به نام خدا

اللَهُمَّ إنّی اَعوذُ بِکَ مِن عِلمٍ لایَنفَع (خداوندا! پناه می‌برم به تو از علمی که نفعی نداشته باشد)

طراحی الگوریتم

Algorithm Design

حمیدرضا نیرومند

http://niroomand.ir

فهرست

نقش الگوریتم‌ها در پردازش

۱-۱- تعریف چند اصطلاح:

- **مسأله[1]:** مانعی که رسیدن به هدف را مشکل می‌کند.

- **انواع هدف:**
  - **Purpose (مقصود):** به هدف غایی و نهایی از انجام یک کار گفته می‌شود؛ مانند «برنده شدن» در یک بازی فوتبال
  - **Goal (هدف):** به اهداف کوتاه‌مدتی که برای رسیدن به هدف نهایی تعریف می‌شوند گفته می‌شود؛ مانند «گل زدن» در بازی فوتبال برای رسیدن به «برنده شدن»
  - **Objective (هدف عینی/مأموریت):** اهداف قابل‌مشاهده که برای رسیدن به Goalها تعریف می‌شوند؛ مثلاً برای گل زدن «باید در تیم چینش مناسبی داشت».
  - **Policy (خط مشی):** سیاست‌هایی که باید برای رسیدن به اهداف برنامه‌ریزی شوند؛ مانند «تأمین بودجه تیم»، «آنالیز کردن قدرت تیم مقابل» و...

- **تعریف عمومی الگوریتم:**

یک روش مؤثر برای حل یک مسأله که در قالب توالی محدودی[2]از دستورالعمل‌ها بیان شده است.

- **تعریف اختصاصی و کامپیوتری الگوریتم:**

یک الگوریتم یک رویه‌ی[3]محاسباتی خوب‌بیان‌شده[4]است، که مقادیر یا مجموعه‌ای[5] از مقادیر را به عنوان

---

[1] Problem

[2] Finite
[3] Procedure
[4] Well-defined
[5] Set

ورودی[1] می‌گیرد و مقادیر یا مجموعه‌ای از مقادیر را به عنوان خروجی[2] تولید می‌کند؛ بنابراین می‌تواند گفت: یک الگوریتم، یک توالی[3] از گام‌های محاسباتی است که ورودی را به خروجی تبدیل می‌کند.

می‌توان الگوریتم را ابزاری برای حل یک مسأله محاسباتی که جزئیات آن به خوبی بیان شده، دانست، و یا به ساده‌ترین بیان: راهی برای کشف رابطه‌ی بین ورودی و خروجی!

مثال: فرض کنید، قرار است تعدادی عدد را مرتب کنید. «مسأله‌ی مرتب‌سازی به شکل زیر تعریف می‌شود».

ورودی: توالی‌ای از n عدد $(a_1, a_2, \dots, a_n)$

خروجی: توالی مرتب اعداد ورودی $(a'_1, a'_2, \dots, a'_n)$ با این شرط که:

$$(a'_1 \leq a'_2 \leq \cdots \leq a'_n)$$

برای مثال ورودی: (31, 41, 59, 26, 41, 58) با توجه به الگوریتم مرتب‌سازی، خروجی زیر را برمی‌گرداند:

خروجی: (26, 31, 41, 41, 58, 59)

به آن توالی ورودی در اصطلاح یک «نمونه»[4] از مسأله مرتب‌سازی گفته می‌شود. در کل یک نمونه از یک مسأله شامل ورودی‌هایی است که نیاز دارند راه‌حلِ[5] مسأله روی آن‌ها اعمال شود.

از آنجا که بسیاری از برنامه‌ها از مرتب‌سازی به عنوان گام میانی استفاده می‌کنند، الگوریتم‌های مرتب‌سازی[6] یکی از عملیات پایه‌ای و اساسی در علم کامپیوتر[7] هستند؛ بنابراین در این کتاب تعدادی الگوریتم مرتب‌سازیِ خوب مورد بررسی قرار گرفته است.

اینکه کدام الگوریتمِ مرتب‌سازی برای یک کاربرد خاص مناسب است، بستگی دارد به:

۱. تعداد آیتم‌هایی که قرار است مرتب شوند.

---

[1] Input
[2] Output
[3] Sequence
[4] Instance
[5] Solution
[6] Sorting Algorithms
[7] Computer Science

۲. آیتم‌هایی که از قبل مرتب شده بوده‌اند.

۳. محدودیت‌های[۱] احتمالی که روی آیتم‌ها وجود دارد.

۴. معماری[۲] کامپیوتر و نوع دستگاه ذخیره‌سازی[۳] که استفاده می‌شود؛ مانند رَم[۴] یا دیسک[۵] یا نوار.[۶]

- الگوریتم صحیح:

به الگوریتمی که به‌ازای هر نمونه از ورودی، یک خروجی صحیح در پایان[۷] ارائه کند، الگوریتم صحیح[۸] گفته می‌شود. در اصطلاح گفته می‌شود: یک الگوریتم صحیح، مسأله محاسباتی مدنظر را «حل[۹]» می‌کند. یک الگوریتم ناصحیح ممکن است به‌ازای برخی ورودی‌ها هیچ پاسخی[۱۰] برنگرداند و یا یک پاسخ اشتباه[۱۱] برگرداند. بر خلاف آنچه انتظار دارید الگوریتم‌های ناصحیح می‌توانند گاهی مفید واقع شوند، البتّه اگر نرخ‌خطای[۱۲] آن‌ها را کنترل کنیم.

یک الگوریتم می‌تواند به زبان انگلیسی یا در قالب یک برنامه‌ی کامپیوتری[۱۳] و یا حتی در قالب یک طراحی[۱۴] سخت‌افزاری[۱۵] توصیف شده باشد. تنها محدودیت این است که توصیف[۱۶] باید یک بیان دقیق از روال محاسباتی که باید انجام شود، ارائه کند.

۲-۱- برخی مسائل که با الگوریتم‌های این کتاب، قابل حل هستند:

۱-۶-۱- تحلیل ژن‌های روی DNA انسان که در پروژه ژن انسانی[۱۷] کاربرد فراوانی داشته است.

۲-۶-۱- اینترنت دسترسی افراد به حجم عظیمی از اطلاعات را آسان کرده، امّا مدیریت و نگه‌داری این حجم از داده چالشی است که نیاز به الگوریتم‌های پیچیده دارد، همین‌طور یافتن

---

[1] Restrictions
[2] Architecture
[3] Storages Devices
[4] Main Memory
[5] Disk
[6] Tape
[7] Half
[8] Correct
[9] Solve
[10] Answer
[11] Incorrect
[12] Error Rate
[13] Computer Program
[14] Design
[15] Hardware
[16] Specification
[17] Human Genome Project

یک مسیر خوب بین Client و Server یک چالش دیگر به حساب می‌آید که نیاز به الگوریتم‌های خاص خود دارد.

۳-۶-۱- تجارت الکترونیک امکان ارائه کالا و خدمات به صورت الکترونیکی را فراهم کرده است امّا چالش‌هایی مانند حفظ حریم خصوصی اشخاص (از قبیل شماره کارت اعتباری، پسوردها و ...) چالشی است که نیاز به الگوریتم‌های خاص خود دارد.

۴-۶-۱- شرکت‌های تجاری مانند یک شرکت نفت (که نیاز به بهترین جا برای حفر چاه دارد)، برای یافتن بهترین راه‌حل‌ها جهت کسب سود بیشتر با الگوریتم‌های خاص خود درگیرند و یا حتی یک کاندیدای انتخابات سیاسی نیاز دارد بداند در چه کمپین‌هایی بیشتر هزینه کند، تا شانس برنده شدن در انتخابات[1]را افزایش دهد.

۵-۶-۱- مسائلی مانند یافتن کوتاه‌ترین مسیر بین دو Node در یک شبکه، مسائل پرکاربردی هستند که الگوریتم‌های فراوانی برای آن‌ها ارائه شده است.

۶-۶-۱- حتی مسأله ساده‌ای مانند لیست کردن قطعات تشکیل‌دهنده‌ی یک قطعه به شرط اینکه هر قطعه فرزند، خود از قطعات دیگری تشکیل شده باشد، یک مسأله چالشی است که نیاز به الگوریتم‌های خاص خود دارد.

۳-۱- ساختمان‌های داده[2]مورد بحث در این کتاب:

یک ساختمان داده روشی برای ذخیره‌کردن و سازماندهی داده‌ها جهت آسان‌سازی دسترسی به آن‌ها و ویرایش آن‌هاست. همیشه یک نوع ساختمان داده برای همین اهداف پاسخ‌گو نیست؛ بنابراین، دانستن نقاط قوّت و محدودیت‌های ساختمان داده‌ها بسیار مهم است. در این کتاب ساختمان داده‌های مختلفی مورد استفاده قرار می‌گیرند.

۴-۱- تکنیک:

هرچند این کتاب می‌تواند مانند یک «کتاب آشپزی»[3]برای الگوریتم‌ها مورد استفاده قرار بگیرد امّا، به‌هرحال ممکن است شما با مسأله‌ای مواجه شوید که نمی‌توانید به راحتی یک الگوریتم از-پیش-منتشر-شده برای آن پیدا کنید. (مانند بسیاری از مسائل و تمرین‌های این کتاب).

---

[1] Election
[2] Data Structures
[3] Cookbook

این کتاب به شما تکنیک‌های طراحی و تحلیل الگوریتم (یعنی ماهیگیری) را یاد می‌دهد؛ بنابراین شما می‌توانید الگوریتم‌های دلخواه خود را توسعه داده و نشان دهید که پاسخ صحیحی برمی‌گردانند و کارایی آن‌ها را نیز درک کنید.

۵-۱- مسائل سخت:[۱]

اکثر این کتاب درباره‌ی الگوریتم‌های کارا[۲] است. معیار[۳] معمول ما برای کارایی، «سرعت»[۴] است؛ یعنی چقدر طول می‌کشد تا یک الگوریتم نتیجه‌اش را تولید کند.

- **مسائل NPC:** مسائلی هستند که هیچ راه‌حل کارایی برای آن‌ها قابل تصور نیست. به این مسائل در اصطلاح NPC یا NP-Complete گفته می‌شود.
- **مسائل NP:** دسته مسائلی هستند که نمی‌توان در یک زمان چندجمله‌ای، یک پاسخ قطعی برای آن پیدا کرد، امّا اگر یک پاسخ کاندید برای آن‌ها ارائه شود می‌توان، در یک زمان چندجمله‌ای صحّت آن پاسخ را تأیید یا رد نمود؛ مانند اینکه نمی‌دانیم فرضاً با ۱۰ میلیون تومان پول چه شغلی راه‌اندازی کنیم که به سوددهی بینجامد.
- **مسائل P:**[۵] دسته مسائلی هستند که می‌توان در یک زمان چندجمله‌ای، برای آن یک راه‌حل قطعی پیدا کرد.

۶-۱- تمرین:

۱- درباره‌ی قانون مور تحقیق کنید.

۲- یک مثال از دنیای واقعی بزنید که نیاز به مرتب‌سازی داشته باشد و یک مثال از دنیای واقعی که نیاز به مسیر پوشا داشته باشد.

پاسخ: مانند دریافت حقوق بازنشستگان که بر اساس حروف الفبا مرتب شده‌اند و یا مراحل پخت یک غذا که حتماً باید به ترتیب انجام شوند.

مسیر پوشا هم مانند خرید کردن که می‌توان خرید لوازم را طوی انجام داد که ابتدا آن‌هایی که در یک مسیر و به ترتیب هستند انجام شوند و بعد مابقی و یا مسافرت رفتن از یک شهر به شهر دیگر که ترتیب

---

[۱] Hard Problems

[2] Efficient

[3] Measure

[4] Speed

[۵] Polynomial

در آن رعایت می‌شود. و یا خرید ماهیانه پوشاک و مواد غذایی و یا پرداخت قبوض و ... که همه را باید با یک مبلغ مشخص انجام و پوشش داد.

۳- به جز سرعت چه معیار دیگری برای کارایی سراغ دارید که ممکن است شخص از آن در دنیای واقعی استفاده کند.

پاسخ: هزینه، کیفیّت، فضا و مکان و...

۴- یک ساختمان داده که قبلاً با آن آشنا شده‌اید انتخاب کنید و درباره‌ی نقاط قوّت و محدودیت‌های آن بحث کنید. (مثال: درباره‌ی ساختمان داده شئ[1]صحبت کنید)

پاسخ: مانند آرایه‌ها، پشته، لیست پیوندی، متغیر[2] عددی، متغیر رشته‌ای و ...

یادآوری:

LIFO: Last Input Frist Output

ساختمان داده‌ای که در آن، آخرین ورودی، اولین خروجی است؛ مانند پشته

FIFO: First Input First Output

ساختمان داده‌ای که در آن، اولین ورودی، اولین خروجی است؛ مانند نوارکاسِت یا صف نانوایی

۵- مسأله کوتاه‌ترین مسیر و مسأله فروشنده دوره‌گرد چه شباهت‌ها و چه تفاوت‌هایی دارند؟

پاسخ: شباهت‌ها: در هردو بحث کوتاه‌بودن مسیر بر اساس وزن یا همان مسافت مطرح است.

تفاوت‌ها: نقاط آغاز و پایان در مسأله فروشنده دوره‌گرد یکی است و همچنین پوشابودن مسیر در این مسأله مطرح است در حالی‌که در مسأله کوتاه‌ترین مسیر، لزوماً این‌طور نیست.

۶- مسأله‌ای در دنیای واقعی بیان کنید که فقط بهترین راه‌حل است که پاسخ مسأله است، همین‌طور مسأله‌ای مثال بزنید که یک راه‌حل تقریبی[3] نیز پاسخ‌گو است.

پاسخ: مانند استفاده از روش شیمی‌درمانی که در بعضی از موارد سرطانی، بهترین راه‌حل است و یا شرکت در مزایده و یا آتش‌نشانی که از بهترین راه‌حل‌ها برای خاموش کردن انواع آتش‌سوزی‌ها استفاده می‌کند.

---

[1] Object
[2] Float
[3] Approximately

۷-۱- پردازش موازی[1]:

تا سال‌ها ما روی افزایش سرعتِ کلاک[2] پردازش‌گر حساب می‌کردیم امّا محدودیت‌های فیزیکی[3] باعث می‌شود این افزایش سرعت تا حد مشخصی ممکن باشد؛ برای مثال یکی از این محدودیت‌ها این است که اگر سرعت کلاک بیش از حد افزایش پیدا کند، دمای پردازش‌گر به حدی بالا خواهد رفت که خطر ذوب‌شدن قطعات را درپی دارد؛ بنابراین امروزه برای محاسبات سنگین به‌جای افزایش سرعت کلاک از چندین هسته[4] در پردازش‌گر استفاده می‌شود. این حالت را می‌توان چند کامپیوتر در یک چیپ[5] واحد تصور کرد و در اصطلاح به آن «کامپیوتر موازی»[6] گفته می‌شود. برای اینکه از کامپیوترهای چند هسته‌ای بهترین کارایی را استخراج کنیم به الگوریتم‌هایی نیاز داریم که به آن‌ها الگوریتم‌های موازی یا الگوریتم‌های چندرشته‌ای[7] گفته می‌شود و در فصل ۲۷ در این زمینه بحث خواهد شد.

۸-۱- الگوریتم به‌عنوان یک فناوری:

در نظر بگیرید کامپیوترها به‌طور نامحدود، سریع و حافظه‌ی آن‌ها آزاد است. آیا دلیلی برای مطالعه الگوریتم‌ها وجود می‌داشت؟ پاسخ بله است. اگر بر فرض محال این دو محدودیت را نداشته باشیم امّا روش پیشنهادی در این الگوریتم بالاخره باید نمایش[8] داده شود و از آن مهم‌تر پیاده‌سازی شود و همین‌طور ثابت شود که این روش پاسخ صحیحی برای مسأله است، اگر راه‌حل پیشنهادی بیش‌ازحد پیچیده[9] باشد، در پیاده‌سازی[10] یا نمایش یا اثبات درستی آن با مشکل مواجه خواهیم شد. البتّه که کامپیوترها به‌طور نامحدود، سریع و حافظه‌ی آن‌ها بی‌نهایت نیست؛ بنابراین محاسبه زمان و فضای حافظه اجرای یک الگوریتم یک *منبع محدود*[11] به حساب می‌آید. شما باید از این منابع به‌طور هوشمندانه استفاده کنید و از الگوریتم‌هایی استفاده کنید که در بحث زمان و فضا کارا هستند.

تمرین: درباره‌ی مسأله *فضا-زمان*[12] نیوتن تحقیق کنید.

---

[1] Parallelism

[2] Clock

[3] Physical Limitations

[4] Core

[5] Chip

[6] Parallel Computer

[7] Multithreaded Algorithms

[8] Demonstrate

[9] Complex

[10] Implementation

[11] Resource Bounded

[12] Time Space

۹-۱- کارایی[1]:

الگوریتم‌های مختلفی که برای حل یک مسأله مشابه پیشنهاد می‌شود، معمولاً در «کارایی» به شدّت با هم متفاوتند. این تفاوت‌ها زمانی که سخت‌افزارها و نرم‌افزارهای دیگر مورد استفاده برای اجرای آن‌ها با هم متفاوت باشد بیشتر نیز خواهد شد؛ به عنوان مثال، در فصل ۲ ما دو الگوریتم برای مرتب‌سازی خواهیم دید که اولی به نام «مرتب‌سازی درجی»[2] شناخته می‌شود که این الگوریتم زمان C_1n^2 را برای مرتب‌سازی n آیتم نیاز دارد، (C_1 عدد ثابتی است که وابسته به n نیست) که می‌گوییم این الگوریتم با n^2 متناسب[3]است، دومین الگوریتم «مرتب‌سازی ادغامی»[4]است که زمان $C_2n\log n$ دارد (C_2 یک عدد ثابت غیر وابسته به n است و منظور از $\log n$، $\log_2 n$ ($\log n$ در مبنای ۲) است). مرتب‌سازی درجی معمولاً عدد ثابت کمتری نسبت به مرتب‌سازی ادغامی دارد (یعنی $C_1 < C_2$ است). خواهیم دید که عدد ثابت، تأثیر[5] بسیار کمی روی زمان دارد؛ بنابراین می‌توان *زمان اجرای*[6] مرتب‌سازی درجی را n و زمان اجرای مرتب‌سازی ادغامی را $\log n$ تصور کرد. هرچند مرتب‌سازی درجی در تعداد کم سریع‌تر عمل می‌کند؛ برای مثال فرض کنید یک کامپیوتر قوی (کامپیوتر A با قدرت پردازش ۱۰ *میلیارد دستورالعمل در ثانیه*[7]) الگوریتم مرتب‌سازی درجی را اجرا می‌کند، و یک کامپیوتر ضعیف‌تر (کامپیوتر B با قدرت پردازش ۱۰ میلیون دستورالعمل در ثانیه) الگوریتم مرتب‌سازی ادغامی را اجرا می‌کند. هر دو قادر هستند ۱۰ میلیون عدد را مرتب کنند، در این حالت خواهیم دید که کامپیوتر A، ۱۰۰۰ برابر سریع‌تر از کامپیوتر B عمل خواهد کرد، در حالی‌که نمی‌توان از این نتیجه گرفت که الگوریتم مرتب‌سازی درجی سریع‌تر از الگوریتم مرتب‌سازی ادغامی عمل می‌کند.

۱۰-۱- الگوریتم‌ها و دیگر فناوری‌ها:[8]

مثال بالا نشان داد که ما باید الگوریتم‌ها را مانند سخت‌افزار کامپیوتر به‌عنوان یک فناوری در نظر بگیریم. همان‌طور که انتخاب یک سخت‌افزار سریع در کارایی سیستم موثر است، انتخاب الگوریتم‌های کارا نیز در کارایی سیستم تأثیر فراوانی دارد. همان‌طور که پیشرفت‌های سریع در دیگر فناوری‌های مرتبط با کامپیوتر پیش می‌آید، در الگوریتم‌ها نیز پیشرفت‌های سریع و مهمی به وجود می‌آید.

ممکن است لازم باشد الگوریتم‌ها را با توجه به دیگر فناوری‌های پیشرفته مانند موارد زیر انتخاب کنیم:

---

[1] Efficiency
[2] Insertion Sort
[3] Proportional
[4] Merge Sort
[5] Factor
[6] Running Time
[7] 10 billion instructions per second
[8] Algorithms and other Technologies

۱-۱۵-۱-معماری‌های پیشرفته کامپیوتر و تکنولوژی‌های ساخت آن‌ها

۲-۱۵-۱-رابط‌های کاربری گرافیکی[1] با جلوه‌های[2] زیبا و کاربری آسان[3]

۳-۱۵-۱-سیستم‌های شی‌گرا[4]

۴-۱۵-۱-فناوری‌های یکپارچه[5] وب

۵-۱۵-۱-شبکه‌های سریع (کابلی و بی‌سیم)

برای مثال یک سرویس تحت وب را تصور کنید که یک مسیر را بین دو نقطه یافته و به راننده پیشنهاد می‌کند این سرویس علاوه بر این‌که به الگوریتم‌های خاصی برای یافتن کوتاه‌ترین مسیر نیاز دارد با الگوریتم‌های دیگری برای «رِندِر کردن»[6](آماده کردن خروجی نهایی) نقشه و تشخیص آدرس‌ها و اماکن و دیگر الگوریتم‌ها درگیر است. داشتن یک الگوریتم پایه و تکنیک‌های طراحی الگوریتم یکی از ویژگی‌هایی است که *برنامه‌نویس ماهر*[7] را در *برنامه‌نویس تازه‌کار*[8] متمایز می‌کند. امّا این ویژگی کافی نیست، آشنایی با دیگر فناوری‌ها که با الگوریتم‌ها درگیر است نیز برای یک برنامه‌نویس ماهر لازم است.

هرچند ممکن است شما کارهای خود را بدون دانستن الگوریتم‌ها انجام دهید، امّا داشتن یک *پیش‌زمینه‌ی خوب*[9] درباره‌ی الگوریتم‌ها باعث می‌شود، بهتر و بیشتر از قبل کار *انجام دهید*[10][11].

۱۱-۱- تمرین:

۱- یک مثال از یک برنامه کاربردی بزنید که نیاز به محتوای الگوریتمی در سطح اپلیکیشن دارد و در مورد الگوریتم‌هایی که در این زمینه مشکل خاص دارند، بحث کنید.

پاسخ: مانند آپدیت کردن ساعت کنار ویندوز و یا نوع نمایش سه‌بُعدی پنجره‌ها در سطح اپلیکیشن

۲- فرض کنید قرار است در یک ماشین مشابه الگوریتم‌های درجی و ادغامی را اجرا کنیم و به‌ازای n ورودی، الگوریتم‌درجی $8n^2$ گام طی کند و الگوریتم ادغامی$64n\log n$. برای چه مقادیری از n الگوریتم‌های درجی بهتر از الگوریتم‌های ادغامی عمل می‌کنند؟

---

[1] Graphical User Interfaces = UFI: User-Friendly Interface
[2] Effects
[3] Easy-to-Use
[4] Object-Oriented System
[5] Integrated
[6] Rendering
[7] Skilled Programmer
[8] Novice Programmer
[9] Good Background
[10] Task
[11] Accomplish

۳- کوچکترین عدد n در الگوریتمی که زمان اجرای آن $100n^2$ بار است، و الگوریتمی که زمان اجرای آن 2^n است چیست، به شرطی که الگوریتم اول سریع‌تر از الگوریتم دوم عمل کند؟ (هر دو در یک ماشین مشابه اجرا می‌شوند.)

پاسخ: به جای n اعداد ۱ و بیشتر را قرار دهید تا زمانی که $100n^2$ کوچک‌تر 2^n از شود. خواهیم دید که زمانی که n برابر با ۱۵ باشد، اولی ۲۲۵۰۰ خواهد شد و دومی ۳۲۷۶۸ اما اگر n را ۱۴ بگیریم، اولی ۱۶۶۰۰ خواهد شد و دومی ۱۶۳۸۴. پس کوچک‌ترین n، عدد که شرط مسأله را تأمین کند، ۱۵ خواهد بود.

فصل دوم

آغاز کار با الگوریتم‌ها

۲-۱- مقدمه:

این فصل شما را با چارچوبی که ما در کتاب استفاده خواهیم کرد، آشنا می‌کند تا یاد بگیرید چطور درباره‌ی طراحی و تحلیل الگوریتم‌ها بیندیشد:

۲-۲- الگوریتم مرتب‌سازی درجی:

اوّلین الگوریتم، مرتب‌سازی درجی است که مسأله مطرح شده در فصل قبل را حل می‌کند:

ورودی: یک توالی از n عدد $(a_1, a_2, \dots, a_n)$

خروجی: توالی مرتبی به شکل $(a'_1, a'_2, \dots, a'_n)$ به‌طوری‌که $(a'_1 \leq a'_2 \leq \cdots \leq a'_n)$

اعدادی که قرار است مرتب کنیم به نام «کلید»[1]نیز شناخته می‌شوند. اعداد ورودی به شکل یک آرایه‌ی n عنصری به ما داده می‌شوند.

در این کتاب ما الگوریتم‌ها را در غالب برنامه‌هایی که به شکل «شبه‌کد»[2] نوشته شده‌اند توصیف[3] می‌کنیم. شبه‌کدها شبیه زبان‌های C، C++، جاوا، پای‌ثون[4]و پاسکال هستند. اگر با یکی از این زبان‌ها آشنا باشید با کمی تلاش می‌توانید شبه‌کدها را با آن زبان پیاده‌سازی کنید.

تفاوت شبه‌کد با *کد واقعی*[5]این است که در شبه‌کد شفاف‌ترین روش به شکلی مختصر و مفید برای هر الگوریتم بیان می‌شود. تفاوت دیگر شبه‌کد با کد واقعی این است که شبه‌کد معمولاً به چالش‌های[6] *مهندسی*

---

[1] Key
[2] Pseudo Code
[3] Describe
[4] Python
[5] Real Code
[6] Issues

نرم‌افزار[1] چندان توجهی ندارد. چالش‌هایی مانند انتزاع‌داده‌ای،[2] پیمانه‌ای بودن «ماژولاریتی»[3] و *برخورد با خطاها*[4] اغلب در شبه‌کد *نادیده گرفته*[5] می‌شود تا ماهیّت[6] الگوریتم شفاف‌تر، مختصرتر بیان شود.

ما با الگوریتم درجی شروع می‌کنیم که یک الگوریتم کارا[7] برای مرتب‌سازی تعداد کمی عنصر[8] است. مرتب‌سازی درجی روشی است که بسیاری از افراد برای مرتب‌کردن تعدادی کارت در بازی کارت استفاده می‌کنند. اگر به روشی که افراد طی می‌کنند، دقت کنیم آن‌ها کارت‌ها را روی میز پخش می‌کنند و سپس دست چپ خود را که در ابتدا خالی[9] است به‌عنوان محلی برای مرتب‌سازی در نظر می‌گیرند سپس در هر بار یک کارت از روی میز برداشته و در *محل صحیح*[10] خود در دست چپ‌شان درج می‌کنند. هر بار که یک کارت برداشته می‌شود برای یافتن محل صحیح آن از چپ به راست کارت‌های موجود در دست چپ با کارت فعلی مقایسه می‌شود تا به محلی برسیم که کارت موردنظر بزرگ‌تر از آن جایگاه و کوچک‌تر از جایگاه بعدی آن است.

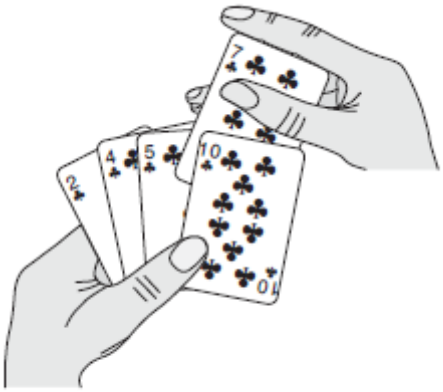


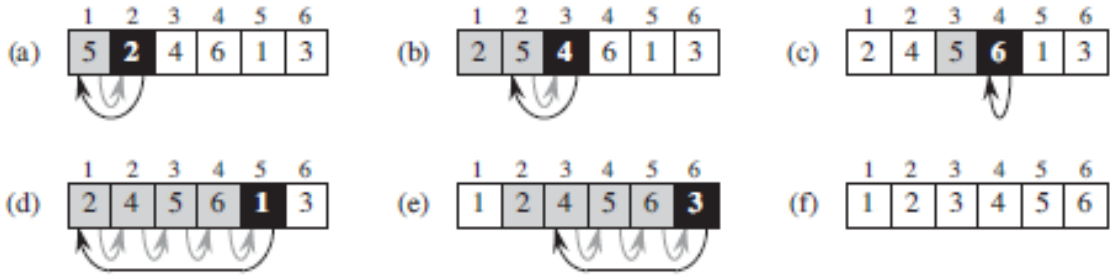
Figure 2.1 Sorting a hand of cards using insertion sort.

شکل ۱-۲ (الگوریتم مرتب‌سازی درجی شبیه به مرتب‌کردن کارت‌ها در بازی کارت یا پاسور است)

---

[1] Software-Engineering
[2] Data Abstraction: منظور از انتزاع، موهومی و مفهومی بودن یک موضوع است (خیلی قابل لمس نیست)
[3] Modularity
[4] Error Handing
[5] Ignored نادیده گرفتن
[6] Essence
[7] Efficient
[8] Element
[9] Empty
[1] Correct Position [0]

ما شبه‌کد را در غالب یک رویه[1](یا تابع) به نامInsertion-Sort ارائه می‌کنیم که به‌عنوان پارامتر[2] ورودی یک آرایه به شکل $A\ [1..n]$ دریافت می‌کند که شامل توالی به طول n از اعدادی است که قرار است مرتب شوند. در کد ما طول آرایه‌ی A به شکل A.Length نمایش داده شده است. این الگوریتم اعداد ورودی را درجا[3]مرتب می‌کند؛ یعنی همان آرایه‌ی A را بدون نیاز به آرایه‌ی دیگری آن‌قدر جابه‌جا می‌کند تا مرتب شوند. شکل زیر مثالی از مرتب‌سازی درجی است و روالی را که طی می‌شود، نشان می‌دهد.



شبه‌کد زیر معروف به الگوریتم مرتب‌سازی درجی است.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

مثال: جدول تغییرات متغیرها فرضاً برای آرایه‌ی $A = \{5, 2, 4, 6, 1, 3\}$:

[1] Procedure
[2] Parameter
[3] In place

جدول تغییرات متغیرها			
A	**J**	**i**	**Key**
{5,2,4,6,1,3}	1	0	2
{5,5,4,6,1,3}		-1	
{2,5,4,6,1,3}	2	1	4
{2,5,5,6,1,3}		0	
{2,4,5,6,1,3}	3	2	6
	4		1
{2,4,5,6,6,3}		3	
{2,4,5,5,6,3}		1	
{2,4,4,5,6,3}		0	
{2,2,4,5,6,3}		-1	
{1,2,4,5,6,3}	5		3
{1,2,4,5,6,6}		4	
{1,2,4,5,5,6}		3	
{1,2,4,4,5,6}		2	
{1,2,3,4,5,6}		1	
	0		

توضیحات مربوط به الگوریتم بالا:

- اندیس j کارت فعلی که قرار است در جای خود در دست چپ درج شود، مشخص می‌کند.
- در هر دور از حلقه for، $A[1..j-1]$ بخشی از آرایه است که تا الان مرتب شده است و بقیه آرایه در $A[j+1..n]$ قرار دارد.
- حلقه‌ی while برای شیفت دادن عناصر به یک خانه بعد از خود استفاده می‌شود تا خانه‌ی مربوط به عنصر فعلی خالی می‌شود.
- یادآوری: هر حلقه شامل بخش‌های زیر است:

```
for(initialization, maintenance & termination, Iteration-Expression)
{
   //… body
}
```

۱- عملیات آغازی که در اصطلاح به آن Initialization گفته می‌شود.

۲- شرط اجرای حلقه که در اصطلاح به آن Maintenance به معنی حفظ و نگه‌داری گفته می‌شود.
۳- عملیات هر گام که در اصطلاح به آن Iteration-Expression گفته می‌شود.
۴- شرط پایان که در اصطلاح به آن Termination گفته می‌شود.
۵- بدنه‌ی حلقه که حاوی کدهایی است که در صورت صحّت شرط، اجرا خواهد شد و در اصطلاح به آن Body گفته می‌شود.

۳-۲- تمرین:

۱- مانند شکل ۲,۲ مدلی رسم کنید که عملیات الگوریتم مرتب‌سازی درجی را روی آرایه زیر نشان دهد.

$$A = \{31, 41, 59, 26, 41, 58\}$$

۲- تابع insertion-sort را طوری بنویسید که به جای مرتب‌سازی صعودی، مرتب‌سازی نزولی انجام دهد.
۳- *مسأله جستجو*[1] را در نظر بگیرید:

ورودی: توالی از n عدد به شکل $A = (a1, a2, \dots, an)$ و یک مقدار به نام V که در حال جستجوی آن هستیم.

خروجی: اندیس i به‌طوری‌که $V = A[i]$ یا مقدار NIL[2] اگر V در آرایه پیدا نشود.

یک تابع برای *جستجوی خطی*[3] بنویسید که این مسأله را حل کند.

پاسخ:

---

[1] Searching Problem

[2] صفر

[3] Linear Search

```cpp
#include<iostream>
using namespace std;
void insert(int [],int);
int search(int [],int,int);
int main()
{
    int a[100],n,key,result;
    cout<<"Please enter number of digits: ";
    cin>>n;
    for(int i=0;i<n;i++)
    {
        cout<<"Please enter a number: ";
        cin>>a[i];
    }
    cout<<"Please enter a number to search: ";
    cin>>key;
    result=search(a,n,key);
    if(result==0)
        cout<<"The key was not found "<<key;
    else
    {
        cout<<"The key was found: ";
        cout<<"a["<<result<<"]="<<key;
    }
    return 0;
}
int search(int a[],int n,int key)
{
    for(int i=0;i<n;i++)
    {
        if(a[i]==key)
            return i;
    }
    return 0;
}
```

۴- فرض کنید دو آرایه باینری به طول n بیت به نام A و B داریم. جمع هرکدام از خانه‌های دو آرایه را در خانه‌ی متناظر آن‌ها در آرایه C ذخیره کنید و سپس آرایه C را چاپ کنید. برای این مسأله یک تابع تعریف کنید. (راهنما: با توجه به اینکه ممکن است جمع دو عدد یک خانه اضافه‌تر نیاز داشته باشد، آرایه C را n+1 خانه‌ای در نظر بگیرید.)

پاسخ:

```c
//Sum of two arrays in binary
#include<stdio.h>
#include<conio.h>
int main(){
    int n;
    printf("Please enter array length:");
    scanf("%d",&n);
    int a[n];
    int b[n];
    int c[n+1];
    printf("Please enter n bits for A:");
    for(int i=n-1;i>=0;i--)
    {
        a[i]=0;
        scanf("%d",&a[i]);
    }
    printf("Please enter n bits for B:");
    for(int i=n-1;i>=0;i--)
    {
        b[i]=0;
        scanf("%d",&b[i]);
    }
    int rem=0;

    for(int i=n;i>=1;i--)
    {
        int r = a[i-1]+b[i-1]+rem;
        if(r==0 || r==1)
        {
            c[i]=r;
        }
        else if(r==2)
        {
            c[i]=0;
            rem=1;
        }
        else if(r==3)
        {
            c[i]=1;
            rem=1;
        }
    }
    c[0]=rem;
    for(int i=0;i<=n;i++)
    {
        printf("%d ",c[i]);
    }
    getch();
}
```

۴-۲- تحلیل الگوریتم‌ها:

منظور از تحلیل[۱] یک الگوریتم پیش‌بینی[۲] منابع مورد نیاز الگوریتم است. معمولاً منابعی مانند حافظه،[۳] *پهنای باند ارتباطی*،[۴] سخت‌افزار کامپیوتر اصلی نگرانی[۵] ما هستند، امّا اغلب‌اوقات *زمان محاسبه*[۶] تنها چیزی است که ما می‌خواهیم *اندازه بگیریم*.[۷]

نکته: منظور از Memory حافظه موقت و یا RAM می‌باشد و منظور از Story حافظه جانبی و یا Hard Disk است.

معمولاً با تحلیل چند الگوریتم کاندید[۸] برای یک مسأله می‌توانیم کاراترین‌شان را *شناسایی کنیم*.[۹] ممکن است تحلیل به این نتیجه برسد که بیش از یک کاندید *قابل اعتماد*[۱۰] است، امّا اغلب می‌توانیم *الگوریتم‌های با سطح پایین‌تر*[۱۱] را *از رده خارج کنیم*.[۱۲] قبل از تحلیل یک الگوریتم باید یک مدل از فناوری پیاده‌سازی[۱۳] الگوریتم که در این کتاب استفاده کرده‌ایم، ارائه کنیم؛ مدلی که منابع آن فناوری و هزینه‌های آن را مشخص کند. در این کتاب ما از یک پردازش‌گر[۱۴] و *ماشینی با حافظه‌ی دسترسی تصادفی*[۱۵] (RAM) برای مدل کردن محاسبات و درک الگوریتم ها استفاده خواهیم کرد. در مدل RAM دستورالعمل‌ها[۱۶] یکی پس از دیگری پردازش می‌شوند و هیچ *عملیات هم‌زمانی*[۱۷] اتفاق نمی‌افتد. مدل RAM شامل دستورالعمل‌هایی است که در کامپیوترهای واقعی وجود دارند؛ مانند دستورالعمل‌های محاسباتی[۱۸] (جمع،[۱۹] تفریق،[۲۰] ضرب،[۲۱] تقسیم،[۲۲] باقیمانده،[۲۳] *جزءِصحیح*[۲۴] و

---

[1] Analyzing
[2] Predicting
[3] Memory
[4] Communication bandwidth
[5] Concern
[6] Computational time
[7] Measure
[8] Candidate
[9] Identify
[10] Viable
[11] Inferior Algorithms
[12] Discard= از رده خارج کردن
[13] Implementation
[14] One Processor
[15] Random Access Machine
[16] Instructions
[17] Concurrent Operations
[18] Arithmetic
[19] Add
[20] Subtract
[21] Multiply
[22] Divide
[23] Remainder

[24] Floor : منظور از کف یا جزءصحیح یک عدد اعشاری، بزرگ‌ترین عدد صحیح کوچک‌تر از آن عدد است؛ به‌طورمثال کف عدد ۵/۷ عدد ۵ می‌شود.

سقف[1] و ...)، دستورالعمل‌های *جابجایی داده*[2] (خواندن یا بارگذاری،[3] ذخیره‌سازی،[4] کپی[5] و دستورالعمل‌های کنترلی (*دستورالعمل‌های شرطی*[6] و انشعاب[7] غیر شرطی،[8] *فراخوانی و بازگشت*[9] تودرتو). هر یک از این دستورالعمل‌ها *مقدار زمانی ثابتی*[10] نیاز دارد.

نمونه سؤال امتحان: قطعه کد زیر مربوط به یک الگوریتم مرتب‌سازی است، آرایه‌ی {۴،۲،۱،۱۰،۹،۷} را با توجه به کد مرتب کنید و در نهایت مشخص کنید قطعه کد مربوط به کدام الگوریتم مرتب‌سازی است؟ (رسم جدول تغییرات متغیرها و شکل کلی مرتب شدن آرایه الزامی است).

۵-۲- تحلیل الگوریتم مرتب‌سازی درجی:

زمانی‌که الگوریتم مرتب‌سازی درجی نیاز دارد، به ورودی بستگی دارد؛ برای مثال، مرتب‌سازی ۱۰۰۰ عدد زمان بیشتری نسبت به مرتب‌سازی ۳ عدد صرف می‌کند. نکته‌ی دیگر اینکه این الگوریتم به‌ازای مقادیری که مرتب‌تر هستند، زمان کمتری صرف می‌کند. عموماً زمانی‌که توسط یک الگوریتم صرف می‌شود، نسبت به اندازه‌ی ورودی رشد[11] می‌کند؛ بنابراین مرسوم[12] است که *زمان اجرای*[13] یک برنامه را *تابعی از اندازه‌ی ورودی*[14] آن در نظر می‌گیرند؛ بنابراین برای تحلیل الگوریتم باید ابتدا واژه‌های «زمان اجرا» و «*اندازه‌ی ورودی*»[15] را با *دقت بیشتری*[16] تعریف کنیم. بهترین تعریف برای «اندازه‌ی ورودی» به مسأله‌ای که درحال تحلیل هستیم، بستگی دارد؛ برای مثال در خیلی از مسائل مانند مرتب‌سازی اندازه‌ی ورودی برابر است با ***تعداد آیتم‌های ورودی***؛[17] (یعنی اندازه n در مسائل مرتب‌سازی)، امّا در مسائل دیگر مانند ضرب دو عدد صحیح، اندازه‌ی ورودی عبارت‌است از: ***تعداد کل لیست‌های***[18] مورد نیاز برای نمایش اعداد ورودی در مبنای ۲.

---

[1] Ceiling : منظور از سقف یک عدد اعشاری، بزرگ‌ترین عدد صحیح بعد از آن عدد است؛ به‌طور مثال سقف عدد ۵/۷ عدد ۶ می‌شود.

[2] Data movement
[3] Load
[4] Store
[5] Copy
[6] Conditional
[7] Branch
[8] Unconditional
[9] Call and return
[10] Constant amount of time
[11] Grows
[12] Traditional
[13] Running time
[14] Function of the size of its input
[15] Size of input
[16] More carefully
[17] Number of items the input
[18] Total number of bits

در ابتدای تحلیل هر الگوریتم باید منظور از «اندازه‌ی ورودی» بیان شود.

{**مطالعه آزاد**: الگوریتم ژنتیک چیست؟ مزایا و معایب آن و ربط این الگوریتم را با قانون تکامل داروین بیان کنید.

یک مسأله بیابید که با الگوریتم ژنتیک حل شود.}

منظور از «زمان اجرا»ی یک الگوریتم به‌ازای یک ورودی خاص تعداد *عملیات اصلی*[1] یا گام‌هایی[2] که در اجرای برنامه طی شده است، می‌باشد. بهتر است زمان اجرا را تعداد گام‌ها در نظر بگیریم تا *غیروابسته به ماشین*[3] باشد. هرچند اجرای هر خط از شبکه زمان متفاوتی نیاز دارد، امّا با زمان اجرای «***i***مین» خط را C_1 در نظر می‌گیریم و منظور از C کلمه Constant به معنی یک زمان ثابت است. ما در الگوریتم مرتب‌سازی درجی «هزینه»ی هر دستورالعمل را یک واحد در نظر می‌گیریم و به‌ازای تعداد دفعاتی که آن دستورالعمل اجرا می‌شود، یک واحد به هزینه اضافه می‌کنیم؛ بنابراین داریم:

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

زمان اجرای الگوریتم برابر است با جمع تعداد دفعات اجرای هر دستورالعمل. توجه کنید که دستورالعملی که C_i گام برای اجرا طول می‌کشد و n بار اجرا می‌شود، کل زمان اجرایش C_in خواهد شد. برای محاسبه $T(n)$ (یعنی کل زمان اجرای الگوریتم) جمعِ ضربِ هزینه در تعداد دفعات اجرا را بدست آورید؛ بنابراین برای الگوریتم بالا داریم:

[1] Primitive operations
[2] Step
[3] Machine independent

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

در این الگوریتم *بهترین حالت*[1] زمانی رخ می‌دهد همه‌ی عناصر آرایه‌ی ورودی مرتب باشند. در این حالت با توجه به اینکه $A[i] \leq Key$ (یعنی هر عنصر، کوچک‌تر یا مساوی عنصر بعدی خود است) حلقه‌ی while هرگز اجرا نمی‌شود؛ پس T(n) در بهترین حالت به‌صورت زیر خواهد بود؛

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

بنابراین می‌توان زمان اجرا را به شکل $an + b$ نمایش داد که ثوابت اعداد a , b به C_i وابسته هستند؛ نتیجه اینکه T(n) یک تابع خطی از n است.

اگر آرایه به‌طور برعکس باشد (یعنی مرتب نزولی) بدترین حالت رخ می‌دهد. در این حالت حلقه‌ی for به‌ازای تمام عناصر ورودی یک‌بار اجرا می‌شود؛ یعنی $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ و به‌طور ساده‌تر می‌توان نوشت:

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

نتیجه اینکه T(n) در بدترین حالت برای الگوریتم مرتب‌سازی درجی عبارتنداز:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

[1] Best-case

می‌توان بدترین حالت را به شکل $an^2 + bn + c$ نشان داد که a, b و c ثوابتی هستند که وابسته به هزینه‌های C_i می‌باشند؛ بنابراین این زمان اجرا، *توان ۲‌ای از* n [1] است.

مثال: T(n) را برای الگوریتم جستجوی خطی به‌ازای n ورودی در بهترین و بدترین حالت به دست آورید.

پاسخ:

بهترین حالت:

هزینه	دفعات اجرا
C1 = 0	1
C2 = 0	1
C3	1
C4 = 0	1
C5	1
C6	1
C7 = 0	1
C8	0
C9 = 0	1

$$T(n) = (0 * 1) + (0 * 1) + (c_3 * 1) + (0 * 1) + (c_5 * 1) + (c_6 * 1) + (0 * 1) + (c_8 * 0) + (0 * 1) = c_3 + c_5 + c_6$$

بدترین حالت:

هزینه	دفعات اجرا
$C_1 = 0$	1
$C_2 = 0$	1
C_3	n
$C_4 = 0$	n – 1
C_5	n – 1
C_6	1
$C_7 = 0$	n – 1
C_8	0
$C_9 = 0$	1

[1] Quadratic function

$$T(n) = (0 * 1) + (0 * 1) + (c_3 * n) + (0 * (n-1)) + (c_5 * (n-1)) + (c_6 * 1) + (0 * (n-1)) + (c_8 * 0) + (0 * 1) = (c_3 * n) + (c_5(n-1)) + c6 = n\,c_3 + n\,c_5 - c_5 + c_6 = n(c_3 + c_5) + c_6 - c_5 = na + b$$

۶-۲- تمرین:

اگر هزینه‌ی اجرای دستورات انتصاب را در آرایه $A = [31, 41, 59, 26, 41, 58]$، ۱ در نظر بگیریم و هزینه‌ی اجرای حلقه‌ها و if و else را ۲ و دیگر عملیات را هم ۱ در نظر بگیریم، حساب کنید به‌ازای آرایه‌ی زیر در بهترین و بدترین حالت، زمان اجرای الگوریتم مرتب‌سازی درجی چقدر خواهد بود؟

۷-۲- تحلیل بهترین و بدترین حالت:

هرچند در تحلیل الگوریتم مرتب‌سازی درجی ما بهترین و بدترین حالت را محاسبه کردیم، امّا ازاین‌پس فقط روی به دست آوردن *زمان اجرای بدترین حالت*[۱] متمرکز می‌شویم که عبارت است از «طولانی‌ترین زمان اجرا برای هر ورودی به اندازه n »؛ چراکه:

۱- زمان اجرای بدترین حالت یک الگوریتم، *بالاترین حد*[۲] زمان اجرا را برای هر ورودی به دست ما می‌دهد و یک ضمانت[۳] برای این است که آن الگوریتم هرگز بیشتر از این زمان طول نخواهد کشید. ما نیاز داریم که یک *برآورد تجربی*[۴] از بدترین زمان اجرا داشته باشیم که امیدوار باشیم که الگوریتم هیچ‌وقت شرایط بدتری را تجربه نمی‌کند.

۲- در برخی الگوریتم‌ها، بدترین حالت خیلی به‌ندرت رخ می‌دهد. (برای مثال در جستجوی یک *پایگاه داده*[۵] برای یک داده خاص درحالی‌که آن داده در Database وجود ندارد)، امّا در برخی کاربردها جستجوها مکرراً به‌دنبال داده‌ای می‌گردند که غایب است.

۳- حالت میانگین[۶] نیز اغلب اوقات مانند بدترین حالت است. برای مثال، فرض کنید ما n عدد تصادفی انتخاب می‌کنیم و الگوریتم مرتب‌سازی درجی را روی آن‌ها اعمال می‌کنیم. چقدر طول خواهد کشید تا در زیر آرایه‌ی $A[1..j-1]$ عنصر $A[j]$ درج شود؟ در حالت میانگین، نیمی از عناصر این زیر-آرایه کوچک‌تر از $A[j]$ و نیمی بزرگ‌تر هستند؛ بنابراین، در حالت میانگین نیمی از عناصر این زیر-آرایه چک می‌شود؛ بنابراین، t_j برابر با $j/2$ است. در نتیجه زمان اجرای حالت میانگین نیز تابع درجه دومی[۷] از اندازه ورودی است؛ دقیقاً مانند زمان اجرای بدترین حالت است.

۴- در برخی مسائل ما به‌دنبال زمان اجرای حالت میانگین یک الگوریتم هستیم، امّا همان‌طورکه در فصل‌های آینده در تکنیک *تحلیل احتمالاتی*[۸] صحبت خواهیم کرد، محدوده[۹] تحلیل حالت میانگین محدود[۱۰] است چراکه گاهی‌اوقات مشخص نیست مقدار ورودی میانگین یک مسأله خاص چیست. در اکثر مواقع ما در نظر می‌گیریم که ورودی‌ها تقریباً به یک اندازه‌ی مساوی هستند، امّا درعمل[۱۱] این فرضیه[۱۲] ممکن است

---

[1] Longest running time
[2] Upper bound
[3] Guarantee
[4] Educated guess
[5] Database
[۶] Average case
[۷] Quadratic function
[8] Probabilistic analysis
[9] Scope
[1] limited [0]
[۱۱] In practice

نقض شود[2] در این مواقع از *الگوریتم‌های تصادفی*[3] استفاده می‌کنیم تا انتخاب‌های تصادفی از کل داده‌ها با اندازه‌های مختلف داشته باشد تا اجازه دهد که با کمک تحلیل احتمالاتی زمان اجرای مورد انتظار را پیش‌بینی کنیم.

۸-۲- نرخ رشد[4]

در تحلیل الگوریتم مرتب‌سازی درجی ما برای اینکه تحلیل الگوریتم ساده‌تر شود برخی مفاهیم را حذف کردیم؛ مثلاً در ابتدا هزینه‌ی واقعی هر عبارت را نادیده گرفتیم و همه را مقدار ثابت C_i در نظر گرفتیم.

سپس مشاهده کردیم که در این حالت نیز جزئیات زیادی به‌دست می‌آید؛ برای مثال زمان اجرای بدترین حالت به‌صورت $an^2 + bn + c$ به دست آمد که به ثوابت[5] a, b و c وابسته است و آن ثوابت خود به هزینه‌های C_i وابسته بودند؛ درنتیجه ثوابت وابسته به C_i را نیز نادیده گرفتیم.

در ادامه ما یک ساده‌سازی دیگر انجام می‌دهیم که به آن نرخ‌رشد[6] یا ترتیب‌رشد[7] گفته می‌شود و عبارت‌است‌از اینکه فقط *جمله‌ی آغازین*[8] فرمول[9] an^2 را در نظر می‌گیریم. چراکه جملات بعدی در اندازه‌های عظیم n ناچیز خواهد بود. همچنین ضریب[10] ثابت جمله‌ی اول یعنی $a.an^2$ را نیز به‌دلیل تأثیر محاسباتی کمی که دارد حذف می‌کنیم.

در بدترین حالت الگوریتم مرتب‌سازی درجی با حذف این موارد تنها n^2 می‌ماند که آن را به شکل θn^2 نشان می‌دهیم و می‌خوانیم «تِتای اِن‌دو».

از این پس وقتی گفته می‌شود یک الگوریتم کاراتر از الگوریتم دیگر است، به این معنی می‌باشد که نرخ رشد زمان اجرای بدترین حالت مربوط به الگوریتم اول کمتر از الگوریتم دوم است. البته دقت کنید که با توجه به ضرایب ثابت و جملات دوم به بعد که حذف کردیم ممکن است الگوریتم دوم برای ورودی‌های کوچک زمان

---

[1] assumption : تفاوت فرضیه و تئوری این است که فرضیه چیزی است که اثبات نشده ولی تئوری چیزی است که روی آن کار و اثبات شده است.

[2] Violated نقض شدن

[3] Randomized algorithm

[4] Order of growth

[5] Constants

[6] Rate of growth

[7] Order of growth

[8] Leading term

[9] Formula

[10] Factor

[11] θ : theta of n-squared

کمتری نسبت به الگوریتم اول صرف کند اما اگر ورودی‌ها به اندازه کافی بزرگ باشد قطعا الگوریتمی که برای مثال $\Theta(n^2)$ است، سریع‌تر از الگوریتمی که $\Theta(n^3)$ است عمل می‌کند.

۹-۲- تمرین:

۱- اگر زمان اجرای یک الگوریتم در بدترین حالت $\frac{n^3}{100} - 100n^2 - 100n + 3$ باشد θ را برای آن نمایش دهید.

پاسخ: طبق مطالب گفته شده در بالا ضرایب ثابت و جملات دوم به بعد را حذف می‌کنیم؛ بنابراین $\theta = n^3$ می‌باشد.

۲- مرتب‌سازی n عدد که در آرایه A ذخیره شده‌اند را به این شکل که ابتدا کوچک‌ترین عدد را یافته و با $A[1]$ جا به جا کنید و سپس دومین عدد کوچک را با $A[2]$ جا به جا کنید و این کار را تا عنصر n-1ام ادامه دهید، در نظر بگیرید. شبه‌کد این الگوریتم را بنویسید. (راهنما: این الگوریتم به‌نام «*الگوریتم انتخابی*» شناخته می‌شود.) چرا این الگوریتم فقط n-1 عنصر را بررسی می‌کند و نه همه‌ی n عنصر را؟ بهترین و بدترین حالت را برای این الگوریتم به‌دست آورید.

پاسخ: زیرا با هر بار اجرای حلقه for عدد بزرگ‌تر به سمت خانه nاُم و در نهایت در خانه nاُم قرار می‌گیرد. به همین دلیل در خانه آخر عدد بزرگ‌تر قرار دارد و آن خانه مورد بررسی قرار نمی‌گیرد.

۳- جستجوی خطی را در نظر بگیرید. چند عنصر در حالت میانگین باید چِک شود تا عنصر موردنظر پیدا شود؟ (فرض کنید که عنصر در آرایه موجود است.) در بدترین حالت چطور؟ حالت میانگین و بدترین حالت را به شکل θ نمایش دهید. (منظور از θ این است که ضرایب ثابت و جملات دوم به بعد را حذف کنید)

پاسخ: در بهترین حالت یک بار ($\theta = 1$) و در بدترین حالت و در حالت میانگین n بار ($\theta = n$)

۴- چطور می‌توان هر الگوریتمی را طوری تغییر داد که بهترین زمان اجرای خوبی داشته باشیم؟

---

[1] Selection Sort

نرخ رشد توابع

۳-۱- نرخ رشد توابع:

ترتیب یا نرخ رشد زمان اجرای یک الگوریتم که در فصل ۲ تعریف شد، یک ویژگی[1]ساده برای بیان کارایی الگوریتم است و به ما اجازه می دهد که کارایی نسبی الگوریتم‌های جایگزین[2]با الگوریتم مدنظرمان را با هم مقایسه کنیم.

اگر اندازه ورودی n به اندازه کافی بزرگ باشد، زمان اجرای بدترین حالت مرتب‌سازی ادغامی $\Theta(n \lg n)$ است؛ بنابراین، رقابت را نسبت به الگوریتم مرتب‌سازی درجی که زمان اجرای بدترین حالت آن $\Theta(n^2)$ است، خواهد برد. هرچند گاهی اوقات می‌توان به‌طور دقیق[3]زمان اجرای یک الگوریتم را مشخص کرد؛ (همان‌طور که در مورد الگوریتم مرتب‌سازی درجی در فصل۲ این کار را کردیم) اما محاسبه دقیق زمان اجرا آنقدر زحمت لازم دارد که معمولاً ارزشش را ندارد.

اگر اندازه‌ی ورودی خیلی بزرگ باشد، دیگر ضرایب و جملات دوم به بعد چندجمله‌ای آن‌قدر نسبت به جمله‌ی اول ناچیز خواهد بود که در نظر گرفته نمی‌شود؛ بنابراین در این مواقع ما «*کارایی تقریبی*»[4] الگوریتم‌ها را مورد مطالعه قرار می‌دهیم و معمولاً الگوریتمی که کارایی تقریبی آن بهتر باشد، برای همه نوع ورودی‌ها (*به جز ورودی‌های خیلی کوچک*[5]) گزینه مناسبی خواهد بود.

این فصل روش‌های استاندارد مختلفی برای ساده‌سازی تحلیل تقریبی الگوریتم‌ها ارائه می‌کند. در ادامه نگاهی به «*نمادهای تقریبی*»[6] خواهیم داشت.

---

[1] characterization
[2] Alternative
[3] Precision -
[4] Asymptotic Efficiency
[5] Very Small Inputs
[6] Asymptotic Notation

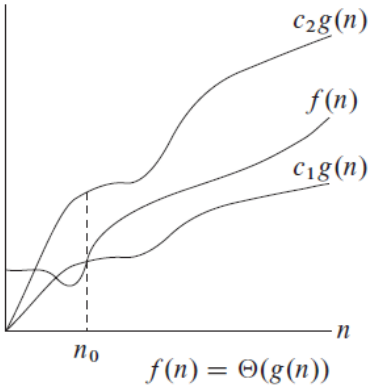
2-3- نمادهای تقریبی (یا نمادهای همبستگی یا نمادهای مُجانبی):

این نمادها برای توصیف زمان اجرای تقریبی یک الگوریتم که در قالب یک تابع تعریف شده و بر روی *اعداد طبیعی*[1] $N = \{ 0, 1, 2, \dots \}$ عمل می‌کند، استفاده می‌شود.

۳-۳- **θ –Notation**:

برای تابع فرضی g(n)، نمادِ $\boldsymbol{\theta}$ ($\boldsymbol{\theta}$(g(n))) به صورت زیر تعریف می‌شود:

$$\theta\big(g(n)\big) = \mathrm{f}(n): \exists\, c_1, c_2, n_0 > 0 : 0 < c_1\, g(n) < f(n) < c_2\, g(n), \forall_n > n_0$$



(a)

این رابطه به این معناست که آهنگ رشد f و g برای مقادیر بزرگ n یکسان است و هیچ‌یک از این دو تابع از دیگری جلو نمی‌زند؛ به‌طور مثال زمان اجرای مرتب‌سازی درجی $an^2 + bn + c$ است بنابراین: $T(n) = \theta(n^2)$. زیرا می‌توان به‌سادگی مقادیر مثبتی برای c_1 و c_2 یافت که:

$$c_1\, n^2 < an^2 + bn + c < c_2 n^2$$

مثال: ثابت کنید $100n^2 + 5n - 4 \neq \theta(n^3)$.

اثبات: باید نشان داد که نمی‌توان مقادیر مثبتی برای c_1 , c_2 و n_0 یافت که رابطه‌ی

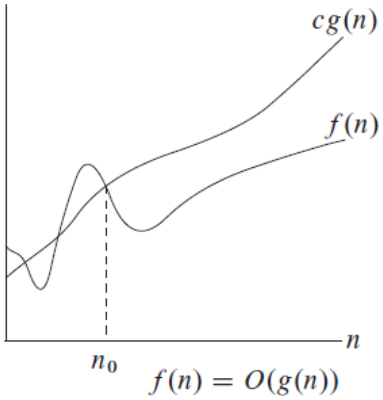
$c_1 n^3 < 100n^2 + 5n - 4 < c_2 n^3$ برای همه مقادیر $n > n_0$ برقرار باشد، به وضوح به‌ازای هر مقدار $c_1 > 0$ و برای مقادیر بزرگ n خواهیم داشت: ($c_1 n^3 \not\leq 100n^2 + 5n - 4$).

[1] Natural Numbers

۳-۴- O –Notation :

نماد $\boldsymbol{\theta}$ به صورت تقریبی، یک تابع را از بالا به پایین محدود[1] می‌کند. وقتی ما فقط یک «*حد بالای تقریبی*[2]» داشته باشیم، از نماد O استفاده می‌کنیم. برای تابع فرضی g(n)، نماد O (O(g(n))) را (که به صورت *بیگ اُ جی اِن*[3] یا O(g(n)) تلفظ می‌شود)، به صورت زیر تعریف می‌شود:

$$\mathrm{O}\big(g(n)\big) = \mathrm{f}(n): \exists\, c_1 n_0 > 0 : 0 < f(n) < \mathrm{c}\, g(n), \forall_n > n_0$$



(b)

یعنی آهنگ رشد تابع g(n) برای مقادیر بزرگ n، بیشتر یا مساوی آهنگ رشد تابع f(n) است، در این صورت می گوییم g(n) *کران بالای مجانبی*[4] برای f(n) است؛ به‌طور مثال:

$$n^2 = O(n^2) \text{ یا } \mathrm{n}log\, n = O(n^2).$$

نکته: منظور از f = O(g(n)) در واقع f ∈ O(g(n)) است.

۳-۵- Ω –Notation :

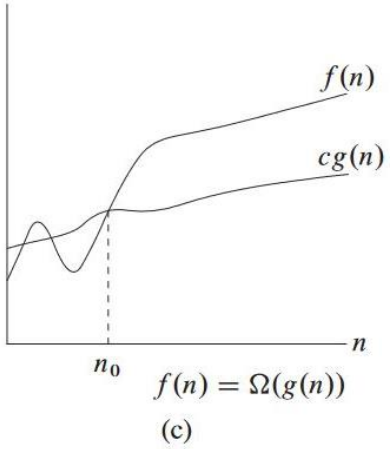
همان‌طورکه نماد O حد بالای تقریبی یک تابع را ارائه می‌کند، نماد Ω حد پایین تقریبی یک تابع را ارائه می‌کند. برای تابع فرضی g(n)، نماد Ω (Ω (g(n))) Ω را که به صورت *بیگ اُمگای جی اِن*[5] یا Ω (g(n))

---

[1] Bounds
[2] Asymptotic Upper bound
[3] "Big-O of g of n"
[4] حد بالای تقریبی
[5] "Big-Omega of g of n"

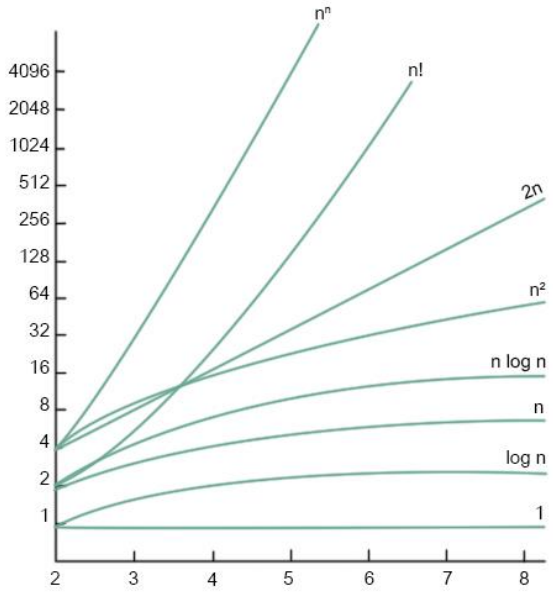
تلفظ می‌شود. تعریف ریاضیاتی این علامت به شکل زیر می‌باشد که در آن تابع g(n) کران پایین مجانبی برای تابع f(n) است.

$$\Omega\big(g(n)\big) = \mathrm{f}(n)\colon \exists\, c_1 n_0 > 0 : 0 < \mathrm{c}\, g(n) < f(n), \forall_n > n_0$$



برای مثال: $n^3 = \Omega(n^2)$ یا و $\sqrt{\boldsymbol{n}} = \Omega(log\, n)$

۶-۳- ترتیب رشد توابع مختلف:



نکته مهم: به زبان ساده، اگر زمان تقریبی دقیق اجرای یک الگوریتم خواسته شد، یعنی $\boldsymbol{\theta}$ و اگر حداکثر زمان تقریبی خواسته شد، یعنی O و اگر حداقل زمان تقریبی خواسته شد، یعنی **Ω**.

۳-۷- تمرین:

۱- فرض کنید $f(n)$ و $g(n)$ توابع غیرمنفی تقریبی باشند، با استفاده از تعریف نماد $\boldsymbol{\theta}$ ثابت کنید:

$$\text{Max}(f(n), g(n)) = \boldsymbol{\theta}(f(n) + g(n))$$

2- ثابت کنید برای هر مقدار ثابت a و b برای هر مقدار داریم:

$$\mathbf{(n + a)^b = \theta(n^b)}$$

3- ثابت کنید که این جمله غلط است: «زمان اجرای الگوریتم A حداقل $O(n^2)$ است».

4- آیا عبارات زیر درست است؟

الف: $2^{n+1} = O(2^n)$

ب: $2^{2n} = O(2^n)$

۳-۸- o –Notation[۱]:

همان‌طورکه گفته شد نماد O (Big o)[۲] ممکن است چندان به تابع اصلی نزدیک نباشد؛ برای مثال $2n^2 = O(n^2)$ به تابع اصلی نزدیک است امّا $2n = O(n^2)$، n^2 به $2n$ چندان نزدیک نیست. برای نمایش حد بالایی که به تابع موردنظر نچسبیده است، از نماد o (Little o) استفاده می‌کنیم و آن را به صورت زیر تعریف می‌کنیم:

$$o(g(n)) = \{ f(n) : for\ any\ positive\ constant\ c > 0,$$
$$there\ exists\ a\ constant$$
$$n_0 > 0\ such\ that\ 0 \leq f(n) < cg(n)\ for\ all\ n \geq n_0 \}.$$

برای مثال o کوچک $2n$ برابر است با $2n = o(n^2)$ که این‌طور می خوانیم: $2n$، اُردرِ n^2 است.

۳-۹- ω –Notation[۳]:

ω کوچک نسبت به Ω بزرگ همان نسبت o کوچک به O بزرگ است؛ یعنی نماد ω کوچک حد پایینی را مشخص می‌کند که به تابع مدنظر نچسبیده است؛ برای مثال اُمگای مربوط به $\frac{n^2}{2}$ برابر است با n؛

ω را در قالب ریاضی به صورت زیر تعریف می‌کنیم:

$$\omega(g(n)) = \{ f(n) : for\ any\ positive\ constant\ c > 0,$$
$$there\ exists\ a\ constant$$

---

[۱] اُ کوچک

[۲] اُ بزرگ

[۳] اُمگای کوچک

$n_0 > 0 \; such \; that \; 0 \leq cg(n) < f(n) \; for \; all \; n \geq \; n_0 \}.$

مثال: اگر $g(n) = 2n^2 + n + 10$ داشته باشیم، نماد θ و Ω و O (طبق شکل ترتیب رشد الگوریتم) به صورت زیر است:

$2n^2 + n + 10 = \theta(n^2)$
$2n^2 + n + 10 = \Omega(n \; logn)$
$2n^2 + n + 10 = \omega \; (n)$
$2n^2 + n + 10 = O(2^n)$
$2n^2 + n + 10 = o(n!)$

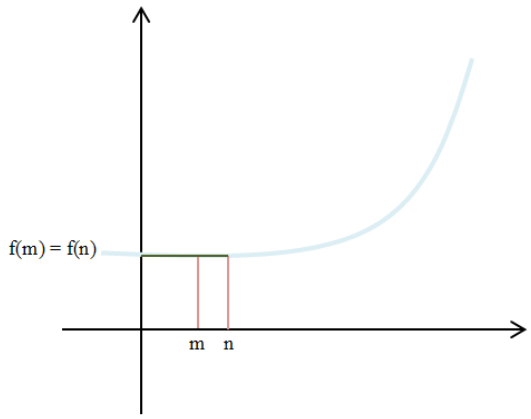
۳-۱۰ نمادهای استاندارد و توابع عمومی:

این بخش مروری دارد به برخی توابع و نمادهای ریاضی استاندارد و روابط بین آن‌ها را کاوش می‌کند و همچنین کاربرد نمادهای تقریبی را مشخص می‌کند.

۳-۱۱ یکنواختی[1]:

تابع f(n) را دارای «*افزایش یکنواخت*[2]» می‌دانیم اگر n و mی داشته باشیم که:

$m \leq n \; و \; f(m) \leq f(n)$



تابع f(n) را دارای «*کاهش یکنواخت*[3]» می‌دانیم اگر n و mی داشته باشیم که:

$m \leq n \; و \; f(m) \geq f(n)$

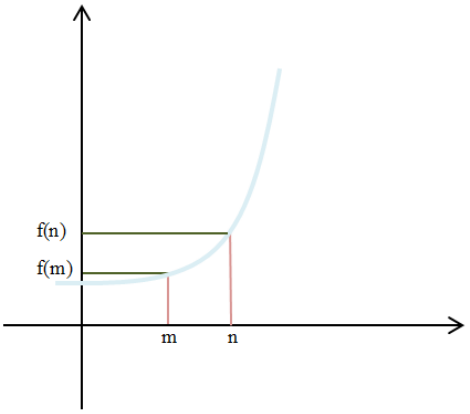
تابع f(n) را «*به شدت افزایشی*[1]» می‌دانیم، اگر n و m به صورت $f(m) < f(n)$ و باشد $m < n$.

---

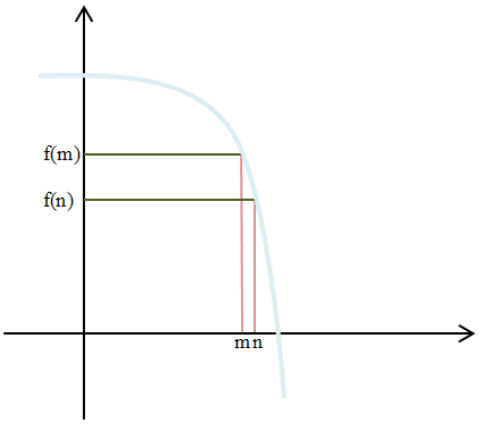
[1] Monotonicity

[2] Monotonically Increasing

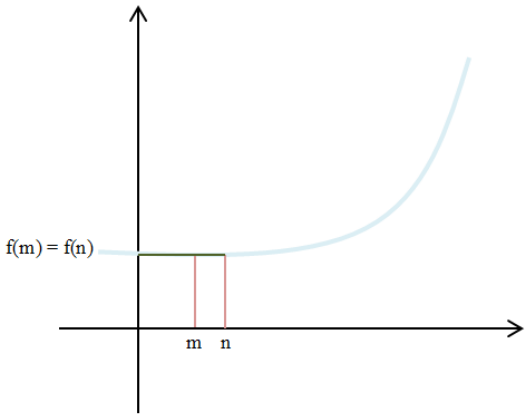
[3] Monotonically Decreasing



تابع f(n) را «*به شدت کاهشی*[2] می دانیم، اگر n و m به صورت $f(m) > f(n)$ و باشد $m < n$.



برای مثال شکل زیر تابع افزایشی است، ولی به شدت افزایشی نیست چراکه می‌توان m و nی پیدا کرد که $f(m) = f(n)$ ولی $m < n$.



[1] Strictly Increasing
[2] Strictly Decreasing

۱۲-۳- باقیمانده[1]

برای هر عدد صحیح a و هر عدد صحیح مثبت n مُد $\frac{a}{n}$ یا باقیمانده تقسیم a بر n به صورت زیر نمایش داده می‌شود[2]

$$a \bmod n = a - n \lfloor \frac{a}{n} \rfloor$$

یعنی a منهای n در کفِ a بر n (منظور از «کف»، حذف بخش اعشاری یک عدد است)

مثال:

$$10 \bmod 3 = 10 - 3 \lfloor \frac{10}{3} \rfloor = 10 - 9 = 1$$

۱۳-۳- تمرین

۱- نشان دهید اگر f(n) و g(n) توبع افزایشی باشند، توابع $f(n) + g(n)$ و f(g(n)) نیز افزایشی است و اگر f(n) و g(n) علاوه‌بر افزایشی بودن منفی نباشند f(n) در g(n) یعنی $f(n).g(n)$ افزایشی است.

۲- ثابت کنید که:

الف) $n! = o(n^n)$

ب) $n! = \omega(2^n)$

۳- برنامه‌ای بنویسید که دو ماتریس ۲*۲ را از ورودی دریافت کند و ماتریس ضرب آن‌ها را نمایش دهد. بررسی کنید که الگوریتم Strassen چطور نتیجه را به دست می‌آورد.

---

[1] Modular Arithmetic

[2] ⌊⌋ : علامت سقف و ⌊⌋ : علامت کف می‌باشند.

فصل چهارم

الگوریتم‌های بازگشتی

۴-۱ الگوریتم بازگشتی[1]:

اگر یک الگوریتم در حین اجرا، خودش[2]را فراخوانی[3]کند به این الگوریتم، الگوریتم بازگشتی گفته می‌شود.

الگوریتم‌های بازگشتی از پرکاربردترین و گاهی پرهزینه‌ترین الگوریتم‌ها در پیاده‌سازی راه حل مسائل هستند.

۴-۲ مثالی از الگوریتم بازگشتی: برنامه‌ی محاسبه فاکتوریل:

یک عدد از ورودی بگیرید و به صورت بازگشتی فاکتوریل آن را حساب کنید.

```
#include<iostream>
using namespace std;

int fact(int n)
{
    if(n==1)
        return 1;
    else
    {
        return n*fact(n-1);
    }
}

int main()
{
    int n;
    cout<<"Please enter a number: ";
    cin>>n;
    cout<<n<<"!="<<fact(n);
}
```

۴-۳ جستجوی دودویی به صورت بازگشتی:

---

[1] Recursive Algorithm

[2] Itself

[3] Call

```c
#include <stdio.h>
int binarySearch(int arr[], int l, int r, int x)
{
   if (r >= l)
   {
        int mid = l + (r - l)/2;
        if (arr[mid] == x)  return mid;
        if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);
        return binarySearch(arr, mid+1, r, x);
   }
   return -1;
}
int main(void)
{
   int arr[] = {2, 3, 4, 10, 40};
   int n = sizeof(arr)/ sizeof(arr[0]);
   int x = 10;
   int result = binarySearch(arr, 0, n-1, x);
   (result == -1)? printf("Element is not present in array")
                 : printf("Element is present at index %d", result);
   return 0;
}
```

۴-۴- اعداد فیبوناچی[1]

این اعداد به صورت زیر تعریف می‌شوند:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_i = f_{i+1} + f_{i-2} \qquad for\ i \geq 2$$

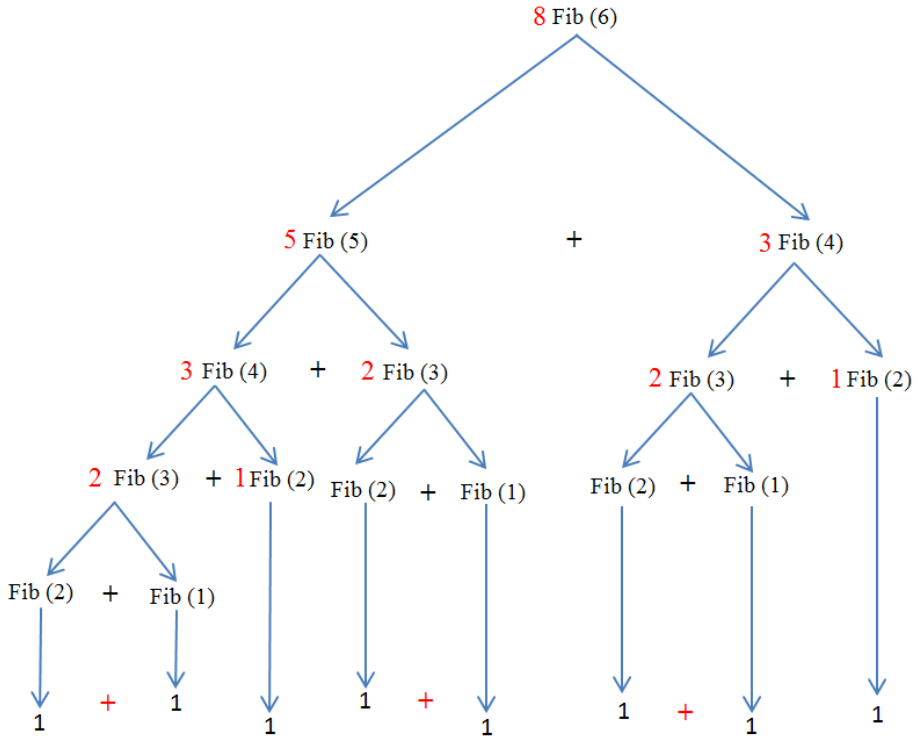
بنابراین توالی زیر تولید خواهد شد:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

به دست آوردن جمله nام از دنباله فیبوناچی یک مسأله بازگشتی مشهور است.

به طور مثال شکل محاسبه جمله هفتم این توالی یعنی عدد ۸، به صورت زیر است:

[1] Fibonacci Numbers



این مسأله به سه روش قابل حل است:

1 - به صورت تکراری (بدون استفاده از الگوریتم بازگشتی):

```cpp
#include<iostream>
using namespace std;
int fib(int n)
{
    int prev = -1;
    int result = 1;
    int sum;
    int i;
    for(i = 0;i <= n;i++)
    {
        sum = result + prev;
        prev = result;
        result = sum;
    }
    return result;
}
int main()
{
    int i;
    int n;
    cout<<"Please enter step number: ";
    cin>>n;
    for(i = 0;i <= n;++i)
    {
        cout<<fib(i)<<"\t";
    }
    return 0;
}
```

2 - به صورت بازگشتی:

```c
#include<stdio.h>
int fib(int n)
{
    if(n<=1)
    return n;
    return fib(n-1)+fib(n-2);
}
int main()
{
    int n=10;
    printf("%d",fib(n));
    getchar();
    return 0;
}
```

3 - با استفاده از برنامه‌نویسی پویا: این روش در فصل‌های آینده توضیح داده خواهد شد.

۵-۴- مطالعه آزاد: نسبت طلایی[1]

اعداد فیبوناچی مربوط به نسبت طلایی φ و مزدوج آن $\hat{\varphi}$ هستند که دو ریشه‌ی معادله‌ی $x^2 = x^2 + 1$ می‌باشند.

نسبت طلایی یا عدد φ در ریاضیات و هنر هنگامی رخ می‌دهد که: «نسبت بخش بزرگ‌تر به بخش کوچک‌تر برابر با نسبت کل به بخش بزرگ‌تر» باشد.

تعریف دیگر آن از این قرار است که: «عددی مثبت است که اگر به آن یک واحد اضافه کنیم به مربع آن خواهیم رسید».

تعریف هندسی آن نیز چنین است: «طول مستطیلی به مساحت واحد که عرض آن یک واحد کم‌تر از طولش باشد» و آن را به شکل زیر نمایش می‌دهیم:

$$\frac{a+b}{a} = \frac{a}{b} = \varphi$$

که a همان طول و b همان عرض مستطیل است و در نهایت φ را برابر عدد زیر در نظر می‌گیریم:

$$\varphi \simeq 1/6180339887$$

روانشناسان بر این باورند که زیباترین مستطیل به دید انسان مستطیلی است که نسبت طول به عرض آن برابر عدد طلایی باشد. دلیل این امر آن است که این نسبت در شبکیه‌ی چشم انسان رعایت شده و هر مستطیلی که این نسبت را دارا باشد به چشم انسان زیبا می‌آید.

---

[1] Golden Ratio

تکنیک‌های طراحی الگوریتم (۱): روش تقسیم و غلبه

۵-۱ طراحی الگوریتم‌ها:

تکنیک‌های طراحی الگوریتم بسیار زیاد هستند. در فصل‌های قبل با تکنیک «افزایشی»[1]در طراحی الگوریتم مرتب‌سازی درجی آشنا شدیم. در این بخش با یک روش[2]طراحی دیگر به نام «تقسیم و غلبه» آشنا می‌شویم و از این روش برای طراحی یک الگوریتم مرتب‌سازی استفاده می‌کنیم که بدترین زمان اجرای آن بسیار کمتر از مرتب‌سازی درجی است.

۵-۲ روش تقسیم و غلبه[3]

بسیاری از الگوریتم‌های مفید در ساختار خود از حالت بازگشتی استفاده می‌کنند؛ یعنی برای حل یک مسأله‌ی فرضی، آن ساختار دائماً خودش را فراخوانی می‌کند تا به حد کافی مسأله کوچک شده و قابل حل شود. این فراخوانی ممکن است یک یا چندین‌بار اتفاق بیفتد. این نوع الگوریتم‌ها نوعاً از روشی استفاده می‌کنند که به تقسیم و غلبه مشهور است؛ یعنی مسأله را به چندین زیرمسأله می‌شکند[4]که آن زیرمسأله شبیه *مسأله‌ی اصلی*[5]در اندازه‌ی کوچکتری است، سپس زیرمسأله را حل می‌کند و پاسخ زیرمسأله‌ها را با هم ترکیب می‌کند تا به حل مسأله‌ی اصلی بینجامد.

روش[6]تقسیم و غلبه با سه گام درگیر است:

۱. **مرحله تقسیم:**[7]مسأله به تعدادی مسأله کوچک‌تر شکسته می‌شود.
۲. **مرحله غلبه:**[1]زیرمسأله‌ها از طریق تقسیم‌های بازگشتی خود را حل می‌کنند، اگر یک زیرمسأله به اندازه‌ی کافی کوچک شده باشد بدون تقسیم اضافه حل می‌شود.

---

[1] Incremental
[2] Approach
[3] Divide-and-conquer
[4] Break
[5] Original problem
[6] Paradigm
[7] Divide

۳. **مرحله ترکیب**:[2] پاسخ زیرمسأله با هم ترکیب می‌شود تا مسأله‌ی اصلی حل شود.

مثال: الگوریتم مرتب‌سازی ادغامی از روش تقسیم و غلبه استفاده می‌کند؛ یعنی داریم:

۴. مرحله تقسیم: توالی n عنصر را به *دو زیر توالی*[3] با $\frac{n}{2}$ عنصر تقسیم می‌کند.

۵. مرحله غلبه: دو زیر توالی را با استفاده از مرتب‌سازی ادغامی مرتب می‌کند.

۶. مرحله ترکیب: دو زیر توالی مرتب شده را با هم ادغام می‌کند تا پاسخ مرتب شده‌ی نهایی را تولید کند.

عملیات تقسیم آن‌قدر ادامه پیدا می‌کند تا تعداد عناصری که باید مرتب شوند ۱ باشد و البتّه طبیعتاً یک عنصر نیازی به مرتب‌سازی ندارد؛ بنابراین اصل مرتب‌سازی زمانی اتفاق می‌افتد که دو عنصر با هم در مرحله بعد ترکیب می‌شوند. مرتب‌سازی ادغامی را با تابع $MERGE\ (A,p,q,r)$ انجام می‌دهیم که در آن، A آرایه‌ای است که باید مرتب شود و p,q,r اندیس‌های آرایه هستند به‌طوری‌که $p \leq q < r$.

تابع فرض می‌کند زیر آرایه $A\ [\ p \ldots q\ ]$ و $A\ [\ q+1 \ldots r\ ]$ مرتب شده هستند. تابع آن‌ها را ادغام می‌کند تا زیرآرایه‌ی $A\ [\ p \ldots r\ ]$ به دست آید.

تابع MERGE زمان $\theta\ (n)$ را صرف می‌کند که در آن $n = r - p + 1$ و در حقیقت تعداد عناصری است که باید مرتب شوند.

شبه‌کد این تابع به صورت زیر است:

---

[1] Conquer
[2] Combine
[3] Subsequence

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 . . n1 + 1] and R[1 . . n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

و کد **C++** این تابع به صورت زیر است:

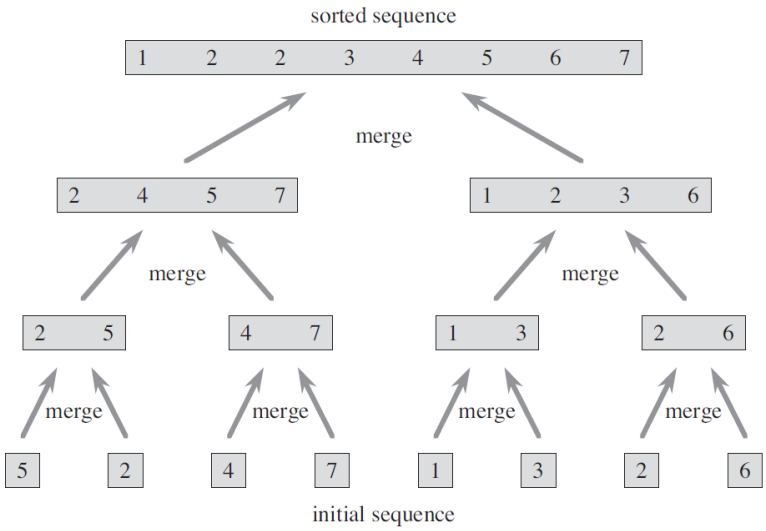
```cpp
#include<stdlib.h>
#include<stdio.h>
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }

        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l+(r-l)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);
        merge(arr, l, m, r);
    }
}
void printArray(int A[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    printf("Given array is \n");
    printArray(arr, arr_size);
    mergeSort(arr, 0, arr_size - 1);
    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}
```

مرجع کدها: سایت http://www.geeksforgeeks.org/merge-sort/

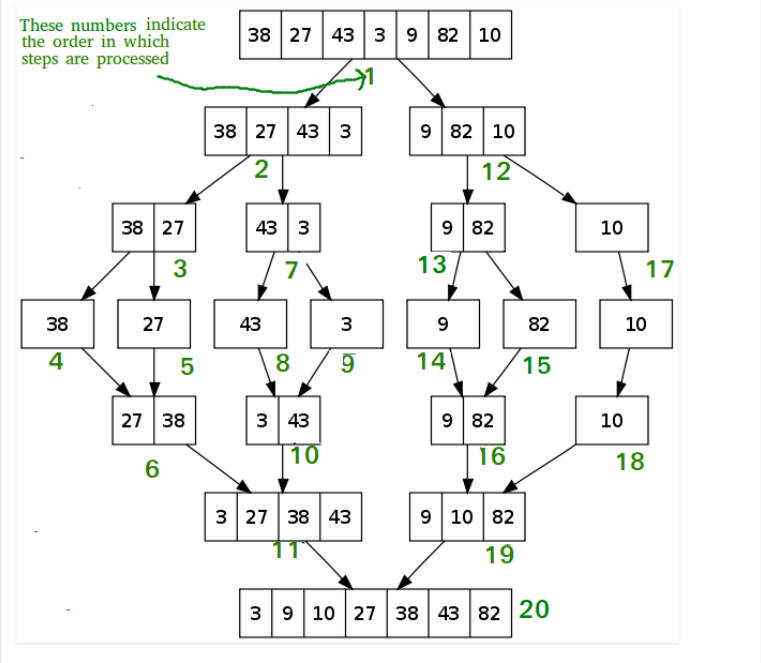
مثال: آرایه‌ی A *با عناصر* ۳۸, ۲۷, ۴۳, ۳, ۹, ۸۲, ۱۰ را در نظر بگیرید. با الگوریتم مرتب‌سازی ادغامی آن را مرتب کنید.

راهنمایی: m را از طریق فرمول $m = int\ \frac{(n-1)}{2}$ به دست آورید.

مثال: آرایه‌ی A *با عناصر* 5,2,4,7,1,3,2,6 را در نظر بگیرید. شکل کلی مرتب‌سازی آرایه A را با کمک الگوریتم مرتب‌سازی ادغامی رسم کنید.



مثال ۲: آرایه‌ی A *با عناصر* 38, 27, 43, 3, 9, 82, 10 را در نظر بگیرید. با الگوریتم مرتب‌سازی ادغامی آن را مرتب کنید.



آرایه‌ی A با عناصر{0, 10, 38, 7, 43, 8, 9, 1, 2, 10} را در نظر بگیرید. شکل کلی مرتب‌سازی آرایه A را با کمک الگوریتم مرتب‌سازی ادغامی رسم کنید.

۳-۵- جستجوی دودویی:

در این جستجو یک آرایه مرتب داریم که در آن به دنبال یک عنصر می‌گردیم، برای یافتن آن ابتدا آرایه را به دو قسمت تقسیم می‌کنیم و عنصر مورد جستجو را با عنصر وسط آرایه مقایسه می‌کنیم، اگر عنصر مورد جستجو بزرگ‌تر از عنصر وسط آرایه بود، تکه‌ی دوم آرایه را به همین روش تا یافتن عنصر موردنظر به دو قسمت تقسیم و مقایسه می‌کنیم.

مثال: با توجه به قطعه کد زیر که تابع جستجوی دودویی است، در آرایه زیر به دنبال عدد 30 بگردید. جدول تغییرات متغیرها را رسم کنید.

```c
#include <stdio.h>
int binarySearch(int arr[], int l, int r, int x)
{
  while (l <= r)
  {
    int m = l + (r-l)/2;
    if (arr[m] == x)
        return m;
    if (arr[m] < x)
        l = m + 1;
    else
         r = m - 1;
  }
  return -1;
}
int main(void)
{
   int arr[] = {1, 5, 8, 20, 21, 30, 40};
   int n = sizeof(arr)/ sizeof(arr[0]);
   int x = 10;
   int result = binarySearch(arr, 0, n-1, x);
   (result == -1)? printf("Element is not present in array")
                 : printf("Element is present at index %d", result);
   return 0;
}
```

جدول تغییرات متغیرها:

Start	Stop	Q	Mid	Result
0	6	30	3	
3	6		4	
4	6			
			5ar	5

تمرین: آرایه ای با ۷ عنصر تعریف کرده و مقادیر آن را از ورودی بخوانید. سپس با مرتب‌سازی ادغامی آن را مرتب کرده و سپس یک عدد از کاربر گرفته و با جستجوی دودویی مشخص کنید آن عدد در کدام خانه از آرایه موجود است؟

فصل ششم

تکنیک‌های طراحی الگوریتم (۲): روش حافظه پویا

۶-۱ مقدمه

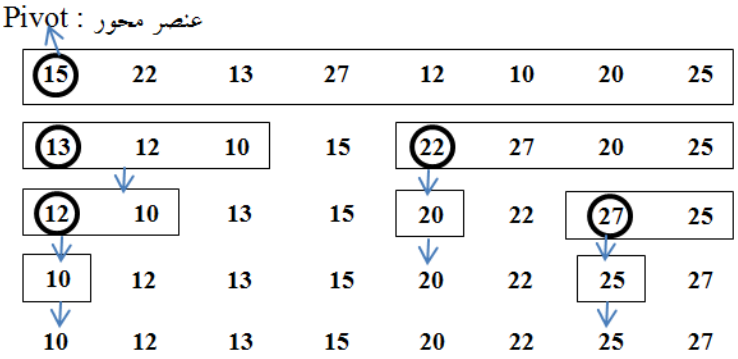
اگر به حل مسأله فیبوناچی به روش بازگشتی دقت کنیم، خواهید دید که اکثر جملات (مانند Fib(3)) چندین بار محاسبه می‌شوند، برای جلوگیری از این پردازش‌های تکراری می‌توان هر جمله‌ای که به دست آمد را در بخشی از حافظه ذخیره کرد و به محض نیاز به آن به جای محاسبه مجدد، از حافظه پاسخ را خواند. به این روش در اصطلاح حافظه پویا گفته می‌شود.

۶-۲ الگوریتم مرتب‌سازی سریع[1]

مثال: اعداد زیر را با Quick Sort مرتب کنید:

15, 22, 13, 27, 12, 10, 20, 25

روش حل:



این الگوریتم را با دو تابع Quick Sort و Partition به شکل زیر می‌توان پیاده‌سازی کرد:

تابع Quick Sort :

[1] Quick Sort

```
5   void quickSort(int arr[], int low, int high)
6   {
7       if (low < high)
8       {
9           int pi = partition(arr, low, high);
10          quickSort(arr, low, pi - 1);
11          quickSort(arr, pi + 1, high);
12      }
13  }
```

تابع Partition:

```
16  int partition (int arr[], int low, int high)
17  {
18      int pivot = arr[high];
19      int i = (low - 1);
20      for (int j = low; j <= high- 1; j++)
21      {
22          if (arr[j] <= pivot)
23          {
24              i++;
25              swap(&arr[i], &arr[j]);
26          }
27      }
28      swap(&arr[i + 1], &arr[high]);
29      return (i + 1);
30  }
```

۳-۶- مسأله ضرب ماتریس‌ها:

برای نگه‌داری عناصر یک ماتریس از یک آرایه دوبعدی و برای ضرب عناصر دو آرایه، از دو حلقه‌ی تودرتو استفاده می‌کنیم.

مثال: اگر arr1 و arr2 به صورت زیر داده شود، عناصر arr3 که ضرب دو ماتریس است، به صورت

زیر محاسبه می‌شوند:

$$arr1 = \begin{bmatrix} 10 & 5 & 3 \\ 7 & 8 & 9 \\ 1 & 2 & 3 \end{bmatrix}$$

$$arr2 = \begin{bmatrix} 5 & 1 \\ 1 & 2 \\ 0 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 10 & 5 & 3 \\ 7 & 8 & 9 \\ 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 5 & 1 \\ 1 & 2 \\ 0 & 3 \end{bmatrix} = \begin{bmatrix} 55 & 29 \\ 43 & 50 \\ 7 & 14 \end{bmatrix}$$

$$arr3 = \begin{bmatrix} 55 & 29 \\ 43 & 50 \\ 7 & 14 \end{bmatrix}$$

ضرب دو ماتریس با استفاده از سه حلقه تودرتو قابل محاسبه است که در کد زیر در خط ۴۶ تا ۵۶ مشخص شده است:

```cpp
#include<iostream>
using namespace std;
int main()
{
    int n1,n2,n3;
    cout<<"Please enter Matrix1 row number (n1) :";
    cin>>n1;
    cout<<"Please enter Matrix1 column number (n2) :";
    cin>>n2;
    cout<<"Please enter Matrix2 column number (n3) :";
    cin>>n3;
    int arr1[n1][n2];
    int arr2(n2][n3];
    cout<<"Please enter arr1 elements: ";
    for(int i=0; i<n1; i++)
    {
        for(int j=0; j<n2; j++)
        {
            cout<<"Please enter a number (arr2["<<i<<"]["<<j<<"]:
            cin>>arr2[i][j];
        }
    }
    int arr3[n1][n3];
    int sum=0;
    for(int i=0; i<n1; i++)
    {
        for(int j=0; j<n3; j++)
        {
            arr3[i][j]=0;
        }
    }
    for(int i=0; i<n1; i++)
    {
        for(int j=0; j<n3; j++)
        {
            for(int k=0; k<n2; k++)
            {
                arr3[i][j]+=arr1[i][k]*arr2[k][j];
            }
        }
    }
    for(int i=0; i<n1; i++)
    {
        for(int j=0; j<n3; j++)
        {
            cout<<arr3[i][j]<<"\t";
        }
        cout<<"\n";
    }
}
```

$$
\begin{aligned}
arr3[0][0] = {} & arr1[0][0] * arr2[0][0] + \\
& arr1[0][1] * arr2[1][0] + \\
& arr1[0][2] * arr2[2][0] + \\
arr3[0][1] = {} & arr1[0][0] * arr2[0][1] + \\
& arr1[0][1] * arr2[1][1] + \\
& arr1[0][2] * arr2[2][1] +
\end{aligned}
$$

در حالت کلی داریم:

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} b_{00} \\ b_{10} \end{bmatrix} = \begin{bmatrix} c_{00} \\ c_{10} \end{bmatrix}$$

$$c_{00} = (a_{00} * b_{00}) + (a_{01} * b_{10})$$

$$c_{10} = (a_{10} * b_{00}) + (a_{11} * b_{10})$$

۴-۶- الگوریتم ضرب Strasson :

همان‌طور که می‌دانیم برای کامپیوتر محاسبه ضرب سخت‌تر از محاسبه جمع است؛ بنابراین در حل یک مسأله اگر بتوان از تعداد ضرب‌ها کاست، آن الگوریتم بهینه‌تر خواهد بود.

الگوریتم Strasson ماتریس‌ها را به روشی ضرب می‌کند که تعداد ضرب‌ها به ویژه در ابعاد بالا بسیار کاهش پیدا می‌کند.

الگوریتم ضرب Strasson برای دو ماتریس ۲*۲ به شکل زیر خواهد بود:

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix}$$

$$m_0 = (a_{00} + a_{11})(b_{00} + b_{11})$$

$$m_1 = (a_{10} + a_{11})b_{00}$$

$$m_2 = a_{00}(b_{01} - b_{11})$$

$$m_3 = a_{11}(b_{10} - b_{00})$$

$$m_4 = (a_{00} + a_{01})b_{11}$$

$$m_5 = (a_{10} - a_{00})(b_{00} + b_{01})$$

$$m_6 = (a_{01} + a_{11})(b_{10} + b_{11})$$

ماتریس c از روابط بالا به این صورت حاصل می‌گردد:

$$c = \begin{bmatrix} m_0 + m_3 - m_4 + m_6 & m_2 + m_4 \\ m_1 + m_3 & m_0 + m_2 - m_1 + m_5 \end{bmatrix}$$

مثال:

$$\begin{bmatrix} 5 & 3 \\ 2 & 1 \end{bmatrix} * \begin{bmatrix} 4 & 2 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 23 & 16 \\ 9 & 6 \end{bmatrix}$$

$m_0 = (5 + 1)(4 + 2) = 6 * 6 = 36$
$m_1 = (2 + 1) * 4 = 3 * 4 = 12$
$m_2 = 5 * (2 - 2) = 5 * 0 = 0$
$m_3 = 1 * (1 - 4) = 1 * -3 = -3$
$m_4 = (5 + 3) * 2 = 8 * 2 = 16$
$m_5 = (2 - 5)(4 + 2) = -3 * 6 = -18$

$$m_6 = (3-1)*(1+2) = 2*3 = 6$$

در نتیجه ماتریس C خواهد بود:

$$c = \begin{bmatrix} 36+(-3)-16+6 & 0+16 \\ 12+(-3) & 36+0-12+(-18) \end{bmatrix} = \begin{bmatrix} 23 & 16 \\ 9 & 6 \end{bmatrix}$$

شبه‌کد الگوریتم Strasson :

```
void strasson (int n, n*n-matrixA, n*n-matrixB, n*n-matrixC)
{
     if(n<=threshold)
           compute C=A*B using the standard algorithm;
     else
     {
      partition A into 4 submatix A00, A01, A10, A11;
      partition B into 4 submatix B00, B01, B10, B11;
      compute C=A*B using strasson's method; // strasson(n/2, A00+A11, B00+B11, M0)
     }
}
```

نمونه سؤال: الگوریتم Strasson چیست و چگونه مسأله را حل می‌کند؟ از چه الگوریتمی برای حل مسأله‌اش استفاده کرده است؟ چه مزایایی نسبت به حالت عادی حل مسأله دارد؟

پاسخ: این لگوریتم روش جدیدی برای حل مسأله ضرب دو ماتریس n*n مربع است که با کاهش تعداد ضرب‌ها باعث می‌شود، پردازش‌ها کاهش یافته و سریع‌تر نتایج به دست آید. کاربرد عملی این الگوریتم در ضرب ماتریس‌های بزرگ است، هرچند که نسبت به الگوریتم‌های جدیدتر کمی کندتر است.

تمرین: یک الگوریتم جدیدتر و سریع‌تر نسبت به Strasson پیدا کنید.

۵-۶- ضرب اعداد بزرگ:

دو عدد u و v هر کدام n رقمی را در نظر بگیرید. برای محاسبه u*v می‌توان به روش زیر عمل کرد:

$$m = \frac{n}{2}$$

$$u = x * 10^m + y$$

$v = w * 10^m + z$

$u * v = (x * 10^m + y)(w * 10^m + z) = xw10^{2m} + (xz + yw) * 10^m + zy$

مثال: دو عدد ۱۰۲۴۳۶ و ۷۲۴۳۰۰ را با هم ضرب کنید.

$$m = \frac{n}{2} = \frac{6}{2} = 3$$

$u = 102436 = 102 * 10^3 + 436$, $v = 724300 = 724 * 10^3 + 300$

$uv = (102 * 724) * 10^6 + (102 * 300 + 436 * 724) * 10^3 + (300 * 436)$

۶-۶- برنامه‌نویسی با حافظه پویا[1]

برنامه‌نویسی پویا یک روش نوشتن الگوریتم‌هاست که در آن مسأله اصلی را به استفاده از یک فرمول بازگشتی حل می‌کنیم؛ یعنی در ابتدا برای مسأله، یک ضابطه بازگشتی می‌یابیم. سپس با استفاده از یک حافظه سعی بر آن داریم. که در ابتدا مسأله را برای مقادیر اولیه حل کنیم سپس با ترکیب حل‌های مقدماتی‌تر حل مسأله را در حالت بالاتر دنبال می‌نماییم. این روند را تا آنجا ادامه می‌دهیم که به پاسخی برای مسأله اصلی دست یابیم.

روش DP از آنجا ابداع گردید که بتواند راه‌حل‌های تکراری را که ممکن است در روش D&C (تقسیم و غلبه) به وجود آید را حذف نماید.

در اینجا راه‌حل، راه‌حلی Button-Up است.

۶-۷- محاسبه تراز چند دانشجو:

فرمول محاسبه تراز:

$$T = 1000z + 5000$$

فرمول محاسبه z:

$$z = \frac{x - \bar{x}}{s}$$

در این فرمول:

x نمره خام داوطلب در درس موردنظر است.

$\bar{x}$ میانگین نمرات کل داوطلبین در درس موردنظر است.

s همان انحراف از معیار یا Standard Deviation است.

---

[1] Dynamic Programming

فرمول محاسبه s:

$$s = \sqrt{\frac{\sum_{i=0}^{n}(x_i - \bar{x})^z}{N}}$$

N تعداد کل شرکت‌کنندگان است.

مثال: فرض کنید سه نفر در یک درس فرضی، نمره‌های ۱۰۰ و ۲۰۰ و ۳۰۰ را گرفته‌اند. تراز هر کدام از این سه نفر را حساب کنید.

$$n = 3\,, x_1 = 100\,, x_2 = 200\,, x_3 = 300$$

$$\bar{x} = \frac{100 + 200 + 300}{3} = 200$$

$$\sum_1^3 = \frac{(100 - 200\,)^2}{3} = \frac{10000}{3} = 3333.33$$

$$\sum_2^3 = \frac{(200 - 200\,)^2}{3} = \frac{0}{3} = 0$$

$$\sum_3^3 = \frac{(300 - 200\,)^2}{3} = \frac{10000}{3} = 3333.33$$

$$s = \sqrt{\frac{\sum_1^3(\,x_i - \,\bar{x})^2}{3}} = \sqrt{3333.33 + 3333.33\,} = 81.65$$

$$z_1 = \frac{x_1 - \,\bar{x}}{s} = \frac{100 - 200}{81.65} = -1.22$$

$$z_2 = \frac{x_2 - \,\bar{x}}{s} = \frac{200 - 200}{81.65} = 0$$

$$z_3 = \frac{x_3 - \bar{x}}{s} = \frac{300 - 200}{81.65} = 1.22$$

$$T_1 = 1000(\,-1.22) + 5000 = 3780$$

$$T_2 = 1000(\,0\,) + 5000 = 5000$$

$$T_3 = 1000(1.22) + 5000 = 6220$$

اگر به محاسبات بالا دقت کنید؛ برای هر نفر به صورت تکراری $\bar{x}$ و s مجددا محاسبه شده است. این درحالی است که محاسبه‌ی هر یک از این‌ها در آمار بالا پردازش بالایی نیاز دارد. برای رفع این مشکل می‌توان از برنامه نویسی پویا استفاده کنیم؛ به این صورت که وقتی برای نفر اول $\bar{x}$ و s محاسبه شد، در بخشی از حافظه نگه داشته شود و برای نفرات بعد از همان داده‌ها استفاده شود.

۸-۶- مسأله انتخاب k شیئ از n شیئ (محاسبه $\binom{n}{k}$):

فرض کنید در یک کیسه ۴ گوی با رنگ‌های مختلف (سیاه، سفید، قرمز و آبی) وجود دارد، به چند حالت می‌توان ۲ گوی از کیسه خارج کرد؟

فرمول:

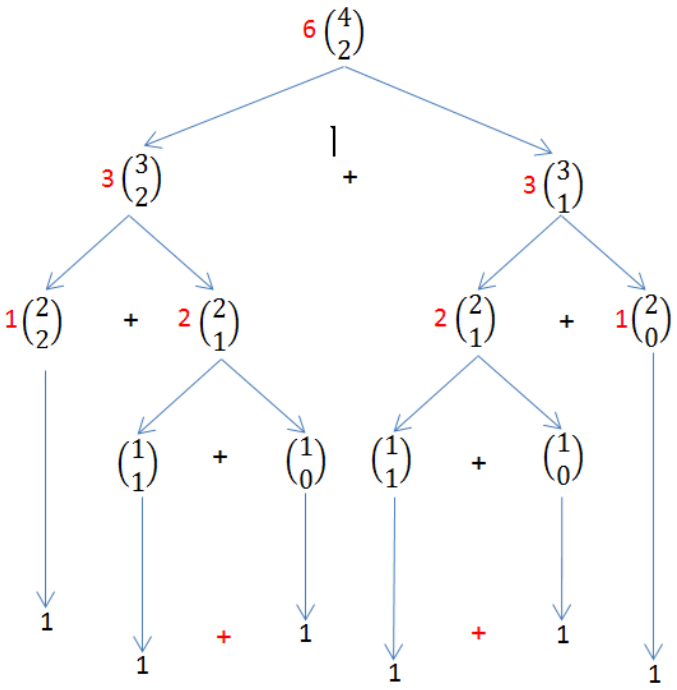
$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

$$\binom{4}{2} = \frac{4!}{(4-2)!\,2!} = \frac{4*3*2!}{2*1*2!} = \frac{4*3}{2*1} = 6$$

برای نوشتن برنامه مسأله ثابت می‌کنیم که:

$$\binom{n}{k} = \begin{cases} 1 & k = 0 \; or \; k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & else \end{cases}$$

با مثال زیر ثابت می‌شود که این رابطه صحیح است، در نتیجه با یک تابع بازگشتی می‌توان مسأله را پیاده‌سازی کرد.



اما بدون استفاده از برنامه‌نویسی پویا، مسأله، بهینه نخواهد بود، چرا که برخی محاسبات چندین بار انجام شده است. برای بهینه کردن این مسأله، برنامه‌نویسی پویا را به شکل زیر پیاده‌سازی می‌کنیم:

آرایه‌ای به شکل زیر ایجاد می‌کنیم که مقادیر به دست آمده از پردازش‌های تکراری در آن ذخیره می‌شود و هرگاه به پاسخ آن پردازش نیاز بود، از آرایه خوانده و استفاده می‌کنیم.

$\binom{n}{k}$	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	...	1		
5	1	5	⋮			1	
6	1	6					

پیاده‌سازی مسأله بدون استفاده از برنامه‌نویسی پویا:

```cpp
#include<iostream>
using namespace std;
int comb(int n, int k)
{
    if(n==k || k==0)
        return 1;
    return comb(n-1,k-1)+comb(n-1,k);
}
int main()
{
    cout<<comb(8,3);
    return 0;
}
```

پیاده‌سازی مسأله با استفاده از برنامه‌نویسی پویا:

```cpp
#include<iostream>
using namespace std;
long long int c[1000][1000];
long long int comb(int n, int k)
{
    if(n==k || k==0)
        return 1;
    else
    {
        if(c[n][k]==-1)
        {
            long long int x=comb(n-1,k-1)+comb(n-1,k);
            c[n][k]=x;
        }
        else
            return c[n][k];
    }
}
int main()
{
    for(int i=0;i<1000;i++)
        for(int j=0;j<1000;j++)
            c[i][j]=-1;
    cout<< comb(1000,100);
    return 0;
}
```

۹-۶- حل مسأله فیبوناچی با استفاده از برنامه‌نویسی پویا:

برای حل این مسأله یک آرایه به تعداد جمله خواسته شده (nتعریف می‌کنیم و یک بار برای همیشه جمع هر دو خانه را در خانه بعد از آن می‌ریزیم (یا جمع هر دو خانه را در خانه قبل آن‌ها می‌ریزیم)، با این‌کار آرایه‌ای به شکل زیر خواهیم داشت:

0	1	1	2	3	5	8	13	21	33	…

که به‌راحتی خانه nاُم آرایه که برابر با پاسخ مسأله هست بدون نیاز به محاسبهِ تکراریِ جمله‌هایِ قبل از آن به دست می‌آید.

```cpp
#include<stdio.h>
int fib(int n)
{
    long int f[n+1];
    long int i;
    f[0]=0;
    f[1]=1;
    for(i=2; i<=n; i++)
    {
        f[i]=f[i-1]+f[i-2];
    }
    return f[n];
}
int main()
{
    int n=1000;
    printf("%d",fib(n));
    getchar();
    return 0;
}
```

تمرین: برنامه‌ی فیبوناچی را با استفاده از بازگشتی و برنامه‌نویسی پویا نوشته و از لحاظ زمان اجرا با هم مقایسه کنید.

پاسخ: بازگشتی: با توجه به این‌که عملیات اصلی در تابع بازگشتی جمع fib(n – 1) + fib(n - 2) و به‌ازای n، T(n - 1) + T(n - 2) + O(1) بار اجرا می‌شود؛ بنابراین این روش $O(2^n)$ است.

در روش دوم: با توجه به این‌که عملیات اصلی f [i] = f [i - 1] + f [i - 2] است و این حلقه n*c بار تکرار می‌شود. این روش نیز O(n) است.

۱۰-۶- مسأله فلوید:

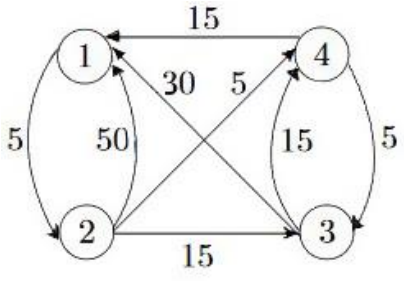
یکی از مهم‌ترین مسائل در دنیای امروز یافتن کوتاه‌ترین مسیر بین دو نقطه است؛ برای مثال در گراف جهت‌دار ابرکوتاه‌ترین مسیر بین دو روتر از الگوریتم‌های یافتن مسیر استفاده می‌شود.

الگوریتم‌های بسیار زیادی برای یافتن کوتاه‌ترین مسیر ارائه شده است که هر کدام از آن‌ها مزایا و معایبی دارد؛ برای مثال یکی پیاده‌سازی آسان ولی T(n) بالایی دارد و برعکس.

کاربرد فلوید در نرم‌افزارهایی مانند گوگل مپ و نرم‌افزارهای مشابه برای یافتن کوتاه‌ترین مسیر بین دو شهر است.

یکی از الگوریتم‌ها **فِلوید-وارشال** است. با یک مثال روش کار آن را توضیح می‌دهیم.

مثال: کوتاه‌ترین مسیر در گراف زیر را با روش فلوید به دست آورید.



گام ۱: ماتریس مجاورت (m) را برای همه‌ی گره‌ها به‌دست آورید. (ماتریس مجاورت، ماتریسی است که نشان می‌دهد از هر گره به گره دیگر یک مسیر مستقیم وجود دارد یا خیر؟ اگر وجود داشت، وزن یال نوشته

می‌شود و اگر وجود نداشت علامت ∞ و طبیعتا از هر گره به همان گره، وزن یال برابر با صفر خواهد بود.)
D_0 را برابر m بگیرید.

$$D_0 = m = \begin{bmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{bmatrix}$$

دقت کنید که سطر و ستون‌ها همان رأس‌ها هستند؛ یعنی:

$$D_0 = M = \begin{array}{c} \begin{matrix} V_1 & V_2 & V_3 & V_4 \end{matrix} \\ \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix} \end{array} \begin{matrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{matrix}$$

گام ۲: D_1 را طوری بسازید که مشخص‌کننده مینیمم مسیرهایی باشد که فقط از v_1 می‌گذرد و یا مسیر مستقیم هستند.

$$D_1 = \begin{bmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & 0 & \infty & \infty \\ 0 & 0 & 0 & 0 \\ 0 & v_1 & 0 & 0 \\ 0 & v_1 & 0 & 0 \end{bmatrix}$$

گام ۳: D_2 را طوری بسازید که شامل کوتاه‌ترین مسیرهایی باشد که از v_1 یا v_2 عبور می‌کنند و یا مستقیم هستند.

$$D_2 = \begin{bmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & 0 & v_2 & v_2 \\ 0 & 0 & 0 & 0 \\ 0 & v_1 & 0 & 0 \\ 0 & v_1 & 0 & 0 \end{bmatrix}$$

گام ۴: D_3 را طوری بسازید که شامل کوتاه‌ترین مسیرهایی باشد که از v_1 یا v_2 یا v_3 می‌گذرد و یا مستقیم است.

$$D_3 = \begin{bmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & 0 & v_2 & v_2 \\ v_3 & 0 & 0 & 0 \\ 0 & v_1 & 0 & 0 \\ 0 & v_1 & 0 & 0 \end{bmatrix}$$

گام ۵: D_4 را طوری بسازید که شامل کوتاه‌ترین مسیرهایی باشد که از v_1 یا v_2 یا v_3 یا v_4 عبور کند و یا مستقیم باشد.

$$D_4 = \begin{bmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & 0 & v_4 & v_2 \\ v_4 & 0 & v_4 & 0 \\ 0 & v_1 & 0 & 0 \\ 0 & v_1 & 0 & 0 \end{bmatrix}$$

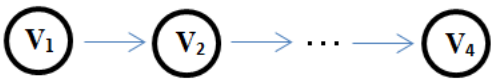
گام ۶: ماتریس P را در حین ساختن ماتریس‌های D به صورت زیر پر کنید. (همین‌طور که از D_1 به D_k پیش می‌روید)، وقتی کوتاه‌ترین مسیر به دست آمد، رأس جدیدی که باعث به وجود آمدن مسیر کوتاه‌تر شده را در P یادداشت کنید. اگر از یک رأس به رأس دیگر مسیر مستقیم وجود داشت و این مسیر کوتاه‌ترین مسیر بود در P، صفر قراردهید.

$$P = \begin{bmatrix} 0 & 0 & v_4 & v_2 \\ v_4 & 0 & v_4 & 0 \\ 0 & v_1 & 0 & 0 \\ 0 & v_1 & 0 & 0 \end{bmatrix}$$

گام آخر: برای به دست آوردن کوتاه‌ترین مسیر بین دو رأس به ماتریس P نگاه کنید. اگر بین آن دو رأس نام یک رأس دیگر نوشته شده، مسأله را برای یافتن کوتاه‌ترین مسیر بین آن رأس و ردس مبدأ تبدیل کنید و آن‌قدر این کار را تکرار کنید تا نهایتاً به صفر برسید و صفر به معنی این است که بین رأس یک مانده به آخر و رأس آخر مسیر مستقیمی وجود دارد که کوتاه‌ترین مسیر است.

مثال: کوتاه‌ترین مسیر بین v_1 و v_4 را به دست آورید.

پاسخ: طبق ماتریس P، v_1 به v_4 از v_2 می‌گذرد، پس داریم:



یعنی مسأله از یافتن v_1 به v_4 به یافتن کوتاه‌ترین مسیر از v_2 به v_4 تبدیل شد. مجدداً طبق ماتریس P خانه‌ی مربوط به v_2 به v_4 صفر است؛ یعنی از v_2 به v_4 مسیر مستقیمی وجود دارد که کوتاه‌ترین مسیر است، پس در نهایت کوتاه‌ترین مسیر بین v_1 و v_4 مسیر زیر خواهد بود:

الگوریتم فلوید با سه حلقه‌ی تو در تو به شکل زیر قابل پیاده‌سازی است:

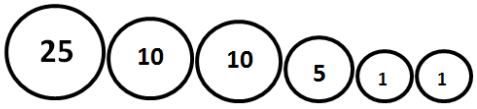
```cpp
#include <iostream>
#include <conio.h>
using namespace std;
void floyds(int b[][7])
{
    int i, j, k;
    for (k = 0; k < 7; k++)
    {
        for (i = 0; i < 7; i++)
        {
            for (j = 0; j < 7; j++)
            {
                if ((b[i][k] * b[k][j] != 0) && (i != j))
                {
                    if ((b[i][k] + b[k][j] < b[i][j]) || (b[i][j] == 0))
                    {
                        b[i][j] = b[i][k] + b[k][j];
                    }
                }
            }
        }
    }
    for (i = 0; i < 7; i++)
    {
        cout<<"\nMinimum Cost With Respect to Node:"<<i<<endl;
        for (j = 0; j < 7; j++)
        {
            cout<<b[i][j]<<"\t";
        }

    }
}
int main()
{
    int b[7][7];
    cout<<"ENTER VALUES OF ADJACENCY MATRIX\n\n";
    for (int i = 0; i < 7; i++)
    {
        cout<<"enter values for "<<(i+1)<<" row"<<endl;
        for (int j = 0; j < 7; j++)
        {
            cin>>b[i][j];
        }
    }
    floyds(b);
    getch();
}

```

تکنیک‌های طراحی الگوریتم (۳): روش‌های حریصانه[1]

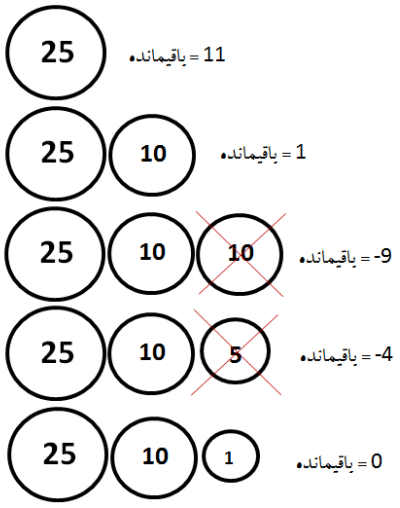
دسته‌ای از مسائل وجود دارد که برای حل آن‌ها در طول اجرای برنامه باید همیشه کوچک‌ترین یا بزرگ‌ترین مقدار از یک مجموعه مقدار انتخاب شود. این مسائل با استفاده از الگوریتم‌های حریصانه حل می‌شود.

۱-۷- مسأله خرد کردن پول:

مثال ۱: فرض کنید تعدادی سکه به صورت زیر داریم و می‌خواهیم ۳۶ تومان با استفاده از این سکه‌ها خرد کنیم، با این شرط که کمترین تعداد سکه به دست آید:

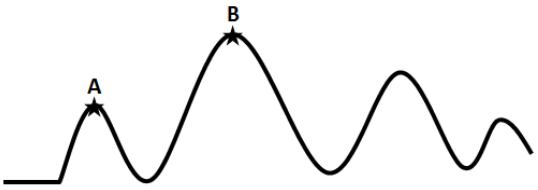


برای حل این مسأله ابتدا سکه‌ها را از بزرگ به کوچک مرتب می‌کنیم و سپس به‌طور حریصانه ابتدا بزرگ‌ترین سکه را انتخاب می‌کنیم و باقیمانده سکه‌های انتخاب شده تا مبلغ موردنظر را به دست آورده و ملاک انتخاب سکه‌های بعدی قرار می‌دهیم؛ بنابراین مراحل به شکل زیر خواهد بود:



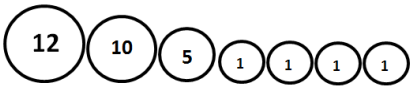
[1] Greedy Algorithms

نکته مهم: راه‌حل حریصانه همیشه بهترین پاسخ را به دست نمی‌آورد، چرا که بزرگ‌ترین مشکل این روش ***بهینه‌گی محلی***[1]است. بهینه‌گی محلی یعنی ممکن است یک پاسخ در مقطعی از زمان درست به نظر برسد امّا اگر بازه‌ی زمانی افزایش پیدا کند، آن پاسخ نسبت به کل، پاسخ صحیحی نیست. به شکل زیر دقت کنید:

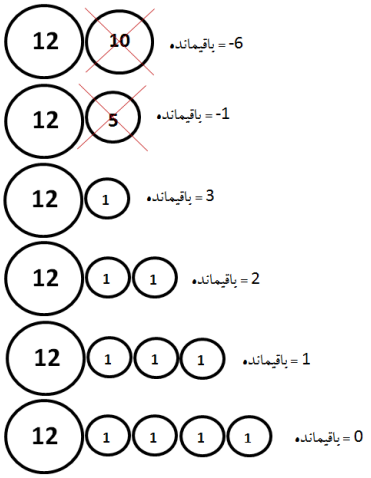


فرض کنید که قرار است برنامه‌ای بنویسید که بلندترین قله را در میان یک‌سری عدد پیدا کند، اگر شما برنامه را طوری نوشته باشید که به محض رسیدن به نقطه‌ای که $nums\ [i] < nums\ [i-1]$ است، آن را قله اعلام کند، با توجه به شکل خواهید دید که در نقطه‌ی A برنامه متوقف خواهد شد، در حالی‌که نقطه‌ی B نیز این ویژگی را دارد و از نقطه‌ی A بلندتر است.

مثال ۲: با استفاده از سکه‌های زیر ۱۶ تومان پول خرد به دست آورید، با همان شرط که حداقل سکه‌ها به دست آید.



راه‌حل: اگر با همان الگوریتم قبل مسأله را حل کنیم، خواهیم دید که پاسخ به صورت زیر خواهد بود:



در این حالت می‌بینیم که پاسخ ۱، ۱، ۱، ۱، ۱۲ به‌عنوان پاسخ نهایی انتخاب می‌شود، در حالی که پاسخ بهینه‌تر ۱، ۵، ۱۰ است.

تمرین: برای مسأله بالا یک راه‌حل معتبر ارائه کنید.

---

[1] Local Optima

۲-۷- مسأله کد هافمن[1]:

الگوریتم هافمن برای فشرده‌سازی داده‌ها استفاده می‌شود.

فرض کنید قرار است رشته‌ی زیر را فشرده کنیم:

$abaabcdaabbccddeda$

هافمن به روش زیر این رشته‌ها را فشرده می‌کند:

گام ۱: فرکانس (تعداد تکرار) هر کاراکتر را به دست آورید، پس داریم:

$a:6 \quad b:4 \quad c:3 \quad d:4 \quad e:1$

(فرکانس[2] تعداد تکرار هر چیز معمولاً در واحد زمان است و نام فارسی آن بسامد می‌باشد.)

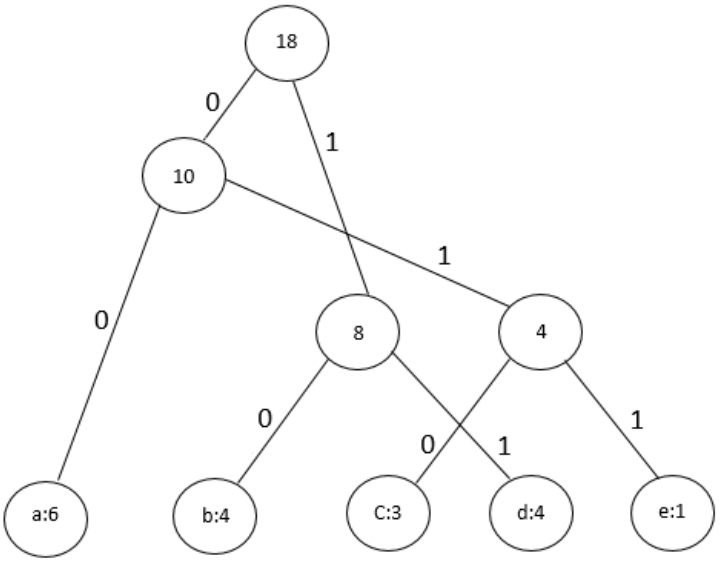
گام ۲: از سمت چپ دو کاراکتری را که کمترین تکرار را دارند انتخاب کنید و جمع آن‌ها را به دست آورید. (d:4 و b:4 انتخاب شده‌اند و جمع آن‌ها یعنی ۸ در بالای آن‌ها نوشته شده است)

گام ۳: مجدداً از سمت چپ دو کاراکتر دیگر را که کمترین تکرار را دارند، انتخاب و جمع آن‌ها را به دست آورید. فقط توجه کنید که دایره‌ی مربوط به جمع‌های قبلی نیز به‌عنوان یک کاراکتر قابل انتخاب خواهند بود.

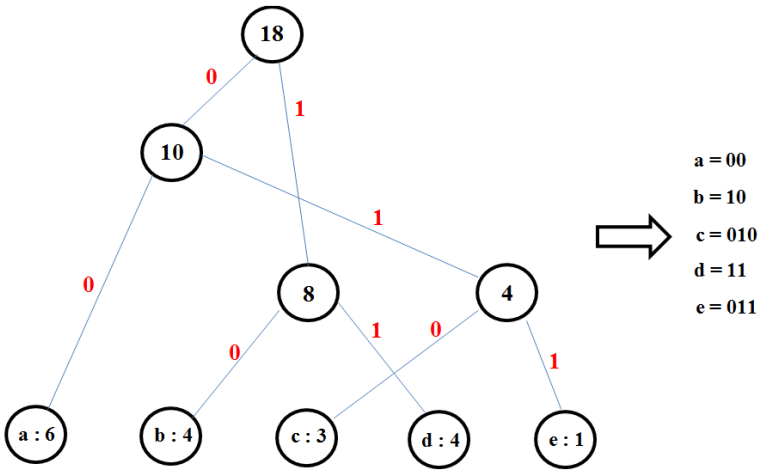
گام ۴: این کار را تکرار کنید تا کاراکتری برای جمع باقی نماند.

گام ۵: با توجه به گام‌های بالا درخت هافمن به شکل زیر از پایین به بالا ساخته خواهد شد. از ریشه شروع کنید و به یال سمت چپ، صفر و به یال سمت راست، یک نسبت دهید.

[1] Hafman
[2] Frequency: Cycles Per Time Unit



گام آخر: کد هافمنِ هر کاراکتر، رشته‌ای از صفر و یک‌ها از ریشه به سمت آن کاراکتر خواهد بود.



با توجه به توضیحات داده شده به جای ذخیره کردن یا ارسال کاراکتر a که ۸ بیت اشغال خواهد کرد، کد هافمن آن یعنی 00 را ذخیره یا ارسال می‌کنیم که مشخص است که حداقل ۶ بیت به‌ازای هر a فشرده‌سازی داریم. در بقیه حروف نیز به همین صورت.

نکته: باید یک فضا هم برای ذخیره کردن کد هر حرف در نظر بگیریم که در مقصد بفهمیم که برای مثال منظور از 00 کاراکتر a بوده است.

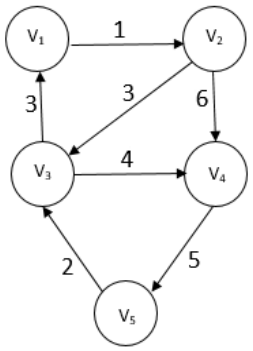
۳-۷- مسأله درخت پوشای کمینه[1]

درخت پوشای کمینه با درخت فراگیر مینیمم در گراف‌های ارزش‌دار (وزن‌دار) ساخته می‌شود.

فرض کنید گراف، یک گراف هم‌بند باشد (یعنی بین هر دو رأس متمایز آن، یک مسیر وجود داشته باشد) منظور از یک درخت پوشا از این گراف، درختی است که شامل همه رئوس این گراف باشد ولی فقط بعضی از یال‌های آن را در برگیرد.

منظور از درخت پوشای مینیمم (برای گراف هم‌بند وزن‌دار) درختی است که بین درخت‌های پوشای آن گراف، مجموع وزن یال‌های آن، کمترین مقدار ممکن باشد.

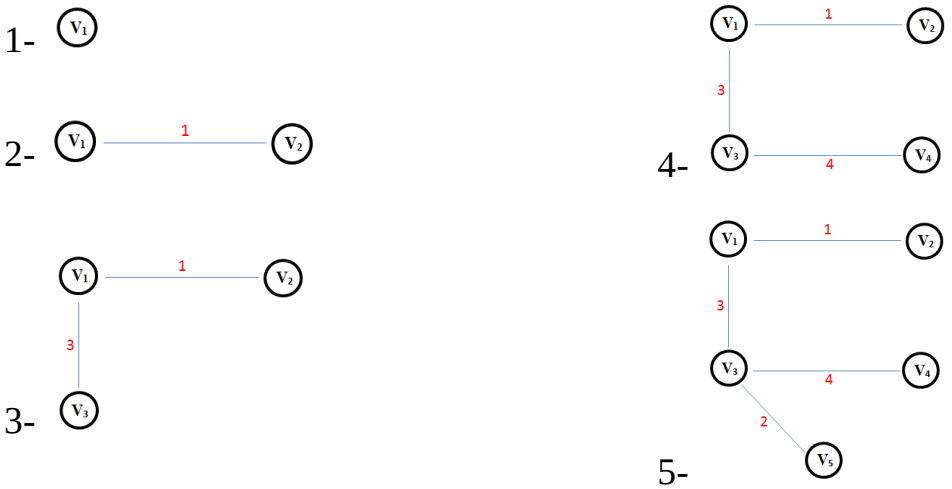
مثال: گراف هم‌بند زیر را در نظر بگیرید. گراف MST (درخت پوشای کمینه) آن را به دست آورید.



راه‌حل: برای حل مسأله از v_1 شروع می‌کنیم و به‌طور حریصانه یالی که کمترین وزن را دارد و به یکی از رأس‌های پیمایش‌شده متصل است، به عنوان ادامه مسیر انتخاب می‌کنیم. فقط باید توجه کرد که مسیر لازم نیست حتماً به صورت حلقه و به‌هم پیوسته باشد.

مراحل به شکل زیر خواهد بود:

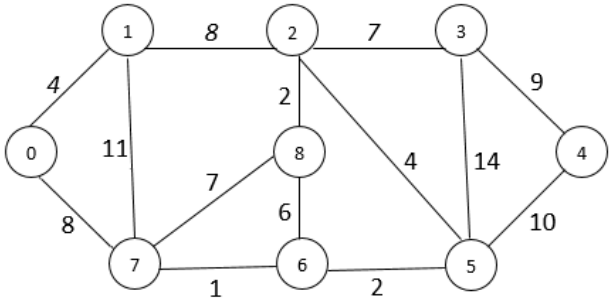
[1] Minimum Spanning Tree



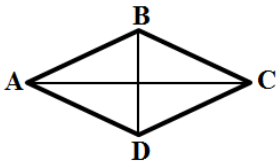
با استفاده از تابع Prim به صورت زیر می‌توان این الگوریتم را پیاده‌سازی کرد:

```c
#include <stdio.h>
#include <limits.h>
#define V 5
int minKey(int key[], bool mstSet[])
{
   int min = INT_MAX, min_index;

   for (int v = 0; v < V; v++)
     if (mstSet[v] == false && key[v] < min)
         min = key[v], min_index = v;

   return min_index;
}

int printMST(int parent[], int n, int graph[V][V])
{
   printf("Edge   Weight\n");
   for (int i = 1; i < V; i++)
      printf("%d - %d    %d \n", parent[i], i, graph[i][parent[i]]);
}

void primMST(int graph[V][V])
{
     for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

     for (int count = 0; count < V-1; count++)
     {
        int u = minKey(key, mstSet);

        mstSet[u] = true;

         for (int v = 0; v < V; v++)

           if (graph[u][v] && mstSet[v] == false && graph[u][v] <  key[v])
              parent[v]  = u, key[v] = graph[u][v];
     }
     printMST(parent, V, graph);
}


int main()
{
   /* Let us create the following graph
           2    3
       (0)--(1)--(2)
        |   / \   |
       6| 8/   \5 |7
        | /     \ |
       (3)-------(4)
             9          */
   int graph[V][V] = {{0, 2, 0, 6, 0},
                      {2, 0, 3, 8, 5},
                      {0, 3, 0, 0, 7},
                      {6, 8, 0, 0, 9},
                      {0, 5, 7, 9, 0},
                     };

    primMST(graph);
    return 0;
}
```

تمرین: در گراف هم‌بند زیر درخت MST را به دست آورید.



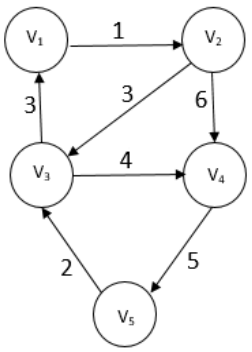
۱-۷- مسأله فروشنده دوره‌گرد[1]



صورت این مسأله به این شکل است: اگر یک فروشنده دوره‌گرد از نقطه A شروع کند و فواصل بین نقاط مشخص باشد، کوتاه‌ترین مسیر که از تمام نقاط یک‌بار بازدید کند و به A بازگردد، کدام است؟

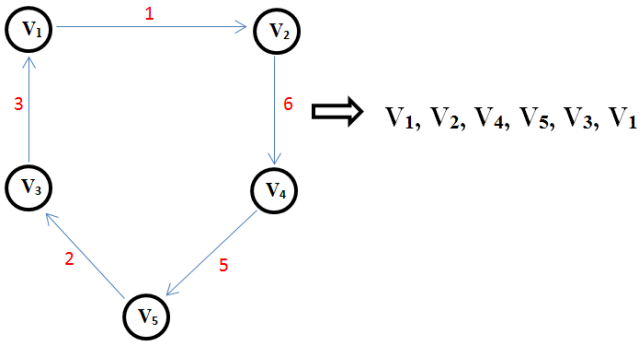
در مسأله فروشنده دوره‌گرد باید از رأس v_1 به رأس‌های دیگر مسیر به‌هم پیوسته‌ای را پیدا کنید که ضمن گذر از همه رئوس به رأس v_1 برگردد. ضمناً این مسیر کمینه نیز باشد و خیلی مهم است که گراف جهت‌دار باشد.

مثال: بهترین مسیر برای فروشنده دوره‌گرد در گراف زیر چیست؟



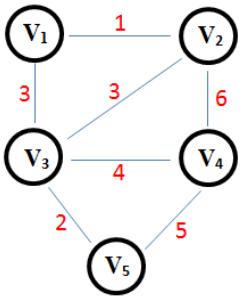
پاسخ:

[1] Travelling Salesman Problem



۲-۷- الگوریتم کراسکال:

این الگوریتم نیز برای به دست آوردن درخت پوشای کمینه استفاده می‌شود، با این تفاوت که مراحل آن با کمی تفاوت به شکل زیر است:



مرحله ۱: یال‌ها را برحسب وزن‌شان مرتب کنید.

$(v_1, v_2 : 1)$

$(v_3, v_5 : 2)$

$(v_1, v_3 : 3)$

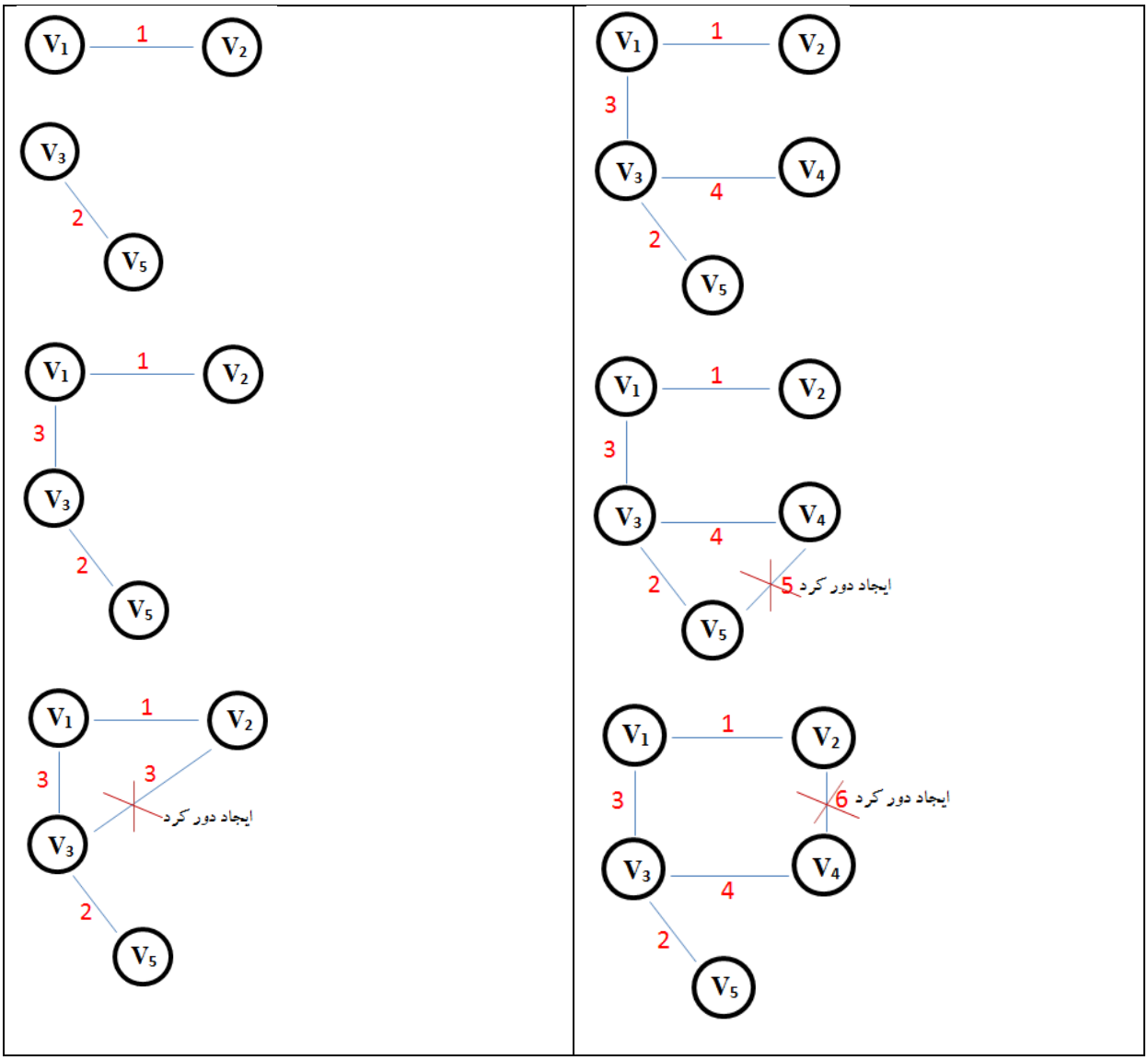
$(v_2, v_3 : 3)$

$(v_3, v_4 : 4)$

$(v_4, v_5 : 5)$

$(v_2, v_4 : 6)$

مرحله ۲: هر یالی که ایجاد دور نکرد، انتخاب کنید.



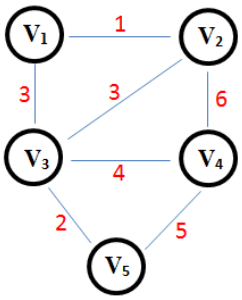
۳-۷- الگوریتم دکسترا:

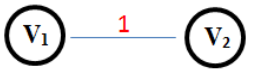
این الگوریتم نیز یکی از الگوریتم‌های پیمایش گراف است که، مسأله «کوتاه‌ترین مسیر از مبدأ واحد» را برای گراف‌های وزن‌داری که یال یا وزن منفی ندارند حل می‌کند و در نهایت با ایجاد درخت کوتاه‌ترین مسیر، کوتاه‌ترین مسیر از مبدأ به همه رأس‌های گراف را به دست می‌دهد.

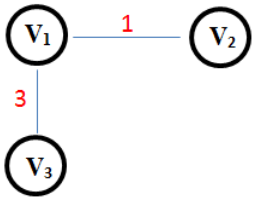
توضیح:

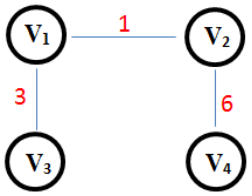
از رأس V_1 شروع کنید و پس از آن رأسی را انتخاب کنید که مجموع یال‌های رسیده تا آن رأس، کمینه شود.

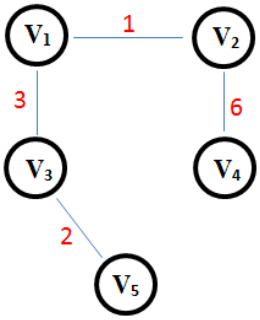
مثال: درخت پوشای کمینه را با استفاده از الگوریتم دکسترا به دست آورید.



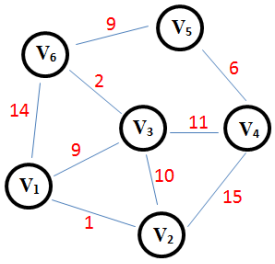








تمرین: با استفاده از الگوریتم دکسترا کوتاه‌ترین مسیر بین V_1 و V_5 را در گراف زیر به دست آورید.



فصل هشتم

تکنیک‌های طراحی الگوریتم (۴): الگوریتم‌های عقبگرد[1]

بازگشت به عقب یا عقبگرد، تکنیکی در برنامه‌نویسی است که در آن برای حل یک مسأله، از یک گراف جهت‌دار (درخت جهت‌دار) استفاده می‌شود. در این مسائل ابتدا یک فضای حالت برای مسائل در نظر گرفته می‌شود و آن فضای حالت به صورت گرافی که، هر رأس آن یک حالت و هر یال آن یک انتخاب از آن حالت به حالت بعد است، نمایش داده می‌شود. مهم‌ترین مسأله در مسائل عقبگرد، تشخیص بن‌بست در یک شاخه و هرس کردن[2] آن شاخه است.

مسائلی وجود دارند که نه با تقسیم و غلبه و نه با برنامه‌نویسی پویا حل نمی‌شوند. الگوریتم حریصانه نیز در این مسائل به جواب بهینه منتهی نمی‌شود. **تنها راه** یافتنِ راه‌حلِ بهینه، تولید کردن تمام راه‌حل‌ها و آزمایش آن‌هاست، که روش‌های عقبگرد به این صورت عمل می‌کند. در راه‌برد عقبگرد سعی می‌کنیم بن‌بست‌ها را شناسایی کنیم و از دنبال کردن مسیرهای تکراری جلوگیری کنیم. به‌طور خلاصه راه‌برد عقبگرد عبارت است از انتخاب دنباله‌ای از اشیا از یک مجموعه توأم با برآورده کردن یک شرط یا محدودیت.

راه‌برد عقبگرد در حقیقت نوعی جستجو است.

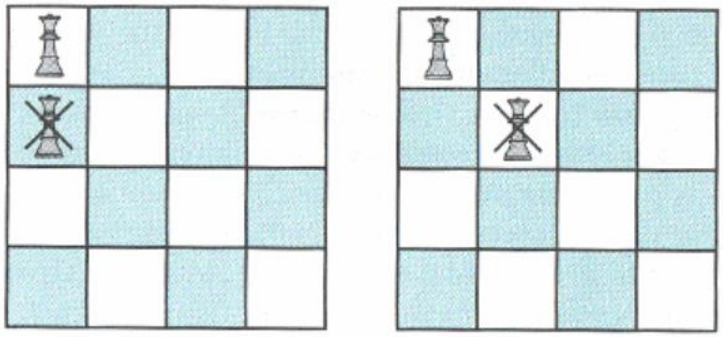
۱-۸- مسأله ۴ وزیر:

صورت مسأله: فرض کنید یک صفحه شطرنج ۴*۴ داریم و می‌خواهیم ۴ وزیر را به‌طوری روی صفحه قرار دهیم که هیچ وزیری امکان زدن وزیر دیگر را نداشته باشد؛ یعنی هیچ وزیری نباید در دید مستقیم افقی یا عمودی یا قطری وزیرهای دیگر باشد.

---

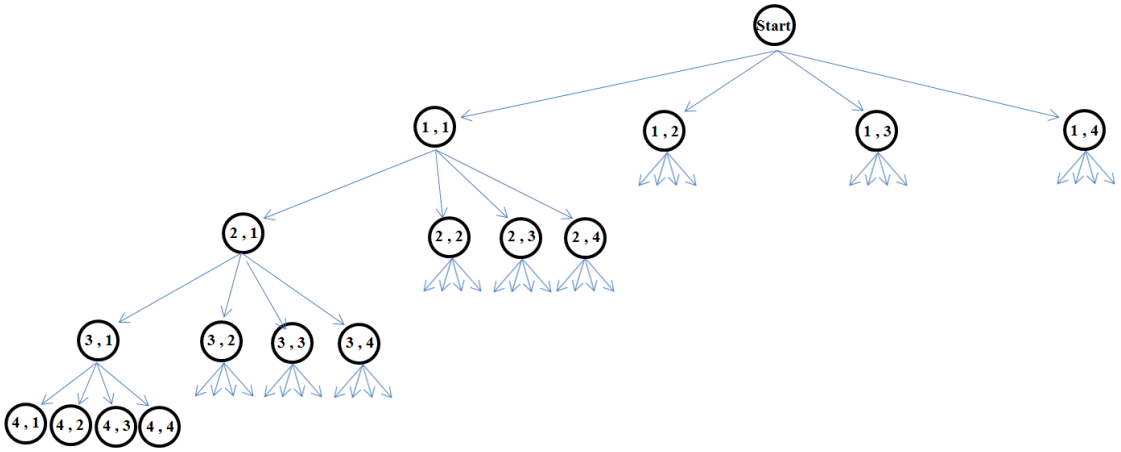
[1] Back Tracking

[2] Pruning



راه‌حل:

۱. درخت فضای حالت[۱] را به شکل زیر رسم کنید.



۲. همان‌طور که مشخص است، این درخت ۲۵۶ = ۴*۴*۴*۴ برگ خواهد داشت.

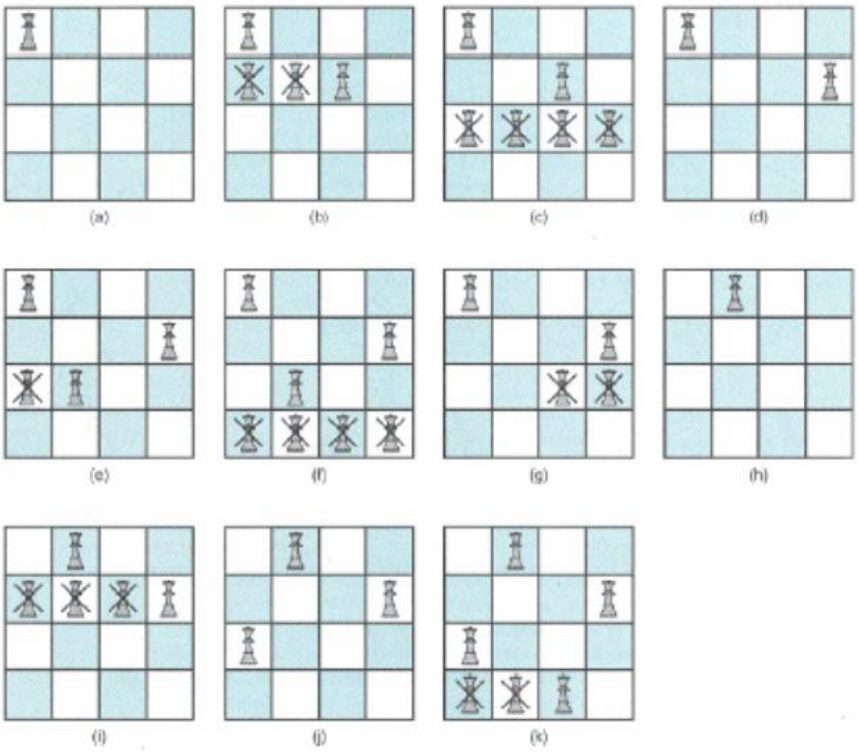
۳. هر سطر برای یک وزیر در نظر گرفته می‌شود؛ یعنی سطر شماره i برای وزیر iاُم.

۴. در حقیقت مسأله اصلی این است که چه شماره ستونی (j) را برای وزیر iاُم انتخاب کنیم.

۵. هر مسیر از ریشه تا برگ یک راه‌حل کاندید است که ممکن است صحیح یا غلط باشد.

۶. بنابراین این مسأله در واقع به مسأله جستجوی درخت فضای حالت تغییر می‌کند و تبدیل می‌شود.

[۱] State Space Tree



۷. به دو صورت می‌توان یک درخت را پیمایش کرد:

- جستجوی ساده:

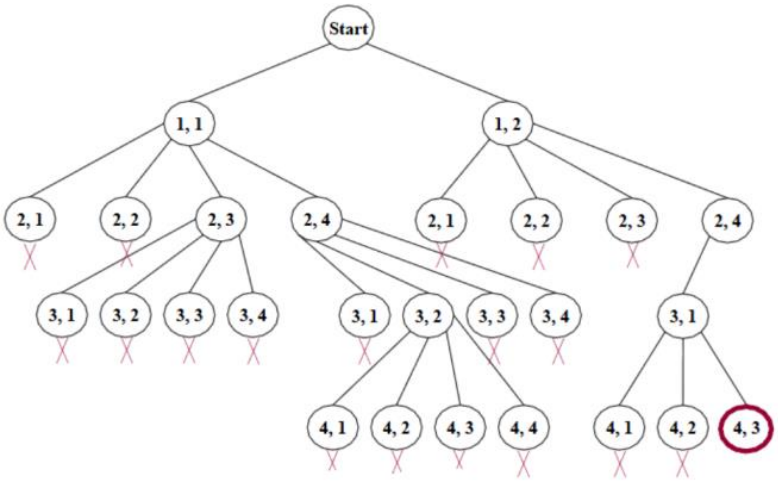
یعنی همه گره‌ها را تک به تک بررسی کنیم و ببینیم که پاسخ صحیح هست یا خیر، که در این حالت پردازش بسیاری لازم دارد.

- جستجوی با راه‌برد عقبگرد:

یعنی گره‌ای که در آن احتمال پاسخ وجود ندارد را باز نکنید و آن شاخه را از نقطه‌ای که احتمال پاسخ بودن را نقض می‌کند، هرس کنید، که در این حالت خواهیم دید بسیاری از پردازش‌ها با هرس کردن شاخه‌ها حذف خواهد شد.

- به گره‌ای که احتمال رسیدن به پاسخ وجود داشته باشد، «گره امید بخش» گفته می‌شود.

با توجه به نکات بالا پاسخ این مسأله به شکل زیر خواهد بود:



یعنی:

	1		
			2
4			
		3	

الگوریتم کلی مسأله ۴ وزیر:

```
void checknode ( node v)
{
   node u ;

   if ( promising ( v ) )
        if ( there is a solution at v )
                write the solution ;
        else
                for ( each child u of v )
                        checknode ( u ) ;
}
```

الگوریتم ۴ وزیر به صورت بهینه شده:

در این الگوریتم گره‌هایی که احتمال پاسخ ندارند، باز نمی‌شوند.

```
void expand ( node v )
{
   node u ;

   for ( each child u of v )
        if ( promising ( u ) )
                if ( there is a solution at u )
                        write the solution ;
                else
                        expand ( u ) ;
}
```

می‌توان این مسأله را به مسأله n وزیر تعمیم داد.

تمرین: تابع Promising را به زبان فارسی توصیف کنید؛ یعنی منظور از امید بخش بودن یک گره چیست؟

پاسخ: تفاضل شماره ستون فرزند (k) و والد (i) نباید از ۱ کمتر باشد؛ یعنی:

$$| col\ (i) - col\ (k)| > 1$$

۲-۸- مسأله حاصل جمع زیرمجموعه‌ها:

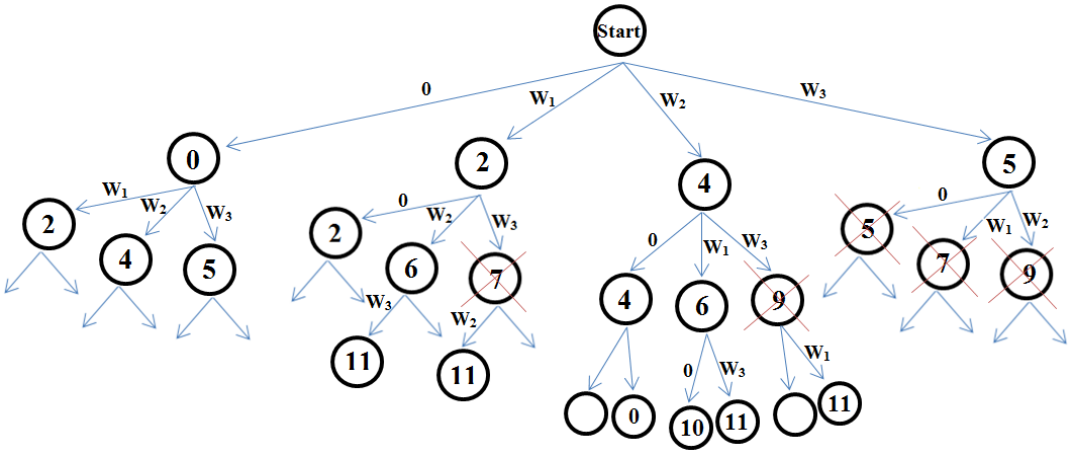
صورت مسأله: مجموعه‌ای مانند $\{w_1, w_2, \dots, w_n\}$ داریم که می‌خواهیم زیرمجموعه‌ای از این مجموعه انتخاب کنیم؛ به این شرط که $\sum wi = w$.

مثال: از مجموعه‌ی زیر، زیر مجموعه‌ای انتخاب کنید که جمع آن‌ها ۶ شود:

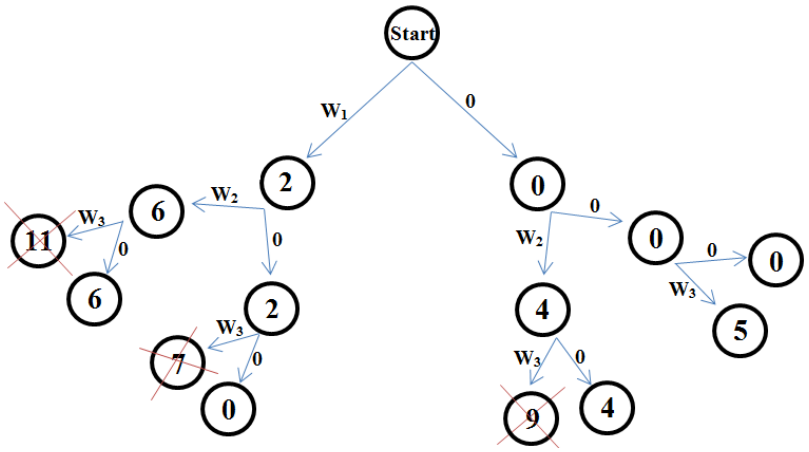
$$\{2, 4, 5\}$$

راه‌حل: درخت حالت این مسأله به صورت زیر است:

$$\{w1: 2, w2: 4, w3: 5\}$$

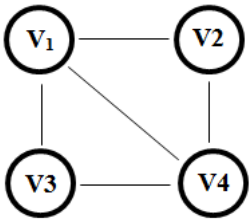


همان‌طور که مشاهده می‌کنیم درخت فضای حالت این مسأله بسیار پیچیده خواهد شد، می‌توان مسأله را به صورت درخت دودویی در نظر گرفت که در هر حال، یکی از عناصر انتخاب می‌شود یا نمی‌شود؛ بنابراین داریم:

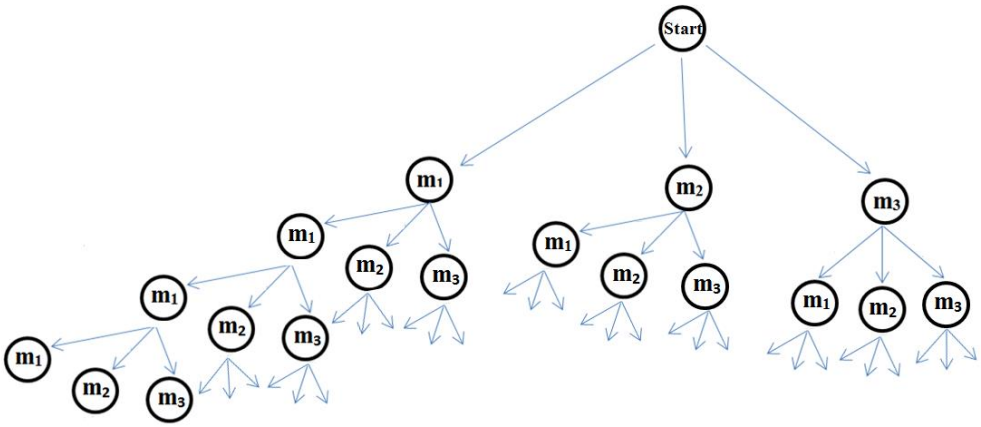


۳-۸- مسأله رنگ‌آمیزی گراف با m رنگ (M–Coloring):

صورت مسأله: گراف زیر را با سه رنگ طوری رنگ کنید که هیچ دو رنگی مجاور هم قرار نگیرند.



این مسأله در رنگ‌آمیزی نقطه‌های جغرافیایی یا سلول‌بندی شبکه‌های مخابراتی کاربرد دارد. برای به دست آوردن پاسخ مسأله، درخت فضای حالت را به صورت زیر رسم می‌کنیم:



پس از ساخت گراف، تک‌تکِ مسیرها بررسی می‌شوند و با توجه به یال‌های گراف تصمیم می‌گیریم که این مسیر پاسخگوی مسأله هست یا خیر. به طور مثال، مسیر اول عبارت است از:

m1,m1,m1,m1

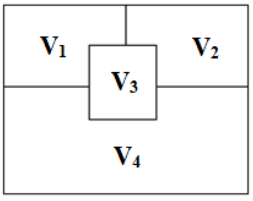
که قطعاً پاسخ مسأله نیست (چون طبق گراف نباید رنگ v1 و v2 یکسان باشد. مسیر دوم:

m1,m1,m1,m2

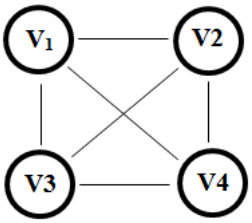
این نیز پاسخ درستی نخواهد بود و به همین صورت تمام مسیرها بررسی می‌شود.

اما این گراف می‌تواند طبق الگوریتم عقبگرد، هرس شود و به طور مثال وقتی بعد از m1 دوباره m1 انتخاب می‌شود، این شاخه، «غیر امیدبخش» تشخیص داده شود و هرس شود و برگ‌های آن بررسی نشود...

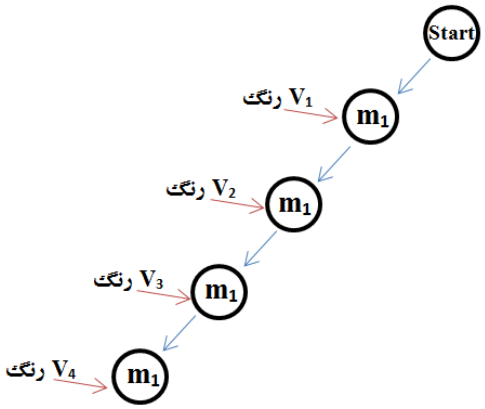
مثال: نقشه‌ی زیر حداقل به چند رنگ نیاز دارد تا مسأله M–Coloring در مورد آن صدق کند.



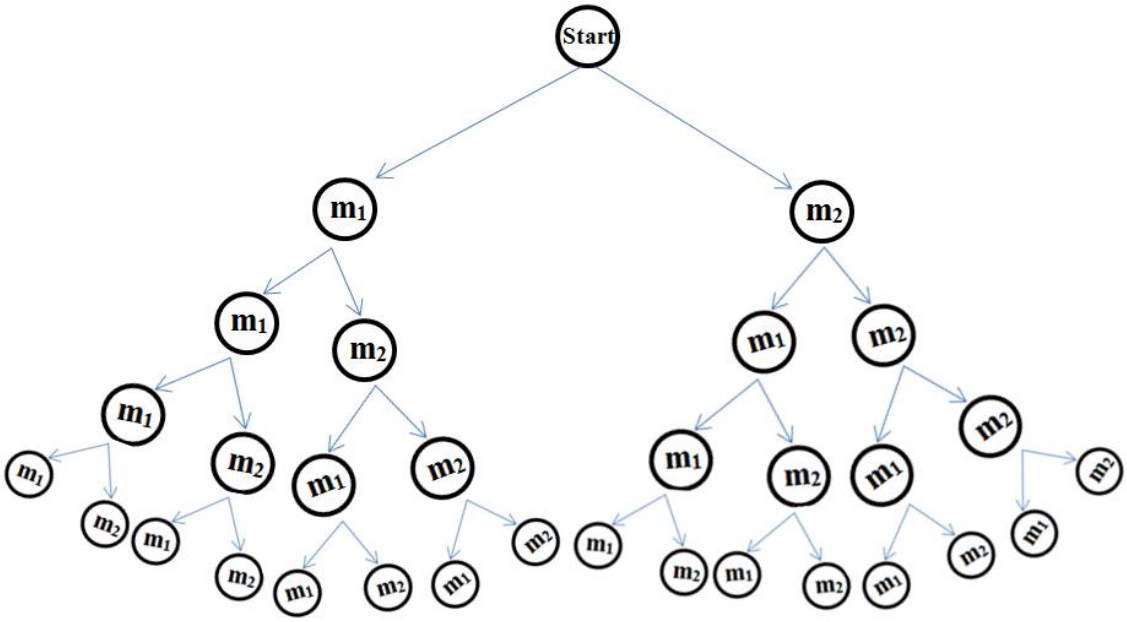
راه‌حل: برای حل مسأله، آن را به گراف زیر تبدیل می‌کنیم:



بنابراین به یک مسأله M–Coloring تبدیل شد که باید درخت حالت مسأله آن را رسم کنیم. مسأله را یک‌بار، با یک رنگ حل می‌کنیم، که داریم:



سپس با دو رنگ:

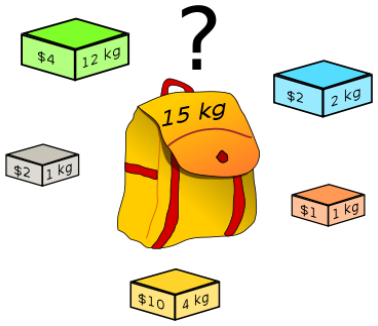


سپس سه رنگ و سپس چهار رنگ.

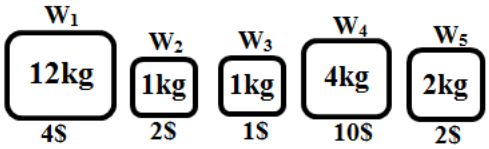
اگر به همین صورت با دو، سه و چهار رنگ ادامه دهیم، به این نتیجه خواهیم رسید که فقط با چهار رنگ، امکان انتخاب رنگ‌هایی وجود دارد که مجاور نباشد.

۸-۴ مسأله کوله‌پشتی[1]

صورت مسأله: فرض کنید قرار است یک جهانگرد از بین اشیاء زیر حداقل اشیائی را انتخاب کند که وزن آن‌ها بیش از ۱۵ کیلوگرم نشود و در عین حال سودمندترین اشیاء باشد. (منظور از ارزش هر شیئ میزان نیازمندی جهانگرد به آن شیئ است)



مسأله را به صورت زیر در نظر بگیریم قابل فهم‌تر است:



برای حل مسأله ابتدا تناسب سود به وزن را به صورت زیر محاسبه می‌کنیم:

$$w_1 = \frac{4}{12} = 0.3$$

$$w_2 = \frac{2}{1} = 2$$

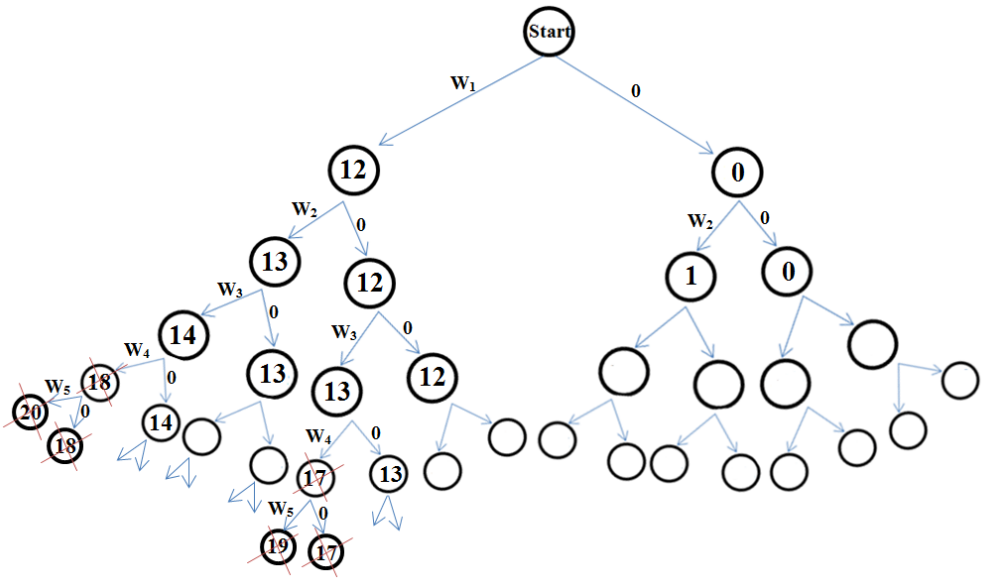
$$w_3 = \frac{1}{1} = 1$$

$$w_4 = \frac{10}{4} = 2.5$$

$$w_5 = \frac{2}{2} = 1$$

[1] Knapsack

درخت فضای حالت را به‌صورت زیر رسم می‌کنیم:



تمرین: این مسأله را با روش حریصانه حل کنید.

۵-۸- انواع مسأله کوله‌پشتی:

کوله‌پشتی صفر و یک (۱ ، ۰): در این دسته مسائل کلِ یک شیئ را می‌توانیم یا نمی‌توانیم انتخاب کنیم.

کوله‌پشتی کسری: در این دسته مسائل می‌توانیم، کسری از یک شیئ را انتخاب کنیم.

تمرین: برای کوله‌پشتی کسری یک مثال از دنیای واقعی بزنید...