فصل ۶

روش حریصانه

روش حریصانه (Greedy) یکی از روش‌های ساده، جالب و سریع برای حل دقیق مسئله‌هاست. البته باید مسئله دارای شرائط خاصی باشد تا بتوان از این روش برای حل آن استفاده کرد. متأسفانه تعداد کمی از مسئله‌ها راه‌حل حریصانه دارند، از جمله‌ی آن‌ها مسئله‌های مهمی هستند که راه‌حل تعدادی از آن‌ها به‌دلیل نوآوری قابل توجه به‌نام طراح‌های آن‌ها ثبت شده‌اند: مانند الگوریتم هافمن (Huffman) برای پیدا کردن کدهای بهینه برای فشرده‌سازی متن‌ها، الگوریتم دایکسترا (Dijkstra) برای پیدا کردن همه‌ی کوتاه‌ترین مسیرها از یک رأس در یک گراف، و الگوریتم‌های پریم (Prim) و کروسکال (Kruskal) برای پیدا کردن درخت پوشای کمینه در یک گراف، و الگوریتم Fleury برای پیداکردن یک دور اولری در گراف.

روش حریصانه اگر منجر به پیدا کردن جواب دقیق مسئله نشود یک روش تقریبی برای حل آن مسئله است که در صورتی که بتوان اثبات کرد که جواب به‌دست آمده در بدترین حالت چه‌مقدار با جواب دقیق فاصله دارد، بسیار با ارزش است و یکی از روش‌های مهم در طراحی الگوریتم‌های تقریبی «قابل اثبات» است. در صورتی‌که چنین اثباتی وجود نداشته باشد، روش حریصانه در واقع یک روش مکاشفه‌ای (heuristic) برای حل مسئله است که ممکن است در عمل کاربرد فراوان داشته‌باشد ولی از نظر علمی چندان با ارزش نیست.

مسائلی که با روش حریصانه قابل حل هستند دارای خصوصیات زیر می‌باشند:

- مسئله بهینه‌سازی (optimization) است.

- برای حل بهینه‌ی مسئله باید زیرمسئله‌های آن‌را نیز به‌صورت بهینه حل کرد (optimal subproblem). مسئله‌هایی که به‌روش پویا قابل حل بودند نیز دو ویژگی فوق را داشتند.

- انتخاب حریصانه (greedy choice) دراین‌گونه مسائل بهترین انتخاب است و عوض نمی‌شود. (تفاوت با روش پویا در این ویژگی است.)

به‌طور کلی می‌توان یک الگوریتم حریصانه را مطابق زیر بیان کرد:

```
GREEDY(C)
    ▷ C: the set of all candidates
1   MAKENULL(S)    ▷ S is the solution
2   while  not SOLUTION(S)  and  not EMPTY(C)
3          do x ← an element in C maximizing SELECT(x)
4              C ← C − {x}
5              if FEASIBLE (S ∪ {x})
6                 then S ← S ∪ {x}
7   if SOLUTION(S)
8     then return (S)
9     else return "NO solution"
```

۱.۶ ویژگی‌های انتخاب حریصانه

انتخاب نامزد درصورت نداشتن تضاد با انتخاب‌های انجام‌شده انتخاب نهایی است و عمل پس‌گرد (backtracking) برای تغییر انتخاب صورت نمی‌گیرد. این انتخاب براساس تابع انتخاب (selection function) و براساس مقادیر محلی صورت می‌گیرد. در واقع الگوریتم حریصانه به‌گونه‌ای است که انتخاب مبتنی بر بهینه‌سازی محلی (local optimization) منجر به بهینه‌سازی سراسری (global optimization) می‌شود که هدف مسئله است.

۲.۶ انتخاب فعالیت‌ها (Activity Selection Problems)

N فعالیت مختلف $a_1, a_2, \cdots$ و t_n داده شده‌اند که همگی از یک منبع غیرقابل اشتراک استفاده می‌کنند. برای هر فعالیت دو پارامتر زمان شروع و زمان پایان مشخص شده است. هدف پیدا کردن بیشترین تعداد این فعالیت‌هاست که بتوانند از منبع استفاده کنند.

ورودی: برای هر فعالیت t_i (برای $1 \leq i \leq n$) مقادیر زیر داده شده‌اند:

- زمان شروع: s_i
- زمان پایان: f_i

هدف: انتخاب حداکثر فعالیت های ممکن.

مثال: هر فعالیت می‌تواند یک تقاضا برای استفاده از مثلاً سالن اجتماعات یک دانشگاه باشد.

ارائه‌ی یک روش ساده برای حل مساله:

می توان برای هرفعالیت موجود، تمام فعالیت‌های دیگر که با آن در تضاد هستند را حذف نموده و این روش را تکرار کرد و کلیه‌ی مجموعه فعالیت‌های ممکن را به‌دست آورد. در پایان بزرگ‌ترین مجموعه‌ی ممکن جواب است. الگوریتم بسیار کند و از مرتبه‌ی $O(n!)$ است چون همه‌ی جای‌گشت‌ها را بررسی می‌کند.

روش حریصانه

در الگوریتم حریصانه به‌دنبال روشی برای انتخاب یک فعالیت مناسب هستیم که در صورتی‌که با انتخاب‌های تناقضی نداشته باشد آن‌را به‌صورت نهایی انتخاب کنیم. برای این کار می‌توانیم ترتیب انتخاب‌ها را براساس ترتیب اولیه، طول فعالیت‌ها، زمان شروع، یا زمان پایان (از کوچک به بزرگ یا برعکس) انجام دهیم.

مشخص است که در دو مورد اول به مشکل بر می‌خوریم، مثلاً در مورد ترتیب انتخاب براساس طول فعالیت‌ها، اگر از سه فعالیت، کوچک‌ترین آن‌ها با دو فعالیت دیگر (که از هم مجزا هستند) اشتراک داشته باشد، ما فقط همان کوچک‌ترین فعالیت را می‌توانیم انتخاب کنیم، در صورتی‌که دو فعالیت دیگر جواب بهینه است. برای ترتیب‌های دیگر، به‌جز آخرین ترتیب نیز می‌توان مثال نقض پیدا کرد.

بنابراین، به نظر می‌رسد که اگر فعالیت‌ها را به‌ترتیب پایان زمان‌شان انتخاب کنیم و پس از انتخاب یک فعالیت، همه‌ی فعالیت‌های متضاد با آن را حذف کنیم، جواب بهینه به‌دست می‌آید. البته درستی این الگوریتم حریصانه را باید اثبات کنیم.

مراحل کلی اثبات درستی یک الگوریتم حریصانه:

۱. اثبات می‌کنیم که یک راه حل بهینه وجود دارد که شامل اولین انتخاب الگوریتم پیشنهادی است.

۲. با حذف انتخاب اولیه و انتخاب‌های متضاد با آن یک زیرمسئله به‌دست می‌آید که باید آن‌را نیز به‌صورت بهینه و به‌همین روش حل کنیم.

در مسئله‌ی انتخاب فعالیت‌ها، ابتدا یک زیر مسئله‌ی کلی را تعریف می‌کنیم و سپس به اثبات دقیق درستی الگوریتم ارائه‌شده می‌پردازیم. یک زیر مسئله، شامل k عدد فعالیت است (در مسئله‌ی اصلی $k = n$). بدون کم شدن از کلیت مسئله می‌توان فرض کرد که شماره‌ی فعالیت‌های یک زیرمسئله به‌همان ترتیب زمان‌های پایان آن‌ها (از کوچک به بزرگ) است. یعنی فرض می‌کنیم که $f_1 \leq f_2 \leq \cdots \leq f_k$.

لم ۸. یک راه‌حل بهینه برای مسئله وجود دارد که شامل کار t_1 است.

اثبات. برهان خلف: فرض A یک راه‌حل بهینه برای مسئله‌ی داده‌شده است که t_1 متعلق به آن نیست. فرض کنید t_i فعالیت با کم‌ترین زمان پایان در A است. بنا به فرض داریم: $f_1 \leq f_i$. حال اگر فعالیت t_i را از A حذف کنیم و به‌جای آن t_1 را قرار دهیم، مجموعه‌ی حاصل نیز یک جواب برای مسئله است، چون t_1 حداکثر می‌تواند با t_i در A اشتراک داشته باشد و با بقیه‌ی فعالیت‌ها اشتراکی ندارد. چون تعداد فعالیت‌های موجود در A و در جواب جدید برابرند، این جواب نیز بهینه است. پس ما یک جواب بهینه‌ی شامل t_1 پیدا کردیم. □

با انتخاب t_1 و سپس حذف کلیه‌ی فعالیت‌هایی که با آن اشتراک دارند، یک زیر مسئله‌ی کوچک‌تر به‌دست می‌آید که آن‌را نیز به همین روش حل می‌کنیم. بنابراین این الگوریتم به یک مرتب‌سازی اولیه نیاز دارد و بقیه‌ی الگوریتم از مرتبه‌ی $O(n)$ انجام می‌شود. پس پیچیدگی الگوریتم $O(n \lg n)$ است.

مثال: الگوریتم پیشنهادی بر روی داده‌های نمونه به‌صورت زیر عمل کرده است.

فعالیت i	زمان شروع (s_i)	زمان پایان (f_i)	انتخاب یا حذف
۱	۱	۴	انتخاب اول
۲	۳	۵	حذف
۳	۰	۶	حذف
۴	۵	۷	انتخاب دوم
۵	۳	۶	حذف
۶	۵	۹	حذف
۷	۶	۱۰	حذف
۸	۸	۱۱	انتخاب سوم
۹	۸	۱۲	حذف
۱۰	۲	۱۳	حذف
۱۱	۱۲	۱۴	انتخاب چهارم

۳.۶ مسئله‌های کوله‌پشتی

دیدیم که مسئله‌های کوله‌پشتی در حالت کلی NP-Complete هستند و برای برخی از آن‌ها راه‌حل پویا با زمان چندجمله‌ای وجود دارد. البته این راه‌حل‌ها چون مقدار حافظه‌ی مصرفی‌اشان متناسب است با مقدار اعداد ورودی این راه‌حل واقعاً چندجمله‌ای نیست و به‌آن شبه چندجمله‌ای (psuedo polynomial) می‌گویند.

برای برخی از مسئله‌های کوله‌پشتی راه‌حل حریصانه وجود دارد.

۱.۳.۶ مسئله‌ی خرد کردن پول

می‌خواهیم n تومان را با سکه‌های ۱، ۲ و ۵ تومانی خرد کنیم که مجموع تعداد سکه‌هایی که استفاده می‌کنیم حداقل شود. از هر یک از این سکه‌ها به‌تعداد زیادی در اختیار داریم. این الگوریتم معمولاً برای این مسئله ارایه می‌شود:

> در ابتدا هرچه بتوانیم سکه‌های ۵ تومانی برمی‌داریم تا جایی‌که مقدار باقی‌مانده از ۵ تومان کم‌تر شود. سپس آن‌قدر سکه‌ی ۲ تومانی برمی‌داریم تا مقدار باقی‌مانده از دو تومان کم‌تر شود و در نهایت مقدار باقی‌مانده را از سکه‌های یک تومانی برمی‌داریم.

این یک الگوریتم حریصانه است، چون در هرمرحله بزرگ‌ترین سکه‌ای که می‌تواند انتخاب می‌کند. آیا این الگوریتم همواره جواب بهینه را می‌دهد؟ جواب این سوال برای این سکه‌ها مثبت است. اثبات این امر به‌تمرین واگذار می‌شود. برای مقادیر دیگری از سکه‌ها (برابر ۱، c، c^2 و ...) هم این الگوریتم درست کار می‌کند. اما اگر به جای سکه‌های ۱، ۲ و ۵ تومانی سکه‌های دیگری در اختیار داشتیم، الگوریتم لزوماً درست عمل نمی‌کند. مثلاً اگر به‌جای سکه‌ی ۲ تومانی سکه‌ی ۴ تومانی داشتیم، الگوریتم ۸ تومان را با یک سکه‌ی ۵ تومانی و سه سکه‌ی ۱ تومانی (مجموعا ۴ سکه) خرد می‌کند، درحالی‌که فقط با دو سکه ۴ تومانی بهتر خرد می‌شود. روشن است که در حالت کلی این مسئله همان مسئله‌ی کوله‌پشتی است که ارزش هر بار ۱ و تعداد هر نوع بار بینهایت است. این مسئله را قبلاً حل کرده‌ایم.

۲.۳.۶ مسئله‌ی کوله‌پشتی با بارهای قابل تقسیم

مسئله‌ی کوله‌پشتی در صورتی‌که بارها را بتوانیم تقسیم کنیم (fractional knapsack problem) نیز راه‌حل حریصانه دارد.

فرض کنید N عدد بار داده شده است که وزن و ارزش بار iام به‌ترتیب برابر W_i و C_i است و می‌خواهیم یک کوله‌پشتی به‌اندازه‌ی M را با این بارها پر کنیم به‌طوری‌که ارزش بارهای انتخابی بیشترین شود. در این مسئله مجازهستیم که یک بار را به نسبت دل‌خواه به دو قسمت تقسیم کنیم.

الگوریتم ساده‌ی زیر این کار را انجام می‌دهد:

> بارها را به‌ترتیب ارزش در واحد وزن مرتب می‌کنیم. یعنی فرض می‌کنیم $\frac{W_1}{C_1} \geq \frac{W_2}{C_2} \geq \cdots \frac{W_n}{C_n}$. بارها را به‌همین ترتیب مورد بررسی قرار می‌دهیم. اگر بار i می‌تواند کاملاً در کوله‌پشتی باقی‌مانده قرار بگیرد این کار را می‌کنیم و سراغ بار بعدی می‌رویم. اگر وزن بار iام بیشتر از مقدار باقی‌مانده از کوله‌پشتی است، این بار را به‌نسبت مورد نظر تقسیم می‌کنیم تا کوله‌پشتی کاملاً پر شود. واضح است که این الگوریتم درست کار می‌کند. (اثبات دقیق را به‌خودتان وا می‌گذاریم.)

۴.۶ مسائل زمان‌بندی

در حالت کلی این مسائل NP-Complete هستند اما برای برخی از آن‌ها با روش حریصانه راه حل دقیق وجود دارد.

۱.۴.۶ حالت ساده

یک سرویس‌دهنده و چند کار (job) آماده‌ی گرفتن سرویس داده شده‌اند. زمان مورد نیاز هر کار داده شده است. می خواهیم به تمام این کارها به‌ترتیبی سرویس دهیم و یک پارامتر سیستم را بهینه سازیم. پارامترهای سیستم می‌تواند، متوسط زمان پاسخ، حداکثر یا متوسط زمان انتظار باشد. مثالی از این مسئله را در صف‌های مختلف در عمل می‌توان دید. چنین صف‌هایی هم در سیستم عامل داریم و بسیاری از الگوریتم‌های زمان‌بندی در این‌جا کاربرد دارند.

به‌بیان دیگر، یک پردازنده، n عدد کار d_1، d_2، $\cdots$، و t_n در زمان صفر داده شده‌اند، به‌طوری‌که زمان مورد نیاز کار t_i برابر s_i است (service time). می خواهیم یک زمان‌بندی یا ترتیب اجرا (scheduling) ارائه دهیم به‌طوری‌که متوسط زمان پاسخ (average response time) حداقل شود. زمان پاسخ کار t_i برابر است با مدت زمان انتظار این کار باضافه‌ی d_i.

می‌توان به‌طور شهودی دید که اگر کارها را به‌ترتیب افزایشی زمان سرویس آن‌ها در صف قرار دهیم، در مجموع زمان‌های پاسخ کمینه می‌شود. برای اثبات، فرض می‌کنیم که $d_1 \leq d_2 \leq \cdots d_n$

لم ۹. برای مسئله‌ی زمان‌بندی بهینه‌ی یک پردازنده، یک راه‌حل بهینه وجود دارد که در آن اولین کاری که از پردازنده سرویس می گیرد، کار t_1 است.

اثبات. اثبات با برهان خلف: فرض کنید که اولین کار t_i باشد ($i > 1$). بنابراین کار t_1 بین کارهای دیگر در صف قرار می‌گیرد. فرض کنید که t_j کاری است که قبل از t_1 انجام می‌شود. نشان می‌دهیم که با تعویض دو کار t_1 و t_j مجموع زمان‌های پاسخ بدتر نمی‌شود.

بدیهی است که تعویض دو کار مجاور تأثیری بر زمان پاسخ کارهای دیگر ندارد. بنابراین فقط مجموع زمان‌های پاسخ کارهای t_1 و t_j مهم است. اگر D زمان اتمام کار جلوی کار t_j باشد. مجموع زمان‌های پاسخ این دو کار در حالت اول برابر است با $A = D + d_j + D + d_j + d_1 = 2D + 2D_j + d_1$ و در حالت دوم برابر است با $B = 2D + 2D_1 + d_j$. بدیهی است که $A - B = d_j - d_1 \geq 0$ (چون d_1 کم‌ترین زمان اجرای کارهاست.)

با این‌ترتیب، روشن است که کار t_1 را می‌توان همواره با کار جلوی‌اش تعویض کرد و هر بار معیار مورد نظر بهتر شود (یا بدتر نشود) تا آن‌جا که این کار اولین کار صف بشود. پس یک راه‌حل بهینه وجود دارد که t_1 اولین کار سرویس گیرنده است. □

با داشتن دو ویا چند پردازنده نیز به همین ترتیب عمل می کنیم. اگر درمسئله‌ی فوق روابط پیش‌نیازی وجود داشته باشد، در حالت کلی مسئله NP-Complete است ولی برای یک یا دو پردازنده راه‌حل چندجمله‌ای وجود دارد.

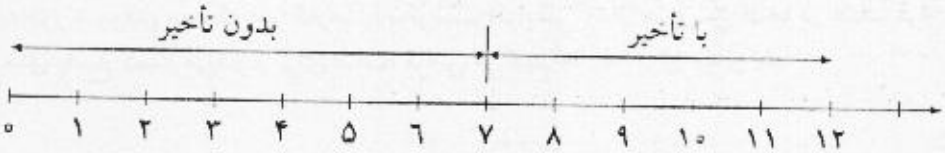
۲.۴.۶ زمان‌بندی کارها با جریمه‌ی تاخیر

n کار با شماره های ۱ تا n داده شده‌اند که زمان اجرای هر کدام ۱ واحد زمان است. d_i مهلت انجام کار i است و اگر این کار بعد از زمان d_i به‌اتمام برسد جریمه‌ای برابر W_i به آن تعلق می‌گیرد. فرض می‌کنیم که مهلت‌ها اعداد صحیح هستند. هدف تعیین یک زمان‌بندی برای اجرای همه‌ی این کارهاست به‌طوری‌که مجموع جریمه‌ها کمینه شود.

مشاهدات ما از مسئله: فرض می کنیم زمانبندی بهینه را داریم . این زمانبندی را به‌صورت شکل زیر را در نظر میگیریم که در آن هر کار در یکی از بازه‌های زمانی به‌اندازه‌ی ۱ واحد تخصیص داده شده است.

چون به هر کار با تأخیر فقط یک مقدار جریمه تعلق می‌گیرد و میزان تأخیر در این جریمه تأثیر ندارد، روشن است که می‌توان در زمان‌بندی بهینه (یا هر زمان‌بندی داده‌شده) همه‌ی کارهای با تأخیر (late tasks) را در انتهای زمان‌بندی و کارهای بدون تأخیر (early tasks) را در ابتدای آن انجام داد بدون آن‌که در میزان جریمه تغییری داده شود. این کار با تکرار عمل مجاز جابه‌جایی یک کار بدون تأخیر، مانند t_a با یک کار با تأخیر که بلافاصله قبل از t_a اجرا می‌شود انجام داد.

هم‌چنین در زمان‌بندی داده‌شده می‌توان کارهای بدون تاخیر را بر حسب مهلت‌هایشان (از کوچک به بزرگ) مرتب کرد. زیر هر دو کار بدون تأخیر و مجاور t_a و t_b که t_b بلافاصله بعد از t_a اجرا شود، ولی داشته باشیم $d_a > d_b$ را می‌توان با هم جابه‌جا کرد و هر دو کار هنوز قبل از مهلت‌شان اجرا شوند (با توجه به این‌که t_b قبل از



زمان d_b اجرا می‌شود و داریم $d_a \geq d_b + 1$ پس با اجرای t_a یک واحد زمانی دیر تر هنوز آن کار قبل از مهلتش اجرا می‌شود. اجرای زودتر t_b مشکلی ایجاد نمی‌کند.)

با این مشاهدات الگوریتم حریصانه‌ی زیر را برای این مسئله پیشنهاد می‌کنیم:

۱. کارها را به‌ترتیب جریمه‌هایشان از بزرگ به کوچک مورد بررسی قرار می‌دهیم . فرض کنید کار t_i را انتخاب کرده‌ایم.

۲. آخرین بازه‌ی زمانی‌ای را پیدا می‌کنیم که t_i را بتوانیم در آن قبل از مهلتش انجام دهیم. برای این کار از بازه‌ی $[d_i - 1, d_i]$ آغاز و بازه‌های سمت چپ آن‌را به‌ترتیب راست به چپ مورد بررسی قرار می‌دهیم. سمت راست‌ترین بازه‌ی خالی محل قرار گرفتن کار t_i است.

۳. اگر بازه‌ی زمانی خالی برای اجرای بدون تأخیر برای کار t_i پیدا نشود، این کار با تأخیر انجام می‌شود و بعد از تعیین همه‌ی کارهای بدون تأخیر زمان‌بندی می‌شود.

لم ۱۰. الگوریتم حریصانه‌ی ارائه‌شده یک جواب بهینه برای مسئله‌ی فوق به‌دست می‌آورد.

اثبات. برای اثبات، باید زیر مسئله را به‌درستی تعریف کنیم. فرض می‌کنیم $W_1 \geq W_2 \geq \cdots \geq W_n$. زیرمسئله شامل کارهای t_i تا t_n است که باید در یک بازه‌ی زمانی nتایی قرار داده‌شوند. با توجه به این‌که قبلاً تکلیف کارهای t_1 تا t_{i-1} مشخص شده است، فرض می‌کنیم که تعدادی (حداکثر برابر $i - 1$ عدد) از بازه‌ها قبلاً پر شده‌اند و قابل استفاده نیستند.

نشان می‌دهیم که اگر امکان اجرای کار t_i قبل از مهلتش باشد، یک راه‌حل بهینه وجود دارد که در آن t_i بدون تأخیر اجرا می‌شود.

فرض کنید که یک راه حل بهینه وجود دارد که در آن t_i با تأخیر اجرا می‌شود. کاری که در بازه‌ی $[d_{i-1}, d_i]$ قرار دارد را t_j بنامید. می‌دانیم که $d_j > d_i$ پس t_j بدون تأخیر اجرا می‌شود. با جابه‌جایی t_i و t_j در زمان‌بندی جدید t_j با تأخیر اجرا می‌شود که با توجه به این‌که می‌دانیم $W_i \geq W_j$ مجموع جریمه‌ها بدتر نمی‌شود. اگر t_j هم بدون تأخیر اجرا شود، وضعیت بهتر می‌شود.

با تخصیص t_i حداکثر یک بازه‌ی جدید اشغال می‌شود و با کم‌شده این کار، یک زیرمسئله‌ی کوچک‌تر ایجاد می‌شود که به‌همین روش حل می‌شود. □

مثال

کار	۱	۲	۳	۴	۵	۶	۷
d_i	۴	۲	۴	۳	۱	۴	۶
W_i	۷۰	۶۰	۵۰	۴۰	۳۰	۲۰	۱۰

مجموع جریمه‌هایی که تعلق می‌گیرد ۵۰ است.

۵.۶ الگوریتم هافمن

فایلی حاوی n نویسه داده شده است. می‌خواهیم برای هر نویسه کدی طراحی کنیم به‌طوری‌که با استفاده از این کدها (به‌جای کدهای مثلاً ۸ بیتی قبلی) اندازه‌ی فایل جدید (برحسب بیت) کمینه شود. تعداد بیت‌های کدهای طراحی شده می‌تواند متفاوت و دل‌خواه باشد. این بهترین روش برای فشرده‌سازی فایل‌هاست با این شرط که هر نویسه کدگذاری شود. روش‌های دیگری وجود دارند که زیر رشته‌ها را به‌صورتی به کد تبدیل می‌کنند که ممکن است از این الگوریتم بهتر عمل کند.

در این مسئله با توجه به این‌که فایل داده‌شده است، بنابراین تعداد تکرار و یا احتمال وقوع هر نویسه را داریم. این الگوریتم را در مواردی مثلاً ارسال اطلاعات ترتیبی و به‌صورت «برخط» (online) بر روی شبکه در صورتی‌که احتمال وقوع نویسه‌ها با تقریب خوبی داشته باشیم، بدون آن‌که کل فایل داده‌شده باشد نیز استفاده می‌شود. توجه کنید که نویسه‌ها لزوماً نباید نویسه‌های مثلاً اسکی باشند.

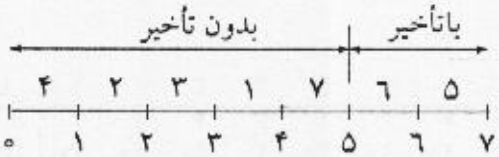
مسئله به‌صورت انتزاعی از قرار زیر است. n نویسه‌ی a_1 تا a_n و احتمال وقوع f_i برای هر a_i داده شده است ($\sum_i f_i = 1$). می‌خواهیم به‌هر نویسه‌ی a_i کدی به‌طول ℓ_i نسبت دهیم به‌طوری که متوسط طول کدها (یعنی $\sum_{i=1}^{n} f_i\ell_i$) کمینه شود.

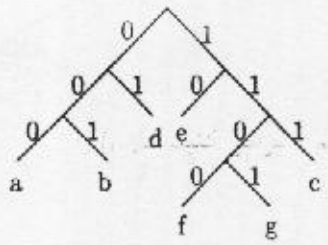
برای آن‌که از فایل دریافت‌شده‌ی کدگذاری شده و نیز جدول کدها بتوانیم فایل اصلی را به‌دست آوریم، روشن است که نباید هیچ کدی زیر رشته‌ی کد دیگر باشد. به‌عبارت دیگر کدها نباید خاصیت پیشوندی داشته‌باشند. با وجود این خاصیت می‌توان کدها را به‌صورت یک درخت دودویی کامل مدل کرد که در آن برگ‌های سمت چپ و راست هر گره به‌ترتیب دارای برچسب‌های ۰ و ۱ هستند و هر نویسه یک برگ این درخت است به‌طوری‌که بیت‌های مسیر از ریشه به آن نویسه کد آن نویسه است. به‌این درخت، درخت هافمن می‌گویند. مثلاً در شکل ۱.۶ کدهای a تا g به‌ترتیب برابرند با: ۰۰۰، ۰۰۱، ۱۱۱، ۰۱، ۱۰، ۱۱۰۰، و ۱۱۰۱. توجه کنید که اگر کدی پیشوند کد دیگر باشد، دیگر نمی‌تواند در درخت برگ باشد.

روشن است که متوسط عمق برگ‌های درخت هافمن همان متوسط طول کدهاست.

الگوریتم هافمن با دریافت احتمال وقوع نویسه‌ها، یک درخت هافمن با حداقل متوسط عمق برگ‌ها ایجاد می‌کند. الگوریتم در ابتدا یک گره برای هر نویسه‌ی ورودی ایجاد می‌کند و برای هر گره احتمال وقوع آن نویسه‌را نسبت می‌دهد. به‌صورت حریصانه هر بار دو گره‌ی x و y با کم‌ترین احتمال وقوع (f_x و f_y) را پیدا می‌کند، این دو گره را فرزندان یک گره‌ی جدید (به نام مثلاً z) با احتمال وقوع $f_x + f_y$ می‌کند. x و y را حذف و به‌جای آن‌ها z را اضافه می‌کند. الگوریتم این کار را تکرار می‌کند تا این که تنها یک گره با احتمال وقوع ۱ که همان ریشه‌ی درخت است حاصل شود.

پیاده‌سازی الگوریتم به‌صورت زیر است:





شکل ۱.۶: کدگذاری درست را می‌توان با یک درخت دودویی نمایش داد.

```
HUFFMAN(C)
Ceate an empty Q
for all c ∈ C
    do ALLOCATE-NODE (x)
        f[x] ← f[c] INSERT (x, Q)
for i ← 1 to n − 1
    do z ← ALLOCATE-NODE()
        x ← Left[z] ← EXTRACT-MIN (Q)
        y ← Left[z] ← EXTRACT-MIN (Q)
        f(z) ← f(x) + f(y)
        INSERT (Q, z)
```

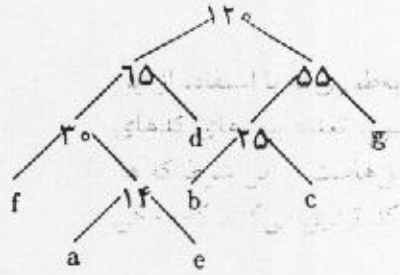
بهترین داده‌ساختار برای Q یک صف اولویت است که اعمال فوق را با $O(\lg n)$ انجام می‌دهد. پس الگوریتم فوق از $O(n \lg n)$ است.

مثال

فایلی به‌اندازه‌ی ۱۲۰ حاوی نویسه‌های زیر است.

نویسه	فرکانس تکرار
a	۵
b	۱۳
c	۱۲
d	۳۵
e	۹
f	۱۶
g	۲۰

اعمال الگوریتم هافمن درخت شکل ۲.۶ را ایجاد می‌کند. کدها از روی درخت به‌دست می‌آید. توجه کنید که کدهای بهینه یکتا نیستند.



شکل ۲.۶: درخت هافمن برای مثال.

لم ۱۱. الگوریتم هافمن به‌درستی یک درخت با حداقل متوسط عمق برگ‌ها ایجاد می‌کند.

اثبات. مانند دیگر الگوریتم‌های حریصانه نشان می‌دهیم که اولین انتخاب درست است. فرض کنید که x و y دو گره هستند که دارای کوچک‌ترین احتمال‌های وقوع (f_x و f_y) هستند. نشان می‌دهیم که یک درخت بهینه وجود دارد که در آن x و y برادرند و در بیشترین عمق در درخت قرار دارند.

باز از برهان خلف استفاده می‌کنیم. فرض کنید که یک کد بهینه وجود دارد که مطابق ادعای ما نیست و در آن دو گره‌ی دیگر برای نویسه‌های مثلاً b و c برادر هم هستند و بیشترین عمق را دارند. بدون از دست دادن کلیت مسئله می‌توانیم فرض کنیم که $f(b) < f(c)$ و هم‌چنین $f(x) < f(y)$. بنابراین خواهیم داشت: $f(x) <= f(b)$ و $f(y) <= f(c)$ (چون طبق فرض x و y کم‌ترین احتمال وقوع را دارند.)

نشان می‌دهیم که با جابه‌جا کردن برچسب‌های گره‌های x و b متوسط عمق درخت بدتر نمی‌شود. می‌دانیم که عمق x از b بیشتر نیست و چون $f(x) <= f(b)$ بنابراین ادعای ما درست است. به‌دلیل مشابه می‌توانیم برچسب‌های y و c را با هم تعویض کنیم. پس یک درخت بهینه حاصل شد که در آن x و y برادر هم و در بیشترین عمق قرار دارند. بنابراین کدهای x و y یک طول دارند، پس می‌توانیم هر دو را با یک نویسه‌ی z با احتمال وقوع $f(x) + f(y)$ تعویض کنیم. این همان کاری است که الگوریتم انجام می‌دهد و یک زیر مسئله با یک نویسه‌ی کم‌تر ایجاد می‌شود که آن‌را نیز به‌همین روش می‌توان حل کرد. □

۶.۶ الگوریتم حریصانه‌ی تقریبی برای مسئله‌ی بسته‌بندی

n شیء داده شده‌اند که حجم شیء iام برابر v_i است. v_iها اعداد حقیقی بین صفر و یک هستند. می‌خواهیم این اشیاء را در صندوقچه‌هایی که حجم هرکدام از آن‌ها برابر با ۱ است، بسته‌بندی کنیم به‌طوری‌که تعداد کل صندوقچه‌ها حداقل شود. این مسئله به‌نام بسته‌بندی (bin packing) معروف است. ساده ترین الگوریتم حریصانه که برای این مسئله به‌نظر می‌رسد، به این صورت است:

برای قراردادن شیء iام، اگر یکی از صندوقچه‌هایی غیر خالی به اندازه‌ی v_i جای خالی داشت، شیء iام را در آن قرار می‌دهیم و گرنه، آن را در یک صندوقچه‌ی جدید قرار می‌دهیم.

اما متأسفانه این الگوریتم در بسیاری از موارد اشتباه می‌کند. مثلاً اگر حجم اشیاء برابر ۰/۶، ۰/۳، ۰/۳، ۰/۲، ۰/۲، ۰/۲، ۰/۲ باشند، این الگوریتم این اشیاء را در ۳ صندوقچه بسته‌بندی می‌کند: در صندوقچه‌ی اول اشیاء با حجم‌های ۰/۶ و ۰/۳، در صندوقچه‌ی دوم اشیاء با حجم‌های ۰/۳، ۰/۲، ۰/۲، و ۰/۲ و در صندوقچه‌ی سوم یک شیء با حجم ۰/۲ قرار گرفته است. درحالی که می‌توانستیم این کار را با دو صندوقچه نیز انجام دهیم، به این صورت که در صندوقچه‌ی اول اشیاء با حجم‌های ۰/۲، ۰/۲ و ۰/۲ و در صندوقچه دوم اشیاء با حجم‌های ۰/۳، ۰/۳، ۰/۲ و ۰/۲ را قرار دهیم. دیدیم که این الگوریتم ممکن است جواب بهینه را به‌دست ندهد. اما اکنون یک سوال پیش می‌آید: جوابی که این الگوریتم به ما می‌دهد تاچه حد نزدیک به جواب بهینه است؟ قضیه ی زیر تا حدودی به این پرسش پاسخ می‌دهد:

قضیه ۹. اگر حداقل تعداد صندوقچه‌های لازم برای بسته‌بندی اشیاء داده‌شده برابر با OPT باشد، الگوریتم فوق این اشیاء را با حداکثر ۲×OPT صندوقچه انجام می‌دهد.

اثبات. در الگوریتم ما ممکن نیست که بیش از یک صندوقچه باقی بماند که کم‌تر از نصف آن پر شده باشد. (چرا؟) بنابراین اگر $S = \sum_i v_i$، الگوریتم ما از حداکثر $\lceil 2S \rceil$ عدد صندوقچه استفاده می‌کند. از طرف دیگر هرطوری که اشیاء را بسته‌بندی کنیم، حداقل به S تا صندوقچه نیاز داریم. بنابراین $OPT >= S$ بنابراین ثابت کردیم که الگوریتم ما حداکثر از ۲×OPT عدد صندوقچه استفاده می‌کند. □ بنابراین الگوریتم حریصانه‌ای که برای این مسئله ارائه دادیم، زیاد هم بد نیست! جوابی که این الگوریتم به‌ما می‌دهد حداکثر دو برابر مقدار بهینه است. در واقع با استفاده از یک روش پیچیده‌تر می‌توان ثابت کرد که این الگوریتم حداکثر ۱/۷ برابر مقدار بهینه را بدست می‌آورد. البته مثال‌هایی هم می‌توان ساخت که الگوریتم ما دقیقاً ۱/۷ برابر مقدار بهینه را به‌دست آورد. بنابراین با استفاده از این الگوریتم نمی‌توانیم بیشتر از این پیش برویم.

حالا سعی می‌کنیم که کمی الگوریتم فوق را اصلاح کنیم تا نتیجه‌ی بهتری بگیریم. یک کار عاقلانه این است که ابتدا اشیا ی را به‌ترتیب نزولی حجم‌شان مرتب کنیم و سپس همان الگوریتم را در مورد آن‌ها به‌کار ببریم. به‌نظر می‌رسد که با این اصلاح، الگوریتم قدری بهتر شده است، اما هنوز خیلی وقت‌ها اشتباه می‌کند. ثابت شده است که جوابی که این الگوریتم به ما می‌دهد از OPT+۴ بیشتر نیست. البته اثبات این موضوع چندان ساده نیست.

البته ذکر این نکته ضروری است که مسئله‌ی بسته‌بندی یک مسئله‌ی NP-Complete است و انتظار ارائه‌ی یک راه‌حل حتی چندجمله‌ای برای آن بیهوده است.

۷.۶ تمرین‌ها

۱. ثابت کنید که الگوریتم خرد کردن پول اگر سکه های موجود ۱، ۲ و ۵ تومانی باشد، جواب بهینه را به‌دست می‌آورد.

۲. ثابت کنید که اگر سکه های موجود ۱، C، ... و تومانی باشند الگوریتم خرد کردن پول درست کارمی‌کند.

۳. فرض کنید که سکه هایی با ارزش تومان وجود دارند و از هر کدام از آنها نیز به تعداد نامحدودی در دسترس داریم. می‌خواهیم با استفاده از این سکه ها L تومان پول را خرد کنیم به طوری که تعداد سکه هایی که استفاده می‌کنیم حداقل باشد. را برابر با حداقل تعداد سکه های لازم برای خرد کرد L تومان پول می‌گیریم. ثابت کنید که: با استفاده از روابط یک الگوریتم برای خرد کردن پول طراحی کنید.

۴. در مسئله کوله‌پشتی، فرض کنید که این شرط را داریم که: ترتیب بسته‌ها وقتی که برحسب وزن‌شان به‌طور صعودی مرتب شوند، همان ترتیبی است که اگر آن‌ها را برحسب ارزش‌شان به‌طور نزولی مرتب کنیم. ثابت کنید که اگر این شرط برقرار باشد، الگوریتم حریصانه درست کار می‌کند.

۵. در مسئله کوله و یک، فرض کنید که n نوع جنس داریم که وزن و ارزش جنس iام به ترتیب برابر و است. از هرکدام از این اجناس نیز فقط یک عدد موجود است. را به این صورت تعریف می‌کنیم: اگر تنها K نوع جنس اول، دوم،... و Kام را در اختیار داشته باشیم، مقدار حداکثر ارزش اجناسی که می‌توان از بین این K جنس انتخاب کرد و در کوله پشتی با ظرفیت W جا داد با است. رابطه ی بازگشتی زیر را برای ثابت کنید و با استفاده از آن الگوریتمی برای حل مسئله کوله پشتی صفر و یک بیابید:

۶. فرض کنید که در یک گراف وزندار کوچکترین زیردرخت فراگیر را بدست آورده‌ایم. حالا می‌خواهیم زیردرخت فراگیری از این گراف را بدست آوریم که پس از کوچکترین زیردرخت فراگیر، از بقیه ی زیردرختهای فراگیر این گراف کم وزن تر باشد. به عبارت دیگر می‌خواهیم زیردرخت فراگیری را بدست آوریم که از نظر کم وزن بودن در مرتبه ی دوم قرار گرفته است. الگوریتم برای حل این مسئله بیابید.

۷. مسئله برنامه ریزی چند پردازنده (Multiprocessor Soheduling) به این صورت مطرح می‌شود: فرض کنید K تا کامپیوتر داریم که سرعت همه‌ی آن‌ها باهم برابر است. می‌خواهیم n تا برنامه را روی این کامپیوترها اجرا کنیم. زمان اجرای برنامه ی iام بر روی هرکدام از این کامپیوترها برابر با t ثانیه است. می‌خواهیم ببینیم که باید هر کدام از برنامه ها را روی کدام کامپیوتر اجرا کنیم که زمان اجرای همه ی برنامه ها، یعنی اولین لحظه‌ای که اجرای تمام برنامه ها به پایان رسیده است، حداقل باشد. یک الگوریتم حریصانه برای این مسئله پیشنهاد می‌کنیم: برنامه ها را به ترتیب نزولی زمان اجرایشان مرتب می‌کنیم. سپس اولین برنامه را به اولین کامپیوتر، دومی را به دومین کامپیوتر،... و Kامین برنامه را به Kامین کامپیوتر اختصاص می‌دهیم. پس از این اولین کامپیوتری که کارش تمام شد، اولین برنامه‌ای که هنوز به هیچ کامپیوتری اختصاص داده نشده است را به آن کامپیوتر می‌دهیم. این کار را تا جایی ادامه می‌دهیم که تمامی برنامه ها بر روی کامپیوترها اجرا شوند. اولا، ثابت کنید که الگوریتم فوق درست نیست ! یعنی مثال نقضی پیدا کنید که الگوریتم فوق نتواند جواب بهینه را برای آن بدست آورد. ثانیا، ثابت کنید که اگر جواب بهینه برای مسئله برابر با T باشد، الگوریتم فوق جوابی برای مسئله بدست می‌آورد که زمان آن از ۲T بیشتر نیست.

۸. یک گراف G داده شده است. می‌خواهیم راسهای این گراف را رنگ‌آمیزی کنیم به طوری که شرایط زیر برقرار باشند: الف - هیچ دو راس مجاوری هم رنگ نباشند. ب - تعداد رنگهایی که استفاده می‌کنیم حداقل باشد. ساده ترین الگوریتم حریصانه‌ای که می‌توان برای این مسئله پیشنهاد کرد به این صورت است: ابتدا یک ترتیب دلخواه مانند راس رئوس گراف در نظر می‌گیریم. راس ۱ با رنگ شماره ی ۱ رنگ آمیزی می‌کنیم و پس از آن برای i از ۲ تا n به این صورت جلو می‌رویم: برای رنگ آمیزی راس i از اولین رنگی که می‌توانیم استفاده می‌کنیم، یعنی از اولین رنگی که در مجاورت این راس ظاهر نشده است. اگر چنین رنگی موجود نبود، یک رنگ جدید را به مجموعه ی رنگهااضافه می‌کنیم و از آن برای رنگ آمیزی راس استفاده می‌کنیم. به سادگی می‌توانید

مثال نقضی برای الگوریتم فوق بیابید. (در واقع هیچ الگوریتم حریصانه‌ای وجود ندارد که جواب بهینه را برای این مسئله پیدا کن.) برابر با تعداد رنگهایی می‌گیریم که الگوریتم فوق برای رنگ آمیزی گراف از آن استفاده می‌کند، اگر ترتیب اولیه ی O را برای رئوس انتخاب کرده باشیم. همچنین C (G) را برابر با حداقل تعداد رنگهایی که برای رنگ آمیزی گراف G لازم است می‌گیریم ثابت کنید که: الف - برای هر گراف G، ترتیب O وجود دارد که شود. به عبارت دیگر در هر گراف اگر ترتیب اولیه O، ترتیب مناسبی باشد، الگوریتم درست عمل می‌کند. ب - برای هر عدد حقیقی ، گراف G و ترتیب O برای رئوس آن وجود دارد که نسبت از بیشتر شود. به عبارت دیگر الگوریتم حریصانه می‌تواند به مقدار دلخواهی اشتباه کند.

۹. گراف بدون جهت G داده شده است. مسئله پوشش راسی (Vertex-Cover) به این صورت مطرح می‌شود: کوچکترین زیرمجموعه ی C از مجموعه ی رئوس گراف را پیدا کنید که برای هر یک از یالهای گراف، حداقل یکی از دو سر این یال در مجموعه ی C باشد. یک الگوریتم مکاشفه‌ای ساده برای پیدا کردن کوچکترین پوشش راسی به این صورت است: ۱) C را مساوی مجموعه ی تهی و 'E را مساوی مجموعه یالهای گراف قرار بده. ۲) یال دلخواه uv از مجموعه ی 'E را انتخاب کن ۳) دو راس u و v را به مجموعه ی C اضافه کن ۴) تمامی یالهایی که لااقل یکی از دو سرشان u یا v باشد را از مجموعه ی 'E حذف کن. ۵) اگر 'E مجموعه تهی نیست، به مرحله ی ۲ برگرد. ۶) مجموعه ی C را به عنوان خروجی الگوریتم برگردان.

ابتدا مثال نقضی پیدا کنید که الگوریتم فوق درست کار نکند و سپس ثابت کنید که برای هر گراف G جوابی که الگوریتم فوق می‌دهد حداکثر دوبرابر مقدار بهینه عضو دارد.

۱۰. راننده‌ای می‌خواهد با ماشینش از یک شهر به شهر دیگری سفر کند. ماشین با باک بنزین پر می‌تواند n کیلومتر حرکت کند. در مسیر مسافرت این راننده k تا پمپ بنزین وجود دارد. (پمپ بنزین اول در شهر مبدأ و پمپ بنزین k ام در شهر مقصد است) فاصله‌ی پمپ بنزین iام با پمپ بنزین $i+1$ام برابر با کیلومتر است. اگر اعداد n و و... و داده شده باشند، می‌خواهیم حداقل تعداد پمپ بنزین هایی را پیدا کنیم که راننده می‌تواند با پرکردن باک بنزین ماشینش در این پمپ بنزینها مسافرت خود را انجام دهد یک الگوریتم حریصانه برای حل این مسئله پیشنهاد کنید و آن را ثابت کنید.

۱۱. n نقطه با طول‌های و و... و روی محوز طول‌ها داده شده‌اند. الگوریتمی طراحی کنید که کمترین تعداد بازه‌های به‌طول ۱ را پیدا کند که تمامی این نقاط را بپوشانند. درستی الگوریتم خود را ثابت کنید.

۱۲. در یک فروشگاه n مشتری منتظر هستند تا کارشان انجام شود. کار مشتری شماره ی i به‌اندازه‌ی d_i دقیقه طول می‌کشد. یک نفر کارمند متصدی انجام کار این مشتری‌هاست و در هر لحظه می‌تواند کار فقط یک مشتری را انجام دهد. می‌خواهیم ببینیم که کارمند این فروشگاه باید کار این مشتری‌ها را به چه ترتیبی انجام دهد تا مجموع زمان معطل شدن این مشتری ها کمینه شود. (زمان معطل شدن یک مشتری برابر با زمانی‌است که انجام کار او به اتمام می‌رسد.) یک الگوریتم حریصانه برای حل این مسئله پیدا کنید و سپس درستی الگوریتم خود را اثبات نمایید.

۱۳. n برنامه با طول‌های l_1 l_2 ، dots، و l_m باید روی یک نوار مغناطیسی ذخیره شوند. می‌دانیم که احتمال این‌که بخواهیم برنامه ی شماره ی i را از روی نوار بخوانیم برابر با p_i است. اگر برنامه‌ها روی نوار مغناطیسی به ترتیب شماره‌هایشان ذخیره شده باشند (یعنی برنامه‌ی اول در ابتدای نوار، برنامه‌ی دوم پس از آن، و ...) مقدار زمانی که طول می‌کشد تا بتوانیم برنامه‌ی شماره‌ی i را از روی نوار بخوانیم، متناسب با $\sum_{k=1}^{i} l_k$ است. بنابراین زمان خواندن یک برنامه از روی نوار به طور متوسط متناسب با $T = \sum_{i=1}^{n} p_i \sum_{k=1}^{i} l_k$. هدف این است که

ترتیب برنامه‌ها روی نوار مغناطیسی را طوری تعیین کنیم که T حداقل شود.

الف - با ارائه‌ی یک مثال ثابت کنید که اگر برنامه‌ها را به‌ترتیب صعودی طول‌شان روی نوار ذخیره کنیم، ترتیب ذخیره‌شدن برنامه‌ها لزوماً بهینه نیست.

ب - با ارائه‌ی یک مثال ثابت کنید که اگر برنامه‌ها به ترتیب نزولی طول‌شان روی نوار ذخیره شوند، ترتیب ذخیره شدن برنامه‌ها لزوماً بهینه نیست.

ج - ثابت کنید که اگر برنامه‌ها را به‌ترتیب نزولی مقدار p_i بر روی نوار ذخیره کنیم، مقدار T مینیمم می‌شود، یعنی این ترتیب، بهینه است.