طراحی الگوریتم

Algorithms Unlocked

تالیف: توماس اچ. کورمن

ترجمه: محسن کجباف

انتشارات پندار پارس

سرشناسه	:	کورمن، تامس Cormen, Thomas H
عنوان و نام پدیدآور	:	طراحی الگوریتم/ تالیف توماس اچ. کورمن ؛ ترجمه محسن کجباف.
مشخصات نشر	:	تهران : پندار پارس ، ۱۳۹۴.
مشخصات ظاهری	:	۲۱۸ ص.
شابک	:	۱۸۰۰۰۰ ریال: ۸-۰۳-۸۲۰۱-۶۰۰-۹۷۸
وضعیت فهرست نویسی	:	فیپا
یادداشت	:	عنوان اصلی: .Algorithms Unlocked, ۲۰۱۳
موضوع	:	الگوریتم‌های کامپیوتری
شناسه افزوده	:	کجباف، محسن، ۱۳۶۵ -، مترجم
رده بندی کنگره	:	۹/۷۶QA الف/ ۱۳۹۴۷ ۹ک
رده بندی دیویی	:	۰۰۵/۱
شماره کتابشناسی ملی	:	۴۱۹۱۹۴۹

انتشارات پندارپارس

دفتر فروش: انقلاب، ابتدای کارگرجنوبی، کوی رشتچی، شماره 14، واحد 16 **www.pendarepars.com**

تلفن: 66572335 - تلفکس: 66926578همراه: 09122452348 **info@pendarepars.com**

نام کتاب	:	**طراحی الگوریتم**
ناشر	:	**انتشارات پندار پارس**
تالیف	:	**توماس کورمن**
ترجمه	:	**محسن کجباف**
چاپ نخست	:	**اردیبهشت 95**
شمارگان	:	**500 نسخه**
طرح جلد	:	**سارا یعسوبی**
چاپ، صحافی	:	**روز**
قیمت	:	**18000 تومان** **شابک** : 978-600-8201-03-8

مقدمه

الگوریتم مجموعه‌ای متناهی از دستورالعمل‌ها است، که به ترتیب خاصی اجرا می‌شوند و مسئله‌ای را حل می‌کنند. به عبارت دیگر یک الگوریتم، روشی گام به گام برای حل مسئله است. واژه الگوریتم از نام ریاضیدان و ستاره‌شناس نامی ایرانی، ابوجعفر محمد بن موسی خوارزمی(الخوارزمی) گرفته شده است.

پیش نیاز درس طراحی الگوریتم ها، درس ساختمان داده ها می باشد. مطالب پیش نیاز این درس، آشنایی اولیه با مفاهیم ساختمان داده ها و الگوریتم ها است. هدف این درس آموزش روش‌های تجزیه و تحلیل و طراحی الگوریتم‌ها می‌باشد. در این درس دانشجویان می‌آموزند که چگونه یک مسئله را تحلیل نموده و انواع الگوریتم‌های احتمالی برای حل آن را پیدا کنند. سپس راه حل‌های الگوریتمی مبتنی بر هر نوع را یافته، آنها را از نظر پیچیدگی محاسباتی تحلیل و مقایسه نموده و بر اساس اندازه و ویژگی‌های ورودی مسئله، بهترین آنها را برای یک کاربرد خاص مهندسی انتخاب نمایند.

به دلیل اهمیت این درس در دوره کارشناسی در رشته‌های کامپیوتر، فن‌آوری اطلاعات، علوم کامپیوتر تصمیم گرفتم کتاب Algorithms Unlocked که مولف آن پرفسور Thomas Cormen است را ترجمه نمایم. مولف این کتاب، نویسنده اول کتاب بسیار مشهور Introduction To Algorithms (این کتاب با عنوان CLRS شهرت یافته است) می‌باشد. کتاب حاضر بنا به گفته پرفسور Thomas Cormen در مقدمه کتاب، پیش نیازی برای کتاب Introduction To Algorithms است. کتاب Algorithms Unlocked به طور کامل برگرفته از کتاب Introduction To Algorithms است. این کتاب به طور شیوا و با زبان بسیار ساده، مباحث مربوط به تحلیل و طراحی الگوریتم‌ها را بیان کرده است. امیدورام این اثر در بالابردن دانش خوانندگان عزیز در زمینه طراحی الگوریتم‌ها، موثر باشد.

با توجه به این که هیچ ترجمه‌ای عاری از نقص نیست، از همه خوانندگان عزیز تقاضا دارم پیشنهادهای خود را از طریق پست الکترونیکی M_kajbaf@yahoo.com با اینجانب در میان بگذارند. همچنین از مدیریت محترم انتشارات پندارپارس جناب آقای مهندس حسین یعسوبی که با چاپ این اثر در انتشارات پندارپارس موافقت نمودند، کمال تشکر و سپاسگزاری را دارم.

محسن کجباف

اسفندماه94

تقدیم به شهدای مدافع حرم

سردار شهید حسین همدانی

سردارشهید هادی کجباف

سردار شهید سیدحمید تقوی فر

سردار شهید جبار دریساوی

و

سایر شهدای مدافع حرم

فهرست

فصل 1: الگوریتم‌ها چه هستند؟ 1
درستی 2
الگوریتم‌های کامپیوتری برای افراد غیرکامپیوتری 4
الگوریتم‌های کامپیوتری برای افراد کامپیوتری 5
مطالعه‌ی بیشتر 6
فصل 2: شرح و ارزیابی الگوریتم‌های کامپیوتری 9
شرح الگوریتم‌های کامپیوتری 9
چگونگی مشخص کردن زمان‌های اجرا 16
ثابت های حلقه 20
فصل 3: الگوریتم‌هایی برای مرتب‌سازی و جست‌وجو 25
جست‌وجوی دودویی 28
مرتب‌سازی به روش انتخابی 32
مرتب‌سازی درجی 36
مرتب‌سازی ادغامی 41
مرتب‌سازی سریع 52
فصل 4: یک کران پائین برای مرتب‌سازی و چگونگی گذر از آن 67
اصولی برای مرتب سازی 67
کران پائین در مرتب‌سازی مقایسه‌ای 69
پایین آوردن کران پائین با مرتب‌سازی شمارشی 70
مرتب‌سازی مبنایی 75
فصل 5: گراف جهت‌دار بدون دور 80
گراف جهت‌دار بدون دور 84
مرتب‌سازی توپولوژیک 85
چگونگی نمایش یک گراف جهت‌دار 89
زمان اجرای مرتب‌سازی توپولوژیکی 91
مسیر بحرانی در نمودار PERT 92
کوتاه‌ترین مسیر در یک گراف جهت‌دار بدون دور 97
مطالعه بیشتر 101
فصل 6: کوتاه‌ترین مسیرها 103
الگوریتم Dijkstra 105
الگوریتم Dijkstra ثابت حلقه‌ی زیر را نگهداری می‌کند 109
اجرای ساده آرایه 111
اجرای هیپ دودویی 112
اجرای هیپ فیبوناچی 114
الگوریتم بلمن-فورد 115
الگوریتم فلوید- وارشال 119
فصل 7: الگوریتم رشته‌ها 127
طولانی‌ترین زیردنباله‌ی مشترک 128
تغییر شکل دادن یک دنباله به دنباله‌ی دیگر 134
تطبیق رشته 143

مطالعه‌ی بیشتر 151
فصل 8 : مبانی رمزنگاری 153
رمزهای جایگشتی ساده 155
رمزنگاری کلید متقارن 157
padهای یک زمانه 157
رمزهای بلوکی و دنباله‌سازی 159
توافق بر سر اطلاعات مشترک 160
رمزنگاری کلید عمومی 161
سیستم رمزی RSA 163
چگونگی انجام محاسبات با اعداد بزرگ 167
چگونگی یافتن عدد اول بزرگ 168
چگونگی یافتن عددی که نسبت به عدد دیگر اول باشد 169
چگونگی محاسبه‌ی معکوس در حساب پیمانه‌ای 170
چگونگی رساندن سریع یک عدد به توان یک رقم 170
رمزنویسی چندگانه 172
محاسبه‌ی اعداد تصادفی 173
مطالعه‌ی بیشتر 174
فصل 9 : فشرده‌سازی داده 177
کدهای هافمن 180
پیشرفت‌های LZW 197
مطالعه‌ی بیشتر 198
فصل 10 : مسائل سخت 199
کامیون‌های قهوه‌ای 199
دسته‌های NP، p ، NP-completeness 203
مسائل تصمیم‌گیری و ساده‌سازی‌ها 205

فصل 1

الگوریتم‌ها چه هستند؟

الگوریتم‌ها چه هستند و چرا باید مورد توجه قرار گیرند؟ اجازه دهید با این آغاز کنیم که غالبا پرسیده می‌شود: الگوریتم چیست؟ پاسخ معمول عبارت است از مجموعه‌ای از گام‌ها برای انجام دادن یک کار.

شما در انجام کارهای روزانه‌ی خود با الگوریتم‌ها سروکار دارید، یک الگوریتم برای مسواک زدن دندان‌های خود دارید: باز کردن درب تیوپ خمیر دندان، برداشتن مسواکتان، مالیدن خمیر دندان بر روی مسواک به اندازه مورد نظرتان، بستن درب تیوپ خمیر دندان، جای دادن مسواک در یک چهارم دهانتان، به بالا و پائین حرکت دادن مسواک برای چند ثانیه و غیره. و به همین صورت، اگر مجبور به انجام کاری باشید یک الگوریتم برای انجام آن خواهید داشت.

به هر روی، این کتاب در مورد الگوریتم‌هایی که در کامپیوترها و به‌صورت عمومی‌تر در دستگاه‌های محاسباتی اجرا می‌شوند، نوشته شده است. همانگونه که الگوریتم‌هایی که به‌کار می‌بندید بر زندگی روزمره‌تان اثر می‌گذارند، الگوریتم‌هایی که در کامپیوترها اجرا می‌شوند، به همان صورت بر آن اثر گذارند.

آیا از GPS خود برای مسیریابی هنگام سفر استفاده می کنید؟ GPS برای این منظور الگوریتمی که ما بر آن نام کوتاه‌ترین مسیر را نهاده‌ایم به کار می‌برد. آیا از طریق اینترنت محصولاتی را خریداری می‌نمایید؟ پس، از یک وب‌سایت ایمن که یک الگوریتم رمزنگاری را اجرا می‌کند، استفاده می‌کنید. وقتی محصولاتی را از طریق اینترنت خریداری می‌کنید، آیا آنها از طریق سرویس‌های حمل ونقل خصوصی تحویل شما می‌شوند؟ این سرویس از الگوریتم‌هایی برای تخصیص بسته‌ها به هر کامیون و سپس برای تشخیص اینکه هر راننده می‌بایست کدام بسته را تحویل دهد، استفاده می‌کند. الگوریتم بر روی کامپیوترها، در هر کجا که هستند بر روی لپ‌تاپ - برروی سرورها- برروی گوشی‌ها- برروی سیستم‌های تعبیه شده (مانند سیستم‌های خودرو - اجاق ماکروویو یا سیستم‌های کنترلی) فارغ از مکان آنها، اجرا می‌شوند.

چه وجه تمایزی بین الگوریتمی که بر روی یک کامپیوتر اجرا می‌شود و یک الگوریتم که شما اجرا می‌کنید وجود دارد؟ وقتی که یک الگوریتم به صورت دقیق توضیح داده نشده باشد، ممکن است قادر به اجرای آن باشید، ولی یک کامپیوتر قادر به این کار نیست. برای نمونه، اگر به سمت محل کار خود

در حال رانندگی باشید، الگوریتم رانندگی تا محل کار ممکن است بگوید، اگر ترافیک بد است مسیر جایگزین را انتخاب کن. در این بین اگرچه معنی ترافیک بد را می‌دانید ولی یک کامپیوتر این را نمی-داند.

بنابراین یک الگوریتم کامپیوتر مجموعه‌ای است از گام‌های متوالی برای انجام وظیفه‌ای که به اندازه کافی توضیح داده شده؛ به‌گونه‌ای که کامپیوتر می‌تواند آن را انجام دهد. اگر حتی مقداری برنامه‌نویسی در جاوا، C، C++، پایتون، فورترن، متلب و از این گونه برنامه‌ها انجام داده باشید، درکی از آن سطح دقت دارید. اگر تاکنون حتی یک برنامه کامپیوتری ننوشته باشید، پس از دیدن اینکه چگونه الگوریتم‌ها را در این کتاب توضیح می‌دهم، شاید حسی نسبتا سطحی از دقت را پیدا کنید.

اجازه دهید به سراغ پرسش بعد برویم؛ از الگوریتم کامپیوتری چه چیزی می‌خواهیم؟ الگوریتم‌های کامپیوتری مسائل محاسباتی را حل می‌کنند. از الگوریتم کامپیوتری دو چیز می‌خواهیم: دادن یک ورودی به یک مسئله (ورودی داده شده می‌بایست همیشه یک راه‌حل صحیح برای مسئله تولید کند)، و نیز استفاده بهینه از منابع محاسباتی در زمان انجام آن. اجازه دهید این دو ضرورت را امتحان کنیم:

درستی

تولید یک راه‌حل صحیح برای یک مسئله به چه معنی است؟ معمولا می‌توانیم به صورت دقیق مشخص کنیم که راه‌حل صحیح مستلزم چیست. برای نمونه اگر GPS راه‌حل درستی برای پیدا کردن بهترین مسیر سفر را تولید کند، آن راه‌حل می‌بایست مسیری از بین همه مسیرهای ممکن باشد که میان مبدأ تا مقصد وجود دارد و شما را در کوتاه‌ترین زمان به آنجا برساند و شاید مسیری است که کوتاه‌ترین فاصله را دارد و یا مسیری است که شما را زودتر رسانده و از پرداخت عوارض جلوگیری کند. البته اطلاعاتی که GPS استفاده می‌کند، ممکن است با واقعیت تطابق نداشته باشند، به جز در حالتی که GPS شما به اطلاعات آنی ترافیک دسترسی دارد. این دستگاه برای محاسبه زمان سفر در طول یک مسیر ممکن است از این فرض استفاده کند که زمان سفر یک جاده برابر با نسبت فاصله جاده به محدودیت سرعت در آن مسیر است. در این صورت، اگر جاده پر ترافیک باشد و به دنبال سریع‌ترین راه ممکن برای رسیدن به مقصد باشید ممکن است GPS اطلاعات اشتباهی به شما بدهد.

هنوز می‌توانیم بگوییم که الگوریتم مسیریابی که GPS استفاده می‌کند صحیح می‌باشد؛ هرچند که حتی ورودی الگوریتم صحیح نباشد. برای ورودی داده شده به الگوریتم مسیریابی نتیجه محاسباتی الگوریتم سریع‌ترین مسیر می‌باشد.

اکنون برای برخی مسائل بسیار سخت و حتی غیرممکن است که بگوییم خروجی الگوریتم صحیح است. برای نمونه تشخیص نوری کاراکترها را در نظر بگیرید، آیا این تصویر 6*11 پیکسل، عدد 5 است یا حرف S؟

برخی افراد ممکن است بگویند عدد 5 برخی دیگر بگویند حرف S، بنابراین چگونه می‌توانیم تشخیص دهیم که تصمیم کامپیوتر صحیح بوده است یا اشتباه؟ نخواهیم توانست. در این کتاب بر روی الگوریتم‌های کامپیوتری که دارای راه‌حل‌های قابل فهم می‌باشند، تمرکز خواهیم کرد. گاهی می‌توانیم بپذیریم که الگوریتم کامپیوتر ممکن است پاسخ اشتباهی تولید کند، همچنین می‌توانیم کنترلی بر اینکه این موضوع چند بار اتفاق می‌افتد داشته باشیم.

پاسخ این است که این کار را با ادغام دو ایده انجام می‌دهیم. ابتدا مشخص می‌کنیم این الگوریتم به عنوان تابعی از اندازه‌ی ورودی‌اش، چقدر زمان صرف می‌کند. در نمونه‌ی مربوط به مسیریابی، ورودی باید شکلی از یک نقشه‌ی راه باشد و اندازه‌ی آن به تعداد تقاطع‌ها و تعداد مسیرهای مرتبط کننده‌ی تقاطع‌های درون نقشه بستگی دارد. (اندازه‌ی فیزیکی شبکه‌ی راه مهم نیست زیرا می‌توان تمام فواصل را با اعداد شرح داد و تمام اعداد، اندازه‌ی مشابهی را در ورودی اشغال می‌کنند. طول یک مسیر هیچ تأثیری بر اندازه‌ی ورودی ندارد). نمونه‌ای ساده‌تر جست‌وجوی یک لیست خاصی از عنصرها برای مشخص کردن اینکه آیا یک عنصر خاص در لیست وجود دارد یا نه؛ اندازه‌ی ورودی معادل تعداد عناصر لیست است.

دوما ما بر سرعت رشد تابعی که زمان اجرا را مشخص می‌کند، با اندازه‌ی ورودی، تمرکز می‌کنیم. سرعت رشد زمان اجرا در فصل 2 با نمادهایی مواجه می‌شویم که از آن‌ها برای شرح زمان اجرای یک الگوریتم استفاده خواهیم کرد. اما جالب‌ترین نکته در نحوه نگرش ما به موضوع این است که ما تنها به یک عبارت با بیشترین تاثیر در زمان اجرا توجه می‌کنیم و ضرایب را در نظر نمی‌گیریم. به بیان دیگر، ما بر «درجه‌ی رشد» زمان اجرا تمرکز می‌کنیم. مثلا فرض کنید می‌توانستیم مشخص کنیم که اجرای خاص یک الگوریتم خاص برای جست‌وجوی لیستی از n عنصر به 50n+125 سیکل ساعت ماشینی نیاز دارد. عبارت 50n بر عبارت 125 غالب است، و این وقتی است که n به اندازه کافی بزرگ باشد که در این حالت از $n \geq 3$ آغاز شده و از نظر غلبه، حتی برای اندازه‌های لیستی بزرگ‌تر هم افزایش می‌یابد. بنابراین هنگام شرح زمان اجرای این الگوریتم فرضی، عبارت 125 را در نظر نمی-گیریم که درجه‌ی پائینی دارد. آنچه می‌تواند باعث تعجب شما شود این است که ما ضریب 50 را هم

حذف می‌کنیم و نشان می‌دهیم که زمان اجرا به‌طور خطی با سایز ورودی n رشد می‌کند. به عنوان یک مثال دیگر اگر الگوریتمی نیاز به $20n^3+100n^2+300n+200$ سیکل ساعت ماشینی داشته باشد، خواهیم گفت که زمان اجرای آن به صورت n^3 افزایش خواهد یافت. باز، همچنان که مقدار ورودی n در حال افزایش است عبارات درجه پائین $100n^2$، 300n و 200 بی اهمیت‌تر می‌شوند.

در عمل ضرایبی که نادیده می‌گیریم مهم هستند، اما آنها وابستگی شدیدی به فاکتورهای بیرونی دارند و این مسئله کاملا محتمل است که اگر دو الگوریتم A و B که دارای یک درجه‌ی رشد بوده و در یک ورودی اجرا می‌شوند را مقایسه کنیم؛ در آن صورت A می‌تواند با ترکیب خاصی از ماشین، زبان برنامه‌نویسی، کامپایلر/ مفسر و برنامه‌نویس سریع‌تر از B اجرا شود در حالی که B می‌تواند با یک ترکیب دیگر سریع‌تر از A اجرا شود. البته اگر الگوریتم‌های A و B راه‌حل‌های درستی را به دست آورند و A همیشه با سرعت دو برابر B اجرا شود، و از طرفی تمام موارد دیگر برابر باشند، در این صورت همیشه ترجیح می‌دهیم A را به جای B اجرا کنیم. با این حال از دیدگاه مقایسه‌ی انتزاعی الگوریتم‌ها ما بر درجه‌ی رشد تمرکز می‌کنیم که توسط ضرایب یا عبارات درجه پائین بیان نمی‌شود. پرسش نهایی که در این فصل می‌پرسیم این است که: چرا باید الگوریتم‌های کامپیوتری را مورد توجه قرار دهم؟ پاسخ به این مسئله بستگی دارد که شما چه کسی هستید.

الگوریتم‌های کامپیوتری برای افراد غیرکامپیوتری

حتی اگر خود را یکی از اعضای جامعه‌ی کامپیوتری به‌شمار نیاورید، الگوریتم‌های کامپیوتری برای شما مهم هستند و هر روز از آن‌ها استفاده می‌کنید. به صورت روزانه از GPSها استفاده می‌کنید، مگر اینکه در یک مأموریت حیات وحش بدون GPS باشید. آیا امروز چیزی را در اینترنت جست‌وجو کردید؟ موتور جست‌وجویی که به کار برده‌اید، خواه گوگل باشد یا بینگ یا هر موتور جست‌وجوی دیگر، از الگوریتم‌های پیشرفته‌ای برای جست‌وجوی وب و تصمیم‌گیری در مورد ترتیب ارائه‌ی نتایج استفاده می‌کنند. آیا امروز با ماشین خود جایی رفته‌اید؟ کامپیوترهای موجود در ماشین‌ها در طول سفر میلیون‌ها تصمیم می‌گیرند که همگی بر مبنای الگوریتم‌ها هستند؛ مگر اینکه خودروی شما از نوع کلاسیک باشد. می‌توان به همین ترتیب، نمونه ارائه کرد.

شما به عنوان کاربر نهایی الگوریتم‌ها باید خود را مکلف کنید تا اطلاعات بیشتری در مورد چگونگی طراحی، شرح و ارزیابی آن‌ها بیاموزید. فرض می‌کنم علاقه‌ی متوسطی به این مسئله دارید، زیرا این کتاب را انتخاب کرده‌اید و به این بخش رسیده‌اید.

الگوریتم‌های کامپیوتری برای افراد کامپیوتری

اگر اهل کامپیوتر باشید، پس توجه بهتری نسبت به الگوریتم‌های کامپیوتری دارید. الگوریتم‌ها نه تنها در مرکز تمام اتفاقات قرار دارند، بلکه نوعی از تکنولوژی هستند که درون کامپیوتر جریان دارند. می‌توانید هزینه‌ی بالایی برای خرید کامپیوتری با جدیدترین و بزرگ‌ترین پردازشگر بپردازید اما نیاز به این دارید که الگوریتم‌های خوبی در آن کامپیوتر اجرا کنید تا پولتان هدر نرفته باشد.

در اینجا نمونه‌ای وجود دارد که نشان می‌دهد چگونه الگوریتم‌ها واقعا یک تکنولوژی هستند. در فصل 3 چند الگوریتم متفاوت را خواهیم دید که لیستی از n عدد را به ترتیب صعودی مرتب‌سازی می‌کند. زمان اجرای برخی از این الگوریتم‌ها به صورت n^2 افزایش می‌یابد. اما زمان اجرای برخی دیگر به صورت n lg n است. Log n چیست؟ در واقع، لگاریتم 2 از n یا $\log_2^n$ است. دانشمندان علوم کامپیوتر درست همانند دانشمندان و ریاضی‌دانان که از $\ln n$ به عنوان شکل مختصر برای الگوریتم طبیعی ($\log_e^n$) استفاده می‌کنند، پی‌درپی از شکل کوتاه خاص خود برای لگاریتم پایه 2 استفاده می‌کنند. اکنون، چون تابع $\lg n$ معکوس یک تابع توانی است، به کندی با n رشد می‌کند. اگر $n = 2^x$ پس $x = \lg n$. مثلا $2^{10} = 1024$ پس lg1024 تنها 10 است؛ همچنین برای 2^{20}=1048576؛ پس $lg1048576$ تنها 20 است. بنابراین رشد $n \lg n$ در برابر n^2 یک فاکتور n را تنها برای یک فاکتور $\lg n$ مبادله می‌کند و این کاری است که باید هر روز انجام دهید.

اجازه دهید تا این نمونه را پخته‌تر کنیم. این عمل را با به کار بردن کامپیوتری سریع‌تر (کامپیوتر A) که یک الگوریتم مرتب‌سازی که زمان اجرای آن برای n مقدار به شکل n^2 افزایش می‌یابد، و در مقابل کامپیوتر کندتر (کامپیوتر B) که زمان اجرای این الگوریتم به صورت $n \lg n$ رشد می‌کند، انجام می‌دهیم. هر کدام از آن‌ها باید ردیفی از 10 میلیون عدد را مرتب‌سازی کند. (هرچند ممکن است 10 میلیون عدد بسیار زیاد به نظر برسد اما اگر اعداد صحیح 8 بایتی باشند در این صورت حدود 80 مگابایت را اشغال می‌کنند که حافظه‌ی یک لپ‌تاپ ارزان هم چندین برابر این مقدار است).

فرض کنید که کامپیوتر A در هر ثانیه 10 میلیارد دستورالعمل را اجرا کند (سریع‌تر از هر تک‌کامپیوتر سریع در زمان نوشتن کتاب) و کامپیوتر B تنها 10 میلیون دستورالعمل را در ثانیه اجرا می‌کند به گونه‌ای که کامپیوتر A در قدرت محاسبه‌ی ردیفی 1000 بار سریع‌تر از کامپیوتر B است. برای اینکه این تفاوت شدیدتر شود فرض کنید که ماهرترین برنامه‌نویس جهان، زبان ماشینی را برای کامپیوتر رمزگذاری می‌کند و رمز حاصل به $2n^2$ دستورالعمل برای مرتب‌سازی n عدد نیاز دارد. باز هم فرض کنید که یک برنامه‌نویس متوسط با استفاده از یک زبان سطح بالا با یک کامپایلر کم بازده برای کامپیوتر B می‌نویسد و رمز حاصل $50 n \lg n$ دستورالعمل دارد. برای مرتب‌سازی 10 میلیون عدد، کامپیوتر A به زمان زیر نیاز دارد:

$$\frac{2.(10^7)^2\ instruction}{10^{10}\ instruction/second} = 20000\ second$$

که بیش از 5/5 ساعت است اما در مورد کامپیوتر B :

$$\frac{50\times10^7\lg10^7 \quad \text{دستورالعمل}}{10^7 \quad \text{ثانیه/ دستورالعمل}} \approx 1163 \text{ ثانیه}$$

که کمتر از 20 دقیقه است. کامپیوتر B با اجرای الگوریتمی که زمان اجرای آن بسیار کندتر رشد می-کند (حتی با یک کامپایلر ضعیف) سرعتی 17 برابر نسبت به کامپیوتر A دارد! مزیت الگوریتم $n\lg n$ هنگام مرتب‌سازی 100 میلیون عدد از این هم بزرگ‌تر است: که الگوریتم n^2 در کامپیوتر A بیش از 23 روز طول می‌کشد؛ اما الگوریتم $n\lg n$ در کامپیوتر B کمتر از 4 ساعت طول می‌کشد. در کل با افزایش اندازه‌ی مسئله، مزیت نسبی الگوریتم $n\lg n$ هم افزایش می‌یابد.

حتی با پیشرفت‌های عظیمی که به صورت پی‌درپی در سخت‌افزار کامپیوترها مشاهده می‌کنیم، عملکرد کلی سیستم افزون بر انتخاب سخت‌افزار سریع یا سیستم‌عامل‌های مؤثر به انتخاب الگوریتم-های مؤثر بستگی دارد. پیشرفت‌های سریع دیگر تکنولوژی‌های کامپیوتری در الگوریتم‌ها هم در حال رخ دادن هستند.

مطالعه‌ی بیشتر

نظر فردی من این است که، شفاف‌ترین و سودمندترین منبع الگوریتم‌های کامپیوتری «مقدمه‌ای بر الگوریتم‌ها» است، که توسط چهار فرد خبره نوشته شده است. این کتاب بر مبنای ابتدای نام چهار نویسنده‌ی خود، به طور اختصاری CLRS نامیده می‌شود. بسیاری از این مطالب را از این کتاب استخراج کردم. هرچند این کتاب بسیار کامل‌تر از کتاب من است اما فرض می‌کند که دست‌کم برنامه-ریزی کامپیوتری اندکی انجام داده است؛ از سویی، پافشاری بر ریاضیات ندارد. اگر دریافتید که با سطح ریاضیات این کتاب راحت هستید و برای بررسی عمیق‌تر آماده هستید، این کتاب می‌تواند بهتر از CLRS باشد.

کتاب جان مک کورمیک با عنوان «9 الگوریتمی که آینده را تغییر دادند»؛ تعدادی از الگوریتم‌ها و جنبه‌هایی از محاسبه را که بر زندگی روزانه‌ی ما تأثیر دارند شرح می‌دهد. روش مک کورمیک نسبت به این کتاب ساده‌تر است. اگر فکر می‌کنید دستاورد من در این کتاب خیلی ریاضیاتی است، توصیه می‌کنم که کتاب مک کورمیک را بخوانید. اگر کمی با ریاضیات آشنا باشید می‌توانید به خوبی درک کنید. اگر فکر می‌کنید CLRS خیلی سخت است، می‌توانید به سراغ کتاب چند جلدی دونالد نات به نام «هنر برنامه‌نویسی کامپیوتری» بروید. هرچند عنوان این سری شبیه به کتابی برای جزئیات

کدنویسی است، اما شامل آنالیزهای عالی و عمیق از الگوریتم‌ها هستند. با این حال آگاه باشید که محتویات TAOCP بسیار زیادند. به هر روی، اگر مشتاقید بدانید واژه الگوریتم از کجا آمده است، نات می‌گوید: از نام خوارزمی گرفته شده است که یکی از ریاضیدانان ایرانی قرن 9 است. افزون بر CLRS چند متن عالی دیگر هم در طی سال‌های گذشته منتشر شدند. نکته‌های فصل 1 کتاب CLRS بسیاری از این متون را فهرست می‌کند. به جای تکرار آن‌ها پیشنهاد می‌کنم به خود کتاب رجوع شود.

فصل 2

شرح و ارزیابی الگوریتم‌های کامپیوتری

در فصل پیش ملاحظه کردید چگونه زمان اجرای یک الگوریتم کامپیوتری را مدیریت می‌کنیم: با تمرکز بر زمان اجرا به عنوان تابعی از اندازه ورودی و به ویژه، با تمرکز بر مرتبه بزرگی رشد زمان اجرا. در این فصل کمی به عقب بازمی‌گردیم و می‌بینیم که چگونه الگوریتم‌های کامپیوتری را شرح می‌دهیم. سپس نمادهایی را خواهیم دید که، برای شرح زمان اجرای الگوریتم‌ها آنها را به کار می‌بریم. در آخر، این فصل را با بررسی چند تکنیک که از آنها برای طراحی و درک الگوریتم‌ها استفاده می‌کنیم، به پایان می‌رسانیم.

شرح الگوریتم‌های کامپیوتری

همیشه گزینه‌ی شرح یک الگوریتم کامپیوتری را به صورت یک برنامه‌ی قابل اجرا در یک زبان برنامه‌نویسی رایج مانند جاوا، C، C++ یا Python در نظر می‌گیریم. در واقع برای انجام این کار چند کتاب درسی لازم است. مشکل کاربرد زبان‌های برنامه‌نویسی واقعی برای مشخص کردن الگوریتم‌ها این است که ممکن است در جزئیات زبان گرفتار شوید و با این کار ایده‌های ورای الگوریتم‌ها مبهم شوند. نگرش دیگری که در "مقدمه‌ای بر الگوریتم‌ها" اتخاذ کردیم، از "شبه کد" استفاده می‌کند که همانند ترکیبی از زبان‌های مختلف برنامه‌نویسی است که انگلیسی هم در آن ادغام شده است. اگر تاکنون از زبان برنامه‌نویسی واقعی استفاده کرده باشید، می‌توانید شبه کد را به سادگی درک کنید. اما اگر هرگز برنامه‌نویسی نکرده‌اید شبه کد، کمی عجیب به نظر می‌رسد.

نگرشی که در این کتاب در پیش گرفتم؛ این است که تلاش نمی‌کنم الگوریتم‌ها را برای نرم‌افزار یا سخت‌افزار شرح دهم، بلکه هدف، توضیح آن برای wetware است. همچنین فرض می‌کنم که هرگز یک برنامه‌ی کامپیوتری ننوشته‌اید، بنابراین الگوریتم‌ها را با استفاده از زبان برنامه‌نویسی واقعی یا حتی شبه کد بیان نخواهم کرد. به جای آن، هر جایی که بتوانم، به صورت توصیف سناریوهای واقعی، آنها را شرح خواهم داد. برای نمایش رویدادها هم از لیست‌ها و لیست‌های درون لیستی (آنچه که فلوچارت کنترلی می‌نامیم) استفاده خواهم کرد. اگر بخواهید الگوریتمی را در یک زبان برنامه‌نویسی واقعی اجرا کنید، اجازه تبدیل توضیحم به یک کد قابل اجرا را به شما خواهم داد.

هرچند تلاش خواهم کرد که توضیحاتم را تا جای ممکن غیرتکنیکی نگه دارم، اما به هر روی، این کتاب در مورد الگوریتم‌های کامپیوتری است. بنابراین باید از اصطلاحات فنی کامپیوتری استفاده کنم. مثلا برنامه‌های کامپیوتری حاوی رویه‌ها[1] (که در زبان‌های برنامه‌نویسی واقعی با عنوان توابع یا رویه‌ها هم شناخته می‌شوند) که نشان می‌دهند، چگونه کاری را انجام دهید. برای اینکه یک رویه، وظیفه تعیین شده برایش را انجام دهد، آن را فراخوانی می‌کنیم. وقتی یک رویه را فراخوانی می‌کنیم، با دادن ورودی از آن استفاده می‌کنیم (دست‌کم یک ورودی، اما برخی از رویه‌ها اصلا ورودی ندارند). ورودی‌ها را به‌عنوان پارامترها در پرانتز و پس از نام "رویه" قرار می‌دهیم.

مثلا برای محاسبه‌ی ریشه‌ی مجذور یک عدد می‌توان یک رویه AQUARE-ROOT (x) را تعریف کرد که در اینجا، ورودی رویه با پارامتر x مشخص می‌شود. فراخوانی یک رویه، ممکن است خروجی تولید کند یا نکند، که این مسئله به چگونگی انتخاب رویه بستگی دارد. اگر رویه به تولید خروجی منجر شود، خروجی را چیزی در نظر می‌گیریم که به عامل فراخوانی ربط دارد. در این رابطه، در زبان محاسبه‌ای می‌گوییم که رویه یک مقدار داده است. بسیاری از برنامه‌ها و الگوریتم‌ها با آرایه‌هایی از داده‌ها کار می‌کنند. آرایه، داده‌های هم‌نوع را در یک موجودیت واحد جمع‌آوری می‌کند. می‌توانید آرایه را مانند جدول در نظر بگیرید، که با توجه به شاخص یک ورودی می‌توانیم در مورد عنصر آرایه موجود در آن شاخص سخن برانیم. برای نمونه، جدول زیر مربوط به 5 رئیس جمهور نخست آمریکاست:

Index	President
1	George Washington
2	John Adams
3	Thomas Jefferson
4	James Madison
5	James Monroe

مثلا، عنصر مشخص شده با شاخص 4 در این جدول جیمز مادیسون است. این جدول را 5 موجودیت مجزا نمی‌دانیم بلکه یک جدول با 5 ورودی است. آرایه هم همین‌گونه است. شاخص‌های آرایه، اعدادی پی‌درپی هستند که می‌توانند از هر عددی شروع شوند؛ اما معمولا آن‌ها را از 1 شروع می‌کنیم. با داشتن نام آرایه و یک شاخص در آن آرایه، این دو را با براکت های مربعی ادغام می‌کنیم تا عنصر آرایه‌ای خاص را نمایش دهیم. مثلا عنصر iام از آرایه A را با A[i] نشان می‌دهیم.

آرایه‌های موجود در کامپیوترها یک ویژگی مهم دیگر هم دارند: زمان‌های دسترسی به هر عنصر آرایه کاملا برابر هستند. به محض اینکه شاخص‌iام از آرایه را به کامپیوتر می‌دهید، کامپیوتر

[1] procedures

می‌تواند به همان سرعت دسترسی به نخستین عنصر و بدون توجه به مقدار i، به عنصر iام دسترسی پیدا کند.

اجازه دهید نخستین الگوریتم را بررسی کنیم: جست‌وجوی آرایه برای یافتن مقدار خاص. پس یک آرایه داریم که می‌خواهیم بدانیم کدام ورودی این آرایه (اگر وجود داشته باشد) مقدار خاص دارد. برای فهمیدن اینکه چگونه می‌توانیم آرایه را جست‌وجو کنیم، اجازه دهید آرایه را شبیه یک قفسه طویل و پر از کتاب در نظر بگیریم و فرض کنیم که می‌خواهید بدانید کتاب نوشته شده توسط Jonathan Swift کجای آن قرار دارد. کتاب‌های موجود در قفسه ممکن است به یکی از چند روش موجود، در کتابخانه مرتب شده باشند، شاید طبق حروف الفبای نام نویسنده، یا نام کتاب و یا شماره تماس، یا شاید این قفسه مانند کتابخانه‌ی من در منزل باشد، که به هیچ روش خاصی سازمان‌دهی نشده است.

اگر نتوانستید فرض کنید که کتاب‌ها در قفسه سازمان‌دهی شده هستند، چگونه می‌خواهید این کتاب را بیابید؟ الگوریتمی که ممکن است استفاده کنم، بدین صورت است که: از انتهای سمت چپ قفسه شروع کرده و سمت چپ‌ترین کتاب را بررسی خواهم کرد. اگر اثر Swift بود، آن را به عنوان پاسخ مسئله قرار می‌دهم؛ وگرنه به سراغ کتاب بعدی می‌روم. این کار آنقدر ادامه می‌یابد تا کتاب را بیابم یا اینکه به آخر قفسه برسم. در این حالت می‌توانم نتیجه بگیرم که در قفسه هیچ کتابی از Jonathan Swift وجود ندارد (در فصل 3، خواهیم دید که وقتی کتاب‌ها در قفسه مرتب شده باشند چگونه باید برای یافتن یک کتاب، آن را جست‌وجو کرد).

اکنون می‌توان این مسئله‌ی جست‌وجو را از نظر محاسباتی شرح داد. اجازه دهید تا کتاب‌های درون کتابخانه را آرایه‌ای از کتاب‌ها در نظر بگیریم. نخستین کتاب از سمت چپ در موقعیت 1 قرار دارد، کتاب بعدی در سمت راست آن در موقعیت 2 و به همین صورت دیگر کتاب‌ها. اگر n کتاب در قفسه داشته باشیم پس کتاب آخر در سمت راست دارای موقعیت n است. می‌خواهیم شماره‌ی موقعیت هر کتاب نوشته شده توسط Jonathan Swift را در کتابخانه بیابیم.

به عنوان مسئله‌ی محاسباتی کلی، آرایه A (قفسه پر از کتاب) با n عنصر (تک تک کتاب‌ها) را داریم و می‌خواهیم بدانیم آیا مقدار x (کتاب جاناتان Swift) در ردیف A وجود دارد یا نه. اگر وجود داشت، می‌خواهیم شاخص i که برای آن A[i]=x را تعیین کنیم (موقعیت مکانی i در قفسه که کتاب Jonathan Swift را در خود جای داده است). همچنین به روشی برای نشان دادن اینکه آرایه A حاوی x نیست، نیاز داریم (قفسه کتابی از این نویسنده ندارد). فرض این نیست که x حداکثر یک بار در آرایه دیده می‌شود (شاید شما چند نسخه از کتاب را داشته باشید)، بنابراین اگر x در ردیف A وجود داشته باشد، ممکن است چندین بار دیده شود. تمام آن چیزی که از یک الگوریتم جست‌وجو می‌خواهیم،

هرگونه شاخصی است که در آن، x را در آرایه بیابیم. فرض خواهیم کرد که شاخص‌های این آرایه از 1 شروع می‌شوند پس عناصرش در دامنه‌ی A[1] و A[n] قرار دارند.

اگر جست‌وجو را از سمت چپ قفسه به دنبال کتابی از Jonathan Swift شروع کنیم و کتاب‌ها را یکی یکی در حرکت به طرف راست قفسه بررسی کنیم؛ این تکنیک را "جست‌وجوی خطی[1]" می‌نامیم. برای یک آرایه در کامپیوتر از ابتدای آرایه شروع کرده و هر عنصر آن را بررسی می‌کنیم (از A[1] تا A[n]) و اگر x را بیابیم مکان آن را ثبت می‌کنیم. رویه‌ای جست‌وجوی خطی که در زیر می‌آید، سه پارامتر که آنها را برای جداسازی با کاما از هم جدا کرده‌ایم را می‌گیرد.

Procedure LINEAR-SEARCH(A, n, x)

Inputs:

- A: an array.
- n: the number of elements in A to search through.
- x: the value being searched for.

Output: Either an index i for which $A[i] = x$, or the special value NOT-FOUND, which could be any invalid index into the array, such as 0 or any negative integer.

1. Set *answer* to NOT-FOUND.
2. For each index i, going from 1 to n, inorder:
   A. If $A[i] = x$, then set *answer* to the value of i.
3. Return the value of *answer* as the outpt.

افزون بر پارامترهای x، n و A رویه جست‌وجوی خطی از متغیری به نام جواب استفاده می‌کند. این روش مقدار اولیه‌ی NOT-FOUND را در مرحله‌ی 1 به جواب اختصاص می‌دهد. مرحله‌ی2 هر عنصر آرایه را از A[1] تا A[n] کنترل می‌کند تا ببیند آیا آن عنصر برابر مقدار x است یا نه. هر بار A[i] برابر x شده، مرحله‌ی 2.A، مقدار کنونی i را به جواب اختصاص می‌دهد. اگر x در آرایه ظاهر شود، مقدار خروجی بازگردانده شده در مرحله 3، آخرین شاخصی خواهد بود که x در آن ظاهر می‌شود، اگر x در آرایه دیده نشود، تست برابری مرحله 2.A هرگز درست نخواهد بود و مقدار خروجی بازگردانده شده "NOT-FOUND" خواهد بود (همان مقدار اختصاص داده شده به جواب در مرحله 1).

پیش از ادامه‌ی بحث مربوط به جست‌وجوی خطی باید به نکته‌ای در مورد چگونگی تعیین اعمال تکراری مانند مرحله 2 اشاره کرد، اجرای اقدامی برای متغیری که مقادیری در دامنه‌ی خاص دارد،

[1] Linear search

در الگوریتم‌ها کاملا امری عادی است. اعمال تکراری را که به کار می‌بریم حلقه[1] و هر بار اجرای درون یک حلقه را تکرار[2] می‌نامیم.

برای حلقه مرحله 2 نوشتم"برای هر اندیس i از 1 تا n، به ترتیب. در عوض از این پس خواهم نوشت " for i = 1 to n"، که کوتاه‌تر است و همچنان همان ساختار را معنی می‌دهد. توجه کنید وقتی حلقه را به این شکل می‌نویسیم، باید به متغیر حلقه (در اینجا i) مقدار اولیه‌ای (در اینجا 1) بدهیم و در هر تکرار حلقه، باید مقدار اخیر متغیر حلقه را در مقابل یک کران (در اینجا n) آزمایش کنیم. اگر مقدار اخیر متغیر حلقه کمتر یا مساوی حد باشد همه چیز در بدنه حلقه را انجام می‌دهیم (در اینجا مرحله 2.A). پس از اینکه یک تکرار در بدنه حلقه اجرا می‌شود، متغیر حلقه را افزایش می‌دهیم (به آن یک اضافه می‌کنیم) و برمی‌گردیم و متغیر حلقه را مقایسه می‌کنیم، سپس مقدار جدید آن را به‌طور پی‌درپی با کران، و نیز متغیر حلقه را با کران مقایسه می‌کنیم، بدنه حلقه را اجرا می‌کنیم و متغیر حلقه را افزایش می‌دهیم تا زمانی که متغیر حلقه به کران برسد. سپس اجرا از گامی که بی‌درنگ به دنبال بدنه حلقه می‌آید، ادامه پیدا می‌کند (در اینجا مرحله3). حلقه به شکل"for i = 1 to n" n تکرار و n+1 مقایسه با کران انجام می‌دهد (زیرا متغیر حلقه در مقایسه n+1ام از کران فراتر می‌رود).

امیدوارم برای شما واضح شده باشد که رویه جست‌وجوی خطی همواره به پاسخ صحیح می‌رسد. به هر روی، ممکن است دریافته باشید که این رویه، کم بازده است، آن به جست‌وجوی آرایه حتی پس از اینکه اندیس i (به طوریکه A[i] = x) را یافت، ادامه می‌دهد. به طور طبیعی، به جست‌وجوی یک کتاب وقتی آن را در قفسه کتابخانه‌تان پیدا کرده‌اید، ادامه نمی‌دهید، این گونه نیست؟ در عوض، می‌توانیم رویه جست‌وجوی خطی را به گونه‌ای طراحی کنیم که زمانی که مقدار x را در آرایه پیدا کرد، جست‌وجو را متوقف کند. فرض می‌کنیم که وقتی می‌گوییم یک مقدار را بدست آورده‌ایم، رویه بی‌درنگ مقدار را به فراخواننده خود می‌دهد که بعدا کنترل می‌کند.

Procedure BETTER-LINEAR-SEARCH (A, n, x)
Inputs and Output: Same as LINEAR-SEARCH.
1. For $i = 1$ to n:
   A. If $A[i] = x$, then return the value of i as the output.
2. Return NOT-FOUND as the output.

باور کنید یا نه، می‌توانیم جست‌وجوی خطی را حتی پربازده‌تر کنیم. مشاهده کنید که هر زمان در مرحله 1 حلقه، رویه جست‌وجوی خطی دو تست را بهتر انجام می‌دهد: تستی در مرحله 1 برای تعیین

[1] loop
[2] iteration

اینکه آیا $i \leq n$ (و اگر این‌گونه باشد، تکرار دیگری از حلقه‌ای انجام می‌دهد) و تست تساوی در مرحله 1.A. در زمینه جست‌وجو در قفسه کتاب، این تست‌ها متناظر با این است که دو چیز را برای هر کتاب بررسی کنید: آیا به انتهای قفسه رسیده‌اید؟ اگر نه، آیا کتاب بعدی اثری از Jonathan Swift است؟ مطمئنا برای رفتن به انتهای قفسه دچار جریمه زیادی نمی‌شوید (مگر اینکه صورتتان را خیلی نزدیک به کتاب‌ها نگه دارید، در حالی که مشغول بررسی آن هستید، دیواری در انتهای قفسه وجود داشته باشد و صورتتان به دیوار برخورد کند!)، ولی در یک برنامه کامپیوتری، معمولا خیلی بد است که به عناصر آرایه پس از انتهای آرایه دستیابی پیدا کنید. برنامه‌ی شما ممکن است در هم بریزد یا داده‌ها را از بین ببرد.

می‌توانید آن را به گونه‌ای ایجاد کنید که تنها یک بررسی برای هر کتابی که می‌بینید، انجام دهید. چه می‌شد، اگر با اطمینان می‌دانستید که در قفسه کتاب‌های‌تان حتما کتابی از Jonathan Swift وجود دارد؟ بنابراین مطمئن بودید که آن را پیدا خواهید کرد و بنابراین هرگز مجبور نبودید، رسیدن به انتهای قفسه را بررسی کنید. تنها لازم بود هر کتاب را چک کنید که اثر Jonathan Swift است یا نه. یا شاید فکر کرده باشید که کتاب‌هایی از او دارید، ولی هرگز نداشته‌اید. پس مطمئن نیستید که قفسه کتاب‌های‌تان، کتابی از او دارد. این چیزی نیست که می‌توانید انجام دهید.

جعبه‌ای خالی به اندازه کتاب بردارید و در طرف باریک آن بنویسید، "سفرهای گالیور اثری از Jonathan Swift " و کتاب سمت راست قفسه را با این جعبه جایگزین کنید. سپس همان‌گونه که از چپ به راست در قفسه جست‌وجو می‌کنید، تنها نیاز دارید که ببینید چیزی که برمی‌دارید، اثر Swift است یا نه. می‌توانید بررسی اینکه به انتهای قفسه رسیده‌اید را فراموش کنید، زیرا می‌دانید که کتابی از سوئفت خواهید یافت. تنها پرسش این است که آیا واقعا کتابی از Swift پیدا می‌کنید یا جعبه خالی که به نام او برچسب زده بودید را پیدا کرده‌اید؟ پس واقعا کتابی از Swift نداشتید. بررسی آن ساده است و در پایان جست‌وجوی‌تان، به جای یک بار برای هر کتاب در قفسه، تنها لازم است یک بار آن را انجام دهید.

نکته دیگری وجود دارد که باید از آن آگاه باشید: اگر تنها کتابی که از Jonathan Swift در قفسه کتابهای‌تان دارید، در انتهای سمت راست باشد، چرا اگر آن را با جعبه خالی جایگزین کنید، جست‌وجوی‌تان به جعبه خالی خواهد انجامید و ممکن است نتیجه بگیرید که کتاب را نداشتید. بنابراین باید یک بررسی دیگر برای آن احتمال انجام دهید، ولی این تنها یک بررسی است نه چندین بررسی برای هر کتاب در قفسه.

در مفهوم الگوریتم کامپیوتر، مقدار x را که به دنبال آن هستیم، را در آخرین مکان قرار خواهیم داد، A[n]، پس از ذخیره محتوای A[n] در متغیر دیگری، هر زمان که x را پیدا کنیم، تست کنیم که آیا

واقعا آن را یافته‌ایم؟. مقداری که در آرایه قرار داده‌ایم را نگهبان[1] می‌نامیم، ولی می‌توانید آن را به عنوان جعبه خالی تصور کنید.

Procedure SENTINEL-LINEAR-SEARCH (A, n, x)
Inputs and Output: Same as LINEAR-SEARCH.
1. Save $A[n]$ into *last* and then put x into $A[n]$
2. Set i to 1.
3. While $A[i] \neq x$, do the following:
   A. Increment i.
4. Restore $A[n]$ from *last*.
5. If $i < n$ or $A[n] = x$, then return the value of i as the output.
6. Otherwise, return NOT-FOUND as the output.

مرحله 3 یک حلقه است، ولی نه حلقه‌ای که از طریق نوعی متغیر حلقه، شمارش می‌شود. در عوض، حلقه تا زمانی که شرطی برقرار شود، تکرار می‌شود. شرط این است: A[i] ≠x روش تغییر چنین حلقه‌ای این است که تست را (در اینجا A[i] ≠x) را انجام دهد و اگر صحیح است، هرچه را در بدنه حلقه است (در آنجا گام 3.A که i را افزایش می‌دهد) انجام دهد. سپس برگردد و تست را انجام دهد و اگر صحیح است، بدنه را اجرا کند. در ادامه با انجام تست و اجرای بدنه، تا اینکه پاسخ تست نادرست شود، پیش رود. سپس از گام بعدی پس از بدنه ادامه یابد (در اینجا گام 4).

رویه جست‌وجوی SENTINEL-LINEAR-SEARCH کمی از دو رویه جست‌وجوی خطی نخست پیچیده‌تر است. از آنجایی که در گام 1، x را در A[n] قرار می‌دهد، مطمئن می‌شویم A[i] برای تست در گام 3 برای x خواهد بود. زمانی که این اتفاق بیفتد، از حلقه گام 3 خارج می‌شویم و اندیس i پس از آن تغییر نخواهد کرد. پیش از انجام هر چیز دیگری، گام 4 مقدار اولیه را در A[n] ذخیره می‌کند (مادرم آموخته است، هر چیز را پس از اتمام کار سر جایش برگردانم). سپس باید تعیین کنیم آیا واقعاً x را در آرایه پیدا کرده‌ایم یا نه، چون x را در آخرین جزء یعنی A[n] قرار می‌دهیم، می‌دانیم که اگر x را در A[i] ای که i < n است، پیدا کنیم، پس حتما x را پیدا کرده‌ایم و می‌خواهیم که اندیس i را به دست آوریم. اگر x را در A[n] پیدا کردیم چه؟ این بدین معنی است که x را پیش از A[n] پیدا نکردیم و بنابراین باید تعیین کنیم که آیا A[n] با x برابر است یا خیر. اگر باشد، اندیس n را می‌خواهیم نمایش دهیم که در اینجا برابر i است. ولی اگر برابر نباشد، NOT-FOUND را نمایش می‌دهیم. گام 5 این تست‌ها را انجام می‌دهد و در صورتی که x اساسا در آرایه وجود داشته باشد، اندیس صحیح را نمایش می‌دهد. اگر تنها به این دلیل که گام 1، x را درآرایه قرار داد، پیدا شد گام 6، NOT-FOUND را می‌دهد. اگرچه جست‌وجوی SENTINEL-LINEAR-SEARCH پس از اتمام

[1] Sentinel

حلقه‌اش باید دو تست انجام دهد، تنها یک تست در هر تکرار حلقه انجام می‌دهد، بنابراین از LINER-SEARCH یا BETTER_LINEAR_SEARCH پر بازده‌تر است.

چگونگی مشخص کردن زمان‌های اجرا[1]

به رویه LINER-SEARCH بازمی‌گردیم تا زمان اجرای آن را بفهمیم. به یاد آورید که قصد داریم زمان اجرا را به عنوان تابعی از اندازه ورودی مشخص کنیم. در اینجا ورودی A با n عنصر است که به دنبال عدد n و مقدار x در آن هستیم. اندازه n و x وقتی آرایه بزرگ شود، بی معنا می‌شود. n تنها یک عدد صحیح تک است و x تنها به بزرگی n عنصر آرایه است، بنابراین خواهیم گفت که اندازه ورودی n، تعداد اجرا در A است. باید فرضیات ساده‌ای درباره اینکه کارها چقدر طول می‌کشند، انجام دهیم. فرض می‌کنیم که هر عملیات مجزا، خواه عملیات حسابی (مانند جمع، تفریق، ضرب یا تقسیم)، مقایسه، تخصیص به یک متغیر، اندیس دادن داخل یک آرایه یا فراخوانی یک رویه مقدار زمان مشخصی می‌گیرد که مستقل از زمان ورودی است. زمان ممکن است برای هر عملیاتی متفاوت باشد، به طوری که مثلا تقسیم، زمان بیشتری از جمع می‌برد، ولی وقتی یک مرحله تنها شامل یک عملیات ساده می‌شود، هر اجرای منفرد از آن مرحله مقدار زمان ثابتی می‌گیرد. چون هر یک از عملیات اجرا شده، از یک گام تا گام دیگر فرق می‌کنند و نیز به دلیل عوامل بیرونی که در آغاز فصل نخست گفته شد، زمان اجرای یک گام ممکن است با گام دیگر تفاوت داشته باشد. بیایید بگوییم که زمان هر اجرا در گام i، برابر t_i است که t_i ثابتی است که به n بستگی ندارد. البته باید در نظر داشته باشیم که برخی گام‌ها چندین بار اجرا می‌شوند. گام 1 و3 تنها یک بار اجرا می‌شوند، ولی گام 2 چطور؟ باید i را در مقابل n به تعداد $n + 1$ بار تست کنیم: n بار که در آن $i \leq n$ و یک بار وقتی i برابر $n + 1$ می‌شود و از حلقه خارج می‌شویم، گام 2.A دقیقا n بار اجرا می‌شود، یک بار برای مقدار i از 1 تا n. از پیش نمی‌دانیم چند بار جواب به مقدار i می‌رسد؛ می‌تواند هیچ بار (اگر x در آرایه وجود نداشته باشد) تا n بار (اگر هر مقدار درآرایه برابر x باشد) باشد. اگر بخواهیم در محاسباتمان دقیق باشیم – معمولا این‌قدر دقیق نیستیم- لازم است بفهمیم که گام 2، دو کار متفاوت انجام می‌دهد که دفعات مختلفی اجرا می‌شود: تست i در برابر n، n+1 بار اتفاق می‌افتد، ولی افزایش i تنها n بار اتفاق می‌افتد. بیایید زمان خط 2 را به t'_2 برای تست و t''_2 برای افزایش، تقسیم کنیم. به طور مشابه، زمان مرحله 2.A را به t'_{2A} برای تست اینکه A[i] = x است یا خیر و t''_{2A} برای تنظیم جواب به i تقسیم خواهیم کرد. بنابراین زمان اجرای جست‌وجوی خطی بین

$$t_1 + t'_2 \cdot (n+1) + t''_2 \cdot n + t'_{2A} \cdot n + t''_{2A} \cdot 0 + t_3$$

[1] Running times

و

$$t_1 + t_2' \cdot (n+1) + t_2'' \cdot n + t_{2A}' \cdot n + t_{2A}'' \cdot n + t_3$$

است.

اینک این محدوده‌ها را بازنویسی می‌کنیم، (عناصری که در n ضرب شده‌اند را با هم جمع می‌کنیم و باقی را هم با یکدیگر)، در این صورت محدوده پایینی زمان اجرا برابر $(t'2 + t''2 + t'2A).n +$ $(t1 + t'2 + t3)$ و محدوده بالایی آن $(t'_2+t''_2+t'_{2A}+t''_{2A})$، $n+(t_1+t'_2+t_3)$ خواهد شد. توجه کنید که هر دو محدوده به فرم c.n+d هستند که c و d ثابت و مستقل از n هستند. یعنی هر دو تابع خطی از n هستند. زمان اجرای جست‌وجوی خطی در محدوده تابع خطی از n از پایین و بالا است.

از نماد خاصی برای نشان دادن اینکه زمان اجرا از بالا با تابع خطی n و از پایین با تابع خطی (احتمالا متفاوتی) از n محدود شده، استفاده می‌کنیم. می‌نویسیم زمان اجرا Θ(n) است. این حرف یونانی تتا است و می‌خوانیم تتای n یا همان تتا n. همان‌گونه که در فصل 1 وعده داده شد، این نشانه قسمت $(t_1+t'_2+t_3)$ و ضریب n (یعنی $t'_2+t''_2+t'_{2A}$ برای کران پایینی و $t'_2+t''_2+t'_{2A}+t''_{2A}$ برای کران بالایی) را نادیده می‌گیرد. اگرچه با مشخص کردن زمان اجرا با Θ(n) دقت را از دست می‌دهیم، مزیت برجسته‌سازی مرتبه رشد زمان اجرا و ممانعت از جزئیات خسته کننده را به‌دست می‌آوریم.

این نماد Θ در کل به توابع گفته می‌شود، نه تنها آنهایی که زمان اجرای الگوریتم‌ها را توصیف می‌کنند و به توابعی به غیر از توابع خطی نیز گفته می‌شود. ایده این است که اگر دو تابع f(n) و g(n) داشته باشیم، می‌گوییم f(n) برابر (g(n)) است اگر برای مقادیر به اندازه کافی بزرگn، f(n) عامل ثابتی از g(n) باشد. بنابراین می‌توانیم بگوییم که زمان اجرای جست‌وجوی خطی وقتی که n به اندازه کافی بزرگ شود، در ضریب ثابتی از n است.

تعریفی از نماد Θ وجود دارد؛ ولی خوشبختانه به ندرت مجبوریم برای استفاده از نماد Θ به آن رجوع کنیم. به آسانی بر عبارت با درجه رشد بالاتر، با حذف عبارات با درجه پایین‌تر و عوامل ثابت، تمرکز می‌کنیم. برای نمونه تابع $n^2/4+100n+50$ Θ(n^2) است؛ در اینجا جملات با مرتبه پایین تر 100و 50 را حذف می‌کنیم و ضریب ثابت $\frac{1}{4}$ را نیز حذف می‌کنیم. اگرچه جملات مرتبه پایین‌تر در مقادیر کوچک‌تر n بر $\frac{n^2}{4}$ برتری دارند، زمانی که n از 400 بزرگ‌تر شود، جمله$\frac{n^2}{4}$50 از 100n+50 فراتر می‌رود، وقتی n=1000، جمله غالب $\frac{n^2}{4}$ با 250000 برابر می‌شود، در حالی که جملات مرتبه پایین‌تر100n+50 تنه برابر 100050 می‌شوند، برای n=2000 تفاوت این دو مقدار 1000000 در مقابل 200050 است. در دنیای الگوریتم‌ها، کمی از نمادها سوءاستفاده می‌کنیم و می‌نویسیم F(n) = Θ(g(n))، تا بتوانیم بنویسیم Θ(n2) =100n+50+$\frac{n^2}{4}$ اکنون به زمان اجرای جست‌وجوی خطی بهتر در

صفحه 14 نگاهی می‌اندازیم؛ این یکی از جست‌وجوهای خطی کمی ماهرانه‌تر است، زیرا از پیش نمی‌دانیم حلقه چند بار تکرار خواهد شد. اگر A[1] با x برابر شود، یک بار تکرار خواهد شد. اگر x در آرایه نباشد، حلقه n بار تکرار می‌شود، که بیشترین حالت ممکن است. هر تکرار حلقه مقدار زمان ثابتی می‌گیرد، بنابراین می‌توانیم بگوییم در بهترین حالت، جست‌وجوی خطی بهتر زمان $\Theta(n)$ را برای جست‌وجوی یک آرایه با n عنصر برد. چرا "بدترین حالت"؟ چون می‌خواهیم الگوریتم‌ها زمان اجرای کمی داشته باشند بدترین حالت، زمانی رخ می‌دهد که الگوریتم بیشترین زمان را در هر ورودی ممکن بگیرد. در بهترین حالت، وقتی A[1] برابر با x می‌شود، جست‌وجوی BETTER-LINEAR-SEARCH تنها مقدار ثابتی زمان می‌برد؛ یعنی i را برابر 1 قرار می‌دهد و بررسی می‌کند که $i\leq n$ باشد، تست A[i]=x درست در می‌آید و رویه مقدار i را می‌دهد که 1 است. این مقدار زمان به n بستگی ندارد. می‌نویسیم که زمان اجرای بهترین حالت LINER-SEARCH، بهتر از $\Theta(1)$ است زیرا در بهترین حالت، زمان اجرای آن ضریب ثابتی از 1 است. به بیان دیگر، زمان اجرای بهترین حالت، ثابتی است که به n بستگی ندارد. پس می‌بینیم که نمی توان نماد Θ را برای عبارت پوششی که تمام حالات زمان اجرای جست‌وجوی BETTER-LINEAR-SEARCH را پوشش می‌دهد، استفاده کرد. نمی‌توان گفت که زمان اجرا همواره $\Theta(n)$ است، زیرا در بهترین حالت برابر $\Theta(1)$ است. و نمی توانیم بگوییم که زمان اجرا همواره $\Theta(1)$ است زیرا در بدترین حالت برابر $\Theta(1)$ است. می‌توانیم بگوییم که تابع خطی از n در تمام حالات کران بالایی است، با این حال برای آن نمادی داریم:$O(n)$.

زمانی که بخواهیم این نماد را بخوانیم، می‌گوییم O بزرگ n یا تنها Oی n. تابع f(n) برابر $O(g(n))$ است. اگر زمانی n به اندازه کافی بزرگ شود، f(n) از بالا محدود به ثابتی ضرب در g(n) باشد. دوباره از نماد سوءاستفاده می‌کنیم و می‌نویسیم $f(n)=O(g(n))$. برای جست‌وجوی خطی بهتر، می‌توانیم عبارت پوششی را به گونه‌ای ایجاد کنیم که در تمام موارد، زمان اجرای آن برابر $O(n)$ شود؛ اگرچه زمان اجرا ممکن است از تابع خطی n بهتر باشد، هرگز بدتر نخواهد بود.

از نماد O برای نشان دادن اینکه زمان اجرا هیچگاه بدتر از تابعی از n ضرب در ثابتی نیست استفاده می‌کنیم، ولی در مورد نشان دادن اینکه زمان اجرا هرگز بهتر از تابعی از n ضرب در ثابتی باشد چطور؟ این کران پایینی است و از نماد Ω که نماد O را منعکس می‌کند، بهره می‌بریم. تابع f(n) برابر $\Omega(g(n))$ است؛ اگر وقتی که n به اندازه کافی بزرگ شود، f(n) از پایین محدود شود به ثابتی ضرب در g(n)، می‌گوییم f(n) امگای g(n) یا فقط f(n) برابر است با امگا g(n). می‌توانیم بنویسیم $f(n) = \Omega(g(n))$؛ از آنجایی که نماد O کران بالایی را می‌دهد، نماد Ω کران پایینی را می‌دهد و نماد Θ هر دو کران پایینی و بالایی را می‌دهد، می‌توانیم نتیجه بگیریم که f(n) تابعی از $\Theta(g(n))$ است؛ اگر و تنها اگر f(n) هم برابر با $O(g(n))$ باشد و هم $\Omega(g(n))$.

می‌توانیم عبارت پوششی در مورد کران پایینی برای زمان اجرای جست‌وجوی BETTER-LINEAR-SEARCH را ایجاد کنیم: در همه حالات برابر با $\Omega(1)$ است. البته این یک عبارت ضعیف است، زیرا انتظار داریم هر الگوریتمی با هر ورودی، حداقل زمان ثابتی را بگیرد. از نماد Ω زیاد استفاده نمی‌کنیم ولی گاهی سودمند خواهد بود. جمله Catch-all برای نمادهای Θ، O و Ω نماد تقربی است. به این علت که این نمادها، زمانی که آرگومان یک تابع به صورت تقریبی[1] به بی‌نهایت نزدیک شود، رشد آن را می‌گیرد. تمام این نمادهای تقربی نعمت حذف جملات مرتبه پایین‌تر و ضرایب ثابت را می‌دهند، به گونه‌ای که می‌توانیم جزئیات کسالت‌بار را نادیده بگیریم و بر چیزی که اهمیت دارد یعنی چگونگی رشد تابع با n، تمرکز کنیم. اینک بیایید به جست‌وجوی SENTINEL-LINEAR-SEARCH برگردیم. درست مانند BETTER-LINEAR-SEARCH، هر تکرار حلقه مقدار زمان ثابتی می‌گیرد و می‌تواند هر جایی از 1 تا n تکرار باشد. تفاوت عمده میان جست‌وجوی SENTINEL-LINEAR-SEARCH و جست‌وجوی BETTER-LINEAR-SEARCH این است که زمان به ازای هر تکرار در جست‌وجوی SENTINEL-LINEAR-SEARCH از زمان هر تکرار در BETTER-LINEAR-SEARCH کمتر است. هردو در بدترین حالت، مقدار خطی زمان را می‌گیرند، ولی ضریب ثابت برای جست‌وجوی SENTINEL-LINEAR-SEARCH بهتر است. اگرچه انتظار خواهیم داشت که جست‌وجوی SENTINEL-LINEAR-SEARCH در عمل سریع‌تر باشد، تنها با یک ضریب ثابت خواهد بود. وقتی زمان اجرای جست‌وجوی BETTER-LINEAR-SEARCH و جست‌وجوی SENTINEL-LINEAR-SEARCH را با استفاده از نماد تقربی بیان می‌کنیم، با هم برابر هستند: $\Theta(n)$ در بدترین حالت، $\Theta(1)$ در بهترین حالت و $O(n)$ در تمام حالات.

ثابت های حلقه[2]

برای سه نوع جست‌وجوی خطی، آسان بود که ببنیم کدام جواب درست را می‌دهد. گاهی این موضوع کمی سخت‌تر است. طیف وسیعی از تکنیک‌ها بیش از آنکه می‌توانم در اینجا پوشش دهم وجود دارد. روش متداول برای نشان دادن درستی، از یک ثابت حلقه استفاده می‌کند، ادعایی که هر بار که تکرار حلقه را آغاز می‌کنیم، نشان می‌دهیم که درست است. برای اینکه ثابت حلقه به ما کمک کند در مورد درستی بحث کنیم، باید سه چیز در مورد آن نشان دهیم:

آغاز: ثبت حلقه پیش از شروع نخستین تکرار حلقه صحیح است.

نگهداری: اگر پیش از تکرار حلقه صحیح باشد، پیش از تکرار بعدی صحیح باقی می‌ماند.

---

[1] asymptotically
[2] Loop invariants

خاتمه: حلقه پایان می‌یابد و وقتی این پیشامد رخ می‌دهد، ثابت حلقه، همراه با این دلیل که حلقه خاتمه یافته، به ما یک ویژگی سودمند می‌دهد.

برای نمونه: در اینجا ثابت حلقه‌ای برای جست‌وجوی BETTER-LINEAR-SEARCH وجود دارد: در شروع هر تکرار در گام 1، اگر x در آرایه A وجود دارد، پس در زیر آرایه (بخشی مربوط به یک آرایه) از A[i] تا A[n] وجود دارد.

حتی لازم نیست این ثابت حلقه نشان دهد که اگر رویه اندیسی غیر از NOT-FOUND بدهد پس اندیس صحیح است: تنها راهی که رویه می‌تواند اندیس i در مرحله 1A را بدهد این است که A[i] برابر x باشد. در عوض، از این ثابت حلقه برای نشان دادن اینکه اگر رویه NOT-FOUND را در گام 2 بدهد، پس x در هیچ جایی آرایه نیست، استفاده خواهیم کرد.

آغاز: در آغاز،i=1 به‌گونه‌ای که زیر آرایه در ثابت حلقه برابر A[1] تا A[n] باشد که کل آرایه است.

نگهداری: فرض کنید در آغاز یک تکرار برای مقدار i، اگر x در آرایه A وجود دارد، پس در زیر آرایه از A[i] تا A[n] وجود دارد. اگر این تکرار را انجام دهیم و چیزی بدست نیاوریم، می‌دانیم که A[i]≠x و بنابراین می‌توانیم با اطمینان بگوییم اگر x در آرایه A وجود دارد، پس در زیر آرایه از A[i+1] تا A[n] است. چون تا پیش از تکرار بعدی افزایش یافته، ثابت حلقه پیش از تکرار بعدی نگهداری خواهد شد.

خاتمه: این حلقه باید تمام شود، این می‌تواند به دلیل بازگشت رویه به گام 1.A یا اینکه i>n، رخ دهد. هم اینک، مسئله را به دلیل اینکه حلقه، با بازگشت رویه به گام 1A پایان می‌یابد، کنترل کرده‌ایم. برای کنترل کردن مسئله برای حالتی که به دلیل i>n حلقه پایان یابد، بر معکوس[1] ثابت حلقه بسنده می‌کنیم. معکوس عبارت "اگر A باشد پس B هم هست" عبارت "اگر B نباشد پس A هم نیست" است. معکوس یک عبارت درست است، اگر و تنها اگر خود عبارت صحیح باشد. معکوس ثابت حلقه این است که "اگر x در زیر آرایه از A[i] تا A[n] موجود نباشد، پس در آرایه A نیز موجود نیست". اکنون، وقتی i>n باشد زیر آرایه از A[i] تا A[n] خالی می‌باشد، و بنابراین این زیرآرایه نمی‌تواند در برگیرنده x باشد. با استفاده از معکوس ثابت حلقه، پس x در هیچ جایی از آرایه A وجود ندارد، و جواب، NOT-FOUND را در گام 2 به خود اختصاص می‌دهد.

دلایل فراوانی برای مشخص کردن درستی اینکه، تنها یک حلقه ساده نیاز است وجود دارد! آیا مجبوریم همه این مراحل را در هنگام نوشتن یک حلقه طی کنیم؟ من که این کار را نمی‌کنم، اما تعدادی دانشمند کامپیوتر هستند که بر این دلایل دشواری برای هر حلقه پافشاری می‌کنند. وقتی که

---

[1] contrapositive

در حال نوشتن یک کد واقعی هستم، درمی‌یابم که در آن طول زمانی که مشغول نوشتن حلقه هستم، در ذهنم یک حلقه ثابت دارم. این حلقه ممکن است در عمق ذهنم باشد به گونه‌ای که حتی وجود آن را درک نکنم، اما اگر مجبور می‌بودم می‌توانستم آن را بیان کنم. وقتی بخواهیم درک کنیم که چرا حلقه-های پیچیده فعالیت‌های صحیح را انجام می‌دهند، ثابت حلقه می‌تواند بسیار کاربردی باشد.

بازگشتی: با تکنیک بازگشتی، یک مسئله را به واسطه حل نمونه‌های کوچک‌تر، همانند همان مسئله، حل می‌کنیم. مثال استاندارد و محبوب از بازگشتی محاسبه !n (n فاکتوریل)، که برای مقدارهای غیر منفی از n به این صورت تعریف می‌شود که n!=1 اگر n = 0 و n! = n. (n – 1). (n – 2). (n – 3) …3.2.1

اگر n >=1 برای مثال 5!= 5.4.3.2.1=120 دقت کنید؛

$$(n-1)! = (n-1) \cdot (n-2) \cdot (n-3) \ldots 3.2.1$$

و به همین صورت

$$n! = n \cdot (n-1)!$$

برای n>=1. !n را به صورت حالتی از یک مسئله کوچک‌تر تعریف کردیم، یعنی !(n–1). می‌توانیم یک رویه بازگشتی برای محاسبه !n به صورت زیر بنویسیم:

Procedure FACTORIAL(n)
Input: An integer $n \geq 0$.
Output: The value of $n!$.
1. If $n = 0$, then return 1 as the output.
2. Otherwise, return n times the value returned by recursively calling FACTORIAL($n - 1$).

روش من در نوشتن گام 2 تاحدی سنگین است؛ در عوض می‌توانم تنها "Otherwise, return n·FACTORIAL(n-1)" را با استفاده از بازگشتی مقدار جواب فراخوان درون یک روش محاسباتی بزرگ، بنویسم.

برای بازگشت به کار، دو ویژگی را باید نگه داشت؛ ابتدا، باید یک یا بیشتر حالت مبنا داشت، به گونه‌ای که، راه حل را مستقیما بدون بازگشت محاسبه می‌کنیم. ثانیا، هر فراخوان بازگشتی از رویه باید روی یک نمونه کوچک‌تر از همان مسئله که سرانجام به حالت پایه می‌رسد، قرار داشته باشد. برای رویه فاکتوریل، حالت پایه زمانی رخ می‌دهد که n=0 باشد، و هر فراخوانی بازگشتی روی یک نمونه در جایی که مقدار n با مقدار 1 کاهش یافته باشد، قرار دارد. تا زمانی که مقدار اصلی n غیر منفی باشد، فراخوانی‌های بازگشتی سرانجام به حالت پایه کاهش خواهند یافت.

استدلال اینکه یک فراخوان بازگشتی کار می‌کند، در ابتدا کاملا آسان به نظر می‌رسد. نکته این است؛ باور کنیم که هر فراخوان بازگشتی، نتیجه صحیح را تولید می‌کند. تا زمانی که مایل هستیم باور کنیم که فراخوانی‌های بازگشتی، کار درست را انجام می‌دهند، استدلال کردن درباره‌ی درستی، اغلب آسان است. استدلال ممکنی که می‌توانیم برای جواب صحیح دادن رویه فاکتوریل نمایش دهیم، در ادامه می‌آید. واضح است زمانی که n=0 باشد، مقدار برگردانده شده 1 است که با !n مساوی می‌باشد. فرض می‌کنیم که وقتی n>=1 باشد فراخوان بازگشتی فاکتوریل (n–1) عمل صحیح را انجام می‌دهد: مقدار !(n–1) را بازمی‌گرداند. پس از آن رویه، مقدار حاصل را در n ضرب می‌کند، در نتیجه مقدار !n را محاسبه و آن را نمایش می‌دهد.

در اینجا یک نمونه از فراخوانی‌های بازگشتی که روی نمونه‌های کوچک‌تر از همان مسئله مشابه بنا نهاده نشده‌اند، اگرچه از نظر ریاضیاتی درست می‌باشند، آورده می‌شود. در واقع این صحیح است که اگر n>=0 پس(n+1)!/(n+1)=!n اما رویه بازگشتی که در ادامه می‌آید و از مزایای این فرمول استفاده می‌کند، حتی زمانی که n>=1، قادر به ارائه جواب نخواهد بود.

Procedure BAD-FACTORIAL(n)
Input and Output: Same as FACTORIAL.
1. If $n = 0$, then return 1 as the output.
2. Otherwise, return BAD-FACTORIAL$(n + 1)/(n + 1)$.

اگر می‌خواستیم BAD-FACTORIAL (1) را فراخوانی کنیم، این رویه یک فراخوان بازگشتی از BAD-FACTORIAL (2) تولید می‌کرد، که این هم یک فراخوان بازگشتی از BAD-FACTORIAL (3) تولید می‌کرد و تا آخر، به گونه‌ای که هرگز نمی‌توانست تا مقدار حالت پایه که n=0 است، کاهش یابد. اگر این رویه را در یک زبان برنامه نویسی واقعی انجام داده و بر یک کامپیوتر واقعی اجرا کنید، به سرعت چیزی مثل "توده انباشته شده خطا" مشاهده می‌کنید. اغلب می‌توانیم الگوریتم‌هایی را که یک حلقه در سبک بازگشتی استفاده می‌کند، بازنویسی کنیم. جست‌وجوی خطی بدون دیده‌بان که به صورت بازگشتی نوشته شده است در ادامه می‌آید.

Procedure RECURSIVE-LINEAR-SEARCH(A, n, i, x)

Inputs: Same as LINEAR-SEARCH, but with an added parameter i.

Output: The index of an element equaling x in the subarray from $A[i]$ through $A[n]$, or NOT-FOUND if x does not appear in this subarray.

1. If $i > n$, then return NOT-FOUND.
2. Otherwise ($i \leq n$), if $A[i] = x$, then return i.
3. Otherwise ($i \leq n$ and $A[i] \neq x$), return RECURSIVE-LINEAR-SEARCH$(A, n, i + 1, x)$.

این کد زیر برنامه‌ای است برای جست‌وجو x در زیر آرایه که از A[i] شروع و تا A[n] ادامه می‌یابد. حالت پایه در گام 1 که این زیر آرایه تهی است اتفاق می‌افتد، و این زمانی است که i>n باشد. زیرا مقدار i در هر یک از فراخوانی‌های بازگشتی گام 3 افزایش می‌یابد، اگر هیچ فراخوان بازگشتی حتی یک مقدار از i در گام 2 را برنگرداند، در این صورت سرانجام i بزرگ‌تر از n می‌شود و ما به حالت پایه می‌رسیم.

مطالعه بیشتر: فصل 2 و 3 از CLRS مقدار عمده‌ای از مطالب این فصل را پوشش می‌دهد. نخستین کتاب درسی از HOPCROFT،Aho و Ullman [AHU74] دانش استفاده از نمادگذاری تقریبی برای تحلیل الگوریتم‌ها را تحت تأثیر قرار می‌دهد. در اثبات درستی برنامه‌ها کمی کار لازم بود؛ اگر می‌خواهید در این زمینه بررسی بیشتر کنید به کتاب‌هایی مثل Gries [Gri8] و Mitchell[Mit96] مراجعه کنید.