روش حریصانه (Greedy) در طراحی الگوریتم‌ها

ویژگی‌ها

- مسئله بهینه‌سازی (optimization) است.
- برای حل بهینه‌ی مسئله باید زیرمسئله‌های آن‌را نیز به‌صورت بهینه حل کرد (optimal subproblem).
- انتخاب حریصانه (greedy choice) دراین‌گونه مسائل بهترین انتخاب است و عوض نمی‌شود.

```
GREEDY(C)
      ▷ C: the set of all candidates
      ▷ S is the solution (initially empty)
  1  while  not IsSolution(S)  and  not IsEmpty(C)
  2         do x ← an element in C maximizing Select(x)
  3            C ← C − {x}
  4            if IsFeasible (S ∪ {x})
  5              then S ← S ∪ {x}
  6  if IsSolution(S)
  7    then returnS
  8    elsereturn No Solution
```

انتخاب فعالیت‌ها

- N فعالیت مختلف $t_۱$، $t_۲$، $\cdots$، و t_n
- یک منبع غیرقابل اشتراک
- هر فعالیت زمان شروع s_i و زمان پایان f_i
- هدف: پیدا کردن بیشترین تعداد این فعالیت‌هاست که بتوانند از منبع استفاده کنند.

راه‌حل

فعالیت‌ها را به‌ترتیب پایان زمان‌شان انتخاب کنیم و پس از انتخاب یک فعالیت، همه‌ی فعالیت‌های متضاد با آن را حذف کنیم

اثبات درستی

مراحل کلی اثبات:

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.

فرض می‌کنیم که $f_۱ \leq f_۲ \leq \cdots \leq f_k$.

لم ۱ یک راه‌حل بهینه برای مسئله وجود دارد که شامل کار $t_۱$ است.

مثال

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

مسئله‌ی خرد کردن پول

می‌خواهیم n تومان را با سکه‌های ۱، ۲ و ۵ تومانی خرد کنیم تا مجموع تعداد سکه‌هایی که استفاده می‌کنیم حداقل شود.

اگر سکه‌ها ۱، ۴، و ۵ تومانی باشند، خرد کردن ۸ تومان

اگر سکه‌ها c^0، c^1، c^2، تا c^k برای $c > 1$ باشد می‌توان از الگوریتم حریصانه استفاده کرد.

مسئله‌ی کوله‌پشتی با بارهای قابل تقسیم

(fractional knapsack problem)

N عدد بار، وزن و ارزش بار iام به‌ترتیب برابر W_i و C_i کوله‌پشتی به‌اندازه‌ی M

می‌خواهیم کوله‌پشتی را کاملاً پر کنیم تا ارزش بارهای انتخابی بیشترین شود.

مجازهستیم که یک بار را به نسبت دل‌خواه به دو قسمت تقسیم کنیم.

بارها را به‌ترتیب ارزش در واحد وزن در کوله‌پشتی قرار می‌دهیم. فرض می‌کنیم $\frac{C_1}{W_1} \geq \frac{C_2}{W_2} \geq \cdots \frac{C_n}{W_n}$.

مسائل زمان‌بندی: حالت ساده

یک پردازنده، n عدد کار t_1، t_2، $\cdots$، و t_n در زمان صفر داده شده‌اند، به‌طوری‌که زمان مورد نیاز کار t_i برابر s_i است (service time).

می‌خواهیم یک زمان‌بندی 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.

اگر کارها را به‌ترتیب افزایشی زمان سرویس آن‌ها در صف قرار دهیم، در مجموع زمان‌های پاسخ کمینه می‌شود.

فرض: $d_1 \leq d_2 \leq \cdots d_n$.

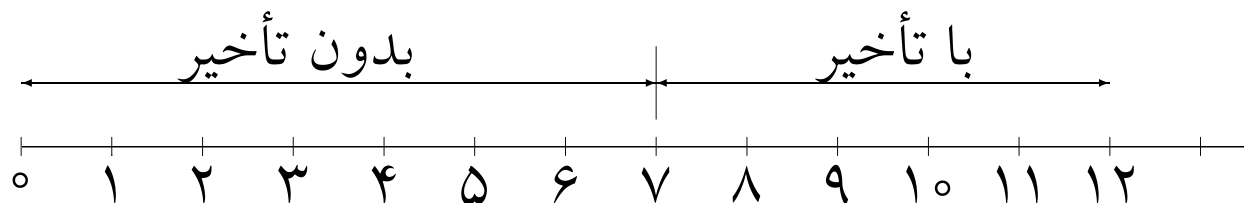
لم ۲ **برای مسئله‌ی زمان‌بندی بهینه‌ی یک پردازنده، یک راه‌حل بهینه وجود دارد که در آن اولین کاری که از پردازنده سرویس می‌گیرد، کار t_1 است.**

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:



الگوریتم

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 $[d_i - ۱, d_i]$ 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 ۳ الگوریتم حریصانه‌ی ارائه‌شده یک جواب بهینه برای مسئله‌ی فوق به‌دست می‌آورد.

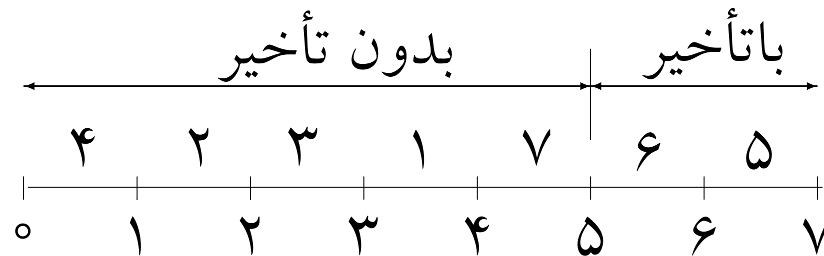
اثبات. فرض می‌کنیم $W_1 \geq W_2 \geq \cdots \geq W_n$.

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. □

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.

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.

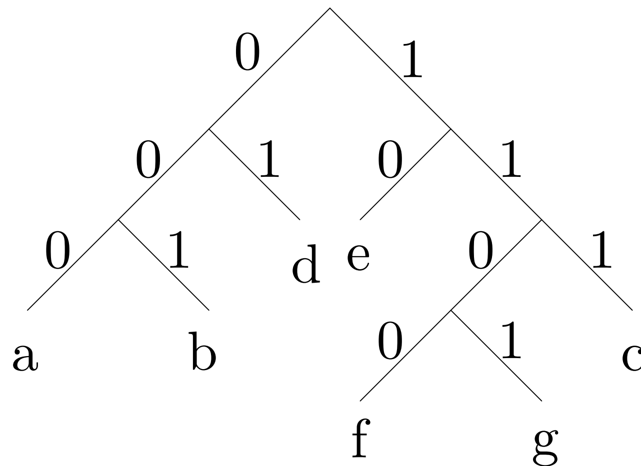
فشرده‌سازی

به‌صورت انتزاعی: n نویسه‌ی $a_۱$ تا a_n و احتمال وقوع f_i برای هر a_i داده شده است ($\Sigma_i f_i = ۱$). می‌خواهیم به‌هر نویسه‌ی a_i کدی به‌طول ℓ_i نسبت دهیم به‌طوری که متوسط طول کدها (یعنی $\Sigma_{i=۱}^{n} f_i \ell_i$) کمینه شود.

ادغام فایل‌ها

خاصیت مهم کدها

کدها نباید پیشوندی هم باشند.



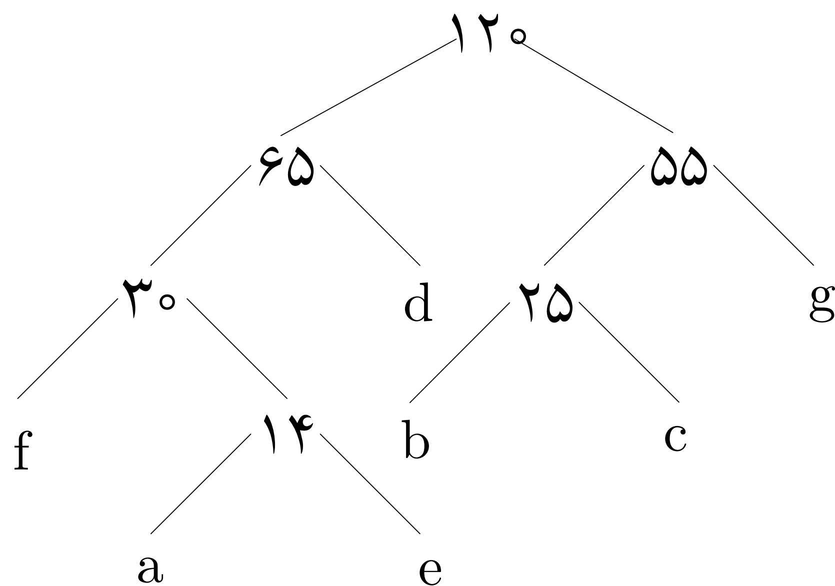
متوسط عمق برگ‌های درخت هافمن همان متوسط طول کدهاست.

```
HUFFMAN(C)
 1  Ceate an empty Q
 2  for all c ∈ C
 3      do ALLOCATENODE (x)
 4          f[x] ← f[c]
 5          INSERT (x, Q)
 6  for i ← 1 to n − 1
 7      do z ← AllocateNode()
 8          x ← Left[z] ← ExtractMin (Q)
 9          y ← Right[z] ← ExtractMin (Q)
10          f(z) ← f(x) + f(y)
11          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



لم ۴ الگوریتم هافمن به‌درستی یک درخت با حداقل متوسط عمق برگ‌ها ایجاد می‌کند.

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.

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.

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:

درحالی که می‌توانستیم این کار را با دو صندوقچه نیز انجام دهیم

قضیه‌ی ۱ 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 $S = \Sigma_i v_i$، الگوریتم ما از حداکثر $\lceil ۲S \rceil$ عدد صندوقچه استفاده می‌کند. از طرف دیگر هرطوری که اشیاء را بسته‌بندی کنیم، حداقل به S تا صندوقچه نیاز داریم. بنابراین OPT$\geq S$ بنابراین ثابت کردیم که الگوریتم ما حداکثر از ۲×OPT عدد صندوقچه استفاده می‌کند. □