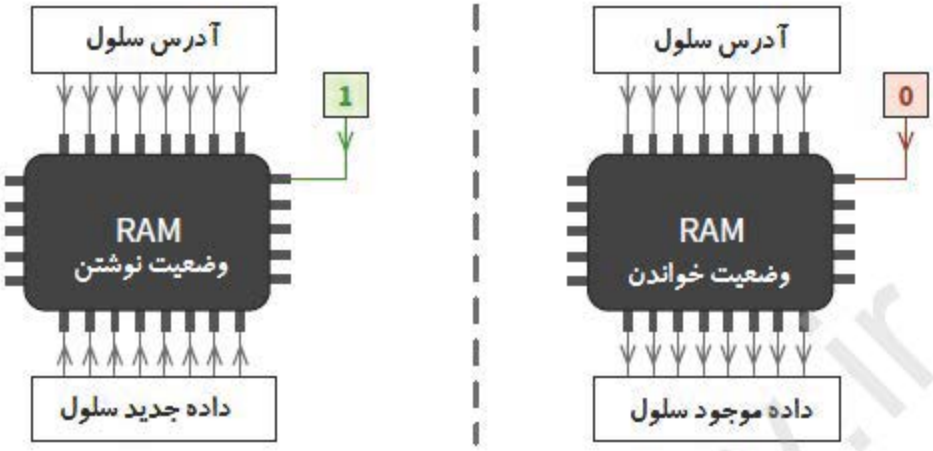
هر سلول حافظه یک عدد دودویی ۸ رقمی را ذخیره می‌کند که یک **بایت**[3] نامیده می‌شود. در وضعیت «خواندن»، حافظه بایت ذخیره‌شده در سلول را بازیابی کرده و آن را از طریق سیم‌های انتقال داده به‌عنوان خروجی برمی‌گرداند (شکل ۷-۳).

---

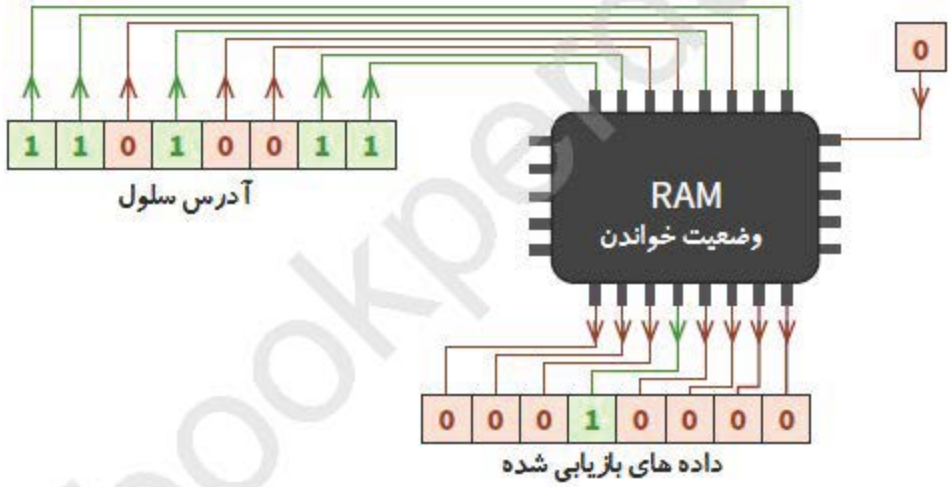
[1] Central Processing Unit (CPU)

[2] اعداد دودویی در مبانی ۲ بیان می‌شوند. پیوست I چگونگی این کار را شرح می‌دهد.

[3] Byte



شکل ۷-۲: حافظه می‌تواند در وضعیت خواندن یا نوشتن عمل کند



شکل ۷-۳: خواندن عدد ۳۲ از آدرس ۲۱۱ حافظه

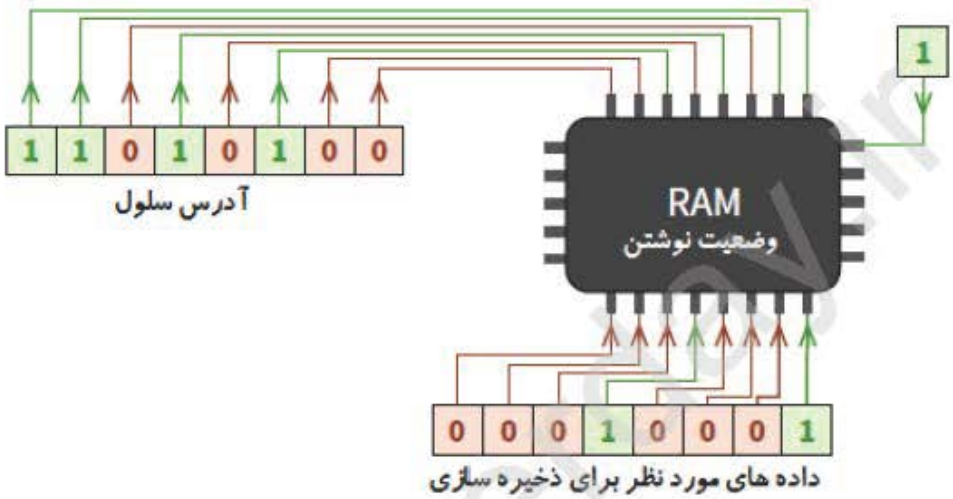
وقتی حافظه در وضعیت «نوشتن» قرار دارد، یک بایت را از این سیم‌ها دریافت کرده و آن را در سلول مشخص‌شده می‌نویسد (شکل ۷-۴).

گروهی از سیم‌ها را که برای انتقال داده‌هایی واحد استفاده می‌شوند یک **باس**[1] می‌نامند. هشت سیمی را که برای انتقال آدرس استفاده می‌شوند **باس آدرس** می‌نامند. هشت سیم دیگر را که برای انتقال

---

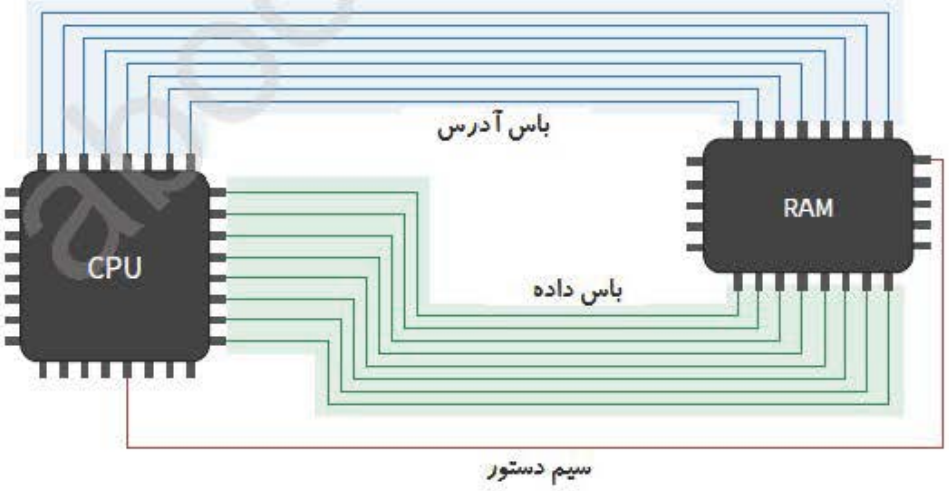
[1] Bus

داده‌ها به بیرون و درون سلول حافظه استفاده می‌شوند **باس داده** می‌نامند. در حالی باس آدرس یک‌طرفه است (فقط برای دریافت داده‌ها استفاده می‌شود)، باس داده دوطرفه است (برای ارسال و دریافت داده‌ها استفاده می‌شود).



شکل ۷- ٤: نوشتن عدد ۳۳ در آدرس ۲۱۲ حافظه

در هر کامپیوتر، CPU و RAM به‌صورت مداوم در حال تبادل داده‌ها هستند: CPU مستمراً دستورات و داده‌ها را از حافظه واکشی کرده و خروجی و محاسبات جزئی متناسب را در آن ذخیره می‌کند.



شکل ۷- ۵: سیم‌کشی CPU به RAM

CPU

مؤلفه‌ی CPU دارای سلول‌های حافظه‌ی درونی است که **ثبات**[۱] نام دارند. با استفاده از این ثبات‌ها، CPU می‌تواند عملیات ساده‌ی ریاضی را انجام دهد. هم‌چنین امکان تبادل داده‌ها بین RAM و این ثبات‌ها وجود دارد. موارد زیر نمونه‌هایی از عملیات قابل‌اجرا توسط CPU هستند:

- کپی کردن داده‌ها از حافظه‌ی شماره‌ی ۲۲۰ در ثبات شماره‌ی ۳.
- اضافه کردن عدد موجود در ثبات شماره‌ی ۳ به عدد موجود در ثبات شماره‌ی ۱.

مجموعه‌ی تمام دستوراتی را که یک CPU می‌تواند انجام دهد **مجموعه‌ی دستورالعمل**[۲] می‌نامند. به هر عملیات در مجموعه‌ی دستورالعمل یک شماره اختصاص‌یافته است. کد کامپیوتری الزاماً یک دنباله از اعداد است که عملیات CPU را نمایش می‌دهند. این عملیات در قالب اعداد در RAM ذخیره می‌شوند. ما داده‌های ورودی/خروجی، محاسبات جزئی و کد کامپیوتری را در کنار هم در RAM نگهداری می‌کنیم[۳].

شکل ۷-۶ نشان می‌دهد که بر اساس دستور کار CPU، چگونه برخی از دستورالعمل‌های CPU به اعداد نگاشت می‌شوند. با پیشرفت فناوری تولید CPU، تعداد دستوارت قابل پشتیبانی در آن نیز افزایش‌یافته است. امروزه، مجموعه‌ی دستورالعمل CPU بسیار بزرگ است. بااین‌حال، مهم‌ترین عملیات مذکور ده‌ها سال است که وجود دارند.

عملکرد CPU در یک چرخه‌ی بی‌پایان انجام می‌شود، زیرا CPU همیشه در حال واکشی و اجرای یک دستورالعمل از حافظه است. در هسته‌ی این چرخه ثبات PC یا شمارنده‌ی برنامه[۴] قرار دارد[۵]. این ثبات یک ثبات ویژه است که آدرس حافظه‌ی دستور بعدی را که باید اجرا شود، نگهداری می‌کند. در این راستا، CPU کارهای زیر را انجام می‌دهد:

۱. واکشی دستور در آدرس حافظه‌ی داده‌شده توسط PC.
۲. اضافه کردن PC به میزان ۱ واحد.
۳. اجرای دستورالعمل.
۴. برگشت به مرحله‌ی ۱.

---

[۱] Register

[۲] Instruction Set

[۳] کد می‌تواند خودش را با اضافه کردن دستورالعمل‌هایی که موجب بازنویسی بخشی از کد در RAM می‌شوند، اصلاح کند. ویروس‌های کامپیوتری اغلب این کار را انجام می‌دهند تا تشخیصشان توسط نرم‌افزارهای ضدویروس سخت‌تر شود. این امر همسانی شگفت‌انگیزی با ویروس‌های زیستی دارد که DNA خود را تغییر می‌دهند تا از دید سیستم ایمنی میزبان مخفی شوند.

[۴] Program Counter (PC)

[۵] این عبارت را با مخفف آشنای PC به معنای کامپیوتر شخصی (Personal Computer) اشتباه نگیرید.

4004 Instruction Set

BASIC INSTRUCTIONS

MNEMONIC	OPR $D_3 D_2 D_1 D_0$	OPA $D_3 D_2 D_1 D_0$	DESCRIPTION OF OPERATION
NOP	0 0 0 0	0 0 0 0	No operation.
INC	0 1 1 0	R R R R	Increment contents of register RRRR.
ADD	1 0 0 0	R R R R	Add contents of register RRRR to accumulator with carry.
LD	1 0 1 0	R R R R	Load contents of register RRRR to accumulator.
LDM	1 1 0 1	D D D D	Load data DDDD to accumulator.
CLC	1 1 1 1	0 0 0 1	Clear carry.
IAC	1 1 1 1	0 0 1 0	Increment accumulator.
DAC	1 1 1 1	1 0 0 0	Decrement accumulator.

شکل ۷-۶: بخشی از داده‌های Intel 4004 که نشان می‌دهد عملیات چگونه به اعداد نگاشت می‌شوند. این CPU اولین نمونه‌ی عرضه‌شده به بازار در جهان در سال ۱۹۷۱ است.

زمانی که CPU شروع به کار می‌کند، PC به مقدار پیش‌فرض خود مقداردهی می‌شود. این مقدار آدرس اولین دستورالعملی است که باید به‌وسیله‌ی ماشین اجرا شود. این دستورالعمل معمولاً یک برنامه‌ی درونی غیرقابل تغییر است که مسئول بارگذاری عملکردهای پایه‌ای کامپیوتر است[1].

بعد از شروع به کار، CPU این چرخه‌ی واکشی را تا زمان خاموش شدن کامپیوتر ادامه می‌دهد. ولی اگر CPU فقط قادر به دنبال کردن یک فهرست مرتب و ترتیبی از عملیات بود، به یک ماشین‌حساب ساده تبدیل می‌شد. شگفت‌انگیز بودن CPU به‌این‌علت است که می‌توان به آن دستور داد مقدار جدیدی را در PC نوشته و موجب شود اجرا به‌جایی دیگر در حافظه منتقل شود. این انتقال می‌تواند شرطی باشد. به‌عنوان نمونه، یک دستورالعمل CPU می‌تواند بگوید «*مقدار PC را به آدرس شماره‌ی ۲۰۰ تغییر بده اگر ثبات شماره‌ی ۱ برابر صفر است.*» این امر به کامپیوتر اجازه می‌دهد دستوراتی مانند مورد زیر را انجام دهد:

```
if x = 0
    compute_this()
else
    compute_that()
```

این تمام‌کاری است که انجام می‌شود. شما چه یک وب‌سایت را بازکرده، یک بازی را بازی کرده، یا یک صفحه‌ی گسترده را ویرایش کنید، محاسبات همیشه یکسان هستند: یک دنباله از عملیات ساده که می‌توانند فقط جمع، مقایسه، یا جابجایی داده‌ها در نقاط مختلف حافظه باشند.

[1] در بسیاری از کامپیوترهای شخصی، این برنامه را BIOS می‌نامند.

با بسیاری از این عملیات ساده، می‌توانیم رویه‌های پیچیده‌ای را بیان کنیم. به‌عنوان‌مثال، کد بازی قدیمی مهاجمان فضایی[1] حدود ۳۰۰۰ دستورالعمل ماشین داشت.

شکل ۷-۷: مهاجمان فضایی که در سال ۱۹۷۸ عرضه شد، اغلب به‌عنوان تأثیرگذارترین بازی ویدیویی تاریخ شمرده می‌شود

ساعت CPU: در دهه‌ی ۱۹۸۰، مهاجمان فضایی بسیار محبوب شد. مردم این بازی را بر روی ماشین‌های آرکید[2] مجهز به یک CPU از نوع 2MHz بازی می‌کردند. این عدد نشان‌دهنده‌ی **ساعت** CPU است: تعداد عملیات اصلی قابل انجام در ثانیه. با یک ساعت ۲ میلیون هرتزی (2MHz)، CPU می‌تواند تقریباً دو میلیون عملیات اصلی را در ثانیه انجام دهد. یک دستورالعمل ماشین برای کامل شدن نیازمند پنج تا ده عملیات اصلی است؛ بنابراین، ماشین‌های آرکید آن زمان می‌توانستند *صدها هزار* دستورالعمل ماشین را در هر *ثانیه* انجام دهند.

با پیشرفت فناوری مدرن، کامپیوترهای رومیزی معمولی و تلفن‌های هوشمند اغلب دارای CPU از نوع 2GHz هستند. این ماشین‌ها می‌توانند صدها میلیون دستورالعملی ماشین را در هر ثانیه اجرا کنند. چون

---

[1] Space Invaders: یک بازی قدیمی و جزء اولین بازی‌های کامپیوتری فراگیر که در سال ۱۹۷۸ ارائه شد.م.

[2] Arcade Machine: ماشین‌های آرکید در مکان‌های عمومی نصب‌شده و بازیکنان با سکه یا اسکناس آن‌ها را فعال کرده و به بازی می‌پرداختند. این ماشین‌ها با معرفی کنسول‌های بازی خانگی محبوبیت خود را از دست دادند. م.

اخیراً CPUهای دارای چندین هسته این عدد را افزایش داده‌اند. یک CPU چهارهسته‌ای از نوع 2GHz می‌تواند نزدیک به یک میلیارد دستورالعمل ماشین را در هر ثانیه اجرا کند. به نظر می‌رسد CPUها در آینده با هسته‌های هر چه بیشتری تجهیز خواهند شد[1].

معماری‌های CPU

آیا تابه‌حال فکر کرده‌اید چرا نمی‌توانید یک CD پلی‌استیشن را در کامپیوتر رومیزی خود گذاشته و شروع به بازی کنید؟ یا چرا نمی‌توانید برنامه‌های iPhone را روی یک Mac اجرا کنید؟ دلیل این امر بسیار ساده است: معماری‌های متفاوت CPU.

امروزه معماری x86 تقریباً استاندارد است، بنابراین‌یک کد مشابه را می‌توان بر روی اغلب کامپیوترهای شخصی اجرا کرد. بااین‌حال و به‌عنوان‌مثال، تلفن‌های همراه دارای CPUهایی با معماری‌های متفاوت و بسیار کاراتر از بعد مصرف انرژی هستند. تفاوت در معماری CPU به معنی مجموعه دستورالعمل‌های متفاوت است، ازاین‌رو، از راه متفاوتی برای کدگذاری دستورات در قالب اعداد استفاده می‌شود. اعدادی که ترجمه‌ای از دستورالعمل‌های CPU کامپیوتر رومیزی شما به شمار می‌روند، دستورالعمل‌های معتبری برای CPU تلفن همراه شما به‌حساب نمی‌آیند و برعکس.

معماری ۳۲بیتی در مقایسه با ٦٤بیتی: اولین CPU که Intel 4004 نام داشت بر اساس یک معماری ۴بیتی ساخته‌شده بود. این بدان معنی است که این CPU می‌توانست عملیات (جمع، مقایسه و جابجایی) اعداد دودویی را تا ۴ رقم در یک دستورالعمل ماشین انجام دهد. باس‌های داده و آدرس این CPU فقط ۴ سیم به ازای هر مورد داشتند.

کمی بعدازآن، CPUهایی با ۸ بیت به‌صورت گسترده در دسترس قرار گرفتند. این CPUها در کامپیوترهای شخصی اولیه که DOS را اجرا می‌کردند، استفاده می‌شدند[2]. کامپیوتر مخصوص بازی معروف Game Boy در دهه‌های ۱۹۸۰ و ۱۹۹۰ نیز پردازنده‌ی ۸ بیتی داشت. یک دستورالعمل ساده در این CPUها بر روی اعداد دودویی ۸ رقمی قابل‌اجرا است.

پیشرفت سریع فناوری امکان ارائه شدن معماری‌های ۱۶ بیتی و ۳۲ بیتی را فراهم کرد. ثبات‌های CPU آن‌قدر بزرگ شدند که بتوانند اعداد ۳۲ بیتی را در خود جا دهند. ثبات‌های بزرگ‌تر منجر به باس‌های داده و آدرس بزرگ‌تر می‌شوند. یک باس آدرس ۳۲ سیمی اجازه می‌دهد 2^{32} (۴ گیگابایت) سلول حافظه را آدرس‌دهی کنیم.

---

[1] یک CPU با ۱۰۰۰ هسته توسط پژوهشگران در سال ۲۰۱۶ ارائه‌شده است.

[2] سیستم‌عامل دیسکی (Disk Operating System). سیستم‌های عامل را به‌زودی توضیح خواهیم داد.

عطش ما برای بیشتر کردن قدرت محاسبات ادامه دارد. برنامه‌های کامپیوتری پیچیده‌تر شده‌اند و حافظه‌ی بیشتری مصرف می‌کنند. چهار گیگابایت RAM خیلی کم‌ارزش شده است. آدرس‌دهی بیش از ۴ گیگابایت حافظه با آدرس‌های عددی قابل جا دادن در ثبات‌های ۳۲ بیتی قابل انجام نیست. این امر موجب به وجود آمدن معماری ۶۴ بیتی شده که امروزه بسیار گسترش‌یافته است. با CPUهای ۶۴ بیتی اعداد بسیار بزرگ را می‌توان با یک دستورالعمل واحد در CPU پردازش کرد. ثبات‌های ۶۴ بیتی آدرس‌های یک فضای حافظه‌ی غول‌آسا را ذخیره می‌کنند: 2^{64} بیت که بیش از ۱۷ *میلیارد گیگابایت* است.

بیگ‌اِندیان و لیتل اِندیان: برخی طراحان کامپیوتر گمان می‌کنند ذخیره کردن اعداد از چپ به راست در RAM و CPU منطقی است، به این روش **لیتل اِندیان**[۱] می‌گویند. سایر طراحان کامپیوتر ترجیح می‌دهند اعداد را در حافظه از راست به چپ بنویسند که به نام **بیگ اِندیان**[۲] شناخته می‌شوند. دنباله‌ی دودویی `1-1-0-0-0-0-0-0-1` بر اساس نوع اِندیان اعداد متفاوتی را نشان می‌دهد[۳]:

- بیگ اِندیان: $2^7 + 2^1 + 2^0 = 131$.
- لیتل اِندیان: $2^0 + 2^6 + 2^7 = 193$.

بسیاری از CPUهای امروزی لیتل اِندیان هستند، ولی هنوز تعداد زیادی کامپیوتر بیگ اِندیان نیز وجود دارند. اگر داده‌های تولیدشده توسط CPU لیتل اِندیان نیاز به تفسیر در یک CPU بیگ اِندیان داشته باشند، باید روشی را برای جلوگیری از **عدم تطابق اِندیان**[۴] انتخاب کنیم. برنامه‌نویسان به‌صورت دستی اعداد دودویی را دست‌کاری می‌کنند، به‌ویژه در زمان تجزیه‌ی داده‌های خارج‌شده از سوییچ‌های شبکه باید به این نکته توجه داشته باشید. اگرچه امروزه اغلب کامپیوترها لیتل اِندیان هستند، ترافیک اینترنت در قالب بیگ اِندیان استانداردشده است؛ زیرا بسیاری از مسیریاب‌های اولیه‌ی شبکه CPUهایی از نوع بیگ اِندیان داشته‌اند. داده‌های بیگ اِندیان زمانی که در قالب داده‌های لیتل اِندیان خوانده می‌شوند، به‌هم‌ریخته به نظر می‌رسند و برعکس.

---

[۱] Little-Endian

[۲] Big-Endian

[۳] اِندیان استعاره‌ای به اختلاف دو قوم لی‌لی‌پوتی بر سر جهت بریدن سر تخم‌مرغ آبپز درکتاب سفرهای گالیور نوشته‌ی جاناتان سویفت است. یکی از این دوقوم تخم‌مرغ آبپز را به روش سنتی از سر پهنتر باز می‌کردند که به **بیگ اِندیان** معروف بودند و قوم دیگر معتقد به بریدن سر کوچکتر تخم‌مرغ بودند که به ایشان **لیتل اِندیان** گفته می‌شد. م.

[۴] Endianness Mismatch

امولاتور: گاهی اوقات اجرا کردن کدی که برای یک CPU متفاوت طراحی‌شده بر روی کامپیوتر خودتان مفید است. با این روش شما می‌توانید یک برنامه‌ی iPhone را بدون داشتن iPhone آزمایش کرده، یا بازی موردعلاقه‌تان را که مخصوص سوپر نینتندو[۱] است بازی کنید. برای انجام این کارها نرم‌افزارهایی به نام **امولاتور**[۲] وجود دارند.

یک امولاتور رفتار ماشین هدف را تقلید می‌کند: درواقع امولاتور وانمود می‌کند که همان CPU، RAM و سایر سخت‌افزارها را دارد. دستورالعمل‌ها به‌وسیله‌ی برنامه‌ی امولاتور کدگشایی‌شده و در ماشین امولاتور اجرا می‌شوند. همان‌گونه که می‌توانید تصور کنید، تقلید از یک ماشین درون یک ماشین دیگر وقتی‌که معماری‌های متفاوتی دارند بسیار پیچیده، ولی ممکن است. شما می‌توانید یک امولاتور برای گیم بوی[۳] تهیه کرده و کامپیوتر خودتان را وادار سازید که یک گیم بوی مجازی بسازد. به این طریق می‌توانید بازی‌ها را دقیقاً همانند یک گیم بوی فیزیکی بازی کنید.

۷-۲- کامپایلرها

ما می‌توانیم کامپیوترها را برای انجام MRI، شناسایی صدای خودمان، کاوش سایر سیاره‌ها و انجام بسیاری وظایف پیچیده‌ی دیگر برنامه‌ریزی کنیم. قابل‌ذکر است هر کاری که یک کامپیوتر می‌تواند انجام دهد، درنهایت به‌وسیله‌ی دستورالعمل‌های ساده‌ی CPU که فقط شامل جمع و مقایسه اعداد هستند، انجام می‌شود. برنامه‌های پیچیده‌ی کامپیوتری مانند مرورگرهای وب، نیازمند چندین میلیون یا میلیارد دستورالعمل ماشین هستند.

ولی ما به‌ندرت برنامه‌های خود را به‌صورت مستقیم در قالب دستورالعمل‌های CPU می‌نویسیم. نوشتن یک بازی کامپیوتری واقع‌گرایانه‌ی سه‌بعدی به این روش برای یک انسان تقریباً غیرممکن است. برای این‌که بتوانیم دستوراتمان را به روشی «طبیعی‌تر» و فشرده‌تر بیان کنیم، **زبان‌های برنامه‌نویسی**[۴] را به وجود آورده‌ایم. ما کدهای خود را با این زبان‌ها می‌نویسیم[۵]. سپس از یک برنامه به نام **کامپایلر**[۶] برای ترجمه‌ی دستوراتمان به دستورالعمل‌های ماشین قابل‌اجرا به‌وسیله‌ی یک CPU استفاده می‌کنیم.

---

[۱] Super Nintendo: کنسول ۱۶ بیتی که در سال‌های ۱۹۹۰ تا ۱۹۹۳ عرضه می‌شد. م.

[۲] Emulator

[۳] Game Boy: یک کنسول بازی دستی با صفحه نمایش سیاه و سفید که توسط شرکت نینتندو در سال ۱۹۸۹ ساخته و عرضه شد. بیش از ۷۵ میلیون نمونه از این کنسول در دنیا به فروش رفت. م.

[۴] Programming Languages

[۵] در فصل بعد مطالب بیشتری در مورد زبان‌های برنامه‌نویسی یاد خواهیم گرفت.

[۶] Compiler

برای این‌که کار یک کامپایلر را توضیح دهیم، یک مثال ساده‌ی ریاضی را در نظر بگیرید. اگر قصد داشته باشیم از فردی بخواهیم فاکتوریل پنج را محاسبه کند، می‌توانیم به این صورت سؤال کنیم: $5! = ?$

بااین‌حال، اگر فردی که از او سؤال کرده‌ایم معنی فاکتوریل را نداند، سؤال ما بی‌معنی خواهد بود. باید آن را با عبارات ساده‌تر جایگزین کنیم: $5 \times 4 \times 3 \times 2 \times 1 = ?$

اگر فردی که از او سؤال کرده‌ایم، فقط جمع بلد باشد چه؟ باید عبارت خود را بازهم ساده‌تر کنیم:

$$5+5+5+5+5+5+5+5+5+5+5+5+5+5+$$
$$5+5+5+5+5+5+5+5+5+5+5+5+5=?$$

هم‌زمان با این‌که ما در حال بازنویسی محاسبات خود با قالب‌های ساده و ساده‌تر هستیم، تعداد عملیات موردنیاز آن بیشتر و بیشتر می‌شود. همین اتفاق در مورد کد کامپیوتری نیز می‌افتد. کامپایلر دستورالعمل‌های پیچیده‌ی یک زبان برنامه‌نویسی را به دستورالعمل‌های معادل CPU ترجمه می‌کند. با ترکیب این فرایند با قدرت کتابخانه‌های خارجی، می‌توانیم برنامه‌های پیچیده‌ی شامل میلیاردها دستورالعمل CPU را در چند خط کد که به‌سادگی قابل‌فهم و اصلاح هستند، بنویسیم.

آلن تورینگ[۱]، پدر محاسبات، کشف کرد که ماشین‌های ساده می‌توانند آن‌قدر قدرتمند باشند که هر چیز *محاسبه‌پذیری* را محاسبه کنند. برای این‌که یک ماشین قدرت محاسبات عمومی داشته باشد، باید بتواند برنامه‌ای شامل دستورالعمل‌های زیر را دنبال کند:

- خواندن و نوشتن در حافظه
- انجام دستورات شرطی: اگر یک آدرس حافظه دارای مقدار مفروضی باشد، به نقطه‌ی دیگری از برنامه برود.

ماشین‌هایی را که چنین قدرت محاسباتی جهانی داشته باشند، **تورینگ کامل**[۲] می‌نامند. برای چنین ماشینی میزان پیچیدگی یا سخت بودن محاسبات اهمیتی ندارد، زیرا این ماشین می‌تواند هر محاسباتی را در قالب دستورالعمل‌های ساده‌ی خواندن/نوشتن و دستورات شرطی بیان کند. با داشتن زمان و حافظه‌ی کافی، این دستورالعمل‌ها می‌توانند هر چیزی را محاسبه کنند.

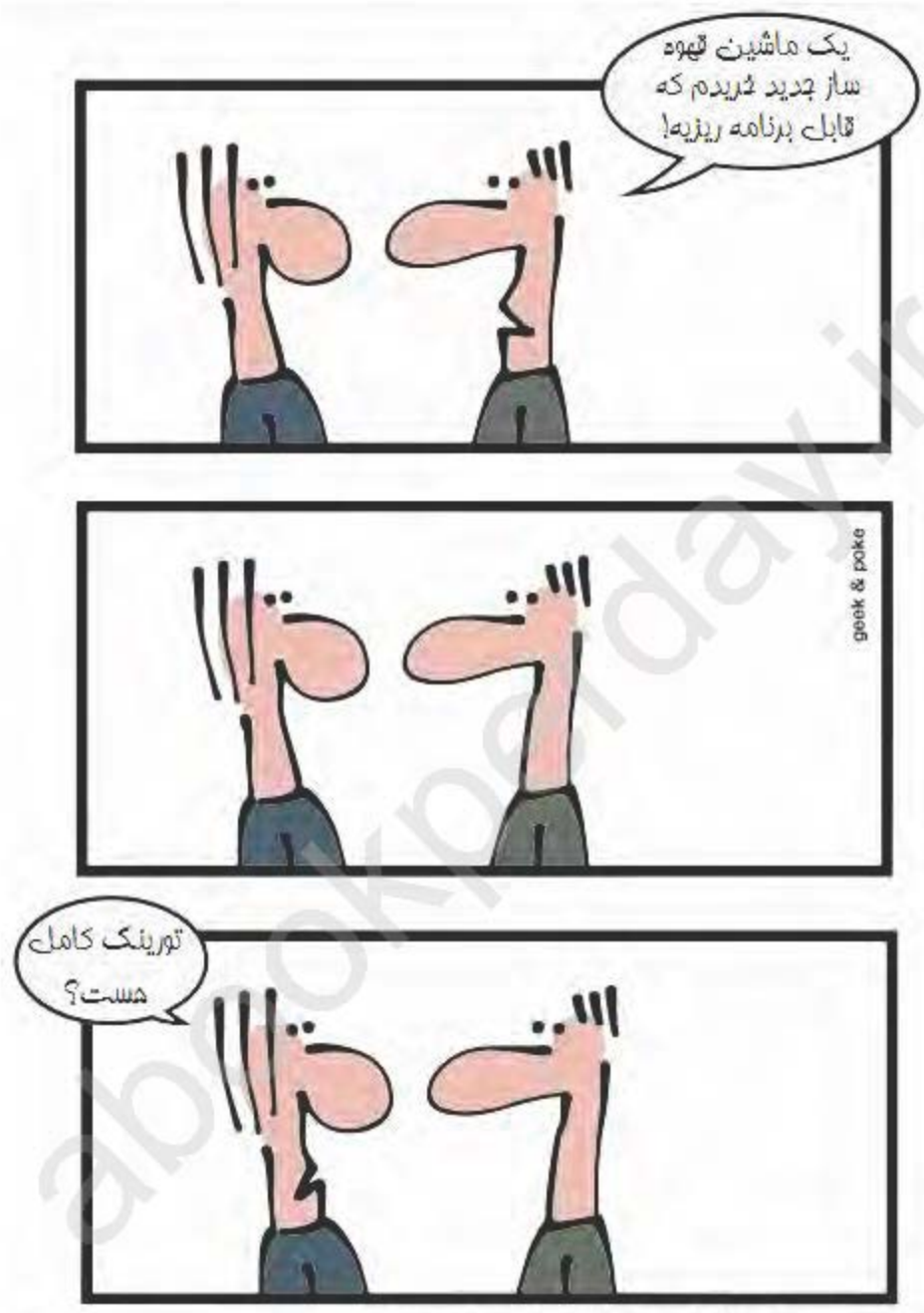
اخیراً نشان داده‌شده است که یک دستورالعمل CPU به نام move (MOV) تورینگ کامل است. این بدان معنی است که یکCPU فقط با انجام دستورالعمل MOV می‌تواند هر کاری را انجام دهد. به‌عبارت‌دیگر، هر نوع کدی را می‌توان با استفاده از دستورالعمل MOV نوشت[۳].

---

[۱] Alan Truing (1912-1954): دانشمند انگلیسی معروف به پدر علوم کامپیوتر و هوش مصنوعی. م.

[۲] Turing-Complete

[۳] این کامپایلر هر کد C را به یک کد دودویی منحصر به دستورالعمل MOV تبدیل می‌کند: https://code.energy/mov.



بعضی از طرفداران فناوری ضدحال هستند

شکل ۷-۸: دریافت شده از http://geek-and-poke.com

مفهوم مهمی که باید در اینجا مطرح شود این است که اگر برنامه‌ای را بتوان در یک زبان برنامه‌نویسی کدگذاری کرد، می‌توان آن را بازنویسی کرد تا در هر ماشین تورینگ کامل، صرف‌نظر از اینکه چقدر ساده است، اجرا شود. کامپایلر یک برنامه‌ی جادویی است که به‌طور خودکار کد را از یک زبان پیچیده به زبان ساده‌تر ترجمه می‌کند.

سیستم‌های عامل

برنامه‌های کامپیوتری کامپایل‌شده اساساً دنباله‌ای از دستورالعمل‌های CPU هستند. همان‌گونه که یاد گرفتیم، کدی که برای یک کامپیوتر رومیزی کامپایل‌شده باشد، بر روی یک تلفن هوشمند اجرا نخواهد شد؛ زیرا این ماشین‌ها دارای CPUهایی با معماری متفاوت هستند. حتی، یک برنامه‌ی کامپایل‌شده ممکن است بر روی دو کامپیوتر با معماری یکسان CPU قابل‌استفاده نباشد. علت این امر آن است که برنامه‌ها برای اجراشدن باید با **سیستم‌عامل**[۱] کامپیوتر تعامل داشته باشند.

برنامه‌ها برای تعامل با دنیا باید ورودی و خروجی داشته باشند: باز کردن فایل‌ها، نوشتن یک پیام بر روی صفحه، باز کردن یک ارتباط شبکه و غیره. ولی کامپیوترهای متفاوت سخت‌افزارهای متفاوت دارند. پشتیبانی مستقیم از تمام انواع صفحه‌ها، کارت‌های صدا یا کارت‌های شبکه برای یک برنامه غیرممکن است.

به همین دلیل برنامه‌ها برای اجراشدن به سیستم‌عامل اتکا می‌کنند. برنامه‌ها می‌توانند با کمک سیستم‌عامل با تلاشی اندک با سخت‌افزارهای مختلف کار کنند. برنامه‌ها **فراخوانی‌هایی سیستمی**[۲] خاصی را ایجاد کرده و از سیستم‌عامل درخواست می‌کنند یک عملیات ورودی/خروجی موردنیاز را انجام دهد. کامپایلرها دستورات ورودی/خروجی را به فراخوانی‌های سیستمی مناسب ترجمه می‌کنند.

بااین‌حال، سیستم‌های عامل متفاوت اغلب از فراخوانی‌های سیستمی ناسازگار استفاده می‌کنند. فراخوانی سیستمی برای چاپ گرفتن از چیزی بر روی صفحه با سیستم عامل ویندوز کاملاً متفاوت از انجام این کار با سیستم‌های عامل مک یا لینوکس است.

به همین دلیل اگر برنامه‌تان را بر روی ویندوز با پردازنده‌ی x86 کامپایل کنید، بر روی مک با یک پردازنده‌ی x86 کار نخواهد کرد. به‌این‌ترتیب کدهای کامپایل‌شده در کنار منحصر بودن به یک معماری خاص CPU، به یک سیستم‌عامل خاص نیز محدود هستند.

---

[۱] Operating System

[۲] System Calls

بهینه‌سازی‌های کامپایلر

کامپایلرهای خوب برای بهینه‌سازی کد ماشین تولید شده، سخت کار می‌کنند. اگر این کامپایلرها ببینند که بخشی از کد شما را می‌توان برای کاراتر شدن تغییر داد، این کار را انجام می‌دهند. کامپایلرها ممکن است صدها نوع بهینه‌سازی را قبل از ساختن خروجی دودویی انجام دهند.

به همین دلیل نیاز نیست خوانایی کد خودتان را به‌منظور بهینه‌سازی ریز و دقیق کاهش دهید. درنهایت، کامپایلر تمام بهینه‌سازی‌های ممکن را انجام خواهد داد. به‌عنوان نمونه، یک نفر ممکن است عقیده داشت باشد که این کد:

```
function factorial(n)
    if n > 1
        return factorial(n - 1) * n
    else
        return 1
```

باید به معادل آن تبدیل شود:

```
function factorial(n)
    result ← 1
    while n > 1
        result ← result * n
        n ← n - 1
    return result
```

بله، انجام محاسبات factorial بدون بازگشت از منابع محاسباتی کمتری استفاده می‌کند. بااین‌حال، هیچ دلیلی برای تغییر کدتان با این هدف وجود ندارد. کامپایلرهای مدرن توابع بازگشتی ساده را به‌صورت خودکار بازنویسی می‌کنند. این یک مثال دیگر است:

```
i ← x + y + 1
j ← x + y
```

کامپایلرها به روش زیر از محاسبه‌ی مجدد x+y اجتناب می‌کنند:

```
t1 ← x + y
i ← t1 + 1
j ← t1
```

بنابراین بر نوشتن برنامه‌های تمیز و قابل‌فهم بدون نیاز به توضیحات بیشتر تمرکز کنید. اگر از بُعد کارایی مشکل دارید، از ابزارهای مخصوص تحلیل برای کشف تنگناهای کدتان استفاده و سعی کنید این بخش‌ها را به روش‌های هوشمندانه‌تری بنویسید. وقتتان را برای ریزمدیریت‌های غیرضروری تلف نکنید.

بااین‌حال، وضعیت‌هایی وجود دارند که ممکن است بخواهیم از مرحله‌ی کامپایل عبور کنیم. اجازه بدهید ببینیم چگونه می‌توان این کار را انجام داد.

زبان‌های اسکریپت‌نویسی

برخی زبان‌های برنامه‌نویسی که آن‌ها را **زبان‌های اسکریپت‌نویسی**[۱] می‌نامند، بدون کامپایل مستقیم به کد ماشین اجرا می‌شوند. این زبان‌ها شامل جاوا اسکریپت[۲]، پایتون[۳] و روبی[۴] هستند. کدهای این زبان‌ها به‌صورت مستقیم توسط CPU اجرا نمی‌شوند، بلکه با استفاده از یک **مفسر**[۵] که بایستی بر روی ماشین موردنظر برای اجرای کد نصب‌شده باشد، اجرا می‌شوند.

به دلیل این‌که مفسر عمل ترجمه‌ی کد را به کد ماشین به‌صورت آنی انجام می‌دهد، معمولاً بسیار کندتر از کدهای کامپایل‌شده اجرا می‌شود. از سوی دیگر، برنامه‌نویس می‌تواند کد را بلافاصله اجرا کرده و منتظر انجام فرایند کامپایل نماند. اگر پروژه خیلی بزرگ باشد، کامپایل ممکن است ساعت‌ها به طول بیانجامد.

مهندسان گوگل باید به‌صورت منظم دسته‌های بزرگ کد را کامپایل کنند. این امر موجب می‌شود کدنویسان زمان زیادی را از دست بدهند (شکل ۷-۹). گوگل نمی‌توانست از زبان‌های اسکریپت‌نویسی استفاده کند، چون به کارایی بالاتر کدهای دودویی نیاز داشت؛ بنابراین زبان Go را، یک زبان‌که به‌صورت شگفت‌انگیزی با سرعت زیادی کامپایل می‌شود ولی همچنان کارایی بالایی دارد، ساختند.

تفکیک اجزا و مهندسی معکوس

با داشتن یک برنامه‌ی کامپیوتری کامپایل‌شده، به دست آوردن کد منبع اصلی آن قبل از کامپایل، غیرممکن است[۶]. ولی می‌توان برنامه‌ی دودویی را کدگشایی کرده و اعداد کدگذاری شده‌ی

---

[۱] Scripting Language

[۲] JavaScript

[۳] Python

[۴] Ruby

[۵] Interpreter

[۶] حداقل فعلاً چنین است. با تکامل هوش مصنوعی ممکن است این کار روزی ممکن شود.

دستورالعمل‌های CPU را به دنباله‌ای از دستورالعمل‌های قابل‌فهم برای انسان تبدیل کرد. این کار را **تفکیک اجزا**[1] می‌نامند.

شکل ۷-۹: کامپایل کردن، دریافت شده از http://xkcd.com

می‌توانیم این دستورالعمل‌های CPU را بررسی کرده و سعی کنیم بفهمیم چه‌کاری انجام می‌دهند، این فرایند را **مهندسی معکوس**[2] می‌نامند. برخی از برنامه‌های تفکیک اجزا با تشخیص خودکار و حاشیه‌نویسی فراخوانی‌های سیستمی و توابع پُراستفاده، کمک زیادی به این فرایند می‌کنند. با استفاده از ابزارهای تفکیک اجزا، یک هکر می‌تواند تمام جنبه‌های عملکرد یک کد دودویی را درک کند. من مطمئن هستم که بسیاری از شرکت‌های بزرگ IT آزمایشگاه‌های مخفی مهندسی معکوس دارند و در آنجا نرم‌افزارهای رقبای خود را تحلیل می‌کنند.

هکرهای زیرزمینی اغلب به تحلیل کد دودویی برنامه‌های دارای مجوز مانند ویندوز، فتوشاپ و غیره می‌پردازند تا بفهمند کدام قسمت کد مجوز مربوطه را ارزیابی می‌کند. سپس کد دودویی را اصلاح کرده و یک دستورالعمل JUMP مستقیم به بخشی از کد که بعد از اعتبارسنجی مجوز قرار دارد، در آن قسمت

---

[1] Disassembly

[2] Reverse Engineering

جا می‌دهند. زمانی که کد اصلاح‌شده اجرا می‌شود قبل از بررسی مجوز به دستور JUMP تزریق‌شده می‌رسد و بنابراین مردم می‌توانند این نرم‌افزارهای غیرقانونی و مسروقه را بدون پرداخت پول اجرا کنند.

محققان و مهندسان امنیتی که برای سازمان‌های جاسوسی مخفی دولتی کار می‌کنند نیز آزمایشگاه‌هایی برای مطالعه‌ی نرم‌افزارهای محبوب و پرکاربرد مانند iOS، ویندوز یا اینترنت اکسپلورر دارند. آن‌ها نقاط ضعف امنیتی موجود در این نرم‌افزارها را شناسایی می‌کنند تا از مردم در مقابل حملات سایبری دفاع کرده یا اهداف ارزشمند را هک کنند. معروف‌ترین حمله از این نوع را می‌توان استاکس‌نت[۱] دانست که یک سلاح سایبری ساخته‌شده به‌وسیله‌ی سازمان‌های آمریکایی و اسرائیلی بود. این سلاح برنامه‌های هسته‌ای ایران را به‌وسیله‌ی آلوده کردن کامپیوترهایی که راکتورهای سوخت هسته‌ای زیرزمینی ایران را کنترل می‌کردند، کند کرد.

نرم‌افزارهای متن‌باز

همان‌گونه که پیش‌تر توضیح داده شد، شما می‌توانید از طریق فایل‌های اجرایی دودویی، دستورالعمل‌های خام برنامه‌های موردنظر را تحلیل کنید، ولی نمی‌توانید کد منبع[۲] اصلی مورداستفاده برای ساخت کد دودویی را به دست بیاورید.

بدون کد منبع اصلی، حتی اگر بتوانید کمی کد دودویی را تغییر دهید تا قسمت‌های کوچکی از آن را هک کنید، ولی انجام تغییرات عمده در برنامه مانند اضافه کردن یک ویژگی جدید، عملاً غیرممکن است. برخی از مردم معتقدند که بهتر است کد را با همکاری یکدیگر بسازند، بنابراین کد منبع خود را در اختیار دیگران قرار می‌دهند تا آن را تغییر دهند. این کار ایده‌ی اصلی متن‌باز[۳] است: نرم‌افزاری که هرکسی بتواند آن را به‌صورت رایگان استفاده کرده یا تغییر دهد. سیستم‌های عامل مبتنی بر لینوکس (مانند اوبونتو[۴]، فدورا[۵]، دبیان[۶]) متن‌باز هستند، درحالی‌که ویندوز و مک[۷] متن‌بسته[۸] هستند.

یک ویژگی جالب سیستم‌های عامل متن‌باز این است که هر کس می‌تواند کد منبع را بررسی کرده و به دنبال نقاط ضعف امنیتی بگردد. هم‌اکنون ثابت‌شده است که سازمان‌های دولتی با استفاده از نقاط

---

۱ Stuxnet
۲ Source Code
۳ Open Source
۴ Ubuntu
۵ Fedora
۶ Debian
۷ Mac OS
۸ Closed Source

ضعف امنیتی برطرف نشده در نرم‌افزارهای مورداستفاده‌ی روزمره به نفوذ و جاسوسی در کامپیوتر میلیون‌ها شهروند پرداخته‌اند.

با استفاده از نرم‌افزارهای متن‌باز چشم‌های بیشتری بر کد نظارت کرده و درج کردن درهای پشتی[۱] نظارتی برای شرکت‌های ثالث و سازمان‌های دولتی با اهداف مخرب سخت‌تر می‌شود. زمانی که شما از ویندوز یا مک استفاده می‌کنید، باید به مایکروسافت و اَپل در رابطه با حفظ امنیتتان و این‌که این شرکت‌ها تمام تلاششان را برای جلوگیری از جریان‌های مخرب امنیتی انجام می‌دهند، اعتماد کنید. سیستم‌های متن‌باز در دسترس تمام افراد متخصص امنیتی قرار داشته، بنابراین شانس کمتری برای پنهان ماندن جریان‌های مخرب امنیتی باقی می‌ماند.

۷-۳- سلسله‌مراتب حافظه

می‌دانیم یک کامپیوتر از طریق اجرای دستورالعمل‌های ساده در CPU خود کار می‌کند. می‌دانیم این دستورالعمل‌ها فقط می‌توانند بر روی داده‌های ذخیره‌شده در ثبات‌های CPU کار کنند. بااین‌حال، فضای ذخیره‌سازی آن‌ها معمولاً به کمتر از هزار بایت محدود است. این یعنی ثبات‌های CPU به‌صورت مداوم باید داده‌ها را به RAM منتقل کرده یا ازآنجا دریافت کنند.

اگر دسترسی به حافظه کند باشد، CPU باید در انتظار انجام کارهایش به‌وسیله‌ی RAM بیکار بماند. مدت‌زمانی که برای خواندن و نوشتن داده‌ها در حافظه صرف می‌شود، مستقیماً بر کارایی کامپیوتر تأثیر دارد. افزایش سرعت حافظه می‌تواند همانند افزایش سرعت CPU موجب تقویت کامپیوتر شما شود. دسترسی به داده‌های درون ثبات‌های CPU تقریباً به‌صورت آنی و فقط در یک سیکل انجام می‌شود[۲]. بااین‌حال، RAM کندتر است.

شکاف پردازنده-حافظه

پیشرفت‌های اخیر فناوری موجب افزایش سرعت CPU به‌صورت نمایی شده‌اند. سرعت حافظه نیز افزایش‌یافته است، ولی با شتاب بسیار کمتری. این شکاف کارایی بین CPU و RAM را **شکاف پردازنده-حافظه**[۳] می‌نامند: دستورالعمل‌های CPU «ارزان» هستند و درنتیجه می‌توان تعداد زیادی از

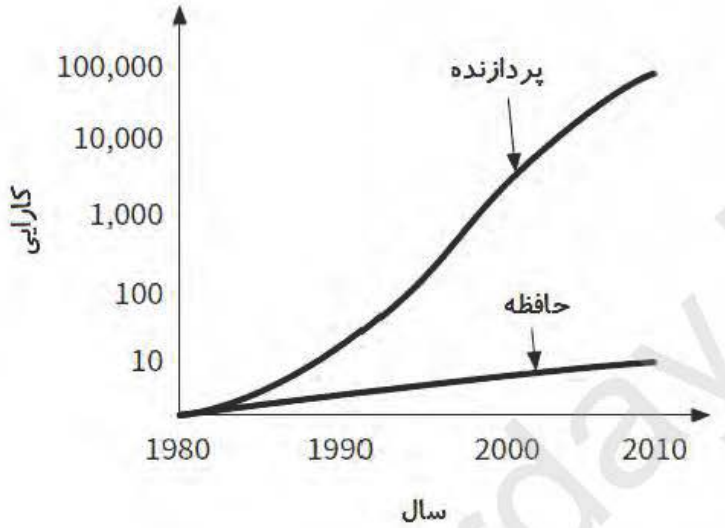
---

[۱] Backdoor

[۲] در یک CPU با ساعت 1GHz، یک سیکل تقریباً یک میلیاردم ثانیه طول می‌کشد، یعنی مدت زمانی که طول می‌کشد تا نور نور از این کتاب به چشم شما برسد.

[۳] Processor-Memory Gap

آن‌ها را انجام داد، درحالی‌که دریافت داده‌ها از RAM زمان بسیار بیشتری به طول می‌انجامد و بنابراین «گران‌تر» است. با عریض‌تر شدن این شکاف، اهمیت دسترسی کارا به حافظه نیز بیشتر شده است.



شکل ۷-۱۰: شکاف پردازنده -حافظه در دهه‌های اخیر

در کامپیوترهای جدید، دریافت داده‌ها از RAM حدود هزار سیکل CPU، تقریباً ۱ میکروثانیه، به طول می‌انجامد[1]. این فرایند به‌صورت شگفت‌انگیزی سریع است ولی در مقایسه بازمان دسترسی CPU به ثبات‌ها بسیار طولانی به نظر می‌رسد. دانشمندان کامپیوتر سعی می‌کنند راه‌هایی برای کاهش تعداد عملیات RAM موردنیاز برای انجام محاسبات پیدا کنند.

موقعیت زمانی و مکانی

در زمان کمینه‌سازی دسترسی به RAM، دانشمندان کامپیوتر به دو نکته توجه دارند:

- **موقعیت زمانی**[2]: زمانی که آدرس حافظه مورد دسترسی قرار می‌گیرد، احتمالاً به‌زودی مجدداً مورد دسترسی قرار خواهد گرفت.
- **موقعیت مکانی**[3]: زمانی که آدرس حافظه مورد دسترسی قرار می‌گیرد، احتمالاً آدرس‌های مجاور آن نیز به‌زودی مورد دسترسی قرار خواهند گرفت.

---

[1] رسیدن امواج صوتی صدای شما به کسی که روبرویتان قرار دارد، حدود ده میکروثانیه طول می‌کشد.

[2] Temporal Locality

[3] Spatial Locality

ذخیره کردن این آدرس‌های حافظه در ثبات‌های CPU خیلی خوب است. این امر از بسیاری از عملیات پرهزینه‌ی RAM جلوگیری می‌کند. بااین‌حال، مهندسان صنعت هیچ راه مناسبی برای طراحی تراشه‌های CPU با ثبات‌های داخلی کافی پیدا نکرده‌اند. بااین‌حال، آن‌ها راهی عالی برای پیدا کردن موقعیت زمانی و مکانی پیداکرده‌اند. بیایید ببینیم چگونه کار می‌کند.

حافظه‌ی نهان L1

ساختن یک حافظه‌ی کمکی بسیار سریع که با CPU تجمیع‌شده باشد، ممکن است. این حافظه را **حافظه‌ی نهان L1**[1] می‌نامند. دریافت کردن داده‌ها از این حافظه و ورود آن‌ها به ثبات‌ها فقط کمی کندتر از دریافت کردن داده‌ها از خود ثبات‌ها است.

با یک حافظه‌ی نهان L1 می‌توانیم محتوای آدرس‌های حافظه با احتمال دسترسی زیاد را درجایی نزدیک به ثبات‌های CPU ذخیره کنیم. به این روش، این محتوا به‌سرعت در ثبات‌های CPU بارگذاری می‌شود. واردکردن داده‌ها از حافظه‌ی نهان L1 به ثبات‌ها فقط حدود ده سیکل CPU طول می‌کشد. این فرایند صدبار سریع‌تر از واکشی داده‌ها از RAM است.

با حدود 10K حافظه‌ی نهان L1 و به‌کارگیری هوشمندانه‌ی موقعیت زمانی و مکانی، تقریباً نصف فراخوانی‌های دسترسی به RAM فقط توسط حافظه‌ی نهان انجام می‌شوند. این ابداع، فناوری محاسبات را دگرگون کرد. تجهیز کردن یک CPU با یک حافظه‌ی نهان L1 زمان انتظار برای داده‌ها را به‌شدت کاهش می‌دهد. به‌این‌ترتیب، CPU زمان بسیار بیشتری برای انجام محاسبات واقعی نسبت به زمان بیکار بودن خواهد داشت.

حافظه‌ی نهان L2

افزایش اندازه‌ی حافظه‌ی نهان L1 موجب می‌شود عملیات واکشی داده‌ها از RAM به‌ندرت انجام شود و درنتیجه زمان انتظار CPU کاهش یابد. بااین‌حال، رشد حافظه‌ی نهان L1 بدون کند ساختن آن مشکل است. بعد از این‌که اندازه‌ی حافظه‌ی نهان L1 به حدود 50k رسید، افزایش بیشتر آن بسیار گران خواهد بود. راه‌حل بهتر، ساختن یک حافظه‌ی نهان اضافی است: **حافظه‌ی نهان L2**[2]. به دلیل این‌که این حافظه می‌تواند کندتر باشد، می‌تواند ازلحاظ اندازه بسیار بزرگ‌تر از حافظه‌ی نهان L1 باشد. یک CPU مدرن

---

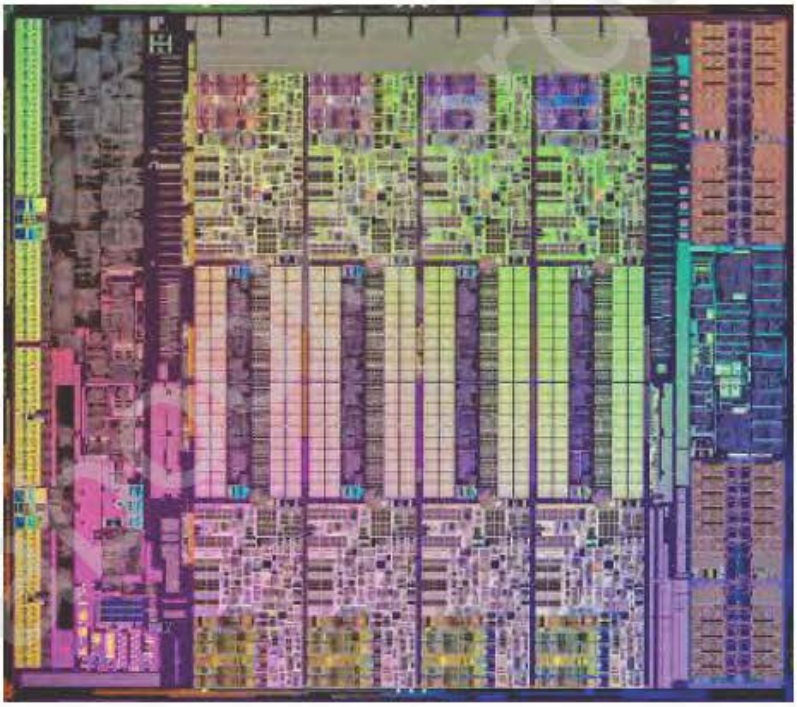
[1] L1 Cache

[2] L2 Cache

حدود 200KB حافظه‌ی نهان L2 دارد. واردکردن داده‌ها از حافظه‌ی نهان L2 به ثبات‌های CPU حدود صد سیکل CPU طول می‌کشد.

آدرس‌هایی را که به‌احتمال بسیار زیاد باید مورد دسترسی قرار بگیرند در حافظه‌ی نهان L1 کپی می‌کنیم. فضاهای حافظه که به احتمال نسبتاً زیاد مورد دسترسی قرار می‌گیرند در حافظه‌ی نهان L2 کپی می‌شوند. اگر یک آدرس حافظه در حافظه‌ی نهان L1 کپی نشده باشد، CPU می‌تواند حافظه‌ی نهان L2 را بررسی کند. تنها درصورتی که در هیچ‌یک از حافظه‌های نهان نباشد، باید به RAM مراجعه کند.

بسیاری از تولیدکنندگان در حال حاضر پردازنده‌هایی را با حافظه‌ی نهان L3 تولید می‌کنند: بزرگ‌تر و کندتر از L2، اما همچنان سریع‌تر از RAM. حافظه‌های نهان L1/L2/L3 بسیار مهم هستند و بیشتر فضای سیلیکونی داخل تراشه‌ی CPU را اشغال می‌کنند.



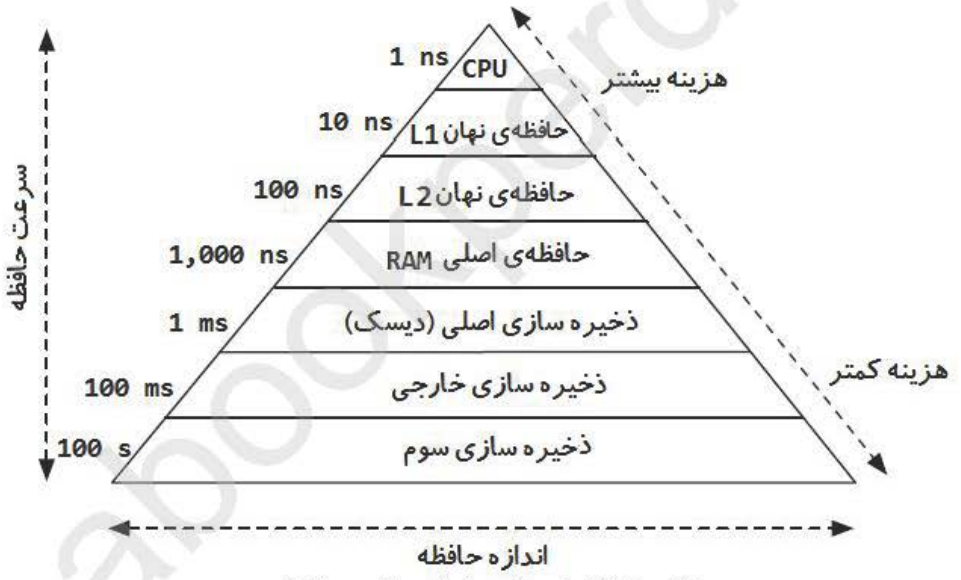
شکل ۷-۱۱: تصویر میکروسکوپی از یک پردازنده‌ی Intel Haskell-E. ساختارهای مربعی در مرکز تصویر 20MB حافظه‌ی نهان L3 هستند.

استفاده از حافظه‌های نهان L1/L2/L3 عملکرد کامپیوترها را به‌طور چشمگیری بهبود می‌دهد. با حافظه‌ی نهان L2 به‌اندازه‌ی 200KB، کمتر از ۱۰ درصد از درخواست‌های حافظه‌ی ارسال‌شده توسط CPU باید مستقیماً از RAM دریافت شوند.

بار دیگری که برای خرید کامپیوتر می‌روید، به یاد داشته باشید که اندازه‌های مختلف حافظه‌ی نهان L1/L2/L3 پردازنده‌های موردنظر خود را مقایسه کنید. هر چه حافظه‌ی نهان بیشتر باشد، CPU بهتر خواهد بود. اغلب بهتر است یک CPU با ساعت کندتر ولی حافظه‌ی نهان بیشتر بخرید.

حافظه‌ی اولیه در مقایسه با حافظه‌ی ثانویه

همان‌گونه که می‌دانید، یک کامپیوتر انواع مختلفی حافظه دارد که در یک سلسله‌مراتب قرارگرفته‌اند. حافظه‌های با بیشترین کارایی ازلحاظ اندازه محدود و گران هستند. با حرکت به سمت پایین در سلسله‌مراتب، فضای حافظه‌ی بیشتری در دسترس است، ولی سرعت نیز کمتر می‌شود.



شکل ۷-۱۲: نمودار سلسله‌مراتب حافظه

بعد از ثبات‌ها و حافظه‌ی نهان CPU، در سلسله‌مراتب حافظه RAM قرار می‌گیرد. این حافظه مسئول ذخیره‌ی داده‌ها و کد تمامی فرایندهای در حال اجرا است. در سال ۲۰۱۷، یک کامپیوتر معمولاً ۱ تا ۱۰ گیگابایت رم دارد. در بسیاری از موارد، این مقدار ممکن است برای متناسب بودن با سیستم‌عامل کامپیوتر و سایر برنامه‌های در حال اجرا کافی نباشد.

در این موارد، ما باید در سلسله‌مراتب حافظه پایین‌تر برویم و از **دیسک سخت**[۱] استفاده کنیم. در سال ۲۰۱۷، کامپیوترها معمولاً دیسک‌های سخت صدها گیگابایتی دارند که بیش‌ازاندازه‌ی کافی برای جا دادن داده‌های همه‌ی برنامه‌های در حال اجرا است. در صورت پر شدن RAM، داده‌های بی‌کار آن را به دیسک سخت منتقل می‌کنیم تا مقداری حافظه آزاد شود.

مشکل این است که دیسک‌های سخت *بسیار* کند هستند. به‌طورمعمول، یک‌میلیون سیکل CPU یا یک میلی‌ثانیه[۲] طول می‌کشد تا داده‌ها بین دیسک و RAM جابجا شوند. ممکن است به نظر برسد که دسترسی به داده‌ها از دیسک سریع است، اما به یاد داشته باشید: دسترسی به RAM تنها هزار سیکل و برای دیسک یک‌میلیون سیکل طول می‌کشد. حافظه‌ی RAM اغلب **حافظه‌ی اولیه**[۳] نامیده می‌شود، درحالی‌که گفته می‌شود برنامه‌ها و داده‌های ذخیره‌شده در دیسک در **حافظه‌ی ثانویه**[۴] قرار دارند.

دسترسی CPU به حافظه‌ی ثانویه به‌صورت مستقیم ممکن نیست. قبل از اجرا، برنامه‌های ذخیره‌شده در حافظه‌ی ثانویه باید در حافظه‌ی اصلی کپی شوند. درواقع، هر بار که کامپیوتر خود را روشن می‌کنید، حتی سیستم‌عامل شما قبل از اینکه CPU بتواند آن را اجرا کند، باید از دیسک روی RAM کپی شود.

هرگز RAM را فرسوده نکنید: مهم است اطمینان حاصل شود که تمام داده‌ها و برنامه‌هایی که یک کامپیوتر در طول فعالیت معمولی مدیریت می‌کند، می‌توانند در RAM آن جا بگیرند. در غیر این صورت، کامپیوتر به‌طور مداوم داده‌ها را بین دیسک و RAM منتقل می‌کند. ازآنجایی‌که این عملیات *بسیار کند* است، کارایی کامپیوتر *آن‌قدر* کاهش می‌یابد که بی‌فایده می‌شود. در این سناریو، کامپیوتر نسبت به زمان انجام محاسبات واقعی، زمان بیشتری را صرف انتظار برای انتقال داده‌ها می‌کند.

هنگامی‌که یک کامپیوتر دائماً داده‌ها را از دیسک به RAM منتقل می‌کند، گفته می‌شود که در **حالت کوبندگی**[۵] است. سِرورها باید همیشه تحت نظارت باشند: اگر شروع به پردازش چیزهایی کنند که در RAM جا نمی‌شوند، حالت کوبندگی ممکن است باعث ازکارافتادن کل سِرور شود. این همان چیزی است که باعث تشکیل صف طولانی در بانک یا صندوق می‌شود، درحالی‌که متصدی نمی‌تواند کاری جز سرزنش کردن سیستم کامپیوتری قرارگرفته در حالت کوبندگی انجام دهد. مقدار RAM ناکافی احتمالاً یکی از دلایل اصلی خطای سرور است.

---

[۱] Hard Disk

[۲] یک عکس استاندارد نور را تقریباً در ۴ میلی ثانیه جذب می‌کند.

[۳] Primary Memory

[۴] Secondary Memory

[۵] Trash Mode

ذخیره‌سازی خارجی و سوم

سلسله‌مراتب حافظه همچنان به سمت پایین ادامه دارد. در صورت اتصال به شبکه، کامپیوتر می‌تواند به حافظه‌ی مدیریت‌شده توسط کامپیوترهای دیگر، چه در شبکه‌ی محلی و چه در اینترنت (معروف به *ابر*[1]) دسترسی پیدا کند؛ اما این کار زمان بسیار بیشتری می‌برد: درحالی‌که خواندن یک دیسک محلی یک میلی‌ثانیه طول می‌کشد، دریافت داده‌ها از یک شبکه می‌تواند صدها میلی‌ثانیه طول بکشد. فقط ده میلی‌ثانیه طول می‌کشد تا یک بسته‌ی شبکه از یک کامپیوتر به کامپیوتر دیگر منتقل شود. اگر بسته‌ی شبکه از طریق اینترنت منتقل شود، اغلب برای مدت طولانی‌تری درحرکت است: دویست تا سیصد میلی‌ثانیه، یعنی معادل با مدت‌زمان یک‌چشم بر هم زدن.

در پایین سلسله‌مراتب حافظه، **ذخیره‌سازی سوم**[2] را داریم: دستگاه‌های ذخیره‌سازی که همیشه آنلاین و در دسترس نیستند. ما می‌توانیم ده‌ها میلیون گیگابایت داده را باقیمت مناسب در کارتریج‌های نوار مغناطیسی یا سی‌دی‌ها ذخیره کنیم. بااین‌حال، دسترسی به داده‌ها در این رسانه، مستلزم آن است که فردی رسانه را بردارد و آن را در دستگاه خواننده قرار دهد. این کار می‌تواند چند دقیقه یا چند روز طول بکشد[3]. ذخیره‌سازی سوم فقط برای بایگانی داده‌هایی مناسب است که به‌ندرت نیاز به دسترسی به آن دارید.

روندها در فناوری حافظه

بهبود قابل‌توجه در فناوری مورداستفاده در حافظه‌های «سریع» (مواردی که در بالای سلسله‌مراتب حافظه قرار دارند) دشوار بوده است. از سوی دیگر، حافظه‌های «کند» دائماً سریع‌تر و ارزان‌تر می‌شوند. هزینه‌های ذخیره‌سازی در دیسک سخت ده‌ها سال است که کاهش‌یافته و به نظر می‌رسد این روند ادامه خواهد داشت.

فناوری‌های جدید نیز دیسک‌ها را سریع‌تر می‌کنند. ما در حال تغییر وضعیت از دیسک‌های چرخان مغناطیسی به درایوهای حالت‌جامد (SSD)[4] هستیم. عدم وجود قطعات متحرک به این درایوها اجازه می‌دهد تا سریع‌تر، قابل‌اطمینان‌تر بوده و انرژی کمتری مصرف کنند.

دیسک‌های دارای فناوری SSD هرروز ارزان‌تر و سریع‌تر می‌شوند، اما هنوز گران هستند. برخی از تولیدکنندگان دیسک‌های ترکیبی را با هر دو فناوری SSD و مغناطیسی تولید می‌کنند. داده‌هایی که

---

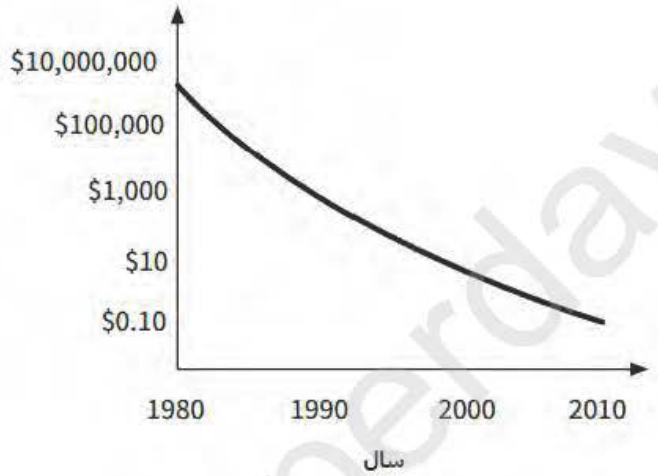
[1] Cloud

[2] Tertiary Storage

[3] از واحد فناوری اطلاعات سازمان بخواهید که در آخرین ساعات کاری هفته از نوارهای مغناطیسی پشتیبان تهیه کند....

[4] Solid State Drives (SSD)

به‌طور مکرر مورد دسترسی قرار می‌گیرند در SSD ذخیره می‌شوند و داده‌هایی که کمتر مورد دسترسی قرار می‌گیرند در قسمت مغناطیسی کندتر نگهداری می‌شوند. هنگامی‌که به داده‌هایی که کمتر مورد دسترسی قرار می‌گرفته‌اند دسترسی بیشتری پیدا می‌شود، در قسمت سریع SSD درایو ترکیبی کپی می‌شوند. این کار شبیه به ترفندی است که CPUها برای تسریع دسترسی به RAM از طریق حافظه‌های نهان داخلی استفاده می‌کنند.



شکل ۷-۱۳: هزینه‌ی فضای ذخیره‌سازی به ازای هر گیگابایت

نتیجه‌گیری

در این فصل، برخی از جنبه‌های بسیار اساسی نحوه‌ی کار کامپیوترها را توضیح دادیم. دیدیم هر چیزی را که قابل‌محاسبه باشد می‌توان در قالب دستورالعمل‌های ساده بیان کرد. آموختیم برنامه‌ای به نام کامپایلر وجود دارد که دستورات محاسباتی پیچیده‌ی ما را به دستورالعمل‌های ساده‌ای که یک CPU می‌تواند انجام دهد، ترجمه می‌کند. کامپیوترها می‌توانند محاسبات پیچیده را با استفاده از حجم عظیمی از عملیات اساسی قابل انجام توسط CPU، انجام دهند.

یاد گرفتیم که کامپیوترهای ما دارای پردازنده‌های سریع، اما حافظه‌ی نسبتاً کندی هستند؛ اما حافظه نه به‌صورت تصادفی، بلکه بر اساس موقعیت مکانی و زمانی قابل‌دسترسی است. این امر اجازه می‌دهد تا از حافظه‌های سریع‌تر برای ذخیره‌ی داده‌های حافظه که احتمال دسترسی بالایی دارند، استفاده شود. ما شاهد اعمال این اصل در چندین سطح از حافظه‌ی نهان بوده‌ایم: از حافظه‌ی نهان L1 تا ذخیره‌سازی سوم.

اصل ذخیره‌سازی در حافظه‌ی نهان موردبحث در این فصل را می‌توان در بسیاری از سناریوها اعمال کرد. شناسایی بخش‌هایی از داده‌ها که بیشتر توسط برنامه‌ی شما استفاده می‌شوند و دسترسی سریع‌تر به این داده‌ها، یکی از پرکاربردترین استراتژی‌ها برای اجرای سریع‌تر برنامه‌های کامپیوتری است.

مراجع

- Structured Computer Organization, by Tanenbaum
  - Get it at https://code.energy/tanenbaum
- Modern Compiler Implementation in C, by Appel
  - Get it at https://code.energy/appel

فصل ۸

برنامه‌نویسی

وقتی کسی می‌گوید «من یک زبان برنامه‌نویسی می‌خواهم که فقط کارهای موردنظرم را برای انجام به آن بگویم» یک آب‌نبات به وی بدهید.
– آلن جی پرلیس[۱]

ما نیازمند این هستم که کامپیوترها ما را درک کنند. این امر دلیل بیان دستوراتمان به یک زبان برنامه‌نویسی است: زبانی که یک ماشین آن را درک می‌کند. شما نمی‌توانید به زبان انگلیسی شکسپیر به یک ماشین بگویید که چه‌کاری انجام دهد، مگر این‌که یک کدنویس استخدام کرده یا در یک فیلم علمی تخیلی باشید. تا امروز، فقط کدنویسان این توانایی را دارند که بدون محدودیت به یک ماشین دستور بدهند چه‌کاری انجام بدهد. با تقویت دانش شما در مورد زبان‌های برنامه‌نویسی، قدرت شما به‌عنوان یک کدنویس نیز افزایش می‌یابد. در این فصل یاد خواهید گرفت:

- **زبان‌شناسی** مخفی حاکم بر کد را درک کنید.
- اطلاعات ارزشمندتان را در درون **متغیرها** ذخیره کنید.
- به راه‌حل‌ها تحت **پارادایم‌های** مختلف فکر کنید.

ما وارد مباحث رسمی نحوی و دستوری علوم کامپیوتر نمی‌شویم. آسوده باشید و به خواندن ادامه بدهید.

۸-۱- زبان‌شناسی

زبان‌های برنامه‌نویسی بسیار متفاوت هستند، اما همه‌ی آن‌ها یک کار انجام می‌دهند: کار با اطلاعات. این زبان‌ها برای انجام این کار بر سه عنصر اصلی تکیه‌دارند. یک **مقدار**[۲] که اطلاعات را نشان می‌دهد.

---

[۱] Alan Jay Perlis (1922-1990): دانشمند آمریکایی علوم کامپیوتر که اولین برنده‌ی جایزه‌ی تورینگ بوده است. وی نقش زیادی در پیشرفت زبان‌های برنامه‌نویسی داشته است.م.

[۲] Value

یک **عبارت**[۱] که یک مقدار را تولید می‌کند. یک **دستور**[۲] که از یک مقدار برای دادن یک دستورالعمل به کامپیوتر استفاده می‌کند.

مقادیر

نوع اطلاعاتی که یک مقدار می‌تواند کدگذاری کند از زبانی به زبان دیگر متفاوت است. در ابتدایی‌ترین زبان‌ها، یک مقدار فقط می‌تواند اطلاعات بسیار ساده‌ای مانند یک عدد صحیح یا یک عدد ممیز شناور[۳] را کدگذاری کند. با پیچیده‌تر شدن زبان‌ها، آن‌ها شروع به استفاده از کاراکترها و رشته‌های دیگر به‌عنوان مقادیر کردند. در C، که هنوز یک زبان بسیار سطح پایین است، می‌توانید یک ساختار تعریف کنید: راهی برای تعریف مقادیری که از گروه‌هایی از مقادیر دیگر تشکیل‌شده‌اند. به‌عنوان‌مثال، می‌توانید یک نوع مقدار به نام «مختصات» تعریف کنید که از دو ممیز شناور ساخته می‌شود: طول و عرض جغرافیایی.

مقادیر آن‌قدر مهم هستند که به‌عنوان «شهروندان درجه‌ی یک» زبان برنامه‌نویسی نیز شناخته می‌شوند. زبان‌ها اجازه‌ی انجام انواع عملیات را با استفاده از مقادیر می‌دهند: مقادیر را می‌توان در زمان اجرا ایجاد کرد، به‌عنوان آرگومان به توابع ارسال کرد و توسط توابع برگرداند.

عبارات

شما به دو روش می‌توانید یک مقدار را به وجود بیاورید: یا با نوشتن یک **نماد راستین**[۴]، یا با فراخوانی یک **تابع**. در اینجا یک عبارت نمونه را با استفاده از یک نماد راستین می‌بینید:

```
3
```

به همین راحتی. با استفاده از نماد راستین 3 مقدار ۳ را به وجود آوردیم. خیلی سرراست. سایر انواع مقادیر را نیز به کمک نمادهای راستین می‌توان به وجود آورد. اغلب زبان‌های برنامه‌نویسی به شما امکان می‌دهند مقدار رشته‌ای *hello world* را با تایپ کردن `"hello World"` بسازید. از سوی دیگر، توابع یک مقدار را با دنبال کردن یک متد یا رویه که جای دیگری کد شده است، تولید می‌کنند. به‌عنوان‌مثال

```
getPacificTime()
```

---

۱ Expression

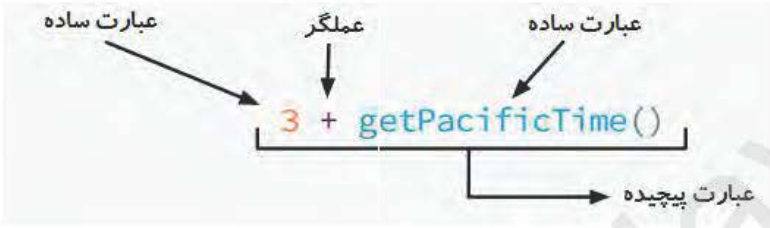
۲ Statement

۳ ممیزهای شناور روش معمولی برای نمایش اعدادی هستند که رقم اعشاری دارند.

۴ Literal

این عبارت یک مقدار برابر با ساعت کنونی در لس‌آنجلس تولید می‌کند. اگر اکنون ساعت ۴ صبح باشد، این متد مقدار 4 را برمی‌گرداند.

عنصر بنیادین دیگر در هر زبان برنامه‌نویسی **عملگر**[1] است. یک عملگر می‌تواند عبارات ساده را به هم پیوند داده و یک عبارت پیچیده تولید کند. به‌عنوان‌مثال، با استفاده از عملگر + می‌توانیم مقداری برابر با ساعت کنونی در نیویورک ایجاد کنیم:



زمانی که در لس‌آنجلس ساعت ۴ صبح باشد، عبارت طولانی ما به عبارت کوتاه‌تر 7 کاهش پیدا می‌کند. در حقیقت، یک عبارت چیزی است که شما می‌نویسید و کامپیوتر می‌تواند آن را به یک مقدار واحد کاهش دهد. عبارات بزرگ را می‌توان با استفاده از عملگرها با سایر عبارات ترکیب کرد و عبارات بزرگ‌تر تولید نمود. درنهایت، حتی پیچیده‌ترین عبارات همیشه باید به یک مقدار واحد تبدیل شوند.

در کنار نمادهای راستین، عملگرها و توابع، عبارات می‌توانند شامل پرانتزها نیز باشند. پرانتزها به شما اجازه‌ی کنترل **اولویت عملگرها**[2] را می‌دهند: $(2 + 4)^2$ به 6^2 تبدیل می‌شود، و این عبارت نیز به 36 تبدیل می‌گردد. عبارت $4^2 + 2$ به $16 + 2$ و سپس به 18 تبدیل می‌شود.

دستورات

درحالی‌که یک عبارت برای نمایش یک مقدار مورداستفاده قرار می‌گیرد، یک دستور برای دستور دادن به کامپیوتر به‌منظور *انجام* یک کار استفاده می‌شود. به‌عنوان‌مثال، این دستور موجب نمایش یک پیام می‌شود: `Print("hello world")`.

مثال‌های پیچیده‌تر شامل دستورات `if`، `while-loop` و `for-loop` هستند. زبان‌های برنامه‌نویسی مختلف از انواع متفاوتی از دستورات پشتیبانی می‌کنند.

---

[1] Operator
[2] Operator Precedence

شکل ۸-۱: دریافت شده از http://geek-and-poke.com[1]

تعاریف: برخی از زبان‌های برنامه‌نویسی دستورات ویژه‌ای به نام **تعاریف**[2] دارند. این دستورات وضعیت برنامه را با اضافه کردن موجودیتی که وجود نداشته است، مانند یک مقدار یا تابع جدید، عوض می‌کنند[3]. برای اشاره به موجودیتی که تعریف کرده‌ایم، باید یک نام به آن اختصاص دهیم. این کار را **انقیاد نام**[4] می‌گویند. به‌عنوان‌مثال، نام getPacificTime باید به تعریف یک تابع درحالی از کد محدودشده باشد.

۸-۲- متغیرها

متغیرها مهم‌ترین نوع انقیاد نام موجود هستند: یک انقیاد بین یک نام و یک مقدار. یک متغیر نامی را به فضایی از حافظه که مقدار در آن ذخیره‌شده است اختصاص می‌دهد، که اصطلاحاً به آن نام مستعار[5] می‌گویند. در اغلب مواقع، یک متغیر با استفاده از عملگر انتساب ساخته می‌شود. در شبه‌کد این کتاب، انتساب‌ها به‌صورت ← نوشته‌شده‌اند، مانند:

---

[1] این کارتون اشاره دارد به اولین برنامه‌ای که معمولاً برنامه‌نویسان می‌نویسند و عبارت hello world را بر روی صفحه چاپ می‌کند. این مسئله امروزه به دست‌مایه‌ای برای طنز و شوخی تبدیل‌شده است. م.

[2] Definitions

[3] گاهی اوقات موجودیت‌ها را می‌توان از کتابخانه‌های خارجی از قبل کدنویسی شده وارد برنامه کرد.

[4] Name Binding

[5] Alias

```
pi ← 3.142
```

در بیشتر زبان‌های برنامه‌نویسی، انتساب‌ها به‌صورت = نوشته می‌شوند. حتی برخی از زبان‌ها نیازمند *اعلام* یک نام برای یک متغیر، قبل از تعریف آن، هستند. در این شرایط و درنهایت شما به چنین وضعیتی می‌رسید:

```
var pi
pi = 3.142
```

این دستور یک قطعه از حافظه را رزرو کرده و مقدار 3.142 را در آن نوشته و نام pi را به آدرس آن قطعه از حافظه اختصاص می‌دهد.

نوع‌دهی متغیر

در اغلب زبان‌های برنامه‌نویسی، متغیرها باید یک نوع مشخص (مانند عدد صحیح، ممیز شناور، یا رشته) داشته باشند. برنامه با استفاده از این روش می‌داند چگونه باید صفرها و یک‌هایی را که از قطعه‌ی مربوطه‌ی حافظه می‌خواند تفسیر کند. این امر به تعیین مکان خطاها در زمان اعمال عملگرها بر روی متغیرها کمک می‌کند. اگر متغیری از نوع «رشته» و متغیر دیگر از نوع عدد صحیح باشد، جمع زدن آن‌ها بی‌معنی خواهد بود.

دو روش برای انجام بررسی نوع وجود دارد: ایستا و پویا. نوع ایستا نیازمند این است که کدنویس نوع هر متغیر را قبل از استفاده مشخص کند. به‌عنوان‌مثال، زبان‌های برنامه‌نویسی مانند C و C++ شما را مجبور می‌کنند که به‌صورت زیر بنویسید:

```
float pi;
pi = 3.142;
```

این دستورات اعلام می‌کنند که متغیری به نام pi فقط می‌تواند داده‌های نمایش‌دهنده‌ی اعداد ممیز شناور را ذخیره کند. زبان‌های دارای نوع ایستا می‌توانند در زمان کامپایل کردن کد بهینه‌سازی‌های اضافه‌ای را انجام داده و خطاهای احتمالی را قبل از اجرای کد شناسایی کنند. بااین‌حال، اعلام نوع در هر بار استفاده از یک متغیر می‌تواند خسته‌کننده باشد.

برخی زبان‌ها ترجیح می‌دهند نوع را به‌صورت پویا بررسی کنند. با استفاده از بررسی پویای نوع، هر متغیر می‌تواند هر نوع مقداری را ذخیره کند، بنابراین به اعلام نوع متغیر نیاز نیست. بااین‌حال، در زمان اجرای کد و قبل از اعمال عملگرها بر روی متغیرها، یک فرایند اضافه‌ی بررسی نوع انجام می‌شود تا از معنادار بودن عملیات بین متغیرها اطمینان حاصل شود.

محدوده‌ی متغیر

اگر تمام انقیادهای نام در تمام نقاط برنامه در دسترس و معتبر بودند، برنامه‌نویسی بسیار سخت می‌شد. با بزرگ‌تر شدن برنامه‌ها، نام‌های مشابه برای متغیرها (مانند time، length یا speed) در بخش‌های غیر مرتبط کد قابل‌استفاده نخواهند بود.

به‌عنوان‌مثال، من می‌توانم یک متغیر length در دو نقطه‌ی برنامه‌ام تعریف کنم، بدون آن‌که توجه کنم این کار باعث بروز یک خطا می‌شود. بدتر از آن، می‌توانم کتابخانه‌ای را وارد برنامه کنم که از یک متغیر به نام length استفاده می‌کند: در این حالت، متغیر length کد من با متغیر length کد واردشده دچار تصادم می‌شود.

برای اجتناب از تصادم، انقیادهای نام در کل کد منبع معتبر نیستند. **محدوده‌ی** متغیر[۱] مشخص می‌کند که در چه جایی معتبر و قابل‌استفاده است. اغلب زبان‌ها به‌گونه‌ای تنظیم‌شده‌اند که یک متغیر فقط در تابعی که تعریف‌شده معتبر است.

زمینه[۲] یا **محیط**[۳] جاری مجموعه‌ای از تمام انقیادهای نام در دسترس در نقطه‌ای خاص از یک برنامه است. معمولاً متغیرهایی که در یک زمینه تعریف می‌شوند به‌محض خروج جریان اجرا از آن زمینه، بلافاصله حذف‌شده و حافظه‌ی کامپیوتر را آزاد می‌کنند. اگرچه این کار توصیه نمی‌شود، ولی شما می‌توانید این قانون را دور زده و متغیرهایی بسازید که همیشه و در همه‌جای برنامه‌تان در دسترس باشند. این متغیرها را **متغیرهای عمومی**[۴] می‌نامند.

مجموعه‌ی تمام نام‌های در دسترس به‌صورت عمومی، **فضای نام**[۵] شما را می‌سازند. شما باید به‌صورت دقیق به فضای نام برنامه‌ی خود توجه داشته باشید. این فضا باید تا حد ممکن کوچک نگه‌داشته شود. در فضاهای نام بزرگ رخ دادن تعارض ساده‌تر است.

وقتی نام‌های بیشتری را به فضای نام خود اضافه می‌کنید، سعی کنید تعداد نام‌های اضافه‌شده را کمینه کنید. به‌عنوان‌مثال، وقتی یک پیمانه‌ی خارجی را وارد می‌کنید، فقط نام توابعی که قصد استفاده دارید را اضافه کنید. پیمانه‌های خوب به کاربر امکان می‌دهند نام‌های کمی را به فضای نام خود اضافه کند. اضافه کردن چیزهای غیرضروری به فضای نام موجب بروز مشکلی به نام **آلودگی فضای نام**[۶] می‌شود.

---

۱ Variable Scope
۲ Context
۳ Environment
۴ Global Variables
۵ Namespace
۶ Namespace Pollution

۸-۳- پارادایم‌ها

یک **پارادایم**[۱] مجموعه‌ای خاص از مفاهیم و عملکردها است که یک حوزه‌ی علمی را تعریف می‌کنند. یک پارادایم نحوه‌ی برخورد شما با یک مسئله، تکنیک‌هایی که استفاده می‌کنید و ساختار راه‌حل شما را مشخص می‌کند. به‌عنوان‌مثال، مکتب نیوتنی و نسبیتی دو پارادایم متفاوت از فیزیک هستند.

روش برخورد شما با مسائلتان، هم در کدنویسی و هم در فیزیک، بر اساس پارادایمی که استفاده می‌کنید تغییر می‌کند. یک پارادایم برنامه‌نویسی دیدگاهی در رابطه با حوزه‌ی کدنویسی است. این دیدگاه سبک و روش کدنویسی شما را جهت می‌دهد.

شما می‌توانید از یک یا چند پارادایم در برنامه‌ی خود استفاده کنید. بهتر است از پارادایم‌هایی استفاده کنید که زبان برنامه‌نویسی مورداستفاده‌ی شما بر اساس آن‌ها بنا است. در دهه‌ی ۱۹۴۰، اولین کامپیوترها به‌صورت دستی با چرخاندن سوئیچ‌ها برای درج صفرها و یک‌ها در حافظه‌ی کامپیوتر برنامه‌ریزی می‌شدند. تکامل برنامه‌نویسی هرگز متوقف نشد و پارادایم‌هایی پدید آمدند تا افراد را قادر به کدنویسی با کارایی، پیچیدگی و سرعت بیشتر کنند.

در برنامه‌نویسی سه پارادایم اصلی وجود دارد: دستوری، اعلانی و منطقی. متأسفانه، اکثر کدنویس‌ها فقط نحوه‌ی کار صحیح با نوع اول را یاد می‌گیرند. آشنایی با هر سه مورد مهم است، زیرا شما را قادر می‌سازد از ویژگی‌ها و فرصت‌هایی که هر یک از زبان‌های برنامه‌نویسی ارائه می‌دهند بهره‌مند شوید. به‌این‌ترتیب، شما قادر خواهید بود با حداکثر اثربخشی کدنویسی کنید.

برنامه‌نویسی دستوری

پارادایم برنامه‌نویسی دستوری[۲] مشخص می‌کند چگونه با استفاده از دستورات خاص به کامپیوتر دستور دهیم که در هر مرحله دقیقاً چه‌کاری باید انجام دهد. هر دستور در این پارادایم وضعیت کامپیوتر را عوض می‌کند. دنباله‌ی دستوراتی که یک برنامه را می‌سازند، یکی پس از دیگری انجام می‌شوند.

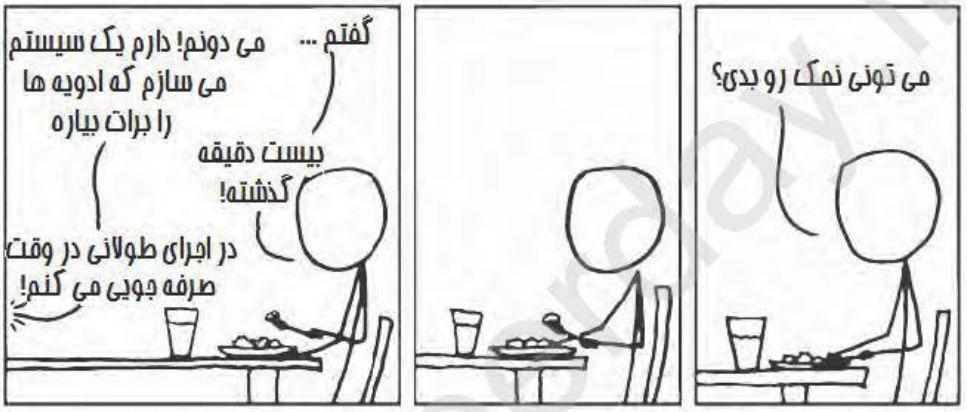
این پارادایم اولین پارادایم برنامه‌نویسی بوده و تعمیمی طبیعی از روش عملکرد کامپیوترها است. محاسبات همیشه به‌وسیله‌ی دستورالعمل‌های CPU که به‌صورت پشت سر هم اجرا می‌شوند، انجام می‌گیرند. درنهایت، هر برنامه‌ی کامپیوتری به‌وسیله‌ی کامپیوتر و تحت این پارادایم اجرا می‌شود.

---

[۱] Paradigm

[۲] Imperative Programming Paradigm

برنامه‌نویسی دستوری تقریباً شناخته‌شده‌ترین پارادایم برنامه‌نویسی است. در حقیقت، بسیاری از برنامه‌نویسان فقط با این پارادایم آشنا هستند. این پارادایم یک تعمیم طبیعی از روش کار انسان نیز هست: ما از این پارادایم برای بیان یک دستور پخت، یک‌روال تعمیر خودرو و سایر رویه‌های روزمره استفاده می‌کنیم. وقتی در انجام یک کار خسته‌کننده تنبل هستیم، این دستورالعمل‌ها را در قالب یک برنامه کدگذاری می‌کنیم و کامپیوتر آن را برای ما انجام می‌دهد. تنبلی برنامه‌نویس علّت بسیاری از اتفاقات مهم است.



شکل ۸-۲: «مشکل عمومی»، از http://xkcd.com

برنامه‌نویسی کد ماشین: برنامه‌نویسان اولیه، که مجبور بودند کدشان را به‌صورت دستی و با استفاده از صفرها و یک‌ها وارد کامپیوتر کنند، نیز تنبل بودند. آن‌ها تصمیم گرفتند با استفاده از نمادهای قابل به یادسپاری مانند CP برای دستورالعمل کپی کردن، MOV برای دستورالعمل جابجا کردن، CMP برای دستورالعمل مقایسه کردن و غیره، روش جذاب‌تری برای برنامه‌نویسی به وجود بیاورند. سپس برنامه‌ای نوشتند که این نمادهای قابل به یادسپاری را به معادل اعداد دودویی آن‌ها، که بعداً قابل‌اجرا توسط کامپیوتر باشد، تبدیل کند. به‌این‌ترتیب، زبان **اسمبلی**[1] یا **ASM** به وجود آمد.

برنامه‌ای که با این نمادهای قابل به یادسپاری نوشته می‌شود بسیار خواناتر از معادل صفر و یک آن است. این نمادهای اولیه و این سبک برنامه‌نویسی امروزه نیز به‌صورت گسترده استفاده می‌شوند. با پشتیبانی CPUهای جدید از دستورالعمل‌های پیچیده‌تر، نمادهای قابل به یادسپاری بیشتری تولید شدند، ولی قوانین بنیادین یکسان هستند.

---

[1] Assembly

زبان اسمبلی برای برنامه‌نویسی سیستم‌هایی مانند مایکروویوهای الکترونیک، یا سیستم‌های کامپیوتری خودرو استفاده می‌شود. این سبک برنامه‌نویسی در بخش‌هایی از کد که به کارایی بسیار زیاد نیاز است و صرفه‌جویی در چند سیکل CPU اهمیت دارد نیز استفاده می‌شود.

به‌عنوان‌مثال، فرض کنید در حال بهینه‌سازی یک سِرور وب با کارایی بالا هستید و به یک مشکل برخورد کرده‌اید. شما می‌توانید این مشکل را به کد ASM تبدیل کرده و آن را بررسی کنید. در بسیاری از اوقات می‌توانیم کد را برای استفاده از چند دستورالعمل کمتر اصلاح کنیم. زبان‌های سطح پایین‌تر گاهی از درج زبان ماشین در کد معمولی زبان‌های برنامه‌نویسی به‌منظور پیاده‌سازی این بهینه‌سازی‌ها پشتیبانی می‌کنند. نوشتن کد به زبان ماشین به شما امکان کنترل کامل عملکرد CPU را در زمان اجرای کد می‌دهد.

برنامه‌نویسی ساخت‌یافته: در ابتدا، برنامه‌ها از دستور GOTO برای کنترل جریان اجرا استفاده می‌کردند. این دستور موجب می‌شود اجرا به نقطه‌ی دیگری از کد منتقل شود. با پیچیده‌تر شدن برنامه‌ها، درک عملکرد برنامه تقریباً غیرممکن شده بود. جریان‌های متفاوت ممکن برای اجرا همگی با دستورهای GOTO و JUMP درهم‌تنیده شده بودند و شرایطی به وجود آمده بود که به آن *کد اسپاگتی*[1] می‌گویند[2]. در سال ۱۹۶۸، دایکسترا اعلامیه معروف خود به نام «دستور GOTO مضر است» را نوشت و موجب بروز یک انقلاب شد. کدها شروع به جدا شدن در قالب بخش‌های منطقی کردند. به‌جای استفاده از GOTO به‌صورت کنترل نشده، برنامه‌نویسان شروع به استفاده از ساختارهای کنترلی (مانند if، else، while، for و ...) کردند. این امر موجب شد برنامه‌ها بسیار راحت‌تر نوشته و خطایابی شوند.

برنامه‌نویسی رویه‌ای: پیشرفت بعدی در هنر کدنویسی، برنامه‌نویسی رویه‌ای[3] بود. این روش امکان تقسیم برنامه‌ها را به چندین **رویه**[4]، به‌منظور اجتناب از تکرار کد و افزایش قابلیت استفاده از آن، به وجود آورد. به‌عنوان نمونه، شما می‌توانید یک تابع بسازید که سیستم متریک را به واحدهای آمریکایی تبدیل کرده، و سپس تابع خود را به‌منظور استفاده‌ی مجدد از کد در زمان نیاز فراخوانی کنید. این امر موجب بهبود بیشتر برنامه‌نویسی ساخت‌یافته شد. استفاده از رویه‌ها، گروه‌بندی قطعات به هم مرتبط کد و مجزا سازی آن‌ها را در قالب بخش‌های منطقی ساده‌تر می‌کند.

---

[1] Spaghetti Code

[2] اگر می‌خواهید کد منبع فرد دیگری را تمسخر کنید، آن را کد اسپاگتی بنامید.

[3] Procedural Programming

[4] Procedure

برنامه‌نویسی اعلانی

پارادایم برنامه‌نویسی اعلانی[1] به شما اجازه می‌دهد نتایج مطلوبتان را بدون درگیر شدن با ریز دستورات در هر گام بیان کرده و به دست آورید. این پارادایم در مورد بیان *آن چیزی* است که می‌خواهید، و نه در مورد چگونگی انجام آن. در بسیاری سناریوها، این روش به برنامه‌ها امکان می‌دهد بسیار کوتاه‌تر و ساده‌تر شوند. خواندن چنین برنامه‌هایی ساده‌تر است.

برنامه‌نویسی تابعی[2]: در پارادایم برنامه‌نویسی تابعی، توابع بیشتر از رویه‌ها مورداستفاده قرار می‌گیرند. در این روش از توابع برای تعریف روابط بین دو یا چند عنصر، خیلی شبیه به معادلات ریاضی، استفاده می‌شود. توابع شهروندان درجه‌ی یک پارادایم تابعی هستند. در این پارادایم با توابع همانند سایر انواع داده‌ای اولیه، مثل رشته‌ها و اعداد، برخورد می‌شود.

یک تابع سایر توابع را به‌عنوان آرگومان دریافت کرده، و توابع دیگری را به‌عنوان خروجی برمی‌گردانند. توابعی را که چنین ویژگی‌هایی داشته باشند، به علت قدرت بیان بالایی که دارند، **توابع مرتبه‌ی بالا**[3] می‌نامند. بسیاری از زبان‌های برنامه‌نویسی رایج از چنین عناصری از پارادایم تابعی استفاده می‌کنند. شما باید از قدرت بیان شگفت‌انگیز این توابع در هرجایی که ممکن است، استفاده کنید.

به‌عنوان نمونه، اغلب زبان‌های برنامه‌نویسی تابعی دارای یک تابع sort درونی هستند. این تابع هر دنباله‌ای از عناصر را می‌تواند مرتب کند. تابع sort یک تابع دیگر را که نحوه‌ی مقایسه‌ی عناصر در فرایند مرتب‌سازی را مشخص می‌کند، به‌عنوان ورودی دریافت می‌کند. به‌عنوان‌مثال، فرض کنید coordinates شامل لیستی از موقعیت‌های جغرافیایی است. با داشتن دو موقعیت، تابع closer_to_home مشخص می‌کند کدام موقعیت به خانه‌ی شما نزدیک‌تر است. شما می‌توانید لیستی از موقعیت‌ها را بر اساس فاصله‌ی آن‌ها تا خانه‌ی خود مرتب کنید:

```
sort(coordinates, closer_to_home)
```

توابع مرتبه‌ی بالا اغلب برای پالایش کردن داده‌ها استفاده می‌شوند. زبان‌های برنامه‌نویسی تابعی دارای یک تابع **پالایش** درونی نیز هستند که مجموعه‌ای از عناصر و یک تابع پالایش کننده را دریافت کرده و مشخص می‌کند هر عنصر باید پالایش شود یا خیر. به‌عنوان‌مثال، برای پالایش کردن اعداد زوج از یک لیست، باید بنویسید:

---

[1] Declarative Programming Paradigm
[2] Functional Programming
[3] High-Order Functions

```
odd_numbers ← filter(numbers, number_is_odd)
```

تابع number_is_odd تابعی است که یک عدد را دریافت کرده و در صورت فرد بودن آن مقدار True و در غیر این صورت False برمی‌گرداند.

کار دیگری که در زمان برنامه‌نویسی انجام می‌شود، به کار بردن یک تابع خاص بر روی تمام عناصر یک لیست است. در برنامه‌نویسی تابعی این کار را **نگاشت**[۱] می‌نامند. زبان‌ها اغلب دارای یک تابع map درونی برای انجام این کار هستند. به‌عنوان‌مثال، برای محاسبه‌ی مجذور هر عدد در لیست، می‌توانیم کار زیر را انجام دهیم:

```
squared_numbers ← map(numbers, square)
```

تابع square تابعی است که مجذور عدد داده‌شده را برمی‌گرداند. فرایندهای نگاشت و پالایش بسیار زیادی انجام می‌شوند، به همین دلیل بسیاری از زبان‌های برنامه‌نویسی راه‌هایی برای ساده‌تر نوشتن این عبارات ارائه می‌کنند. به‌عنوان نمونه، در زبان برنامه‌نویسی پایتون، مجذور اعداد یک لیست را به‌صورت زیر محاسبه می‌کنید:

```
squared_numbers = [x**2 for x in numbers]
```

این پدیده را **قند نحوی**[۲] می‌نامند: اضافه کردن دستورات نحوی که بتوانید با استفاده از آن‌ها عبارات را ساده‌تر و کوتاه‌تر بنویسید. بسیاری از زبان‌های برنامه‌نویسی قالب‌های متعددی برای قند نحوی در اختیار شما قرار می‌دهند. از آن‌ها استفاده و سوءاستفاده کنید.

درنهایت، وقتی به پردازش یک لیست از مقادیر نیاز دارید، به‌گونه‌ای که یک نتیجه‌ی واحد تولید کند، می‌توانید از تابع **کاهش**[۳] استفاده کنید. این تابع یک لیست، یک مقدار اولیه و یک تابع کاهنده را به‌عنوان ورودی می‌گیرد. مقدار اولیه یک متغیر «انباشتگر» را ایجاد می‌کند که به‌وسیله‌ی تابع کاهنده به ازای هر عنصر درون لیست بهنگام‌سازی شده و درنهایت برگردانده می‌شود:

```
function reduce(list, initial_val, func)
    accumulator ← initial_val
    for item in list
        accumulator ← func(accumulator, item)
    return accumulator
```

---

۱ Mapping
۲ Syntactic Sugar
۳ Reduce

به‌عنوان‌مثال، شما می‌توانید از reduce برای محاسبه‌ی مجموع عناصر یک لیست استفاده کنید:

```
sum ← function(a, b): a + b
summed_numbers ← reduce(numbers, 0, sum)
```

استفاده از reduce موجب ساده‌تر شدن کد شما و افزایش خوانایی آن می‌شود. مثال دیگر: اگر sentences لیستی از جملات باشد، و شما بخواهید تعداد کل کلمات این جملات را محاسبه کنید، می‌توانید بنویسید:

```
wsum ← function(a, b): a + length(split(b))
number_of_words ← reduce(sentences, 0, wsum)
```

تابع split یک‌رشته را به لیستی از کلمات تقسیم می‌کند، و تابع length تعداد عناصر یک لیست را می‌شمارد.

توابع مرتبه‌ی بالا فقط توابع را به‌عنوان ورودی نمی‌گیرند، بلکه می‌توانند توابعی را نیز به‌عنوان خروجی تولید کنند. این توابع حتی می‌توانند یک مرجع را به مقداری درون تابعی که تولید کرده‌اند محصور کنند. این فرایند را **بستار**[1] می‌نامند. تابعی که یک بستار دارد، موارد را به «خاطر سپرده» و می‌تواند به محیط مقادیر محصورشده‌ی خود دسترسی داشته باشد.

با استفاده از بستارها، می‌توانیم اجرای یک تابع دارای چندین آرگومان را به چند گام تقسیم کنیم. این فرایند را **آراستگی**[2] می‌نامند. به‌عنوان نمونه، فرض کنید کد شما این تابع sum را دارد:

```
sum ← function(a, b): a + b
```

تابع sum دو پارامتر دریافت می‌کند، ولی با یک پارامتر نیز قابل فراخوانی است. عبارت sum(3) عددی برنمی‌گرداند بلکه یک تابع آراسته‌شده برمی‌گرداند. این عبارت تابع sum را با استفاده از 3 به‌عنوان یک پارامتر فراخوانی می‌کند. مرجع مقدار 3 در تابع آراسته‌شده محصور می‌شود. به‌عنوان نمونه:

```
sum_three ← sum(3)
print sum_three(1)  # prints "4".

special_sum ← sum(get_number())
print special_sum(1)  # prints "get_number() + 1".
```

---

[1] Closure

[2] Currying

توجه داشته باشید که get_number به‌منظور ایجاد تابع special_sum فراخوانی و ارزیابی نمی‌شود. یک مرجع از get_number به special_sum داده می‌شود. تابع get_number فقط زمانی که نیاز به ارزیابی تابع special_sum داشته باشیم، فراخوانی می‌شود. این کار را **ارزیابی سست**[1] می‌نامند که یک ویژگی مهم زبان‌های برنامه‌نویسی تابعی است.

بستارها را می‌توان برای تولید مجموعه‌ای از توابع مرتبط که از یک الگو تبعیت می‌کنند نیز استفاده کرد. استفاده از یک الگو برای تابع می‌تواند کد شما را خواناتر کرده و از تکرار در آن جلوگیری کند. اجازه بدهید یک مثال را ببینیم:

```
function power_generator(base)
    function power(x)
        return power(x, base)
    return power
```

می‌توانیم از power_generator برای تولید توابع مختلف که توان را محاسبه می‌کنند استفاده کنیم:

```
square ← power_generator(2)
print square(2)  # prints 4.

cube ← power_generator(3)
print cube(2)  # prints 8.
```

توجه داشته باشید که توابع برگردانده شده‌ی square و cube مقدار متغیر base را حفظ می‌کنند. این مقدار فقط در محیط power_generator وجود دارد، حتی اگر این توابع برگردانده شده کاملاً مستقل از تابع power_generator باشند. مجدداً: یک بستار تابعی است که به یک متغیر *بیرون* از محیط خود دسترسی دارد.

از بستارها می‌توان برای مدیریت وضعیت درونی یک تابع نیز استفاده کرد. فرض کنید تابعی نیاز دارید که مجموع تمام اعدادی را که به آن داده‌اید محاسبه کند. یک‌راه برای انجام این کار استفاده از یک متغیر عمومی است:

```
GLOBAL_COUNT ← 0
function add(x)
    GLOBAL_COUNT ← GLOBAL_COUNT + x
    return GLOBAL_COUNT
```

---

[1] Lazy Evaluation

همان‌گونه که پیش‌تر دیدیم، از به‌کارگیری متغیرهای عمومی باید اجتناب کرد، زیرا فضای نام برنامه را آلوده می‌کنند. یک رویکرد تمیزتر استفاده از یک بستار برای ایجاد یک مرجع به متغیر انباشتگر است:

```
function make_adder()
    n ← 0
    function adder(x)
        n ← x + n
        return n
    return adder
```

این کار به ما اجازه می‌دهد چندین آدرس را بدون استفاده از متغیرهای عمومی ایجاد کنیم:

```
my_adder ← make_adder()
print my_adder(5) # prints 5.
print my_adder(2) # prints 7 (5 + 2).
print my_adder(3) # prints 10 (5 + 2 + 3).
```

تطابق الگو: برنامه‌نویسی تابعی به شما اجازه می‌دهد با توابع مانند توابع ریاضی برخورد کنید. در ریاضی می‌توانیم رفتار تابع را بر اساس ورودی تعیین کنیم. به الگوی ورودی تابع فاکتوریل دقت کنید:

$$0! = 1$$
$$n! = n \times (n-1)!$$

برنامه‌نویسی تابعی امکان **تطابق الگو**[1] (فرایند شناسایی الگو) را به وجود می‌آورد. به‌سادگی می‌توانید بنویسید:

```
factorial(0): 1
factorial(n): n × factorial(n - 1)
```

در مقابل، برنامه‌نویسی دستوری شما را مجبور می‌کند به‌صورت زیر بنویسید:

```
function factorial(n)
    if n = 0
        return 1
    else
        return n × factorial(n - 1)
```

کدام‌یک تمیزتر به نظر می‌رسد؟ من هر جا که مقدور باشد از نسخه‌ی تابعی استفاده می‌کنم. برخی زبان‌های برنامه‌نویسی *کاملاً* تابعی هستند: تمام کد دقیقاً معادل با توابع صرف ریاضی است. این زبان‌ها بسیار وابسته به زمان هستند و ترتیب دستورات در کد تأثیری بر رفتار آن ندارد. در این زبان‌ها، تمام

[1] Pattern Matching

مقادیر منتسب به متغیرها غیرقابل تغییر هستند. این پدیده را **انتساب واحد**[1] می‌نامند. به دلیل آن‌که هیچ وضعیت برنامه‌ای وجود ندارد، نیاز به تغییر متغیر در هیچ دوره‌ای از زمان نیز وجود ندارد. محاسبات در یک پارادایم کاملاً تابعی، صرفاً به ارزیابی توابع و تطبیق الگوها می‌پردازد.

برنامه‌نویسی منطقی

هر زمان که مسئله‌ی موردنظر شما راه‌حل مجموعه‌ای از فرمول‌های منطقی باشد، می‌توانید از **برنامه‌نویسی منطقی**[2] استفاده کنید. برنامه‌نویس گزاره‌های منطقی را در مورد یک وضعیت بیان می‌کند، مانند مواردی که در بخش ۱-۲ دیدیم. سپس پرس‌وجوهایی برای یافتن پاسخ از مدل ارائه‌شده انجام می‌شود. کامپیوتر وظیفه‌ی تفسیر متغیرهای منطقی و پرس‌وجوها را بر عهده دارد. همچنین یک فضای راه‌حل از گزاره‌ها ایجاد کرده و راه‌حل‌های برآورده‌کننده‌ی همه‌ی آن‌ها را بررسی می‌کند.

بزرگ‌ترین مزیت پارادایم برنامه‌نویسی منطقی این است که فرایند برنامه‌نویسی به حداقل خود می‌رسد. فقط حقایق، گزاره‌ها و پرس‌وجوها به کامپیوتر ارائه می‌شوند. کامپیوتر وظیفه‌ی یافتن بهترین راه برای جستجوی فضای راه‌حل و ارائه‌ی نتایج را بر عهده دارد.

این پارادایم به‌صورت گسترده و به‌خوبی مورداستفاده قرار نمی‌گیرد، اما اگر متوجه شدید که به هوش مصنوعی و پردازش زبان طبیعی نیاز دارید، به یاد داشته باشید که به این پارادایم توجه کنید.

نتیجه‌گیری

با تکامل روش‌های برنامه‌نویسی، پارادایم‌های برنامه‌نویسی جدیدی ظهور کردند. این پارادایم‌ها به کدهای کامپیوتری اجازه بیان و ظرافت بیشتری دادند. هرچه بیشتر از پارادایم‌های مختلف برنامه‌نویسی بدانید، بهتر می‌توانید کدنویسی کنید.

در این فصل، دیدیم که چگونه برنامه‌نویسی از واردکردن مستقیم صفرها و یک‌ها به حافظه‌ی کامپیوتر به نوشتن کد اسمبلی تکامل‌یافته است. سپس برنامه‌نویسی با ایجاد ساختارهای کنترلی مانند حلقه‌ها و متغیرها آسان‌تر شد. دیدیم که چگونه استفاده از توابع موجب سازمان‌دهی بهتر کد می‌شود.

برخی از عناصر پارادایم برنامه‌نویسی اعلانی را دیدیم که در زبان‌های برنامه‌نویسی رایج مورداستفاده قرار می‌گیرند. و درنهایت، به برنامه‌نویسی منطقی اشاره کردیم که پارادایم مطلوب هنگام کار در زمینه‌های بسیار خاص است.

---

[1] Single Assignment
[2] Logic Programming

امیدواریم که جرئت مقابله با هر زبان برنامه‌نویسی جدیدی را داشته باشید. همه‌ی آن‌ها چیزی برای ارائه دارند. اکنون آماده‌شده و شروع به کد نوشتن کنید!

مراجع

- Essentials of Programming Languages, by Friedman
  - Get it at https://code.energy/friedman
- Code Complete, by McConnell
  - Get it at https://code.energy/code-complete

نتیجه‌گیری

آموزش علوم کامپیوتر نمی‌تواند هیچ‌کس را به یک برنامه‌نویس خبره تبدیل کند، همان‌گونه که مطالعه‌ی قلم‌موها و رنگ‌دانه‌ها هیچ‌کس را به یک نقاش خبره تبدیل نمی‌کند.

– اریک اس. ریموند[1]

این کتاب مهم‌ترین مباحث علوم کامپیوتر را به‌صورت بسیار ساده ارائه کرده است. این حداقل چیزی است که یک برنامه‌نویس خوب باید در مورد علوم کامپیوتر بداند.

امیدوارم این دانش جدید شما را تشویق کند تا در موضوعاتی که دوست دارید عمیق‌تر شوید. به همین دلیل است که من پیوندهایی را به برخی از بهترین کتاب‌های مرجع در پایان هر فصل اضافه کرده‌ام.

برخی از موضوعات مهم در علوم کامپیوتر وجود دارند در این کتاب پوشش داده نشده‌اند. چگونه می‌توانید بین کامپیوترهای موجود در شبکه‌ای که کل سیاره زمین را پوشش می‌دهد (اینترنت) به روشی مطمئن ارتباط برقرار کنید؟ چگونه می‌توانید چندین پردازنده را وادار سازید به‌صورت هم‌زمان کار کنند تا یک کار محاسباتی را سریع‌تر انجام دهند؟ یکی از مهم‌ترین پارادایم‌های برنامه‌نویسی، برنامه‌نویسی شیءگرا نیز کنار گذاشته شد. قصد دارم در کتاب بعدی به این بخش‌های گمشده بپردازم.

همچنین، باید برنامه‌هایی بنویسید تا به‌طور کامل آنچه را که دیده‌ایم یاد بگیرید. و این چیز خوبی است. وقتی شروع به یادگیری نحوه‌ی انجام کارهای اساسی با یک زبان برنامه‌نویسی می‌کنید، کدنویسی ممکن است در ابتدا بی‌ارزش باشد. وقتی اصول اولیه را یاد گرفتید، قول می‌دهم که بسیار مفید خواهد بود. پس آماده‌شده و شروع به کدنویسی کنید.

در پایان، می‌خواهم بگویم که این اولین تلاش من برای نوشتن یک کتاب است. نمی‌دانم چقدر خوب پیش رفت. به همین دلیل نظرات شما در مورد این کتاب برای من بسیار ارزشمند خواهند بود. چه چیزی را در مورد آن دوست داشتید؟ کدام قسمت‌ها گیج‌کننده بودند؟ به نظر شما چگونه می‌توان آن را بهبود بخشید؟ برای من یک خط به آدرس hi@code.energy بنویسید.

---

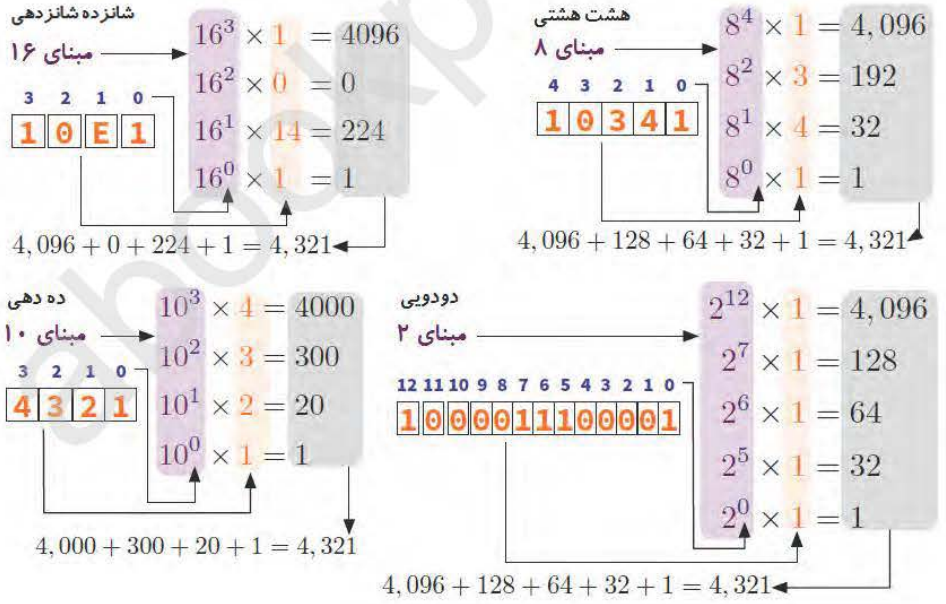
[1] Eric S. Raymond (-1957): برنامه‌نویس، محقق و تاریخ‌نگار فرهنگ هکرهای کامپیوتری .م.

پیوست

I. مبناهای عددی

محاسبات را می‌توان به کار با اعداد کاهش داد، زیرا اطلاعات قابل‌نمایش در قالب اعداد هستند. حروف را می‌توان به اعداد نگاشت کرد، بنابراین متن را نیز می‌توان به‌صورت عددی نوشت. رنگ‌ها ترکیبی از شدت نور قرمز، آبی و سبز هستند، که می‌توان آن را با عدد نشان داد. تصاویر را می‌توان با موزاییک‌هایی از مربع‌های رنگی نمایش داد، بنابراین می‌توان آن‌ها در قالب اعداد نیز بیان کرد.

سیستم‌های عددی قدیمی (مانند اعداد رومی: I، II، III و غیره) اعداد را به‌صورت مجموع ارقام نمایش می‌دهند. سیستم عددی مورداستفاده‌ی امروزی نیز بر اساس مجموع ارقام عمل می‌کند، ولی ارزش هر رقم در موقعیت i در d به توان i ضرب می‌شود، به‌طوری‌که d تعداد ارقام منحصربه‌فرد را نشان می‌دهد. به d **مبنا**[1] گفته می‌شود. ما به‌صورت معمول از $d = 10$ استفاده می‌کنیم، زیرا ده انگشت داریم، ولی این سیستم برای هر مبنای d کار می‌کند:



شکل ۱۰-۱: عدد ۴,۳۲۱ در مبناهای مختلف

[1] Base

II. ترفند گاوس

داستان به‌جایی برمی‌گردد که یک معلم دوره‌ی دبستان به‌عنوان تنبیه از گاوس خواست تمام اعداد بین ۱ تا ۱۰۰ را جمع بزند. در عین ناباوری معلم، گاوس در طی چند دقیقه با جواب ۵۰۵۰ به وی مراجعه کرد. ترفند او بازی کردن با ترتیب عناصر دو برابر مجموع بود:

$$2 \times \sum_{i=1}^{100} i = (1+2+3+\cdots+100)+(1+2+3+\cdots+100)$$
$$= \underbrace{(1+100)+(2+99)+\cdots+(99+2)+(100+1)}_{\text{100 زوج}}$$

$$= \underbrace{101+101+\cdots+101}_{\text{100 بار}} = 10{,}100$$

با تقسیم مقدار فوق به ۲، عدد ۵۰۵۰ به دست می‌آید. می‌توانیم این ترتیب دهی مجدد را به‌صورت رسمی زیر بنویسیم:

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n} (n+1-i)$$

ازاین‌رو

$$2 \times \sum_{i=1}^{n} i = \sum_{i=1}^{n} i + \sum_{i=1}^{n} (n+1-i) = \sum_{i=1}^{n} (i+n+1-i) = \sum_{i=1}^{n} (n+1)$$

در جواب آخر i وجود ندارد، بنابراین $(n+1)$ به تعداد n بار جمع شده است، ازاین‌رو

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

III. مجموعه‌ها

ما از عبارت **مجموعه**[1] برای توصیف تعدادی از اشیاء در کنار هم قرارگرفته استفاده می‌کنیم. به‌عنوان‌مثال، می‌توانیم S را مجموعه‌ی ایموجی‌های صورت میمون تعریف کنیم:

$$S = \{🙈, 🙉, 🙊, 🐵\}.$$

---

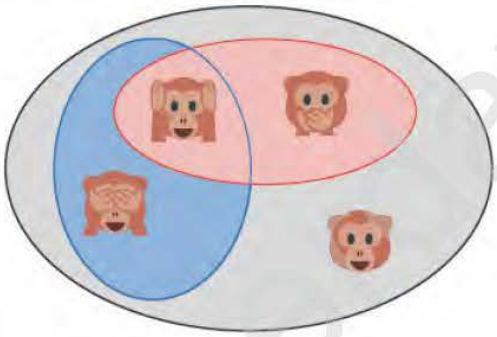
[1] Set

زیرمجموعه: یک مجموعه از اشیاء را که در درون یک مجموعه‌ی دیگر قرار داشته باشد، یک **زیرمجموعه**[۱] می‌نامیم. به‌عنوان‌مثال، میمون‌هایی را که دست‌هایشان را بر روی گوش‌ها و دهانشان گذاشته‌اند در نظر بگیرید:

$$S_1 = \{🙉, 🙊\}$$

هر دو میمون S_1 در S نیز وجود دارند. این پدیده را به‌صورت $S_1 \subset S$ می‌نویسیم. می‌توانیم میمون‌هایی را که دست‌هایشان را بر روی چشم‌ها و گوش‌هایشان گذاشته‌اند در یک مجموعه دیگر قرار دهیم:

$$S_2 = \{🙈, 🙉\}$$



شکل ۱۰-۲: S_1 و S_2 زیرمجموعه‌های S هستند

اجتماع: کدام میمون‌ها به S_1 یا S_2 تعلق دارند؟ جواب میمون‌هایی هستند که در S_3 قرار دارند:

$$S_3 = \{🙈, 🙉, 🙊\}$$

این مجموعه‌ی جدید را **اجتماع**[۲] دو مجموعه‌ی قبلی می‌نامیم. این پدیده را به‌صورت $S_3 = S_1 \cup S_2$ می‌نویسیم.

اشتراک: کدام میمون‌ها به S_1 و S_2 تعلق دارند؟ جواب میمون‌هایی هستند که در S_4 قرار دارند:

$$S_4 = \{🙉\}$$

این مجموعه‌ی جدید را **اشتراک**[۳] دو مجموعه‌ی قبلی می‌نامیم. این پدیده را به‌صورت $S_3 = S_1 \cap S_2$ می‌نویسیم.

مجموعه‌های توانی: توجه داشته باشید که S_3 و S_4 هر دو زیرمجموعه‌های S هستند. هم‌چنین $S_5 = S$ و مجموعه‌ی تهی $S_6 = \{\}$ را نیز در نظر می‌گیریم که هر دو زیرمجموعه‌های S هستند. اگر تمام

---

۱ Subset
۲ Union
۳ Intersection

زیرمجموعه‌های S را بشماریم به $2^4 = 16$ زیرمجموعه می‌رسیم. اگر همه‌ی این زیرمجموعه‌ها را به‌صورت شیء ببینیم، می‌توانیم با کنار هم قرار دادن آن‌ها یک مجموعه‌ی دیگر بسازیم. مجموعه‌ی تمام زیرمجموعه‌های S را **مجموعه‌ی توانی**[1] می‌نامیم:

$$P_S = \{S_1, S_2, S_3, \dots, S_{16}\}$$

IV. الگوریتم کادان

در بخش ۳-۳ مسئله‌ی بهترین تجارت را معرفی کردیم:

> بهترین تجارت: شما قیمت روزانه‌ی طلا را برای یک دوره‌ی زمانی دارید. می‌خواهید دو روز را در این دوره بیابید که اگر در آن‌ها ابتدا طلا بخرید و سپس بفروشید، حداکثر سود را به دست می‌آورید.

در بخش ۳-۷ الگوریتمی با زمان $O(n)$ و فضای $O(n)$ برای حل این مسئله ارائه کردیم. زمانی که جی کادان[2] در سال ۱۹۸۴ الگوریتم زیر را کشف کرد، نشان داد که چگونه می‌توان این مسئله را در زمان $O(n)$ و فضای $O(1)$ حل کرد:

```
function trade_kadane(prices):
    sell_day ← 1
    buy_day ← 1
    best_profit ← 0
    for each s from 2 to prices.length
        if prices[s] < prices[buy_day]
            b ← s
        else
            b ← buy_day
        profit ← prices[s] - prices[b]
        if profit > best_profit
            sell_day ← s
            buy_day ← b
            best_profit ← profit
    return (sell_day, buy_day)
```

دلیل آن است که نیازی به ذخیره کردن بهترین روز خرید به ازای هرروز در ورودی نداریم. فقط کافی است بهترین روز خرید نسبی مرتبط با بهترین روز فروش تاکنون را ذخیره کنیم.

---

[1] Power Set
[2] Jay Kadane (1941-)

COMPUTER SCIENCE DISTILLED

Learn the Art of Solving Computational Problems

By:
Wladston Ferreira Filho

Translated By:
Ali Naserasadi
Computer Group, Zarand Higher Education Complex

Ali Rahnama
Computer Group, Zarand Higher Education Complex

COMPUTER SCIENCE DISTILLED

Learn the Art of Solving Computational Problems

Wladston Ferreira Filho

این کتاب سعی دارد به زبانی بسیار ساده و با بیان مثال‌ها و نکات قابل‌درک و آسان، به واکاوی مهم‌ترین مبانی و مفاهیم علوم کامپیوتر بپردازد. درواقع، هدف اصلی این کتاب ارائه‌ی مطالبی است که ممکن است مورد سؤال بسیاری از افراد بوده ولی به دلیل عدم وجود مرجعی مناسب برای پاسخگویی به آن‌ها، برای ایشان حل‌نشده باقی‌مانده است. به‌این‌ترتیب، این کتاب می‌تواند منبعی مناسب برای دانشجویان جدیدالورود رشته‌ی کامپیوتر، دانش‌آموزان دبیرستانی و هر فرد دیگری که علاقه‌مند به آشنایی با نحوه‌ی عملکرد و مفاهیم بنیادین کامپیوترها است، باشد. باید توجه داشت که تجارب نویسنده‌ی کتاب به‌عنوان یک برنامه‌نویس موفق، نقش چشمگیری در کیفیت مطالب ارائه‌شده داشته است.

Translated By:

Ali Naserasadi
Ali Rahnama

ارشد من
9786220855804