به نام خدا

طراحی الگوریتم‌ها

آرش شفیعی

الگوریتم‌های حریصانه

الگوریتم‌های حریصانه

- مسئله‌های بهینه‌سازی به دنبال جواب بهینه در مجموعه‌ای از جواب‌ها برای یک مسئله می‌گردند. یک جواب بهینه جوابی است که در یک معیار اندازه‌گیری بهترین باشد. برای مثال یک جواب بهینه می‌تواند کوچک‌ترین، بزرگ‌ترین، کوتاهترین، بلندترین و غیره باشد.
- برنامه‌ریزی پویا یکی از روش‌ها برای حل مسئله‌های بهینه‌سازی است.
- الگوریتم‌های حریصانه [1] دسته‌ای دیگر از الگوریتم‌ها برای حل مسئله‌های بهینه‌سازی هستند.

[1] greedy algorithms

الگوریتم‌های حریصانه

- فرض کنید می‌خواهیم در مدت n روز به بیشترین دارایی ممکن دست پیدا کنیم. برای این کار کافی است که در هر روز بیشترین دارایی ممکن را کسب کنیم. بنابراین برای به دست آوردن بیشترین دارایی در مدت n روز باید در روز اول بیشترین دارایی را کسب کنیم و زیر مسئله‌ای که باید در گام بعد حل شود این است که چگونه در مدت $n-1$ روز بیشترین دارایی را کسب کنیم. پس در هر روز یک انتخاب حریصانه انجام می‌دهیم و آن انتخاب، کسب بیشترین دارایی در همان روز است. می‌دانیم که با این انتخاب حریصانه در مدت n روز بیشترین دارایی را کسب خواهیم کرد.

- یک الگوریتم حریصانه مسئله را از بالا به پایین حل می‌کند. برای حل یک مسئله به روش حریصانه در هر گام یک انتخاب بهینه انجام می‌شود و از یک مسئله یک زیرمسئله به دست می‌آید. این فرایند ادامه پیدا می‌کند تا زیرمسئله‌ای باقی نماند. در پایان جواب مسئله مجموعهٔ همهٔ انتخاب‌های بهینه است. به عبارت دیگر در هر گام یک انتخاب حریصانه برای به دست آوردن جواب بهینه صورت می‌گیرد و در پایان مجموعهٔ همهٔ انتخاب‌های حریصانه جواب مسئله است.

الگوریتم‌های حریصانه

- تنها برخی از مسئله‌های بهینه‌سازی را می‌توان به روش حریصانه حل کرد.
- برای مثال مسئله درخت جستجوی دودویی بهینه را در نظر بگیرید. اگر در هر گام برای ساختن درخت جستجوی دودویی بهینه از بین همهٔ کلیدها، کلیدی را به عنوان ریشه در نظر بگیریم که بیشترین احتمال وقوع را داشته باشد، درخت به دست آمده الزاما بهینه نیست.
- همانطور که مشاهده کردیم ریشهٔ یک درخت جستجوی دودویی بهینه ممکن است کلیدی باشد که بیشترین احتمال وقوع را نداشته باشد.

مسئله انتخاب فعالیت‌ها

- در مسئله انتخاب فعالیت‌ها [1] ، تعدادی فعالیت به طور همزمان می‌خواهند انجام شوند به طوری‌که این فعالیت‌ها از یک منبع مشترک استفاده می‌کنند و این منبع مشترک نمی‌تواند به طور همزمان توسط فعالیت‌ها استفاده شود. می‌خواهیم مجموعه‌ای از فعالیت‌ها را انتخاب کنیم، به طوری که بیشترین تعداد فعالیت‌ها بتوانند اجرا شوند.

- فرض کنید شما مسئول زمانبندی یک اتاق کنفرانس هستید. به شما مجموعهٔ $S = \{a_1, a_2, \cdots, a_n\}$ شامل n فعالیت داده شده است که می‌خواهند اتاق کنفرانس را رزرو کنند. در این اتاق در یک زمان تنها یک فعالیت می‌تواند صورت بگیرد.

[1] activity selection problem

مسئله انتخاب فعالیت‌ها

- هر فعالیت a_i یک زمان شروع s_i [1] و یک زمان پایان f_i [2] دارد به طوری‌که $0 \leqslant s_i < f_i < \infty$. اگر فعالیت a_i انتخاب شود، آنگاه این فعالیت می‌تواند در بازهٔ زمانی $[s_i, f_i)$ انجام شود. دو فعالیت a_i و a_j سازگار [3] گفته می‌شوند اگر بازه‌های زمانی $[s_i, f_i)$ و $[s_j, f_j)$ تداخل نداشته باشند. دو فعالیت تداخل ندارند اگر $s_i \geqslant f_j$ یا $s_j \geqslant f_i$. در مسئلهٔ انتخاب فعالیت‌ها، هدف این است که بیشترین تعداد فعالیت‌های سازگار از یک مجموعهٔ فعالیت‌ها انتخاب شوند.

- فرض می‌کنیم فعالیت‌ها بر اساس زمان پایان‌شان مرتب شده‌اند. به عبارت دیگر :

$$f_1 \leqslant f_2 \leqslant f_3 \leqslant \cdots \leqslant f_{n-1} \leqslant f_n$$

---

[1] state time
[2] finish time
[3] compatible

مسئله انتخاب فعالیت‌ها

- برای مثال مجموعه‌ای از فعالیت‌ها در جدول زیر را در نظر بگیرید.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

- مجموعهٔ $\{a_3, a_9, a_{11}\}$ مجموعه‌ای از فعالیت‌های سازگار است ولی بزرگ‌ترین مجموعهٔ فعالیت‌های سازگار نیست چراکه $\{a_1, a_4, a_8, a_{11}\}$ تعداد بیشتری فعالیت را شامل می‌شود. همچنین مجموعهٔ $\{a_2, a_4, a_9, a_{11}\}$ تعداد چهار فعالیت را شامل می‌شود.

مسئله انتخاب فعالیت‌ها

- در اینجا ابتدا سعی می‌کنیم مسئلهٔ انتخاب فعالیت‌ها را توسط برنامه‌ریزی پویا حل کنیم.
- فرض کنید S_{ij} مجموعه‌ای از فعالیت‌ها باشد که پس از اتمام فعالیت a_i آغاز و قبل از شروع فعالیت a_j تمام می‌شوند. فرض کنید می‌خواهیم حداکثر فعالیت‌های سازگار S_{ij} را پیدا کنیم و فرض کنید A_{ij} مجموعه‌ای است که شامل حداکثر تعداد فعالیت‌های سازگار از مجموعهٔ S_{ij} است.
- حال فرض کنید a_k یکی از فعالیت‌ها در مجموعهٔ A_{ij} است. در اینجا دو زیر مسئله داریم : مسئلهٔ یافتن حداکثر فعالیت‌های سازگار در S_{ik} (که شامل فعالیت‌هایی می‌شود که پس از اتمام a_i آغاز و قبل از شروع a_k پایان می‌یابند) و مسئلهٔ یافتن حداکثر فعالیت‌های سازگار در S_{kj} (که شامل فعالیت‌هایی می‌شود که پس از اتمام a_k آغاز و قبل از شروع a_j پایان می‌یابند).

مسئله انتخاب فعالیت‌ها

- در گام اول باید ثابت کنیم مسئله دارای زیرساختار بهینه است.

- فرض کنید a_k در مجموعهٔ جواب A_{ij} است. حال فرض کنید $A_{ik} = A_{ij} \cap S_{ik}$ و $A_{kj} = A_{ij} \cap S_{kj}$، بنابراین A_{ik} شامل فعالیت‌هایی در A_{ij} می‌شود که قبل از شروع a_k پایان می‌یابند و A_{kj} شامل فعالیت‌هایی در A_{ij} می‌شود که پس از اتمام a_k شروع می‌شوند.

- اگر a_k در A_{ij} باشد، الزاما A_{ik} جواب مسئله S_{ik} و A_{kj} جواب مسئله S_{kj} است.

- با استفاده از برهان خلف می‌توانیم اثبات کنیم پاسخ بهینه برای S_{ij} شامل پاسخ بهینه برای دو زیر مسئله S_{ik} و S_{kj} می‌شود. اگر می‌توانستیم مجموعه A'_{kj} از حداکثر فعالیت‌های سازگار S_{kj} را پیدا کنیم به طوری‌که $|A'_{kj}| > |A_{kj}|$ آنگاه می‌توانستیم از A'_{kj} به جای A_{kj} در زیر مسئلهٔ S_{ij} استفاده کنیم و بنابراین داشتیم : $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$ که با فرض اینکه A_{ij} جواب بهینه است در تناقض است.

مسئله انتخاب فعالیت‌ها

- طبق تعریف می‌دانیم حداکثر تعداد فعالیت‌ها در S_{ik} برابر است با A_{ik} و حداکثر تعداد فعالیت‌ها در S_{kj} برابر است با A_{kj} .
- بنابراین داریم $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ و در نتیجه $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$.

مسئله انتخاب فعالیت‌ها

- فرض کنید اندازه مجموعه جواب بهینه برای S_{ij} را با $c[i,j]$ نشان دهیم، بنابراین $|A_{ij}| = c[i,j]$.
- آنگاه می توانیم بنویسیم : $c[i,j] = c[i,k] + c[k,j] + 1$.
- از آنجایی که نمی‌دانیم به ازای کدام a_k جواب بهینه به‌دست می‌آید، بنابراین باید همهٔ a_k ها را در نظر بگیریم تا جواب بهینه به دست آوریم.

$$c[i,j] = \begin{cases} 0 & \text{اگر } S_{ij} = \emptyset \\ \max\{c[i,k] + c[k,j] + 1 : a_k \in S_{ij}\} & \text{اگر } S_{ij} \neq \emptyset \end{cases}$$

- سپس می‌توانیم این مسئله را به روش برنامه‌ریزی پویا حل کنیم.

مسئله انتخاب فعالیت‌ها

- می‌توانیم رابطهٔ به دست آمده را ساده‌تر کنیم.
- در مجموعهٔ S_{ij} همیشه یکی از عناصر به عنوان اولین عنصری است که در مجموعهٔ A_{ij} قرار می‌گیرد.
- می‌خواهیم اولین عنصر از مجموعهٔ S_{ij} که در مجموعهٔ A_{ij} قرار می‌گیرد را انتخاب کنیم.
- پس صورت مسئله را تغییر می‌دهیم.

مسئله انتخاب فعالیت‌ها

- فرض کنید $S_k = \{a_i \in S : s_i \geqslant f_k\}$ مجموعه‌ای از فعالیت‌ها باشد که پس از اتمام a_k آغاز می‌شوند.
- اولین فعالیت در S_k که در بزرگترین مجموعهٔ سازگار فعالیت‌ها قرار می‌گیرد کدام است؟

مسئله انتخاب فعالیت‌ها

- قضیه : فرض کنید S_k مجموعه‌ای غیر تهی از فعالیت‌هاست و a_m فعالیتی است در S_k که کوچک‌ترین زمان پایان را دارد. آنگاه a_m در بزرگ‌ترین مجموعهٔ فعالیت‌های سازگار S_k قرار دارد.

- اثبات : فرض کنید A_k بزرگ‌ترین مجموعهٔ فعالیت‌های سازگار S_k باشد و a_j فعالیتی در A_k باشد که کوچکترین زمان پایان را دارد. اگر $a_j = a_m$ باشد به نتیجه مطلوب رسیده‌ایم یعنی در واقع a_m متعلق به بزرگ‌ترین مجموعهٔ فعالیت‌های سازگار S_k است.

- حال فرض کنیم $a_j \neq a_m$ باشد. مجموعهٔ $A'_k = (A_k - \{a_j\}) \cup \{a_m\}$ را به عنوان مجموعه‌ای که شامل a_j نیست ولی a_m را شامل می‌شود در نظر بگیرید. فعالیت‌های A'_k سازگار هستند، زیرا فعالیت‌های A_k سازگار هستند، و a_j اولین فعالیت در A_k است که به اتمام می‌رسد و $f_m \leqslant f_j$. از آنجایی که $|A'_k| = |A_k|$ ، نتیجه می‌گیریم که A'_k نیز بزرگ‌ترین مجموعه فعالیت‌های سازگار S_k است که a_m را نیز شامل می‌شود.

مسئله انتخاب فعالیت‌ها

- فرض کنید اندازه مجموعه جواب بهینه برای S_k را با $c[k]$ نشان دهیم، بنابراین $|A_k| = c[k]$.

- اگر اولین عضو مجموعهٔ S_k فعالیت a_m باشد، طبق قضیه قبل a_m در A_k است. مجموعه فعالیت‌های باقیمانده S_m است و در اینصورت می توانیم بنویسیم : $c[k] = 1 + c[m]$.

- بنابراین می‌توانیم بنویسیم:

$$c[k] = \begin{cases} 0 & \text{اگر } S_k = \emptyset \\ 1 + c[m] & \text{اگر } S_k \neq \emptyset \text{ و } a_m \text{ اولین عضو } S_k \text{ باشد.} \end{cases}$$

- در اینجا یک رابطهٔ بازگشتی به دست آوردیم که همیشه تنها یک انتخاب بهینه در آن وجود دارد. در صورتی که بتوانیم چنین رابطهٔ بازگشتی برای یک مسئله پیدا کنیم، که در آن همیشه یک انتخاب بهینه وجود داشته باشد، می‌توانیم مسئله را با استفاده از روش حریصانه حل کنیم.

مسئله انتخاب فعالیت‌ها

- بنابراین در هر بار باید فعالیتی را انتخاب کنیم که زمان پایان آن از همهٔ فعالیت‌های دیگر کوچک‌تر باشد.
سپس تنها فعالیت‌هایی را نگه‌داریم که زمان با فعالیت انتخاب شده سازگارند (زمان شروع آنها از زمان پایان فعالیت انتخاب شده کمتر است) و فرایند را تکرار کنیم تا جایی که دیگر فعالیتی برای انتخاب نداشته باشیم.

- برای حل این مسئله می توانیم از یک الگوریتم بازگشتی استفاده کنیم.

- این الگوریتم یک الگوریتم حریصانه است، زیرا در هر مرحله به طور حریصانه فعالیتی انتخاب می‌شود که منجر به تعداد بیشتری فعالیت سازگار شود و در پایان برای مسئلهٔ کلی جواب بهینه پس از حل زیر مسئله ها به صورت بهینه به‌دست می‌آید.

مسئله انتخاب فعالیت‌ها

- الگوریتم‌های حریصانه برخلاف الگوریتم‌های برنامه‌ریزی پویا، به صورت از بالا به پایین [1] انجام می‌شوند. در روش حریصانه برای حل یک مسئله یک انتخاب حریصانه انجام می‌گیرد، و در گام بعدی مسئله برای زیرمسئلهٔ به دست آمده حل می‌شود. بنابراین الگوریتم از حل مسئلهٔ اصلی شروع می‌کند و سپس زیرمسئله‌ها را به ترتیب حل می‌کند. در روش برنامه‌ریزی پویا ابتدا زیرمسئله‌ها حل می‌شوند تا در پایان جواب مسئلهٔ اصلی به دست آید، پس این الگوریتم‌ها به صورت از پایین به بالا [2] انجام می‌شوند.

[1] top-down
[2] bottom-up

مسئله انتخاب فعالیت‌ها

- در الگوریتم زیر، مسئله انتخاب فعالیت توسط حریصانه حل می‌شود.

Algorithm Recursive-Activity-Selector

```
function RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m < = n and s[m] < f[k] do  ▷ find the first activity in S[k] to finish
    m = m + 1
if m < = n then
    return {a[m]} ∪ Recursive-Activity-Selector (s, f, m, n)
else
    return ∅
```

- برای شروع فرض کنید فعالیت a_0 با زمان اتمام $f_0 = 0$ در مجموعهٔ $S_0 = S$ وجود دارد. برای شروع، تابع Recursive-Activity-Selector(s,f,0,n) فراخوانی می‌شود.

مسئله انتخاب فعالیت‌ها

- با فرض اینکه فعالیت‌ها مرتب شده باشند، این الگوریتم در زمان $\Theta(n)$ مسئله را حل می‌کند.

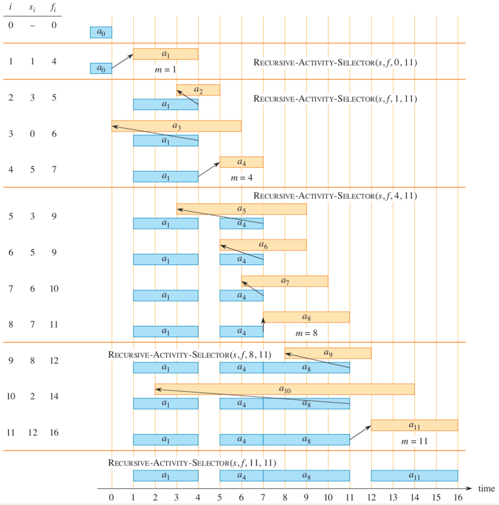
مسئله انتخاب فعالیت‌ها

- یک نمونه از مسئلهٔ انتخاب فعالیت را به صورت زیر در نظر بگیرید.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

مسئله انتخاب فعالیت‌ها

- در مثال زیر این نمونه از مسئله انتخاب فعالیت توسط الگوریتم حریصانه بازگشتی حل شده است.



مسئله انتخاب فعالیت‌ها

– الگوریتم زیر مسئله انتخاب فعالیت را به صورت غیر بازگشتی حل می‌کند.

Algorithm Greedy-Activity-Selector

```
function GREEDY-ACTIVITY-SELECTOR(s, f, n)
A = {a[1]}
k = 1
for m = 2 to n do
    if s[m] >= f[k] then        ▷ is a[m] in S[k] ?
        A = A ∪ {a[m]}          ▷ yes, so choose it
        k = m          ▷ and continue from there
return A
```

مسئلهٔ کوله‌پشتی

- یک الگوریتم حریصانه در هر مرحله انتخابی انجام می‌دهد که در لحظه بهترین انتخاب است و در پایان جواب بهینه نهایی مسئله از این انتخاب‌های بهینه تشکیل شده است. این رویکرد همیشه به جواب بهینه نمی‌رسد اما در برخی مسائل مانند مسئله انتخاب فعالیت جواب بهینه را پیدا می‌کند.
- برای حل یک مسئله به روش حریصانه ابتدا ساختار مسئله مشخص می‌شود و یک جواب بازگشتی برای مسئله بر اساس زیر مسئله‌ها طراحی می‌شود. برخلاف روش برنامه‌ریزی پویا که در آن جواب یک مسئله به چند زیرمسئله بستگی پیدا می کند، در مسئله‌هایی که با روش حریصانه حل می‌شوند، جواب یک مسئله به یک زیر مسئله بستگی دارد. پس از یافتن چنین رابطهٔ بازگشتی که در آن جواب یک مسئله تنها به یک زیرمسئله بستگی دارد، یک الگوریتم بازگشتی حریصانه برای مسئله پیدا می‌شود.

مسئلهٔ کوله‌پشتی

- الگوریتم حریصانه مسئله را به صورت از بالا به پایین حل می‌کند، اما برنامه‌ریزی پویا مسئله را از پایین به بالا حل می‌کند.
- الگوریتم حریصانه در هر گام انتخابی انجام می‌دهد که بهترین انتخاب است و جواب بهینه از این انتخاب‌های بهینه تشکیل شده است.
- در مسئله‌هایی که با الگوریتم حریصانه حل می‌شوند جواب مسئله تنها به جواب یک زیرمسئله بستگی دارد، در حالی که در برنامه‌ریزی پویا، مسئله به چند زیرمسئله بستگی دارد.

مسئلهٔ کوله‌پشتی

- از آنجایی که الگوریتم حریصانه و برنامه‌ریزی پویا شباهت زیادی به یکدیگر دارند و هر دو از زیر مسئله‌های بهینه برای یافتن جواب بهینه مسئله استفاده می‌کنند، ممکن است گاهی برای مسائلی که به روش حریصانه حل می‌شوند، از برنامه‌ریزی پویا استفاده کنیم و یا گاهی به خطا ممکن است بخواهیم مسئله‌ای که به روش برنامه‌ریزی پویا حل می‌شود را به روش حریصانه حل کنیم.

مسئلهٔ کوله‌پشتی

- مسئله کوله پشتی ۱-۰ [1] را در نظر بگیرید. یک دزد که مشغول دزدی از یک مغازه است، می‌خواهد از بین تعدادی کالا که هر کدام ارزش و وزن مشخصی دارند، تعدادی کالا را در کوله پشتی خود که W کیلوگرم ظرفیت دارد بگذارد، به طوری‌که ارزش کالاهایی که با خود می‌برد حداکثر باشد.

- بنابراین این دزد می‌تواند هر زیر مجموعه‌ای از n کالا را بردارد. ارزش کالای i ام برابر است با v_i و وزن آن برابراست با w_i به طوری‌که v_i و w_i دو عدد صحیح هستند. ظرفیت کوله‌پشتی W است و هدف جمع‌آوری تعدادی کالا است که مجموع ارزش آنها حداکثر ممکن باشد.

- به این مسئله، مسئله کوله‌پشتی ۱-۰ گفته می‌شود، زیرا دزد به ازای هر کالا باید یا آن را بگذارد یا بردارد و نمی‌تواند قسمتی از کالا را ببرد و قسمتی را بگذارد.

- در مسئله کوله پشتی کسری [2] دزد می‌تواند یک کالا را به دو قسمت غیر مساوی تقسیم کند و قسمتی از آن را بردارد و قسمتی را با خود ببرد.

---
[1] 0-1 knapsack problem
[2] fractional knapsack problem

مسئلهٔ کوله‌پشتی

- مسئله کوله پشتی ۱-۰ یک مسئلهٔ بهینه‌سازی است که ویژگی آن داشتن زیر ساختار بهینه است، بدین معنی که یک جواب بهینه برای یک مسئله، از جواب بهینه برای زیرمسئله‌های آن تشکیل شده است. درواقع اگر کالاهای پر ارزش با حداکثر مجموع وزن W را در نظر بگیریم که کالای j را در برگرفته‌اند، آنگاه کالاهای کوله‌پشتی بدون کالای j باید پر ارزش‌ترین کالاها با حداکثر مجموع وزن $W - w_j$ باشند.
- همچنین در مسئله کوله‌پشتی کسری، اگر مجموعهٔ پر ارزش‌ترین کالاها با حداکثر وزن W را در نظر بگیریم که مقدار w از کالای j را در برگرفته باشد، آنگاه بقیه کالاهای داخل کوله‌پشتی منهای قسمت w از کالای j با وزن $W - w$ باید پرارزش‌ترین کالاهایی باشند که دزد می‌تواند از بین کالاهای موجود انتخاب کند.
- با وجود اینکه این دو مسئله ساختار بسیار مشابهی دارند، روش حریصانه برای مسئله کوله‌پشتی کسری می‌تواند مورد استفاده قرار بگیرد، در حالی که برای مسئله کوله‌پشتی ۱-۰ راه‌حل حریصانه جواب بهینه به دست نمی‌دهد و باید از روش برنامه‌ریزی پویا استفاده کرد.

مسئلهٔ کوله‌پشتی

- برای حل مسئله کوله‌پشتی توسط روش حریصانه، ابتدا ارزش یک کیلوگرم از هر کالا را به صورت v_i/w_i محاسبه می‌کنیم. وقتی ارزش هر کالا مشخص شد، دزد از با ارزش‌ترین کالا شروع می‌کند و سعی می‌کند کوله‌پشتی خود را پر کند. اگر پرارزش‌ترین کالا به اتمام رسید و هنوز کوله‌پشتی فضای خالی داشت، دزد با دومین کالا پرارزش ادامه می‌دهد و سعی می‌کند آنقدر از آن کالا بر دارد تا کوله پشتی پر شود و اگر کوله‌پشتی با کالا پرارزش دوم پر نشد به سراغ کالا پرارزش سوم می‌رود. این روند آنقدر ادامه پیدا می‌کند تا کوله‌پشتی پر شود. این الگوریتم حریصانه نیاز به مرتب‌سازی کالاها بر اساس ارزش آنها در واحد وزن دارد که این کار با استفاده از یک الگوریتم مرتب‌سازی سریع در زمان $O(n\lg n)$ برای n کالا انجام می‌شود.

مسئلهٔ کوله‌پشتی

- برای اینکه در مسئلهٔ کوله‌پشتی کسری از الگوریتم توصیف شده استفاده کنیم، باید اثبات کنیم الگوریتم درست است.

- می‌توانیم درستی این الگوریتم را با استفاده از برهان خلف ثابت کنیم. فرض کنید کالاهای برداشته شده در کوله‌پشتی بهینه، شامل کالای m که بیشترین چگالی ارزشی [1] را دارد نشود. به عبارت دیگر کالای m که چگالی ارزشی آن v_m/w_m است به طوری که $\forall i, v_m/w_m > v_i/w_i$ در کوله‌پشتی بهینه نباشد. همچنین در مسئلهٔ کوله‌پشتی کسری، الزاما کوله‌پشتی پر می‌شود.

- بنابراین در کوله‌پشتی کالای k وجود دارد که چگالی ارزشی آن از کالای m کمتر است، یعنی $v_k/w_k < v_m/w_m$. در اینصورت می‌توانیم x کیلوگرم از کالای k را از کوله‌پشتی برداریم و x کیلوگرم از کالای m را در کوله‌پشتی قرار دهیم.

- مقدار $x \cdot (v_m/w_m) - x \cdot (v_k/w_k)$ مثبت است، که با فرض اولیه مبنی بر اینکه کوله‌پشتی بهینه است در تناقض دارد، پس کوله‌پشتی الزاما حاوی کالایی با بیشترین چگالی ارزشی است.

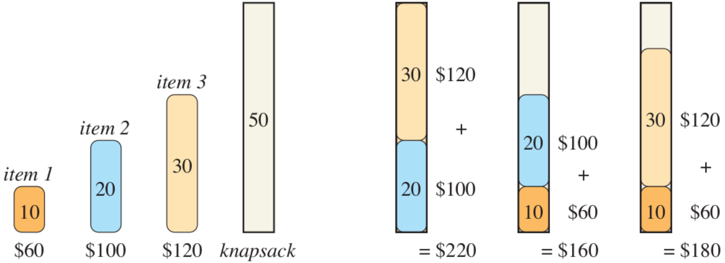
[1] value density

مسئلهٔ کوله‌پشتی

- الگوریتم حریصانه برای مسئله کوله‌پشتی ۱-۰ نمی‌تواند مورد استفاده قرار بگیرد.
- در مسئله کوله‌پشتی ۱-۰ زیر مسئله‌ها باید با یکدیگر مقایسه شوند، در صورتی که جواب بهینه مسئله کوله پشتی کسری تنها به یک زیر مسئله بستگی دارد.
- می‌توان با استفاده از یک مثال نقض نشان داد که الگوریتم حریصانه نمی‌تواند در کوله‌پشتی ۱-۰ مورد استفاده قرار بگیرد.

مسئلهٔ کوله‌پشتی

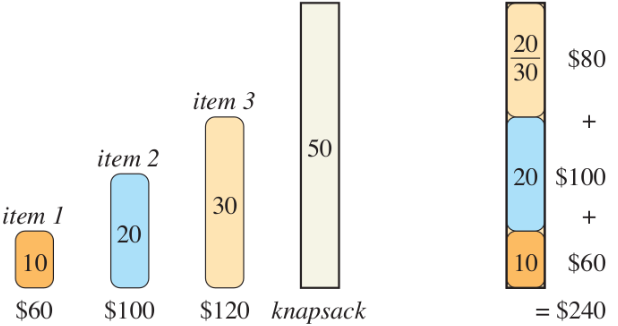
- مثال زیر نشان می‌دهد که روش حریصانه برای مسئله کوله پشتی ۱-۰ الزاما جواب بهینه را به دست نمی‌آورد. با وجود اینکه کالای اول پر ارزش‌ترین کالاست ولی در مسئله کوله پشتی ۱-۰ در جواب بهینه انتخاب نمی‌شود.



- همچنین اگر پر کردن کوله پشتی را با کالاهایی آغاز کنیم که بیشترین قیمت را دارند، ممکن است کالاها با قیمت بالا حجم زیادی را اشغال کرده و کوله پشتی را پر کنند، در صورتی که مجموع ارزش کالاهای کم ارزش‌تر بیشتر باشد.

مسئلهٔ کوله‌پشتی

- در مسئله کوله پشتی کسری توسط یک الگوریتم حریصانه شروع به پر کردن کوله پشتی توسط پرارزش‌ترین کالاها می‌کنیم.



کدهای هافمن

- کدهای هافمن [1] برای فشرده‌سازی داده‌ها استفاده می‌شوند. این کدها بسته به ویژگی داده‌ها، می‌توانند بین ۲۰ تا ۹۰ درصد در میزان حافظه مورد نیاز برای ذخیره‌سازی داده‌ها صرفه جویی کنند.
- الگوریتم حریصانه هافمن جدولی شامل تعداد تکرار حروف دریافت کرده، سپس برای هر حرف یک کد تولید می‌کند، به طوری که ذخیرهٔ یک متن با استفاده از کدهای تولید شده کمترین میزان حافظه را اشغال کند.
- فرض کنید یک فایل داده‌ای داریم که شامل ۱۰۰،۰۰۰ حرف (کاراکتر) است و می‌خواهیم فایل را به صورت فشرده ذخیره کنیم و همچنین می‌دانیم ۶ حرف اول الفبا دارای تعداد تکرارهای ذکر شده در جدول زیر هستند.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101

[1] Huffman codes

کدهای هافمن

- برای مثال حرف a در این فایل ۴۵،۰۰۰ بار و حرف b تعداد ۱۳،۰۰۰ بار تکرار شده‌اند.
- برای نمایش داده‌ها در این فایل راه‌های زیادی وجود دارد. در اینجا از یک کدگذاری دودویی استفاده می‌کنیم. به ازای هر یک از حروف الفبا یک عدد دودویی در نظر گرفته و آن حرف را با کد در نظر گرفته شده نمایش می‌دهیم. به این کدهای دودویی [1] ،به اختصار کد می‌گوییم.
- می‌توانیم از کدهایی با طول ثابت [2] استفاده کنیم، که در اینصورت به تعداد $\lceil \lg n \rceil$ بیت برای نمایش n حرف نیاز داریم. برای ۶ حرف به ۳ بیت نیاز داریم : $a = 000$ ، $b = 001$ ، $c = 010$ ، $d = 011$ ، $e = 100$ و $f = 101$.
- با استفاده از این روش کدگذاری برای یک فایل شامل ۱۰۰،۰۰۰ حرف به ۳۰۰،۰۰۰ بیت نیاز داریم. اما آیا می‌توانیم با تعداد کمتری بیت این فایل را ذخیره کنیم؟

[1] binary character code
[2] fixed-length code

کدهای هافمن

- برای فشرده‌سازی این فایل متنی و ذخیره‌سازی حروف به طور کارامدتر از کدهایی با طول متغیر [1] استفاده می‌کنیم.
- برای نمایش حروفی که تعداد تکرار بیشتری دارند، از کدهای کوتاه‌تر و برای نمایش حروفی که تعداد تکرار کمتری دارند، از کدهای بلندتر استفاده می‌کنیم.

[1] variable-length code

کدهای هافمن

- برای مثال یک روش کدگذاری با کدهای طول متغیر در شکل زیر نمایش داده شده است.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- در اینجا از رشتهٔ بیتی صفر برای نمایش a و رشتهٔ چهاربیتی ۱۱۰۰ برای نمایش حروف f استفاده می‌کنیم.
- برای نمایش یک فایل شامل ۱۰۰،۰۰۰ حرف با تعدادهای تکرار ذکر شده به تعداد بیت زیر نیاز داریم :

$$(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1000 = 224000 \ \ \mathrm{bit}$$

- بنابراین با استفاده از این روش کدگذاری توانستیم به جای ۳۰۰ هزار بیت از ۲۲۴ هزار بیت استفاده کنیم و حدود ۲۵ درصد در فضای حافظهٔ مورد نیاز صرفه‌جویی کنیم.

کدهای هافمن

- در روش کدگذاری استفاده شده، هیچ‌یک از کدها پیشوند کدهای دیگر نبودند. بدین ترتیب به افزودن خط فاصله بین کدها نیازی نداریم و می‌توانیم کدهای یکتا را شناسایی کنیم. به این مجموعهٔ کد، کدهای بدون پیشوند [1] می‌گوییم.
- ثابت شده‌است که کدهای بدون پیشوند بهینه‌ترین روش برای فشرده‌سازی اطلاعات است و هیچ روش کدگذاری بهینه‌تری برای فشرده‌سازی بیشتر وجود ندارد.
- با استفاده از روش کدگذاری می‌توانیم کلمهٔ face را به صورت

$$1100 \cdot 0 \cdot 100 \cdot 1101 = 110001001101$$

ذخیره کنیم. در اینجا عملگر " $\cdot$ " به معنی الحاق دو رشته است.

[1] prefix-free code

کدهای هافمن

- کدهای بدون پیشوند فرایند کدگشایی را ساده می‌کنند. از آنجایی که هیچ کدی پیشوند کد دیگری نیست، کدگذاری یک فایل ابهام ایجاد نمی‌کند. برای کدگشایی یک فایل از ابتدای فایل شروع می‌کنیم و کدها را به ترتیب تبدیل به حروف می‌کنیم.
- برای مثال در کدگشایی رشتهٔ 100011001101 کد 1 یا 10 یا 1000 وجود ندارند و تنها کدی که می‌توان در ابتدای این رشته تشخیص داد، کد 100 است. این کد معادل کلمه cafe می‌باشد.

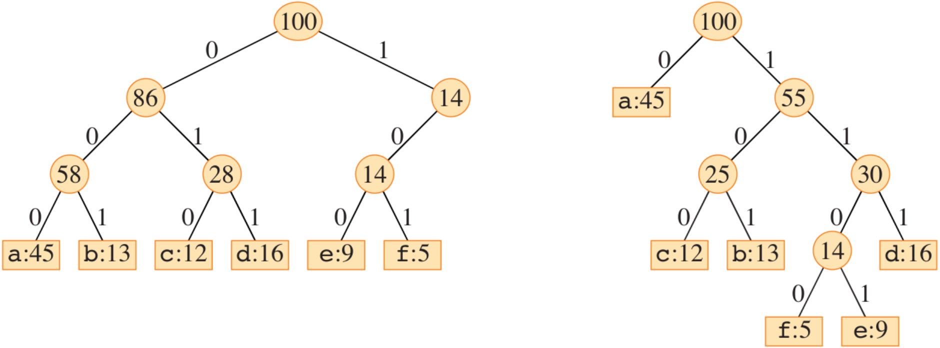
$$100011001101 = 100 \cdot 0 \cdot 1100 \cdot 1101 = cafe$$

کدهای هافمن

- در فرایند کدگشایی برای جستجوی بهینهٔ کدها و حروف متناظر آنها از یک درخت دودویی استفاده می‌کنیم. برگ‌های این درخت دودویی، حروف متناظر با کدهایی هستند که از الحاق کدهای روی یال‌ها از ریشه تا برگ مورد نظر به دست می‌آیند.
- به عبارت دیگر کد مربوط به یک حرف درواقع یک مسیر از ریشه تا حرف مورد نظر است به طوری‌که صفر به معنای رفتن به سمت فرزند سمت چپ و یک به معنای رفتن به فرزند سمت راست است.

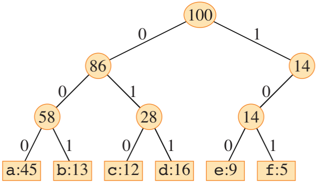
کدهای هافمن

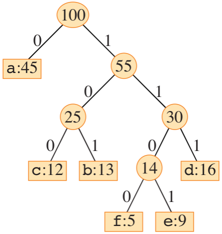
- شکل‌های زیر دو درخت متفاوت برای کدگذاری حروف را نشان می‌دهند.



کدهای هافمن

- یک کدگذاری بهینه همیشه توسط یک درخت دودویی کامل [1] نشان داده می‌شود، بدین معنی که هر رأس میانی در درخت بهینه الزاما دارای دو فرزند است. در شکل زیر در سمت چپ، درخت دودویی کامل نیست، زیرا به ازای کد ۱۱ هیچ حرفی وجود ندارد، پس کدگذاری توسط این درخت نمی‌تواند یک کدگذاری بهینه باشد، اما درخت سمت راست یک درخت کامل را نشان می‌دهد.





- در درخت سمت چپ از کد ۱ برای هیچ حرفی استفاده نشده است و به همین دلیل این کدگذاری نمی‌تواند بهینه باشد.

[1] full binary tree

کدهای هافمن

- یک درخت دودویی کامل با n برگ الزاما $n-1$ رأس غیربرگ دارد.
- اگر C الفبای مورد نظر برای کدگذاری باشد، درختی که برای کدهای بدون پیشوند بهینه به دست می‌آید، دارای $|C|$ برگ است که هر برگ متناظر با یک حرف است و تعداد $|C|-1$ رأس میانی (غیربرگ) در درخت داریم.

کدهای هافمن

- اگر درخت T درختی برای کدهای بدون پیشوند باشد، می‌توانیم تعداد بیت‌های مورد نیاز برای کدگذاری یک فایل را محاسبه کنیم. به ازای هر حرف c در الفبای C ، فرض کنید $c.freq$ تعداد تکرار آن حرف در فایل باشد و فرض کنید $d_T(c)$ عمق برگ متناظر با حرف c در درخت باشد. دقت کنید که $d_T(c)$ طول کد متناظر با حرف c نیز هست. در اینصورت تعداد بیت‌های مورد نیاز برای کدگذاری فایل داده‌ای برابراست با

$$B(T) = \sum_{c \in C} c.freq \times d_T(c)$$

- به مقدار $B(T)$ هزینه [1] درخت T می‌گوییم.

[1] cost

کدهای هافمن

- هافمن یک الگوریتم حریصانه ابداع کرد که کدهای بدون پیشوند بهینه تولید می‌کند. این کدها به کدهای هافمن مشهور هستند.
- ورودی الگوریتم هافمن مجموعهٔ C شامل n حرف است، به طوری که هر عضو $c \in C$ یک حرف است که ویژگی $c.freq$ تعداد تکرار آن را نشان می‌دهد.
- این الگوریتم درخت T را برای تولید کدهای بهینه می‌سازد. این درخت از پایین به بالا تولید می‌شود، بدین معنی که الگوریتم با $|C|$ برگ آغاز می‌کند و با ادغام این برگ‌ها به صورت ساختار درختی، کل درخت را تا ریشه می‌سازد.
- در این الگوریتم از یک صف اولویت استفاده می‌شود که حروف با کمترین تعدادهای تکرار به ترتیب از صف خارج می‌شوند.

کدهای هافمن

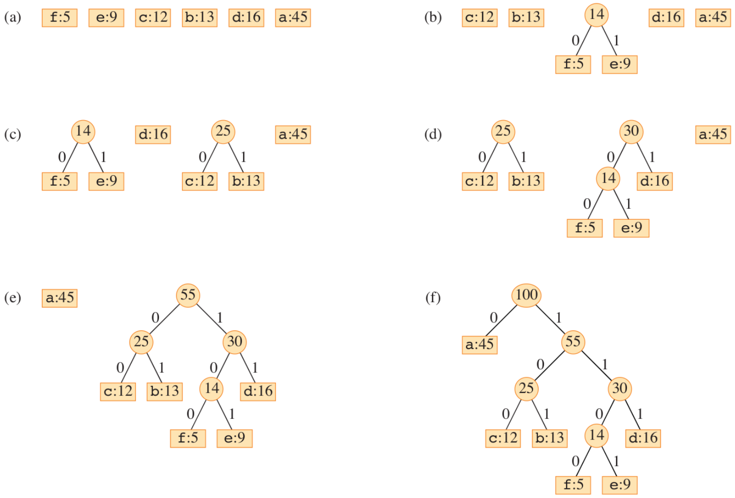
- الگوریتم هافمن به صورت زیر است.

Algorithm Huffman

```
function HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1 do
    allocate a new node z
    x = Extract-Min(Q)
    y = Extract-Min(Q)
    z.left = x
    z.right = y
    z.freq = x.freq + y.freq
    Insert(Q,z)
return Extract-Min(Q)   ▷ the root of the tree is the only node left
```

کدهای هافمن

- برای مثالی که در قبل مطرح کردیم، الگوریتم هافمن به صورت زیر عمل می‌کند.



کدهای هافمن

- زمان الگوریتم هافمن به نحوهٔ پیاده‌سازی صف اولویت بستگی دارد. فرض کنیم با بهینه‌ترین الگوریتم موجود، صف اولویت برای یک الفبا با n حرف در زمان $O(n)$ ساخته می‌شود.
- حلقهٔ اصلی در الگوریتم هافمن $n-1$ بار تکرار می‌شود، زیرا تعداد رئوس غیربرگ $n-1$ است و از آنجایی که در هر بار استفاده از صف اولویت به زمان $\lg n$ نیاز داریم، بنابراین زمان اجرای الگوریتم $O(n \lg n)$ است.
- بنابراین کل زمان اجرای الگوریتم هافمن برای الفبای n حرفی برابراست با $O(n \lg n)$.

کدهای هافمن

- برای اثبات اینکه الگوریتم حریصانه هافمن درست است، نشان می‌دهیم مسئله تعیین کدهای بدون پیشوند بهینه دارای ویژگی انتخاب حریصانه است.
- به عبارت دیگر می‌خواهیم اثبات کنیم دو رأس با کمترین تعداد تکرار الزاما در درخت کدهای بهینه همزاد یکدیگرند و در بیشترین عمق قرار می‌گیرند.

کدهای هافمن

- قضیه : فرض کنید C یک الفبا باشد به طوری‌که $c \in C$ دارای تعداد تکرار $c.freq$ باشد. فرض کنید x و y دو حرف در C با کمترین تعدادهای تکرار باشند. آنگاه یک کدگذاری بدون پیشوند بهینه برای C وجود دارد به طوری‌که x و y طول یکسانی دارند و تنها در بیت آخر متفاوت‌اند، یعنی همزاد یکدیگرند و همچنین در بیشترین عمق درخت قرار دارند.

کدهای هافمن

- اثبات : ایدهٔ اثبات این است که درخت T که یک درخت بهینهٔ بدون پیشوند را در نظر بگیریم و آن را تغییر دهیم تا یک درخت دودویی بدون پیشوند دیگر ساخته شود به طوری‌که در درخت ساخته شده x و y همزاد [1] و در عمق بیشینه در درخت T باشند. چنین درختی که در آن x و y همزاد یکدیگرند یعنی طول یکسان دارند و تنها در بیت آخر متفاوت‌اند، نیز یک درخت بهینه خواهد بود.

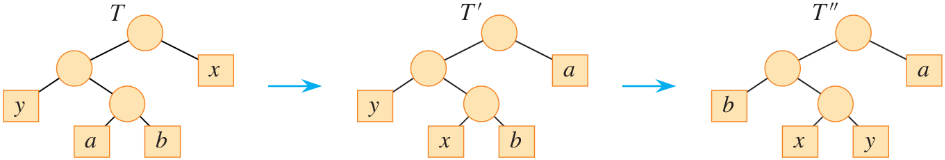
- فرض کنید a و b دو حرف باشند که در درخت T همزاد هستند و در بیشترین عمق درخت قرار دارند. حال فرض کنید $a.freq \leqslant b.freq$ و $x.freq \leqslant y.freq$. از آنجایی‌که $x.freq$ و $y.freq$ کمترین تعدادهای تکرار هستند و $a.freq$ و $b.freq$ دو تعداد تکرار دلخواه هستند، بنابراین خواهیم داشت $x.freq \leqslant a.freq$ و $y.freq \leqslant b.freq$.

- بنابراین می‌توانیم داشته باشیم $x.freq = a.freq$ و $y.freq = b.freq$ که در اینصورت قضیه به طور بدیهی درست است، زیرا a و b دارای کمترین تعداد تکرار هستند. پس فرض می‌کنیم تعداد تکرارهای x و y متفاوت از a و b هستند.

[1] sibiling

کدهای هافمن

- همانطورکه شکل زیر نشان می‌دهد، فرض کنید جای a و x را در درخت T عوض می‌کنیم و درخت T' را به‌دست می‌آوریم و جای b و y را در درخت T' عوض کرده، درخت T'' را به دست می‌آوریم.



- نشان می‌دهیم که هزینهٔ درخت T'' کوچکتر یا مساوی درخت T است. از آنجایی که فرض کردیم درخت T یک درخت بهینه است، بنابراین هزینه درخت T'' و T باید برابر باشد.

- تفاوت هزینهٔ درخت T و T' به صورت زیر خواهد بود.

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
&= (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\geqslant 0
\end{aligned}
$$

کدهای هافمن

- از آنجایی که $a.freq - x.freq$ و همچنین $d_T(a) - d_T(x)$ غیر منفی هستند، بنابراین مقدار $B(T) - B(T')$ مثبت است.
- درواقع $a.freq - x.freq$ غیر منفی است زیرا x یک برگ با حداقل تعداد تکرار است و $d_T(a) - d_T(x)$ غیر منفی است زیرا a یک برگ با عمق بیشینه در درخت T است.
- به همین ترتیب تعویض y و b هزینه را افزایش نمی‌دهد و بنابراین $B(T') - B(T'')$ نیز غیر منفی است.
- بنابراین $B(T'') \leqslant B(T') \leqslant B(T)$ و چون T بهینه است بنابراین داریم $B(T) \leqslant B(T'')$ و بنابراین $B(T'') = B(T)$ در نتیجه T'' یک درخت بهینه است که در آن x و y دو برگ همزاد با عمق حداکثر هستند و قضیه ثابت می‌شود.

کدهای هافمن

- این قضیه در واقع نشان می‌دهد که ساختن درخت بهینه، می‌تواند با انتخاب حریصانه ادغام دو حرف با کمترین تعداد تکرار آغاز شود و ادامه یابد. بنابراین از بین همهٔ انتخاب‌ها برای ادغام الگوریتم هافمن دو حرف با کمترین تعداد را در هر مرحله انتخاب می‌کند که یک انتخاب بهینه است، و همچنین درخت کلی به دست آمده در نهایت یک درخت بهینه خوهد بود.

کدهای هافمن

- حال می‌خواهیم ثابت کنیم ساختن کدهای بدون پیشوند بهینه دارای ویژگی زیر ساختار بهینه است.
- به عبارت دیگر، می‌خواهیم اثبات کنیم اگر رأس z به عنوان پدر رئوس x و y با کمترین تعداد تکرار به همراه بقیه رئوس درخت به جز x و y ، یک درخت کدهای بهینه را تشکیل دهند، آنگاه درختی که در آن x و y به عنوان فرزندان z اضافه شده‌اند نیز درخت کدهای بهینه است.

- قضیه : فرض کنید C یک الفبا باشد به طوری‌که برای هر حرف $c \in C$ تعداد تکرار c برابر با $c.freq$ باشد. فرض کنید x و y دو حرف در C با تعداد تکرار حداقل باشند. فرض کنید C' همان الفبای C باشد به طوری‌که حروف x و y حذف شده و حرف z به آن اضافه شده است، بنابراین $C' = (C - \{x, y\}) \cup \{z\}$ تعداد تکرار همهٔ حروف در C' برابر با حروف C هستند و همچنین $z.freq = x.freq + y.freq$. فرض کنید T' درختی باشد که کدهای بدون پیشوند بهینه C' را نمایش می‌دهد. آنگاه درخت T که از درخت T' به دست آمده و در آن رأس z با یک رأس میانی با دو فرزند x و y به جایگزین شده است، کدهای بدون پیشوند بهینه برای الفبای C را نمایش می‌دهد.

کدهای هافمن

- اثبات : ابتدا نشان می‌دهیم چگونه هزینه $B(T)$ از درخت T بر اساس هزینهٔ $B(T')$ از درخت T' بیان می‌شود.

- به ازای هر حرف $c \in C - \{x, y\}$ ، داریم $d_T(c) = d_{T'}(c)$ و بنابراین $c.freq \times d_T(c) = c.freq \times d_{T'}(c)$.

- چون $d_T(x) = d_T(y) = d_{T'}(z) + 1$ ،بنابراین داریم :

$$\begin{aligned} x.freq \times d_T(x) + y.fraq \times d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \times d_{T'}(z) + (x.freq + y.freq) \end{aligned}$$

- بنابراین نتیجه می‌گیریم $B(T) = B(T') + x.freq + y.freq$ که برابر است با $B(T') = B(T) - x.freq - y.freq$.

- حال از برهان خلف استفاده می‌کنیم.

کدهای هافمن

- فرض کنید T درخت بهینه بدون پیشوند برای C نیست. بنابراین یک درخت T'' بهینه وجود دارد به طوری‌که $B(T'') < B(T)$. درخت T'' دو رأس x و y را به عنوان همزاد درخود دارد.
- حال فرض کنید T''' همان درخت T'' باشد که در آن پدر مشترک x و y با رأس برگ z جایگزین شده است به طوری‌که $z.freq = x.freq + y.freq$
- بنابراین :

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T') \end{aligned}$$

- به این نتیجه رسیدیم که T''' یک درخت بهینه برای C' است، اما با این نتیجه به تناقض می‌رسیم زیرا فرض کردیم T' درخت بدون پیشوند بهینه برای C' است. بنابراین T باید کدهای بدون پیشوند بهینه برای الفبای C را نمایش دهد.

کدهای هافمن

- دو قضیهٔ اثبات شده نشان می‌دهند الگوریتم هافمن کدهای بدون پیشوند بهینه تولید می‌کند.