فصل سوم

روشهای حریصانه (greedy) - آنالیز استهلاکی (amortized)

مقدمه

الگوریتم حریصانه یا آزمند، به ترتیب داده‌ها را گرفته و هر بار عنصری را که طبق معیار معینی «بهترین» به نظر می‌رسد، بدون توجه به انتخاب‌های قبلی و انتخاب‌هایی که در آینده انجام خواهد شد، انتخاب می‌کند. این الگوریتم‌ها بسیار ساده هستند و معمولاً برای مسائل بهینه‌سازی کاربرد دارند.

در این الگوریتم‌ها، مجموعه‌ای وجود دارد و باید از آن مجموعه، مرحله به مرحله اعضا را انتخاب (select) کرد. با هر انتخاب بررسی می‌شود که آیا عضو انتخاب شده، امکان رسیدن به جواب نهایی را دارد یا خیر (feasible). با هر انتخاب بررسی می‌شود که آیا به جواب نهایی رسیده‌ایم یا خیر (Solution).

شکل کلی این الگوریتم‌ها به صورت الگوریتم ۱ می‌باشد:

الگوریتم ۱. شکل کلی الگوریتم‌های حریصانه

```
Set greedy (set c)                          // C مجموعه انتخاب‌های ممکن //
  {
     s = ϕ;                                 // S مجموعه جواب که در ابتدا تهی است //
     while (!solution (s) && c!= ϕ)         // Solution بررسی می‌کند که جواب، حاصل شده یا خیر //
     {
      x=select (c);                         // Select یک عضو از مجموعه C انتخاب می‌کند //
      c=c-{x};                              // آن عضو از C حذف می‌شود //
      if (feasible (S ∪ {x}))               // feasible بررسی می‌کند که آیا امکان اضافه شدن x به جواب وجود یا خیر//
        S=S ∪ {x};
      }
      if (solution (s))                     // اگر امکان وجود داشت، x به مجموعه جواب S اضافه می‌شود //
        return (s);
else                                        // اگر جواب حاصل شد S برمی گردد در غیر این صورت //
        return (ϕ);                         // تهی برگشت داده می‌شود //
}
```

برای آنکه با مفهوم الگوریتم‌های حریصانه بیشتر آشنا شویم یک مثال مطرح می‌کنیم.

مثال: فرض کنید فروشنده‌ای می‌خواهد بقیه پول شما را که ۳۶ واحد است پرداخت کند.

او می‌خواهد تعداد سکه‌های پرداختی حداقل باشد. فرض کنید سکه‌های موجود عبارتند از ۲۵ و ۱۰ و ۵ و ۱ واحدی. فروشنده ابتدا بزرگترین سکه موجود یعنی ۲۵ واحدی را انتخاب می‌کند (بهینه محلی). با انتخاب یک سکه ۲۵ واحدی دیگر جواب حاصل نخواهد شد بنابراین او یک سکه ۱۰ واحدی انتخاب می‌کند تاکنون ۳۵ واحد انتخاب شده. با انتخاب سکه ۱۰ واحدی دیگر و ۵ واحدی جواب حاصل نمی‌شود، بنابراین فروشنده سکه ۱ واحدی را انتخاب می‌کند. بنابراین مجموعه جواب این مسئله {۱ و ۱۰ و ۲۵} می‌باشد که جواب بهینه نیز هست.

حال فرض کنید بجای سکه‌های ۲۵ و ۱۰ و ۵ و ۱ واحدی ، سکه‌های ۱۲ و ۱۰ و ۵ و ۱ واحدی وجود می‌داشت و بقیه پول شما ۱۶ واحد باشد. باروش حریصانه یک سکه ۱۲ واحدی و ۴ سکه ۱ واحدی دریافت می‌کنید یعنی ۵ سکه. حال آنکه جواب بهینه در این حالت شامل یک سکه ۱۰ واحدی، یک سکه ۵ واحدی و یک سکه ۱ واحدی است یعنی فقط ۳ سکه.

پس روش حریصانه برای خرد کردن سکه جواب نمی‌دهد.

در این مثال، مجموع انتخاب‌های ممکن یعنی C، سکه‌های موجود می‌باشد. تابع Select هر بار یک سکه انتخاب می‌کند. تابع feasible با هر انتخاب بررسی می‌کند که آیا انتخاب انجام شده امکان پذیر هست یا خیر. اگر امکان پذیر نباشد آن انتخاب برای همیشه طرد می‌شود. تابع Solution با هر انتخاب بررسی می‌کند که آیا جواب حاصل شده است یا خیر.

الگوریتم‌هایی که به روش حریصانه بررسی می‌شوند عبارتند از:

۱- مساله کوله پشتی غیرصفر و یک ۲- کدهافمن ۳- زمان‌بندی بر مبنای کمینه کردن زمان کل ۴- انتخاب (زمان بعدی) فعالیت‌ها ۵- زمان‌بندی فعالیت‌ها با مهلت معین

الگوریتم‌های حریصانه دیگری مثل کراسکال (راشال) – پریم – دایجسترا در فصل ۶ بررسی می‌شوند.

۱- مساله کوله‌پشتی غیرصفر و یک (کوله پشتی کسری)

یک کوله پشتی که حداکثر ظرفیت آن جرم M است در نظر بگیرید. n تاشی با جرم‌های w_1 و w_2 و ... w_n و ارزش‌های p_1 و p_2 و ... و p_n مفروضند. می‌خواهیم کوله پشتی را با اشیاء پر کنیم طوری که حداکثر سود (ارزش) بدست آید. در ضمن می‌توان از یک شی کسری را انتخاب کرد. بنابراین اگر فرض کنیم از شی i، کسر x_i انتخاب شده است $(0 \le x_i \le 1)$ هدف ماکزیمم کردن عبارت $\sum_{i=1}^{n} p_i x_i$ به شرطی که $\sum_{i=1}^{n} w_i x_i \le M$ باشد.

مثال: کوله پشتی با ظرفیت جرم M=۲۰ و ۳ شی که جرم و ارزش آنها داده شده است مفروضند. کوله پشتی را می‌خواهیم پر کنیم طوری که ارزش بدست آمده ماکزیمم باشد.

$(w_1, w_2, w_3) = (18, 15, 10)$

$(p_1, p_2, p_3) = (25, 24, 15)$

پاسخ: برای این مساله می‌توان با روش حریصانه اشیاء را انتخاب کرده ولی ابتدا باید یک حدس خوب برای انتخاب اشیاء بیابیم. ۳ پیشنهاد وجود دارد:

الف) چون ارزش اشیاء مهم است پس بهتر است اشیاء را به ترتیب بیشترین ارزش انتخاب کنیم به عنوان مثال شی ۱ را کامل انتخاب کنیم که در این صورت ظرفیت باقی مانده کوله پشتی ۲ خواهد بود بنابراین کسر $\frac{۲}{۱۵}$ از شی ۲ را انتخاب کنیم پس:

$$(x_۱, x_۲, x_۳) = \left(۱, \frac{۲}{۱۵}, ۰\right) \Rightarrow \text{ارزش بدست آمده} = ۱\times۲۵ + \frac{۲}{۱۵}\times۲۴ = ۲۸/۲$$

ب) چون وزن اشیاء تاثیر منفی دارد بهتر است اشیا را به ترتیب از کمترین وزن انتخاب کنیم. یعنی شی ۲ را کامل انتخاب کنیم که ظرفیت باقیمانده ۱۰ خواهد شد و از شی ۲ کسر $\frac{۱۰}{۱۵}$ را انتخاب کنیم.

$$(x_۱, x_۲, x_۳) = \left(۰, \frac{۲}{۳}, ۱\right) \Rightarrow \text{ارزش بدست آمده} = ۰\times۲۵ + \frac{۲}{۳}\times۲۴ + ۱\times۱۵ = ۳۱$$

ج) بهتر است اشیاء را به ترتیب نزولی $\frac{p_i}{w_i}$ انتخاب کنیم یعنی آن شی را اول انتخاب کنیم که $\frac{p_i}{w_i}$ آن بیشترین است:

$$\left(\frac{p_۱}{w_۱}, \frac{p_۲}{w_۲}, \frac{p_۳}{w_۳}\right) = \left(\frac{۲۵}{۱۸}, \frac{۲۴}{۱۵}, \frac{۱۵}{۱۰}\right)$$

پس شی ۲ را باید کامل انتخاب کنیم که ظرفیت باقی مانده ۵ خواهد بود و سپس از شی ۳ کسر $\frac{۱}{۲}$ را انتخاب می‌کنیم تا کوله پشتی پر شود پس:

$$(x_۱, x_۲, x_۳) = \left(۰, ۱, \frac{۱}{۲}\right) \Rightarrow \text{ارزش بدست آمده} = ۰\times۲۵ + ۱\times۲۴ + \frac{۱}{۲}\times۱۵ = ۳۱/۵$$

همانطور که ملاحظه می‌شود حدس سوم بهترین است و می‌توان ثابت کرد که حدس سوم همیشه به جواب بهینه خواهد رسید. الگوریتم ۲، کوله پشتی را نشان می‌دهد.

```
void gknapsack (float p[], float w[], float M, float x[], int n)
{
    float rc;                                   // p آرایه ارزش‌ها، w آرایه وزن‌ها، M ظرفیت //
    int i;                                      // x کوله، x کسر انتخاب شده ۰ ≤ x_i ≤ ۱ //
    pwsort (p, w, n);                           // pwsort ، روال مرتب سازی براساس p_i/w_i به صورت نزولی //
    for (i=0; i<n; i++)
      x[i]=0;
    rc=M
    for (i=0 ; i<n; i++)                        // پس از مرتب سازی p[۰]/w[۰] ≥ p[۱]/w[۱] ≥ .. ≥ p[n-۱]/w[n-۱] //
    {
      if (w[i] > rc) break;
      x[i]=1;
      rc= rc-w [i];
```

```
}
if(i<= n-1)
    x[i] = rc/w[i];
}
```

در این الگوریتم اشیاء از صفر تا n-۱ شماره گذاری شده اند. rc ظرفیت باقیمانده کوله است که در ابتدا rc=M می‌باشد. اشیا یکی یکی انتخاب می‌شوند تا اینکه یا اشیاء تمام شوند و یا با گذاشتن شی جاری از ظرفیت مجاز کوله، تجاوز کنیم. در وضعیت اخیر، کسر $\frac{rc}{w[i]}$ از شی i (شیئی که افزودن کامل آن امکان پذیر نیست) به کوله افزوده می‌شود.

مرتبه زمانی الگوریتم، به مرتب‌سازی بستگی دارد که برابر $\theta\left(n\,Log_{۲}^{n}\right)$ می‌باشد.

مثال: چگونه می‌توان با کمک میانه‌گیری، مسئله کوله پشتی کسری را با زمان خطی O(n) حل کرد؟

پاسخ: میانه عناصر $\frac{p_i}{w_i}$ را با مرتبه خطی بیابید و m بنامید. عناصر را ۳ دسته کنید:

G: عناصری که $\frac{p_i}{w_i}$ آنها از m بزرگ‌تر است.

E: عناصری که $\frac{p_i}{w_i}$ آنها با m مساوی است.

L: عناصری که $\frac{p_i}{w_i}$ آنها از m کوچک‌تر است.

سپس W_G را برابر جمع وزن عناصر موجود در G و W_E را برابر جمع وزن عناصر موجود در E تعریف کنید.

- اگر $W_G > M$، آنگاه از G عنصری انتخاب نمی‌شود و روی عناصر موجود در G، الگوریتم را به صورت بازگشتی فراخوانی کنید (با ظرفیت مجاز کوله برابر M)

- در غیر این صورت $(W_G \le M)$، تمام عناصر موجود در G را انتخاب کنید و از مجموعه E حداکثر تعدادی عناصر با محدودیت ظرفیت کوله برابر $M - W_G$ انتخاب کنید.

- اگر $W_G + W_E \ge M$ یعنی کوله پر شده است و کار تمام است.

- در غیر این صورت $(W_G + W_E < M)$، بعد از انتخاب عناصر G و E به طور کامل، روی L با حدودیت ظرفیت $M - W_G - W_E$ فراخوانی بازگشتی انجام دهید.

تحلیل: هر فراخوانی بازگشتی، زمان خطی دارد و خودش ممکن است یک فراخوانی بازگشتی دیگر با داکثر نصف اندازه انجام دهد:

$$T(n) \le T(\frac{n}{۲}) + \theta(n) \Rightarrow T(n) = O(n)$$

۲- کدهافمن:

اگر بخواهیم متنی شامل n کاراکتر را به صورت باینری کد کنیم، $\lceil \mathrm{Log}_2^n \rceil$ بیت به ازای هر کاراکتر نیاز است مثلاً برای کدگذاری حروف الفبای انگلیسی (۲۶ حرف)، ۵ بیت نیاز است.

در این حالت هر کاراکتر دارای طول بیتی یکسان خواهد بود. مثلاً ۰۰۰۰۰ = a و ۰۰۰۰۱ = b و ... .

به راحتی می‌توان چنین متنی را از حالت کد خارج کرد (decode) زیرا هر ۵ بیت نشان دهنده یک کاراکتر است.

کدهافمن یک کد با طول متغیر است. به این معنی که به هر کاراکتر رشته بیتی با طول متفاوت نسبت داده می‌شود. مثلاً ۰۰ = a و ۱۰۱ = b و ۱۰۰۰ = c و ... .

این تخصیص بیت باید دارای خاصیت پیشوند آزاد (prefix – free) باشد. یعنی هیچ کاراکتری نباید پیشوند هیچ کاراکتر دیگری می‌باشد. مثلاً ۰۰ = a و ۰۰۱ = b نادرست است.هافمن با استفاده از درخت دودویی کدی پیشوند آزاد بهینه تولید می‌کند.هافمن براساس تعداد تکرار کاراکترها، به آنها بیت نسبت می‌دهد.

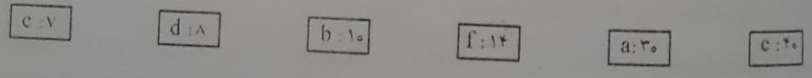
کاراکترهایی که فراوانی آنها زیاد است، رشته بیتی کوچک دریافت می‌کنند. با یک مثال روش‌هافمن را توضیح می‌دهیم.

مثال: فرض کنید متنی شامل حروف a و b و c و d و e و f می‌باشد و فراوانی این حروف در جدول زیر آمده است، می‌خواهیم کدی بهینه برای این حروف تولید کنیم.

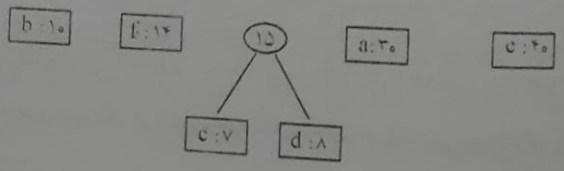
کاراکتر	a	b	c	d	e	f
فراوانی	۳۰	۱۰	۷	۸	۴۰	۱۴

پاسخ: ابتدا دو کاراکتر با کمترین فراوانی انتخاب می‌شوند و با آنها یک درخت ساخته می‌شود و جمع فراوانی آنها در ریشه درخت نوشته می‌شود و آن دو فراوانی از جدول حذف می‌شوند و جمع فراوانی آنها به جدول اضافه می‌شود. سپس دو فراوانی مینیمم دیگر انتخاب می‌شوند و به همین ترتیب. مراحل به شکل زیر است:

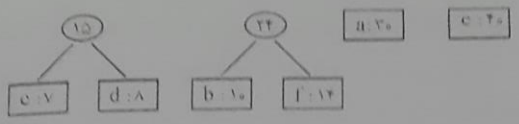
الف) فراوانی‌ها را به ترتیب صعودی مرتب می‌کنیم:



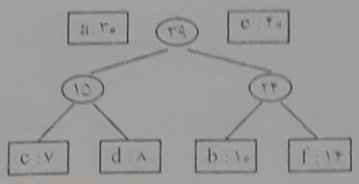
ب) c و d انتخاب می‌شوند و مجموع فراوانی آنها یعنی ۱۵ بعد از ۱۰ قرار می‌گیرد:



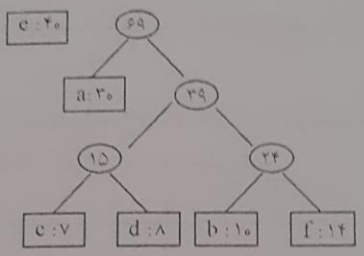
ج) حال b و f انتخاب می‌شوند و با آنها درخت ساخته می‌شود:



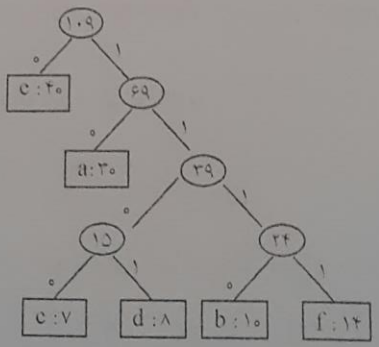
د) با ۱۵ و ۲۴ درخت ساخته می‌شود:



هـ) با ۳۰ و ۳۹ درخت ساخته می‌شود:



و) در نهایت با ۴۰ و ۶۹ درخت می‌سازیم و بعد از ساخت درخت، روی یال‌های چپ ۰ و روی یال‌های راست ۱ می‌نویسیم. بدین ترتیب کد هر کاراکتر بدست می‌آید:



کد حروف مختلف عبارتند از: e = ۰ و a = ۱۰ و c=۱۱۰۰ و d=۱۱۰۱ و b=۱۱۱۰ و f=۱۱۱۱

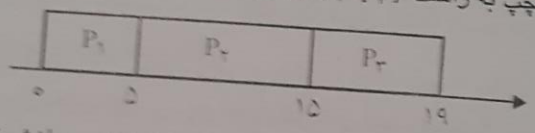
نکته: *در درخت هافمن، نود یک فرزندی وجود ندارد، چون در غیر این صورت کد بهینه تولید نمی‌شود.*

تمرین: الگوریتم هافمن را به کلمه کد مبنای ۳ تعمیم دهید (یعنی از سمبول‌های ۰ و ۱ و ۲ استفاده کنید).

تمرین: فرض کنید یک فایل داده‌ای شامل ۲۵۶ کاراکتر است که با روش عادی هر یک با ۸ بیت نمایش داده می‌شوند. نشان دهید اگر تعداد بیشترین تکرار از کمترین تکرار، کمتر از ۲ برابر باشد، کد هافمن مزیتی به کد عادی ثابت ۸ بیتی ندارد.

۲- زمان‌بندی بر مبنای کمینه کردن زمان کل

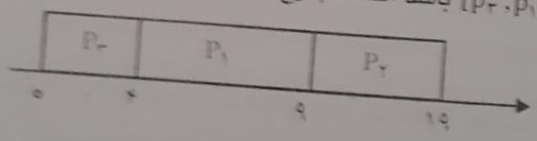
فرض کنید ۳ پردازش که زمان اجرای آنها به ترتیب $p_1 = 5$ و $p_2 = 10$ و $p_3 = 4$ است را می‌خواهیم توسط یک پردازنده اجرا کنیم و می‌خواهیم مجموع زمانهای برگشت، مینیمم شود. منظور از زمان برگشت، فاصله زمانی بین ورود تا خروج یک پردازش است. اگر فرض کنیم ترتیب سرویس‌دهی به این برنامه‌ها به ترتیب از چپ به راست $[p_1, p_2, p_3]$ باشد آنگاه نمودار زمانی زیر را خواهیم داشت:



یعنی پردازش p_1 از زمان ۰ تا ۵، پردازش p_2 از زمان ۵ تا ۱۵ و پردازش ۳ از زمان ۱۵ تا ۱۹ اجرا شده‌اند. مجموع زمانهای برگشت این ۳ پردازش برابر است با:

$5 + 15 + 19 = 39$

ولی اگر ترتیب $[p_3, p_1, p_2]$ باشد آنگاه مجموع زمانهای برگشت برابر است با:



$4 + 9 + 19 = 32$

به راحتی مشاهده می‌شود که اگر پردازشها به ترتیب صعودی زمانشان اجرا شوند، بهینه است. پس الگوریتم این کار درواقع الگوریتم مرتب‌سازی است که مرتبه آن $\theta(n \text{Lgn})$ است.

۴- انتخاب (زمان‌بندی) فعالیتها

فرض کنید مجموعه‌ای از n فعالیت $S = \{1, 2, \ldots, n\}$ پیشنهاد شده است که می‌خواهند از یک منبع مشترک استفاده کنند. هر فعالیت یک زمان شروع و یک زمان پایان دارد. زمان شروع فعالیت i برابر S_i و زمان پایان آن e_i است و $S_i \le e_i$ یعنی فعالیت i در فاصل زمانی $[S_i, e_i)$ انجام می‌شود. وقتی یک فعالیت در حال انجام است، سایر فعالیت‌ها نمی‌توانند انجام شوند. بنابراین دو فعالیت i و j به شرطی هر دو انجام می‌شوند که یا $S_i \ge e_j$ و یا $S_j \ge e_i$ در این صورت i و j را سازگار گویند. هدف، انتخاب بیشترین تعداد از فعالیت‌های سازگار است.

یک الگوریتم حریصانه برای این مسأله به صورت زیر است. فرض شده است که فعالیت‌ها طبق ترتیب صعودی زمان خاتمه‌شان مرتب شده‌اند: $e_1 \le e_2 \le e_3 \le \ldots \le e_n$.

```
Algorithm Activity _ Selection (S , e)
  A={1} , j=1
  for i=2 to n do
    if S[i] ≥ e[j] then
    {
      A = A ∪ {i}
      j = i
    }
return A
```

S و e آرایه‌هایی هستند که زمانهای شروع و خاتمه فعالیت‌ها را ذخیره کرده‌اند. فعالیت‌های انتخاب شده در آرایه A نگهداری می‌شوند. متغیر j معرف آخرین فعالیت انتخاب شده (اضافه شده به A) است. اگر e_i ها از قبل صعودی مرتب باشند، مرتبه الگوریتم فوق $\theta(n)$ است در غیر این صورت، باید مرتب‌سازی انجام شود با مرتبه $O(n\,Log n)$.

مثال: با توجه به جدول فعالیتهای زیر، چه فعالیتهایی قابل انجام هستند؟

i	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱
S_i	۱	۳	۰	۵	۳	۵	۶	۸	۸	۲	۱۱
e_i	۴	۵	۶	۷	۸	۹	۱۰	۱۱	۱۲	۱۳	۱۴

پاسخ:

با توجه به اینکه فعالیت‌ها از اول به صورت صعودی زمان خاتمه‌شان مرتب شده‌اند، لذا نیازی به مرتب‌سازی نیست. فعالیت ۱ انجام می‌شود در زمان $[1,4)$ فعالیت‌های ۲و۳ قابل انجام نیستند چون s_i آنها از ۴ کوچکتر است. فعالیت ۴ در زمان $[5,7)$ اجرا می‌شود. پس فعالیت ۸ در زمان $[8,11)$ و در نهایت فعالیت ۱۱، یعنی فعالیتهای ۱ و ۴ و ۸ و ۱۱ انجام می‌شوند.

۵- زمان‌بندی فعالیت‌ها با مهلت معین (Scheduling with Dead Lines)

مجموعه‌ای از n فعالیت (job) مفروض است. این فعالیت‌ها همگی توسط یک پردازنده باید انجام شوند و هر کدام یک واحد زمانی طول می‌کشند. این فعالیتها هر کدام دارای یک مهلت d_i و یک ارزش یا سود p_i هستند و سود p_i در صورتی حاصل می‌شود که فعالیت مورد نظر در مهلت d_i انجام شود. مثلاً اگر فعالیتی دارای مهلت ۲ و سود ۴۰ باشد، سود ۴۰ فقط در صورتی حاصل می‌شود که این فعالیت در زمان ۱ یا ۲ انجام شود و اگر در زمان ۳ یا بیشتر انجام شود هیچ سودی حاصل نمی‌شود.

مثال: فرض کنید $n = ۴$ فعالیت به شرح $(p_۱, p_۲, p_۳, p_۴) = (۱۰۰, ۱۰, ۱۵, ۲۷)$ و $(d_۱, d_۲, d_۳, d_۴) = (۲, ۱, ۲, ۱)$. انواع زمان‌بندی‌های ممکن و سود حاصل عبارت است از:

	زمان‌بندی	بهره کل
۱	[۳,۱] یا [۱,۳]	۱۰۰+۱۵ = ۱۱۵
۲	[۲,۱]	۱۰۰+۱۰ = ۱۱۰
۳	[۲,۳]	۱۰+۱۵ = ۲۵
۴	[۴,۱]	۱۰۰+۲۷ = ۱۲۷
۵	[۴,۳]	۲۷+۱۵ = ۴۲
۶	[۱]	۱۰۰
۷	[۲]	۱۰
۸	[۳]	۱۵
۹	[۴]	۲۷

ملاحظه می‌شود که جواب ۴ بهینه است و بیشترین سود حاصل ۱۲۷ است. یعنی اول فعالیت ۴ و بعد فعالیت ۱ انجام شوند. دقت کنید فعالیت‌های [۲ و ۴] قابل اجرا نیستند زیرا هر دو باید در زمان ۱ انجام شوند که امکان‌پذیر نیست. ولی فعالیتهای ۱ و ۳ امکان‌پذیر است زیرا هر دو مهلت‌شان زمان ۲ است یعنی می‌توان یکی را در زمان ۱ و دیگری را در زمان ۲ انجام داد.

الگوریتم:

کارها را برحسب ترتیب نزولی سودشان مرتب می‌کنیم و به ترتیب آنها را انتخاب می‌کنیم و هر کار که انتخاب می‌شود را به‌طور موقت به مجموعه فعالیت‌های انتخاب شده تاکنون (j) اضافه می‌کنیم و بعد بررسی می‌کنیم مجموعه جدید، امکان‌پذیر است یا خیر. برای بررسی امکان‌پذیر بودن کافیست کارهای موجود در j را به ترتیب صعودی مهلت‌هایشان مرتب کنیم و فقط امکان‌پذیر بودن این حالت را بررسی کنیم یعنی لازم نیست همه جایگشت‌ها را بررسی کنیم.

مثال: بهترین زمان‌بندی برای مقادیر زیر را جهت به دست آوردن حداکثر سود بیابید.

$(p_۱, p_۲, p_۳, p_۴, p_۵, p_۶, p_۷) = (۴۰, ۳۵, ۳۰, ۲۵, ۲۰, ۱۵, ۱۰)$

$(d_۱, d_۲, d_۳, d_۴, d_۵, d_۶, d_۷) = (۳, ۱, ۱, ۳, ۱, ۳, ۲)$

پاسخ: سودها به صورت نزولی مرتب هستند.

۱- ابتدا مجموعه j را برابر ϕ قرار می‌دهیم $j = \{\ \}$

۲- کار ۱ را انتخاب می‌کنیم $j = \{۱\}$، امکان‌پذیر هست.

۳- کار ۲ را انتخاب می‌کنیم $j = \{۱, ۲\}$ قبول می‌شود زیرا ترتیب [۲,۱] امکان‌پذیر است.

۴- کار ۳ انتخاب نمی‌شود چون ترتیب [۳,۲,۱] با مهلت‌های [۱,۱,۳] امکان‌پذیر نیست.

۵- کار ۴ انتخاب می‌شود و $j = \{۱,۲,۴\}$. این مجموعه با توجه به ترتیب [۲,۱,۴] و مهلت‌های [۱,۳,۳] قابل قبول است.

۶- کار ۵ انتخاب نمی‌شود زیرا ترتیب [۲,۵,۱,۴] با توجه به مهلت‌های [۱,۱,۳,۳] امکان‌پذیر نیست.

۷- کار ۶ انتخاب نمی‌شود زیرا ترتیب [۲,۱,۴,۶] با توجه به مهلت‌های [۱,۳,۳,۳] امکان‌پذیر نیست.

۸- کار ۷ انتخاب نمی‌شود زیرا ترتیب [۲,۷,۱,۴] با توجه به مهلت‌های [۱,۲,۳,۳] امکان‌پذیر نیست.

پس جواب مسأله، مجموعه $j = \{۱,۲,۴\}$ و ترتیب ممکن [۲,۱,۴] یا [۲,۴,۱] است و سود کل $۱۰۰ = ۲۵ + ۳۵ + ۴۰$ است.

بنابراین الگوریتم زمان‌بندی فعالیتها با مهلت معین، ابتدا باید n کار را به ترتیب نزولی سودشان مرتب کند که زمان $\theta(n\,Lgn)$ نیاز دارد و سپس در هر مرحله بررسی کند که آیا انتخاب انجام شده، امکان‌پذیر است یا خیر که مرتبه این عمل $\theta(n^۲)$ است. پس مرتبه کل $\theta(n^۲ + n\,Lgn)$ یا $\theta(n^۲)$ است.

آنالیز استهلاکی (Amortized Analysis)

در این نوع آنالیز، n عمل انجام می‌شود و هزینه زمانی آن روی n عمل سرشکن می‌شود. در واقع ممکن است عملیات انجام شده هزینه‌های مختلفی داشته باشند و برخی هزینه زیادی داشته باشند ولی هزینه متوسط هر عمل کوچک باشد. لازم به ذکر است که آنالیز استهلاکی با آنالیز در حالت میانگین متفاوت است و روش‌های آماری را شامل نمی‌شود. در واقع آنالیز استهلاکی زمان اجرای متوسط هر عمل در بدترین حالت را نشان می‌دهد.

برای این نوع آنالیز ۳ روش مطرح است:

- آنالیز تجمعی (Aggregate Analysis)
- روش حسابرسی (Accounting method)
- روش پتانسیل (Potential Method)

آنالیز تجمعی

در این روش هزینه n عمل را به دست می‌آوریم سپس به n تقسیم می‌کنیم. مثلاً n عمل push، pop و Multipop (k) روی پشته s که در ابتدا خالی است انجام داده‌ایم (Multipop(k)، k عنصر از پشته pop می‌کند، البته اگر کمتر از k عنصر در پشته باشند، همه عناصر pop می‌شوند) درست است که هزینه multipop (k) ممکن است O(k) باشد ولی n عمل از عملیات فوق روی هم هزینه O(n) دارند پس هزینه هر عمل به صورت سرشکن شده برابر $O(۱)$ است.

مثال: فرض کنید آرایه $A[۰..k-۱]$ شامل یک عدد باینری است که $A[۰]$ بیت کم ارزش است بنابراین اگر x مقدار عدد باشد: $x = \sum_{i=۰}^{k-۱} A[i]\cdot ۲^i$.

ابتدا $x = 0$ یعنی عناصر A همگی صفر هستند، می‌خواهیم x را با یک جمع کنیم الگوریتم به صورت زیر است:

```
INC(A)
i ← 0
while i < k and A[i] = 1 do A[i] ← 0   i ← i + 1
if i < k then A[i] ← 1
```

می‌خواهیم هزینه زمانی n عمل INC را به دست آوریم. هزینه هر تغییر بیت را یک فرض کنید.

با اولین عمل INC، یک تغییر بیت داریم (A[0]) با دومین عمل،دو تغییر بیت (A[1], A[0]) با سومین عمل، یک تغییر بیت (A[0]) و ...

ولی می‌توان بهتر به مسئله نگاه کرده بیت A[0] با هر بار اجرای عمل INC، تغییر می‌کند A[1]، $\left\lfloor \frac{n}{2} \right\rfloor$ بار تغییر می‌کند. A[2]، $\left\lfloor \frac{n}{4} \right\rfloor$ بار و به طور کلی A[i]، $\left\lfloor \frac{n}{2^i} \right\rfloor$ بار تغییر می‌کند که $i = 0,1,\ldots,\lfloor Lgn \rfloor$ دقت کنید برای $i > Lgn$، بیت A[i] اصلاً تغییر نمی‌کند. بنابراین تعداد کل تغییر بیت‌ها:

$$\sum_{i=0}^{\lfloor Lgn \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

پس در بدترین حالت هزینه n عمل INC، O(n) است. پس متوسط هزینه هر عمل یعنی هزینه مستهلک شده به‌ازای هر عمل $\frac{O(n)}{n} = O(1)$ است.

تمرین: نشان دهید اگر عمل DEC نیز وجود داشته باشد، آنگاه n عمل می‌تواند هزینه‌ای معادل $\theta(nk)$ داشته باشد.

مثال: دنباله‌ای از n عمل روی یک ساختمان داده اجرا شده است. هزینه عمل i ام برابر i است اگر i توانی از ۲ باشد و در غیر این صورت هزینه عمل برابر ۱ است.

هزینه استهلاکی به ازای هر عمل را بیابید.

پاسخ:

C_i = هزینه عمل i ام

operation	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	...
cost	۱	۲	۱	۴	۱	۱	۱	۸	۱	۱	...

هزینه n عمل $= \sum_{i=1}^{n} C_i \le n + \sum_{j=0}^{\lfloor Lgn \rfloor} 2^j = n + (2n - 1) < 3n$

هزینه میانگین هر عمل $= \frac{\text{هزینه کل}}{\text{تعداد عملیات}} < \frac{3n}{n} = 3$

روش حسابرسی

در این روش به عملیات مختلف، شارژهای مختلفی نسبت می‌دهیم که ممکن است شارژی که نسبت می‌دهیم، از هزینه واقعی عمل کمتر یا بیشتر باشد. مقداری که یک عمل شارژ می‌شود را «هزینه استهلاک» آن عمل گویند.

اگر «هزینه استهلاک» یک عمل از «هزینه واقعی» آن عمل بیشتر باشد، تفاوت این دو به عنوان «اعتبار» به عناصر خاصی از ساختمان داده تخصیص می‌یابد. این اعتبار را می‌توان بعداً برای عملیاتی که هزینه استهلاکشان کمتر از هزینه واقعی‌شان است، صرف کرد.

به عنوان مثال عملیات پشته را در نظر بگیرید. می‌دانیم هزینه واقعی عملیات عبارت است از:

push=۱ ، pop=۱ ، $Multipop(k) = \min(k,s)$

که s تعداد عناصر موجود در پشته هنگام فراخوانی Multipop است.

ولی ما هزینه استهلاکی زیر را به عملیات نسبت می‌دهیم:

push ۲، pop ۰ ، Multipop ۰

با این مقادیر، می‌توان هر دنباله از عملیات را پشتیبانی کرد. هر push که انجام می‌شود، هزینه واقعی‌اش ۱ است، پس از شارژ ۲ مقداری آن، ۱ مقدار برای هزینه‌اش صرف می‌شود و ۱ مقدار در عنصر push شده به عنوان اعتبار ذخیره می‌شود، پس تمام عناصری که در پشته هستند، اعتبار ۱ مقداری دارند. حال اگر pop انجام شود، چون هزینه واقعی pop، ۱ است، این ۱ را از اعتبار عنصری که pop می‌شود، دریافت می‌کند. پس از آنجایی که هزینه استهلاکی push، pop و Multipop ثابت ($O(1)$) است پس هزینه n عمل، $O(n)$ است.

به عنوان مثالی دیگر، عمل INC (جمع عدد k بیتی با ۱) را در نظر بگیرید. هزینه واقعی INC برابر تعداد بیت‌هایی است که عوض می‌شود.

حال فرض کنید هزینه استهلاکی تغییر صفر به یک را ۲ واحد در نظر بگیریم. و برای تغییر یک به صفر، هزینه استهلاکی صفر وقتی بیتی ۱ می‌شود، ۱ واحد از شارژش را صرف هزینه واقعی‌اش می‌کنیم و ۱ واحد را در بیت ذخیره می‌کنیم، بنابراین هر ۱ در عدد دارای یک اعتبار ۱ واحدی است که می‌توان آن را وقتی صرف کرد که بیت مربوطه صفر شود. از آنجایی که هر عمل INC حداکثر یک بیت را ۱ می‌کند پس هزینه استهلاکی هر عمل INC برابر ۲ واحد است، پس هر عمل INC دارای هزینه استهلاکی $O(1)$ است پس n عمل INC دارای هزینه استهلاکی $O(n)$ است.

توجه: تخصیص هزینه استهلاکی به هر عمل، دقت می‌خواهد اگر C_i هزینه واقعی عمل i ام و $\hat{C}_i$ هزینه استهلاکی عمل i ام باشد آنگاه باید $\sum_{i=1}^{n} \hat{C}_i \geq \sum_{i=1}^{n} C_i$

روش پتانسیل

روش قبلی، اعتبار را به عنصر تخصیص می‌داد ولی این روش، اعتبار را به صورت پتانسیل به کل ساختمان داده تخصیص می‌دهد.

فرض کنید D_0 وضعیت اولیه ساختمان داده‌ای است که می‌خواهیم n عمل روی آن انجام دهیم. C_i هزینه واقعی عمل i ام، D_i وضعیت ساختمان داده است پس از انجام عمل i ام روی D_{i-1}. تابع پتانسیلی تعریف می‌کنیم به اسم ϕ که به هر وضعیت ساختمان داده، یک عدد حقیقی نسبت می‌دهد:

$\phi(D_i)$ = یک عدد حقیقی

بنابراین هزینه استهلاکی $(\hat{C}_i)$ عمل i ام برابر است با:

$$\hat{C}_i = C_i + \phi(D_i) - \phi(D_{i-1})$$

پس کل هزینه استهلاکی برابر است با:

$$\sum_{i=1}^{n} \hat{C}_i = \sum_{i=1}^{n} (C_i + \phi(D_i) - \phi(D_{i-1})) = \sum_{i=1}^{n} C_i + \phi(D_n) - \phi(D_0)$$

مثلاً فرض کنید در عملیات پشته (Multipop , pop , push)، تابع پتانسیل ϕ را برابر عناصر موجود در پشته فرض کنیم. در حالت پشته خالی D_0، که حالت شروع است $\phi(D_0) = 0$. از آنجایی که تعداد عناصر موجود در پشته عددی نامنفی است پس همیشه $\phi(D_i) \geq 0 = \phi(D_0)$. پس هزینه کل مستهلک شده برای n عمل با توجه به تابع ϕ، یک کران بالا برای هزینه واقعی است. حال هزینه مستهلک شده را برای push و pop و Multipop به دست می‌آوریم.

اگر عمل i ام روی پشته push باشد و در حال حاضر تعداد s عنصر در پشته باشد آنگاه:

$$\phi(D_i) - \phi(D_{i-1}) = (s+1) - s = 1$$

بنابراین هزینه مستهلک شده push برابر است با:

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1+1 = 2$$

به همین ترتیب نشان دهید که هزینه مستهلک شده pop و Multipop برابر صفر است. پس هزینه مستهلک شده هر عمل $O(1)$ است. بنابراین از آنجایی که هزینه مستهلک شده کران بالای هزینه واقعی است پس هزینه واقعی n عمل برابر $O(n)$ است.