## مسئله  کوله پشتی صفر و یک :

شکلی از صورت مسئله این گونه بیان می شود:

دزدی با یک کوله پشتی وارد یک جواهر فروشی می شود، هر قطعه در این جواهر فروشی دارای ارزش و وزن معینی است. هم چنین حداکثر وزن قابل تحمل توسط کوله پشتی Wمی باشد. مسئله ای که دزد با آن مواجه است، تعیین حداکثر ارزش قطعات است در حالیکه وزن کل آنها از حد معین W فراتر نرود. می توان مسئله را به صورت زیر بیان کرد:

فرض کنید n قطعه وجود داشته باشد، به طوری که:

$S=\{ item_1 , itme_2 , item_3 ,..., item_n \}$

$w_i= item_i$  وز ن

$P_i = item_i$  ارزش

W= حداکثر وزنی که کوله پشتی قادر به تحمل آن است

باید زیرمجموعه A از S به قسمی تعیین شود که مجموع ارزش قطعات A ماکزیمم شود در حالیکه مجموع وزن آنها از W فراتر نرود.

راه حل کورکورانه این است که همه زیرمجموعه های این n قطعه را در نظر بگیریم، زیرمجموعه هایی را که وزن کل آنها ازW فراتر رود، کنار می گذاریم و از میان آنهایی که باقی می ماند، آن را که بیشترین ارزش را دارد، انتخاب می کنیم. تعداد زیرمجموعه های یک مجموعه n عضوی، $2^n$ است، بنابراین زمان الگوریتم کورکورانه حداقل نمایی است.

راه حل های حریصانه:

یک راهبرد حریصانه واضح، این است که قطعاتی با بیشترین ارزش زودتر از همه برداشته شوند. ولی اگر با ارزش ترین قطعات در مقایسه با ارزشی که دارند، وزن بالایی داشته باشند این راهبرد به خوبی کار نمی کند.

برای مثال، فرض کنید سه قطعه داریم، که اولی دارای وزن 25 پوند و ارزش $10 و دومی و سومی هر یک به وزن 10 پوند و ارزش $9 است. اگر W یعنی ظرفیت کوله پشتی 30 پوند باشد، این راهبرد حریصانه فقط $10 سود دارد، حال آنکه حل بهینه، $18 است.

یک راهبرد حریصانه دیگر قرار دادن سبک ترین قطعات در ابتداست. هنگامی که قطعات سبک در مقایسه با وزنی که دارند، کم ارزش باشند، این راهبرد نیز به شکست می انجامد.

برای پرهیز از مشکلاتی که این دو راهبرد حریصانه ایجاد می کنند، یک راهبرد حریصانه پیچیده تر این است که ابتدا قطعاتی با بزرگترین ارزش به ازای واحد وزن برداشته شوند. یعنی قطعات را بر حسب ارزش آنها به ازای واحد وزن مرتب و آنها را به ترتیب انتخاب کنیم. هر قطعه در صورتی در کوله پشتی گذاشته می شودکه وزن کل فراتر از W نرود. با یک مثال نشان می دهیم که این راهبرد نیز الزاما ما را به جواب بهینه نمی رساند.

سه قطعه با مشخصات روبه رو داریم:

| $item_i$ | 1 | 2 | 3 |
|---|---|---|---|
| $p_i$ | $50 | $60 | $140 |
| $w_i$ | 5 | 10 | 20 |

$$item_1 : \frac{\$50}{5} = \$10 \qquad item_2 : \frac{\$60}{10} = \$6 \qquad item_3 : \frac{\$140}{20} = \$7$$

روش حریصانه اخیر، قطعه اول و سوم را انتخاب می کند. که سود کل $190 را می دهد، حال آن که حل بهینه انتخاب قطعه دوم و سوم است که $200 سود را در بردارد.

اما اکنون روش برنامه ریزی پویا را مطرح می کنیم:

از اینجا به بعد شکل دیگری از صورت مسئله را مورد بررسی قرار می دهیم که معادل با شکل قبلی است و به صورت زیر بیان می شود:

n فایل داده ای داریم ومی خواهیم آنها را بر روی حافظه ای با حجم W بایت ذخیره کنیم، فایل i دارای حجم $w_i$ بایت است و زمان $v_i$ دقیقه صرف می کند. هدف ما این است که زیر مجموعه ای از مجموعه فایل های

داده ای را بیابیم به طوری که مجموع زمانهای صرف شده برای آنها بیشینه شود در حالیکه مجموع حجم آنها از فضای حافظه یعنی W فراتر نرود.

تعریف رسمی: دو، n تایی از اعداد مثبت $(v_1,...,v_n)$ و $(w_1,...,w_n)$ و $W>0$ داده شده است، هدف تعیین زیر مجموعه $T \subseteq \{1,...,n\}$ که $\sum v_i$ بیشینه شود با این محدودیت که $\sum w_i \leq W$

توجه: در هر دو شکل صورت مسئله، قطعه و یا فایل، یا به طور کامل برداشته می شوند و یا اصلا برداشته نمی شوند و نمی توان کسری از آنها را برداشت، از این رو به این مسئله، صفر و یک می گویند.

برای استفاده از روش برنامه ریزی پویا، نخست باید نشان دهیم زیر ساختار بهینه برقرار است:

فرض می کنیم $A=\{1, 2,...,i\}$ مجموعه اندیس های فایل های داده ای و W حجم حافظه ما می باشد،

اکنون فرض می کنیم T جواب بهینه مسئله ماست، یعنی $T \subseteq \{1, 2,...,i\}$ به طوریکه مجموع زمانهای فایل های داده ای T بیشینه است در حالیکه مجموع حجم فایل های آن از W فراتر نرود، دو حالت اتفاق می افتد:

حالت اول) اگر $i$ عضو T نباشد، T جواب بهینه برای $(i-1)$ فایل نخست با سایز حداکثر W است. زیرا که اگر T جواب بهینه برای $(i-1)$ فایل نخست با سایز حداکثر W نباشد،می توان جواب بهینه ای هم چون S برای $(i-1)$ فایل نخست با حداکثر سایز W یافت و طبق تکنیک ببر و بچسبان به جای T قرار داد و چون فایل $i$ ام نیز به S اضافه نمی شود، پس S جواب بهینه برای $i$ فایل نخست با سایز حداکثر W نیز هست و این با فرض اینکه T جواب بهینه برای $i$ فایل نخست می باشد، تناقض دارد، پس T جواب بهینه برای $(i-1)$ فایل نخست با سایز حداکثر W است.

حالت دوم) $i \in T$ باشد، اگر T حاوی فایل $i$ باشد، زمان کل حاصل از فایل های موجود در T برابر $v_i$ به اضافه زمان بیشینه است وقتی که فایل ها را بتوان از $(i-1)$ فایل نخست انتخاب کرد( البته با رعایت این محدودیت که مجموع سایز آنها از $W-w_i$ فراتر نرود.) جواب بهینه برای $(i-1)$ فایل نخست با سایز حداکثر $W-w_i$ را R می نامیم.T حتما شامل R است، چرا که در غیر این صورت می توان زیرمجموعه ای چون S از$(i-1)$ فایل نخست با سایز حداکثر $W-w_i$ یافت که مجموع زمان های فایل های S بیشینه شود و از مجموع زمان های فایل های R بیشتر

شود، اکنون طبق تکنیک ببر و بچسبان می توان S را به جای R قرار داد و هنگامی که $v_i$ را به S اضافه می کنیم مجموع زمان های $SU\{v_i\}$ از $T=RU\{v_i\}$ بیشتر شده واین موضوع با فرضمان که T یک جواب بهینه مسئله هست تناقض دارد.

اکنون می توان رابطه بازگشتی زیر را نوشت:

V[i,w]= مجموع زمان بیشینه زیر مجموعه ای از i فایل نخست با سایز حداکثر w

$V[i,w] = axm(V[i-1,w] , v_i+V[i-1,w-w_i])$ for $1\le i\le n$ , $0 \le w\le W$

$V[0,w] = 0$ for $0\le w \le W$ (no item)

$V[i,w] = -\infty$ for $w < 0$ ( illegal)

اکنون به سراغ الگوریتم می رویم:

```
KnapSack (v,w,n,W)
{
    for (w=0 to W)      V[0,w]=0;
    for( i=1 to n)
        for(w=0 to W)
            if((w[i]<=w) and (v[i]+V[i-1,w-w[i]]>V[i-1,w])) {
                V[i,w]=v[i]+V[i-1,w-w[i]];
                Keep[i,w]=1;
            }
            else {
                V[i,w]=V[i-1,w];
```

```
                Keep[i,w]=0;
            }
    K=W;
    for(i=n downto 1)
        if(Keep[i,K]= =1){
            Output i;
            K=K-w[i];
        }
    return V[n,W];
}
```

همانطور که می بینیم مسئله با پر کردن سطر به سطر، ماتریس V[0..n,0..W] حل می شود. هم چنین ماتریس Keep برای ساختن جواب بهینه مورد استفاده قرار می گیرد. در ادامه به ذکر یک مثال می پردازیم:

چهار فایل داده ای به همراه زمان و ارزش هر یک مطابق زیر داده شده است، هم چنین حجم حافظه ما نیز W=10 می باشد، آرایه V، مقدار جواب بهینه و خود جواب بهینه در ادامه آمده است:

| i | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $v_i$ | 10 | 40 | 30 | 50 |
| $w_i$ | 5 | 4 | 6 | 3 |

| V[i,w] | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| i=0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 2 | 0 | 0 | 0 | 0 | 40 | 40 | 40 | 40 | 40 | 50 | 50 |
| 3 | 0 | 0 | 0 | 0 | 40 | 40 | 40 | 40 | 40 | 50 | 70 |
| 4 | 0 | 0 | 0 | 50 | 50 | 50 | 50 | 90 | 90 | 90 | 90 |

همان طور که دیدیم مقدار جواب بهینه 90 و جواب بهینه، زیر مجموعه $\{4,2\}$می با شد.

تحلیل الگوریتم:

الگوریتمی که ارائه شد، از$O(Wn)$ می باشد. توجه شود که میان $n$ و $W$ رابطه ای وجود ندارد، بنابراین برای یک $n$ مفروض، می توان با گرفتن مقادیر بزرگ و دلخواه از $W$، نمونه هایی ایجاد کرد که زمان اجرای آنها به قدر دلخواه بزرگ باشد. هنگامی که $W$ در مقایسه با $n$ بسیار بزرگ باشد، این الگوریتم از الگوریتم کورکورانه که همه زیر مجموعه ها را در نظر می گرفت، بدتر است. این الگوریتم را می توان چنان بهبود بخشید که در بدترین حالت تعداد عناصر محاسبه شده به$O(2^n)$ تعلق داشته باشد.با این بهبود، الگوریتم هیچگاه از روش کورکورانه بدتر نمی شود و غالبا بسیار بهتر است. این بهبود مبتنی بر این واقعیت است که نیازی به تعیین عناصر سطر $i$ ام به ازای هر $w$ میان $0$ و $W$ نیست، بلکه در سطر $n$ ام فقط باید $V[n ,W]$ محاسبه شود. بنابراین، تنها عناصر مورد نیاز در سطر $(n-1)$ ام،آنهایی هستند که برای محاسبه $V[n,W]$ به کار می روند، یعنی$V[n-1,W]$ و $V[n-1,W-w_n]$ . از $n$ به طرف عقب ادامه می دهیم تا تعیین کنیم کدام عناصر مورد نیازند. حال ببینیم کارایی این نسخه در بدترین حالت چقدر است. توجه دارید که حداکثر $2^i$ عنصر را در سطر$(i-1)$ ام محاسبه می کنیم. بنابراین حداکثر تعداد عناصر محاسبه شده عبارت است از :

$$1+2+2^2+...+2^{n-1} = 2^n -1$$

پس تعداد عناصر محاسبه شده در بدترین حالت به$O(2^n)$ تعلق دارد.

با تر کیب این دو نتیجه، تعداد عناصر محاسبه شده در بدترین حالت$O(\text{muminim}(2^n, nW))$ تعلق دارد.

تا کنون کسی برای مسئله کوله پشتی صفر و یک الگوریتمی نیافته است که بدترین حالت آن بهتر از زمان نمایی باشد و در عین حال هنوز کسی عدم امکان آن را نیز اثبات نکرده است.

**مسعود صدیقی**

**آذر ماه 1390**