زمان بندی فاصله های وزن دار

تعریف مسئله:

تعداد n درخواست نشانه گذاري شده داريم بطوريكه هر درخواست i يک زمان شروع S_i و يک زمان پايان f_i را مشخص می کند .هر فاصله i يک مقدار يا وزن v_i نيز دارد. دو فاصله سازگار هستند اگر قسمت اشتراکی نداشته باشند .هدف مسئله فعلی ما انتخاب يک زير مجموعه S از $\{1,\dots,n\}$ از فاصله هاي سازگار دوطرفه می باشد بطوريكه جمع مقادير انتخاب شده ماكسيمم شود.

راه حل كوركورانه:

يک روش بديهی برای بدست آوردن جواب اين است كه تمام زير مجموعه های فاصله ها را در نظر بگيريم و از بين آنها،زيرمجموعه هايی كه فاصله هايش با هم سازگار هستند را بررسی كنيم . زير مجموعه ای كه بيشترين مقدار را به ما می دهد به عنوان جواب مسئله در نظر بگيريم.

هر چند اين روش جواب درست را به ما می دهد ولی چون مجموعه فاصله ها دارای 2^n زيرمجموعه است بنابراين استفاده از اين روش به $O(2^n)$ زمان نياز دارد يعنی از مرتبه نمايی است كه عملا برای nهای بزرگ كار آمد نخواهد بود.

راه حل حريصانه:

انتخاب كار ها بر اساس زمان پايان (كاری كه زودتر تمام می شود اول انتخاب می شود)

هر چند اين راه حل برای شرايط خاصی جواب درست را به ما می دهد ولی در حالت كلی ما را به جواب بهينه نمی رساند.

- ❖ مثال نقض:

999

1

تذكر: اين راه حل حريصانه در حالتی كه همه ی وزن ها برابر يک باشد جواب درست را به ما می دهد.

چند راه حل حريصانه ديگر:

- ✓ *بيشترين سود:* كاری كه بيشترين سود را دارد ابتدا انتخاب شود .
- ❖ *مثال نقض:*

99 99

100

- ✓ *بیشترین نسبت وزن به طول*: کاری که نسبت بین وزن به طول آن بیشتر است ابتدا انتخاب شود.
  - ❖ *مثال نقض:*

99 99

100

- ✓ *کوتاهترین طول*: کاری که کوتاهترین طول را دارد ابتدا انتخاب شود.
  - ❖ *مثال نقض:*

999

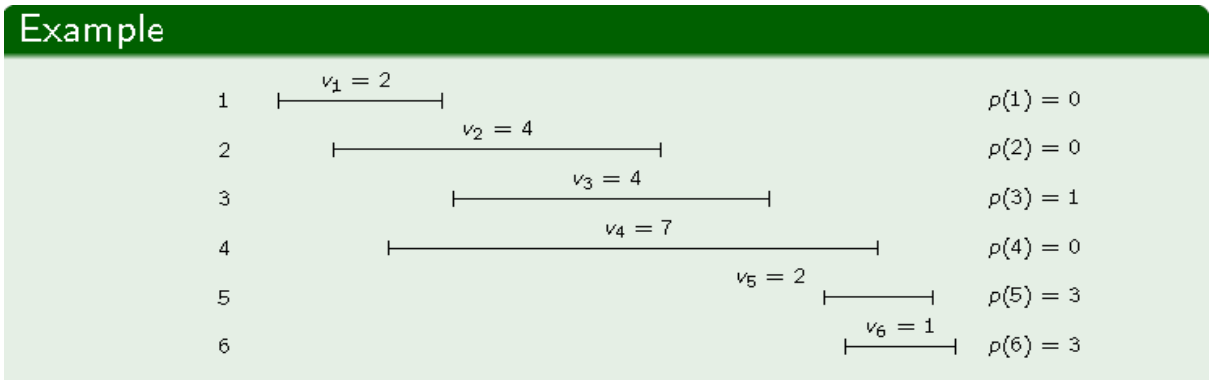
1

- ✓ ...

راه حل بازگشتی:

1. کار ها را بر اساس زمان پایانشان شماره گذاری می کنیم به طوری که به ازای هر $i<j$ داشته باشیم $f_i \leq f_j$
2. $P(j)$ را براي يک فاصله j ، بزرگترين $i<j$ که فاصله هاي i و j با هم سازگار هستند(یعنی قسمت اشتراکی ندارند)،تعریف می کنیم.

ما $P(j)=0$ را زمانی تعریف می کنیم که هیچ درخواست $i<j$ با j سازگار نباشد(حداقل یک i با j قسمت اشتراکی داشته باشد.)مثالی براي عملکرد $P(j)$ در شکل زیر نشان داده شده است.



شکل 1: مثالی از چگونگی عملکرد P(j)

حال یک مسئله ي زمانبندي فاصله هاي وزن دار داده شده است ، بياييد تا راه حل بهينه O را براي آن ارائه کنيم .

مطلب کاملا واضحی که براي O می توان گفت اين است که :

درخواست n (آخرين درخواست) وابسته[1] به O است يا خير.

[1] وابسته بودن n به O یعنی راه حل بهینه O شامل فاصله nام می شود یا نه.

اگر $n \in 0$ باشد بنابراین به طور قطع هیچ فاصله اي بین $P(n)$ و n مشخص نشده است که بتواند وابسته به Oباشد زیرا طبق تعریف $P(n)$ می دانیم که فاصله هاي $P(n)+1,P(n)+2,...,n-1$ همگی قسمت مشترکی با فاصله ي n ام دارند.

به علاوه اگر$n \in 0$ باشد پس باید شامل یک راه حل بهینه براي مسئله اي شامل درخواست هاي $\{1,...,P(n)\}$ باشد زیرا اگر شامل این درخواستها نشود ما می توانستیم انتخاب درخواست O از بین $\{1,...,P(n)\}$ را بدون خطر مورد اشتراك قرارگیري با درخواست n با یک مورد بهتر جایگزین کنیم.(تکنیک *Cut-Paste*)

از طرف دیگر اگر n عضو O نباشد بنابراین O به سادگی برابر می شود با راه حل بهینه براي حل مسئله اي شامل درخواستهای $\{1,...,n-1\}$. این مطلب کاملا مدلل و دلیل پذیر است: فرض می کنیم که O شامل درخواست n نمی شود؛ بنابراین اگر مجموعه درخواستهاي بهینه از بین $\{1,...,n-1\}$ انتخاب نشود ما می توانیم آن را با راه حل بهتري جایگزین کنیم.(تکنیک *Cut-Paste*)

تمامی این پیشنهادات، این بحث را مطرح می کنند که پیدا کردن راه حل بهینه بر روي فاصله هاي $\{1,2,...,n\}$ نیازمند گشتن به دنبال راه حل هاي بهینه از مسائل کوچکتر به شکل $\{1,2,...,j\}$ می باشد. بنابراین براي هر مقدار j بین 1 تا n اجازه بدهید O_j راه حل بهینه را براي مسئله شامل درخواستهاي $\{1,2,...,j\}$ را مشخص کند و اجازه دهید $OPT(j)$ مقدار این راه حل را مشخص کند($OPT(0)$ را بر اساس این قرارداد تعریف می کنیم که، برابر است با مقداربهینه روي یک مجموعه تهی از فاصله ها.) راه حل بهینه اي که ما دنبال آن هستیم دقیقا O_n می باشد .براي راه حل بهینه O_j روي $\{1,2,...,j\}$ دلیلی که در بالا به آن پرداختیم(به طور کلی حالتی که $j=n$ باشد) این مطلب را بیان می کند که چنانچه j عضو O_j باشد در این حالت $OPT(j) = v_j + OPT(P(j))$ و یا چنانچه j عضو O_j نباشد در این صورت $OPT(j)=OPT(j-1)$ زیرا اینها دقیقا دو حالت ممکن انتخاب هستند (j عضو O_j باشد یا نباشد) ، می توانیم بگوییم:

$$OPT(j) = max\ (\ v_j + OPT\ (\ P(j)\ )\ ,\ OPT(\ j-1\ )\ ) \quad *$$

و چگونه تشخیص می دهیم که آیا n وابسته به راه حل بهینه ي O_j می باشد؟ این هم ساده است: وابسته به راه حل بهینه ي O_j خواهد بود اگر و فقط اگر اولین انتخاب هاي بالا حداقل به اندازه ي دومی آنها خوب باشد؛ به عبارت دیگر:

درخواست j وابسته به یک راه حل بهینه بر روي $\{1,2,...,j\}$ است اگر و تنها اگر

$$V_j + OPT\ (\ P(j)\ ) \geq OPT\ (\ j-1\ ) \quad **$$

```
Compute-Opt(j)
    If j = 0 then
        Return 0
    Else
        Return max ( v_j + Compute-Opt(p(j)) , Compute-Opt(j-1) )
    Endif
```

درستی الگوریتم به طور مستقیم به وسیله ي استقراء بر روي j نتیجه می شود:

Copmute-Opt(j) به طور صحیح *OPT(j)* را براي هر$j=1,2,...,n$ محاسبه می کند.

اثبات به وسیله ي تعریف $Opt(0)=0$. حال چند $j>0$ در نظر می گیریم و طبق فرض استقراء *Comput-Opt(i)* به طور صحیح *OPT(i)* را براي تمامی $i<j$ ها محاسبه می نماید .طبق فرضیه استقراء می دانیم که

Compute-Opt(P(j)) = OPT(p(j))

Compute-Opt(j-1) = OPT(j-1)

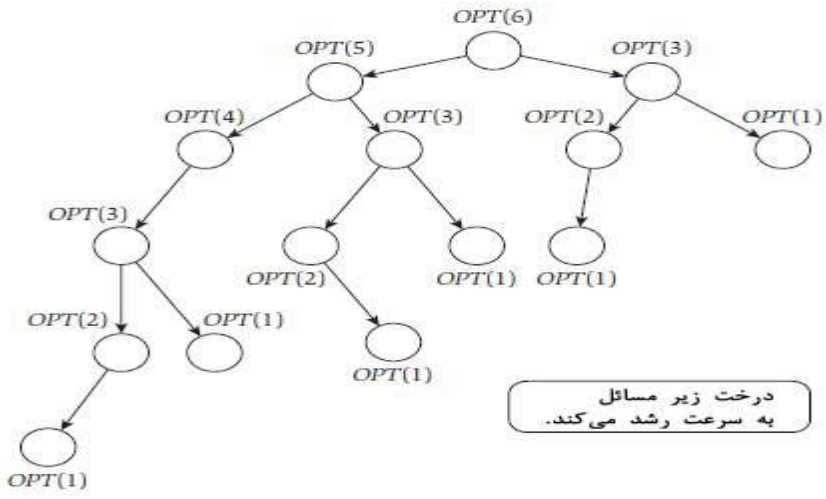
در نتیجه از رابطه * می دهد:

$$OPT(j) = max(v_j + Compute\text{-}Opt(p(j)) , Compute\text{-}Opt(j-1))$$
$$= Compute\text{-}Opt(j)$$

زمان اجرا :

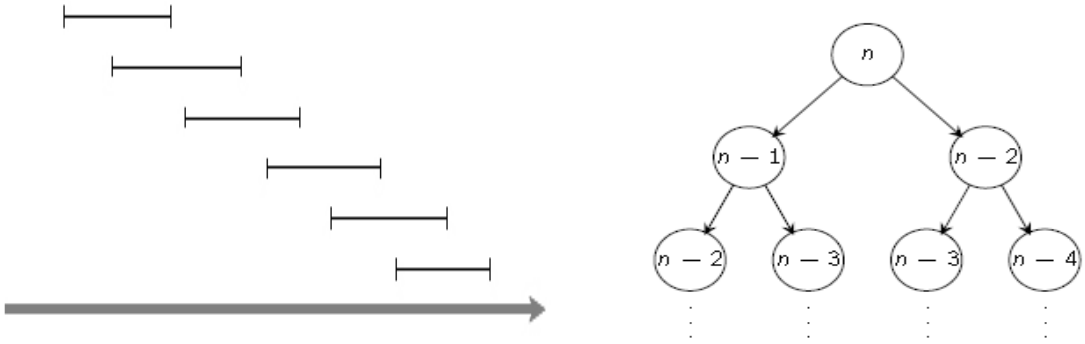
$$T(n) = T(p(n)) + T(n-1) + O(1)$$

متاسفانه ، اگر واقعا بخواهیم الگوریتم *Compute-Opt* را همانطور که نوشتیم پیاده سازي کنیم، در بدترین حالت، اجراي آن یک زمان نمائی را در بر خواهد داشت .براي مثال شکل-2 را مشاهده کنید براي درخت فراخوانی که از فاصله ي شکل-1 نتیجه شده است درخت بنا به شاخه هاي بازگشتی آن به سرعت گسترش می یابد.



شکل 2: درخت فراخوانی

به عنوان یک مثال سخت تر، براي یک فاصله اي که خوب لایه بندي شده است مانند آنچه در شکل-3 موجود است، جایی که $P(j)=j-2$ است براي هر $j=2,3,4,\ldots,n$ می بینیم که *Compute-Opt(j)* فراخوانی هاي بازگشتی جداگانه اي را روي اندازه هاي $j-1,j-2$ تولید می کند .به عبارت دیگر تعداد کل ارجاع ها براي ساخت *Compute-Opt* روي این فاصله مانند اعداد فیبونانچی رشد می کنند که به صورت نمائی در حال زیاد شدن هستند .بنابراین ما به یک راه حل با زمان اجرای چند جمله اي دست نیافتیم.



شکل 3 : مثالی نامناسب برای راه حل بازگشتی و درخت فراخوانی مربوط به آن

زمان اجرا در بدترین حالت :

$$T(n) = T(n-1) + T(n-2) + O(1) = \theta(\varphi^n)$$

که در آن $\varphi = 1.618$ است و عدد طلایی نامیده می شود.

اثبات :

$$T(n) = r^n$$

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

$$r^2 = r + 1$$

$$r^2 - r - 1 = 0$$

$$r = \frac{1 \pm \sqrt{1^2 + 4}}{2} = 1.618 = \varphi$$

Memorize کردن بازگشت:

در حقیقت ما از داشتن یک الگوریتم با مرتبه ي زمانی چند جمله اي زیاد دور نیستیم زیرا الگوریتم بازگشتی *Opt-Compute* در واقع فقط $n+1$ زیر مسئله متفاوت را حل می کند:

Compute-Opt(0),Compute-Opt(1),…,Compute-Opt(n)

دلیل اینکه داراي زمانی نمائی براي اجرا هست به سادگی از افزونگی زمانی است که براي هر یک از فراخوانی ها صرف می کند.(منظور محاسبه هر یک از مقادیر بالا به دفعات زیاد است)

چگونه می توانستیم این افزونگی را برطرف کنیم؟ می توانستیم مقدار *Compute-Opt* را اولین باري که محاسبه کردیم در محلی که بطور سراسري قابل دسترسی باشد ذخیره کنیم و سپس به سادگی براي تمامی فراخوانی هاي بازگشتی آینده ،از این مقدار از پیش محاسبه شده ، استفاده کنیم.

استراتژي فوق را در روندي هوشمندانه تر به نام *M-Compute-Opt* انجام می دهیم.این روند ازیک آرایه *M[0,…,n]* استفاده خواهد کرد *M[j]* با مقدار تهی شروع خواهد شد، اما مقدار *Compute-Opt(j)* را به محض اینکه براي اولین بار تعیین شود نگه خواهد داشت .براي تعیین کردن *OPT(n)* به سراغ *M-Compute-Opt* می رویم.

```
M-Compute-Opt(j)
  If j = 0 then
    Return 0
  Else if M[j] is not empty then
    Return M[j]
  Else
    Define M[j] = max( v_j + M-Compute-Opt(p(j)) , M-Compute-Opt(j-1))
    Return M[j]
  Endif
```

تحلیل نسخه ي Memorize شده:

بسیار مشابه به پیاده سازي الگوریتم قبلی می باشد؛ اگرچه *Memoziation* زمان را کاهش داده است.

زمان اجراي *M-Compute-Opt(n)* از مرتبه ي *O(n)* می باشد.

اثبات:زمانی که براي یک فراخوانی *M-Compute-Opt* صرف شده است،بدون در نظر گرفتن زمان صرف شده در فراخوانی هاي بازگشتی اي که تولید میکند،*O(1)* می باشد.بنابراین زمان اجرا به وسیله ي زمان هاي ثابتی محدود شده است که تعدادي فراخوانی هر دفعه *M-Compute-Opt* را اجرا می کنند .چون پیاده سازي خود هیچگونه کران بالایی براي آن در نظر نگرفته است، ما می خواهیم به وسیله ي یک معیار خوب" پیشرفت "یک کران براي آن پیدا کنیم .بهترین معیار پیشرفت در اینجا تعداد ورودي هایی از *M* است که" تهی "نیستند .در ابتدا این عدد صفر می باشد؛ اما هر مرتبه که بازگشت را احضار می کند، دو فراخوانی بازگشتی به *M-Compute-Opt* را نتیجه می دهد و این باعث پر شدن یک ورودي می شود و بعلاوه تعداد ورودي هاي پر شده را نیز به وسیله ۱ افزایش می دهد .چون *M* فقط *n+1* ورودي دارد، این مطلب دریافت می شود که در بیشترین

حالت $O(n)$ فراخوانی به *M-Compute-Opt* می تواند وجود داشته باشد و بعلاوه همانطور که انتظار داشتیم زمان اجراي *M-Compute-Opt* از مرتبه ي $O(n)$ می شود.

راه حل پویا :

هر چند با استفاده از به خاطر سپاری زمان اجرای $O(n)$ بدست آمده است اما با استفاده از راه حل پویا نیز می توان به این زمان رسید.

```
M[0] = 0
for i = 1 to n
    M[i] = max(vi + M[p(i)], M[i-1])
```

طریقه ی محاسبه با الگوریتم پویا همانند راه حل به خاطر سپاری می باشد با این تفاوت که آرایه M از اول به آخر پر می شود.

محاسبه ي راه حل:

تا کنون ما مقدار یک راه حل بهینه را محاسبه کرده ایم؛ فرض کنید که مجموعه ي کاملا بهینه اي از فاصله ها را می خواهیم.توسعه دادن الگوریتم با اضافه شدن مقدار فاصله ها به طوریکه همچنان یک راه حل بهینه باشد ساده خواهد بود: می توانستیم یک آرایه ي اضافی S را نگهداریم به این ترتیب که $S[i]$ شامل یک مجموعه ي بهینه از فواصل وابسته به $\{1,2,\ldots,i\}$ می باشد.

```
M[0] = 0
S[0] is empty schedule
for i = 1 to n
    M[i] = max(vi + M[p(i)], M[i-1])
    S[j] = v_j + M[p(j)] < M[j-1] ? S[j-1] : S[p(j)] ∪ {j}
```

$S[i]$ به زمان $O(n)$ نیاز دارد تا به روز شود الگوریتم نیز به زمان $O(n)$ نیاز داشت پس زمان کلی مسئله به $O(n^2)$ تغییر می کند که این زمان مناسب نیست بنابراین جواب را از روش دیگری بدست می آوریم.

از رابطه ي ** می دانیم که j مربوط به یک راه حل بهینه براي مجموعه فاصله هاي $\{1,\ldots,j\}$ می باشد اگر و تنها اگر

$$v_j + OPT(p(j)) \geq OPT(j-1)$$

با توجه به این نظریه ما به روند ساده ي بعدي می رسیم که "دنبال کردن به عقب" در میان آرایه M براي پیدا کردن مجموعه فاصله هاي یک راه حل بهینه می باشد.

```
findSolution(int j)
    if (j=0) then
        return empty schedule
    if (vj + M[p(j)] ≥ M[j-1]) then
        return findSolution(p(j)) ∪ {j}
    else
        return findSolution(j-1)
```

به دلیل اینکه *FindSolution* به طور بازگشتی خودش را فقط براي مقدارهاي کوچکتر فراخوانی می کند، باعث ساختن مجموعه اي از $O(n)$ فراخوانی بازگشتی می شود؛و چون هر دفعه زمان ثابتی را براي فراخوانی صرف می کند ، داریم:

از آرایه M از مقادي بهینه زیر مسائل نتیجه می شود که *FindSolution* در زمان $O(n)$ یک راه حل بهینه را باز می گرداند.

اگر بخواهیم راه حل را به صورت غیر بازگشتی بدست آوریم می توانیم به صورت زیر عمل کنیم.

```
M[0] = 0
for i = 1 to n
    M[i] = max(vi + M[p(i)], M[i-1])
    if (vi + M[p(i)] > M[i-1])
        Decision[i] = 1 (* 1 means i included in solution M[i] *)
    else
        Decision[i] = 0 (* 0 means i not included in solution M[i] *)
S = φ, i = n
While (i > 0) do
    if (Decision[i] == 1)
        S = S ∪ {i}
        i = p(i)
   else
        i = i-1
Output S
```