Greedy Algorithms
(chapter 16)

مقدمه

- شبیه به dynamic programming.
- این روش هم برای مسائل بهینه‌سازی استفاده می‌شود.

ایده: وقتی که نیاز است که یک انتخاب انجام شود، آن انتخاب که به نظر می‌رسد در حال حاضر بهترین است.

- انتخاب locally optimal choice به امید آن انجام می‌شود که به یک globally optimal solution برسیم.

- Greedy Algorithm همیشه منجر به یک optimal solution نمی‌شود، ولی گاهی این اتفاق می‌افتد.

- در ادامه الگوریتم‌هایی را می‌بینیم که برای آن‌ها greedy منجر به جواب بهینه می‌شود.
- و همچنین نگاهی می‌کنیم به بعضی general characteristics هایی که در آن‌ها greedy algorithm جواب بهینه می‌دهند.

مثال

Activity Selection

n تا activity که نیاز به استفاده انحصاری از یک منبع مشترک دارند.

به عنوان مثال زمان‌بندی استفاده از یک classroom.

set of activities $S = \{a_1, \ldots, a_n\}$

a_i needs resource during period $[s_i, f_i)$

این بازه یک بازه نیم باز است که

s_i = start time

f_i = finish time

Goal : انتخاب بزرگترین مجموعه ممکن که شامل non overlapping activities است

(mutually compatible)

نکته :

- می توانست که این مسئله objective های بسیار دیگری نیز داشته باشد.

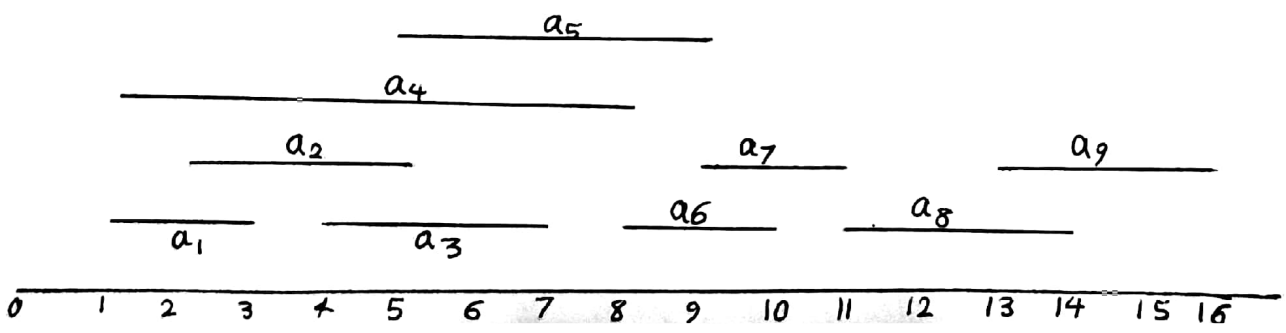
. زمان بندی اتاق برای longest time

. بیشترین درآمد سقف اجاره اتاق

Example : S sorted by finish time

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16

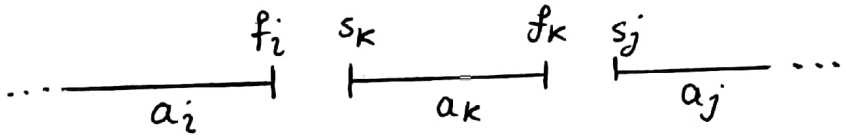
- Maximum-size mutually compatible set: $\{a_1, a_3, a_6, a_8\}$
- Not unique: also $\{a_2, a_5, a_7, a_9\}$



Optimal substructure of activity selection

$$S_{ij} = \{ a_k \in S : f_i \leq s_k < f_k \leq s_j \}$$

= شامل تمام activity هایی که بعد از پایان a_i شروع می‌شوند و قبل از شروع a_j پایان می‌پذیرند.



activity هایی که در S_{ij} هستند، با موارد زیر سازگار هستند

- تمام activity هایی که قبل f_i پایان می‌پذیرند.
- تمام activity هایی که بعد از s_j شروع می‌شوند.

برای نمایش تمام وکلیت مسئله، fictitious activities اضافه می‌کنیم:

$$a_0 = [-\infty, 0)$$

$$a_{n+1} = [\infty, "\infty+1")$$

اهمیتی به $-\infty$ در a_0 یا "$\infty+1$" در a_{n+1} نمی‌دهیم، پس

$$S = S_{0,n+1}$$

Range for S_{ij} is $0 \leq i, j \leq n+1$

فرض کنید که activity ها براساس زمان پایان افزایشی و یکنوا مرتب شده باشند:

$$f_0 \leq f_1 \leq f_2 \leq \cdots \leq f_n \leq f_{n+1}$$

بنابراین:

$$i \geq j \implies S_{ij} = \emptyset$$

- اگر $a_k \in S_{ij}$ وجود داشته باشد.

$$f_i \leq s_k < f_k \leq s_j < f_j \Rightarrow f_i < f_j$$

- امّا

$$i \geq j \Rightarrow f_i \geq f_j \rightarrow \text{تناقض}$$

بنابراین تنها به دنبال S_{ij} هایی هستیم که: $0 \leq i < j \leq n+1$

چون مابقی $S_{ij} = \emptyset$

فرض کنید که یک راه حل برای S_{ij} شامل a_k باشد، دو subproblem دارد:

- S_{ik} شروع بعد از پایان a_i و پایان قبل از شروع a_k
- S_{kj} شروع بعد از پایان a_k و پایان قبل از شروع a_j

راه حل برای S_{ij} = (solution to S_{ik}) $\cup \{a_k\} \cup$ (solution to S_{kj})

تا زمانی که a_k نه در subproblem باشد و نه subproblem ها جدا از هم باشند.

$$|\text{Solution to } S| = |\text{Solution to } S_{ik}| + 1 + |\text{Solution to } S_{kj}|$$

اگر یک جواب بهینه برای S_{ij} شامل a_k باشد باید جواب‌های S_{ik} و S_{kj} نیز بهینه باشند. (cut & paste argument) که در این جواب بهینه به کار برده می‌شود نیز بهینه باشد.

Let A_{ij} = optimal solution to S_{ij}

So $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

فرض می‌کنیم:

- S_{ij} is nonempty
- we know a_k.

Recursive Solution to activity selection

$c[i,j]$ = size of maximum-size subset of mutually compatible activities in S_{ij}.

- $i \geq j \Rightarrow S_{ij} = \emptyset \rightarrow c[i,j] = 0$

اگر $S_{ij} \neq \emptyset$، فرض می‌کنیم که می‌دانیم a_k در زیرمجموعه است، پس:

$$c[i,j] = c[i,k] + 1 + c[k,j]$$

و البته نمی‌دانیم که کدام k مورد استفاده قرار می‌گیرد، بنا براین

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

چرا این range برای k در نظر گرفته شده است؟

چون

$$S_{ij} = \{ a_k \in S : f_i \leq s_k < f_k \leq s_j \}$$

در نتیجه a_k نمی‌تواند a_i یا a_j باشد و همچنین a_k باید در S_{ij} باشد

$$i < k < j$$

از این جا به بعد می‌توانیم مسئله را با استفاده Dynamic programming حل کنیم.

قضیه:

Let $S_{ij} \neq \emptyset$ and let a_m be the activity in S_{ij} with the earliest finish time: $f_m = \min \{ f_k : a_k \in S_{ij} \}$ then:

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.
2. $S_{im} = \emptyset$, so that choosing a_m leaves S_{mj} are the only nonempty subproblem.

اثبات :

۲- فرض کنید که a_k متعلق به S_{im} وجود دارد. پس

$$f_i \leq s_k < f_k \leq s_m < f_m \rightarrow f_k < f_m$$

بنابراین: $a_k \in S_{ij}$ ، a_k زودتری از f_m دارد که متناقض $\underset{time}{finish}$

است با انتخاب a_m .

بنابراین، $a_k \in S_{im}$ وجود ندارد که داشته باشیم.

$$a_k \in S_{im} \Rightarrow S_{im} = \phi$$

۱- فرض کنید که A_{ij} یک زیرمجموعه maximum-size از فعالیت‌ها (activity)های دو به دو سازگار compatible باشد که در S_{ij} است.

order ، activity ها در A_{ij} به صورت یکنوا و اتراشی براساس finish time است.

فرض کنید a_k اولین activity در A_{ij} باشد.

اگر $a_k = a_m$ ، کار تمام است چون a_m مورد استفاده در زیرمجموعه maximum-size است.

درغیراینصورت ، A'_{ij} را می سازیم:

$$A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\} \quad (\text{replace } a_k \text{ by } a_m)$$

ادامه اثبات قضیه:

ادّعا (claim): activities در A'_{ij} همگی disjoint هستند.

اثبات (proof): activities در A_{ij} همگی disjoint هستند، a_k اولین activity در A_{ij} است که پایان می‌یابد.

(بنابراین a_m با هیچ چیز دیگری در A'_{ij}، overlap ندارد.) $f_m \leq f_k$

(پایان اثبات ادعا)

تا زمانی که $|A'_{ij}| = |A_{ij}|$، A_{ij} یک maximum-size subset است، پس A'_{ij} هم چنین است.

(پایان اثبات قضیه)

این عالی است:

	before theorem	after
# of subproblems in optimal solution	2	1
# of choices to consider	$j-i-1$	1

اکنون می‌توانیم به صورت top down حل کنیم:

برای حل یک مسئله S_{ij}.

- $a_m \in S_{ij}$ را به گونه‌ای انتخاب کن که تردیدکمترین زمان پایان را داشته باشد:
(the greedy choice)
- سپس S_{mj} را حل می‌کنیم.

زیرمسائل چه هستند:

↓

- original problem is $S_{0,n+1}$.
- Suppose our first choice is a_{m_1}.
- Then next subproblem is $S_{m_1,n+1}$.
- Suppose next choice is a_{m_2}.
- Next subproblem is $S_{m_2,n+1}$.
- And so on.

هر subproblem به صورت $S_{m_i,n+1}$ است به عبارت دیگر آخرین activity هایی که پایان می‌پذیرند.

و زیرمسئله انتخاب شده زمانهای پایان (finish times) هایی دارند که افزایش می‌یابد.

بنابراین، می‌توانیم هر activity را فقط یکبار در نظر بگیریم براساس ترتیب زمان پایان به صورت یکنواخت افزایشی.

Easy recursive algorithm:

فرض می‌کنیم که activity ها هم اکنون به صورت یکنواخت افزایشی براساس زمان پایان sorted باشند. (اگر unsorted باشند، آنها را در زمان $O(n \lg n)$ مرتب می‌کنیم.

و یک optimal solution برای $S_{i,n+1}$ را return می‌کند.

REC-ACTIVITY-SELECTOR (s, f, i, n)

```
m ← i+1
while m ≤ n and s_m < f_i        ▷ find first activity in S_{i,n+1}
    do m ← m+1
if m ≤ n
    then return {a_m} ∪ REC-ACTIVITY-SELECTOR (s, f, m, n)
    else return ∅
```

Initial call : REC-ACTIVITY-SELECTOR (s, f, 0, n)

ایده : حلقه while دنبال a_{i+1} ، a_{i+2} ، ... ، a_n را تا زمانی که activity a_m را که سازگار با a_i است بیابد، بررسی می‌کند (نیاز هست که $s_m \geq f_i$)

- اگر حلقه با یافتن a_m متوقف شود ($m \leq n$) آن‌گاه به صورت بازگشتی $S_{m,n+1}$ را حل می‌کند و جواب را به همراه a_m return می‌کند.
- اگر حلقه نتواند a_m که سازگار است را بیابد و ($m > n$) ، در این صورت فقط یک empty set را return می‌کند.

در مثال پیش گفته شده باید مجموعه $\{a_1, a_4, a_8, a_{11}\}$ را بدهد.

زمان اجرا : $\Theta(n)$ ← هر activity دقیقا یکبار بررسی می‌شود.

می‌توان آن را iterative کرد ، در حال حاضر recursive است.

GREEDY-ACTIVITY-SELECTOR (s, f, n)

```
A ← {a_1}
i ← 1
for m ← 2 to n
    do if s_m ≥ f_i
        then A ← A ∪ {a_m}
             i ← m        ▷ a_i is most recent addition to A
return A
```

مثال پیش‌گفته شده با جواب $\{a_1, a_4, a_8, a_{11}\}$

زمان اجرا: $\Theta(n)$

Greedy strategy

- انتخاب به نظر بهترین است در همان لحظه‌ای که انجام می‌شود.

- برای activity selection چه می‌کنیم؟

۱- تشخیص optimal substructure

۲- تهیه یک recursive solution

۳- اثبات آنکه در هر مرحله (stage) از recursion یکی از انتخاب‌های بهینه، greedy choice است. بنابراین همیشه safe است که یک greedy choice را انتخاب کنیم.

۴- نشان می‌دهیم که همه زیرمسائل به جز یکی که در نتیجه greedy choice است empty است.

۵- ایجاد یک recursive greedy algorithm

۶- تبدیل آن به یک الگوریتم iterative

در ابتدا، شبیه dynamic programming به نظر می‌رسد.

توسط substructure با دو دیدگاه انجام می‌شود.

- ایجاد و ساختن greedy choice
- باقی گذاشتن تنها یک subproblem

در activity selection، نشان دادیم که greedy choice که بر روی زیر S انجام می‌شود

همیشه ثابت است و fixed است برابر $n+1$.

می‌توانستیم که با یک الگوریتم greedy به صورت زیر شروع کنیم.

- Define $S_i = \{a_k \in S : f_i \leq s_k\}$

- سپس نشان می‌دهیم که greedy choice (شامل اولین a_m تا پایان در S_i)

ترکیب شده است با optimal solution برای S_m ← جواب بهینه برای S_i است.

مراحل کار:

۱- بدست آوردن قالب برای مسئله بهینه سازی به گونه‌ای که بتوانیم یک choice انجام دهیم و فقط یک subproblem برای حل باقی بماند.

۲- اثبات اینکه همواره با انتخاب greedy یک جواب بهینه وجود دارد به بیان دیگر greedy choice همیشه safe است.

۳- نشان می دهیم که greedy choice , optimal solution برای

یک subproblem ← optimal solution برای مسئله اصلی.

هیچ راه کلی وجود ندارد که بگوید یک greedy algorithm بهینه است و دو نکته کلیدی کمک کننده است:

1. greedy-choice property
2. optimal substructure