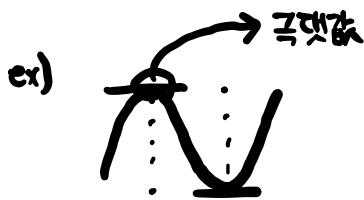


# Greedy Algorithm

→ 최적화 문제에서 최선의 선택을 결정하기 위한 방법으로 DP를 사용하는 것은 대체로 가능할 수 있다.  
이렇기 때문에, 더 단순하면서 효율적인 알고리즘을 선택해야 하는데 Greedy algorithm이 이런 관점에서  
항상 그 순간에서 최선으로 보이는 선택을 하는 알고리즘으로,  
*Globally optimal solution* (전체적으로 최선의 선택)을 위한 *locally optimal solution* (당장 최선의 선택)  
을 만들어 내는 알고리즘이다.

- local optimum (극댓값)



이 극댓값이 최댓값이 되는 경우에 Greedy Algorithm을 사용할 수 있다.  
global optimum

- global optimum (최댓값)

전체 그래프 상에서 정해진 구간내의 최댓값을 의미한다.

그리디 알고리즘이 항상 최선의 결과를 가져다 주진 않지만, 많은 문제에서 최선의 솔루션이 될 수도 있다.

\* 그리디 알고리즘 문제를 보기전에.. DP를 사용하는 문제를 보자.

## 1. Rod-Cutting (톱나무 자르기)

주어진  $n$ 인치 길이의 나무막대와  $P_i$  ( $i=1, 2, \dots, n$ ) 개각의 테이블들로 최대 수익  $R_n$ 을 결정해야 한다.  
이때, 수익은 나무막대를 자르고 토크들을 팔아도 얻을 수 있다.

length i	1	2	3	4	5	6	7	8	9	10
price $P_i$	1	5	8	9	10	17	17	20	24	30

⟨Greedy Approach⟩

• 1 길이당, 최고 가격의 순서대로 취한다.

→ 길이당 가격으로 테이블을 재 설정하면, 아래와 같은 표로 나타난다.

length i	1	2	3	4	5	6	7	8	9	10
price P <sub>i</sub>	1	2.5	2.7	2.3	2	2.8	2.4	2.5	2.7	3

① 4인치짜리 rod의 최대수익은 2인치 2개를 사는 '5.4'가 되지만...

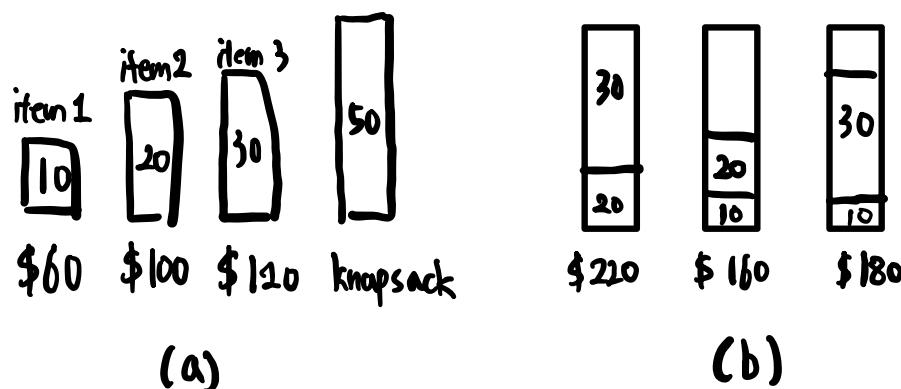
Greedy approach에 의해 현재 주어진 정보에서 최적의 결과를 내는 것(이므로  
3인치 (1,2,3에서 3이 제일 비쌈) + 1인치 (남은 인치 중 제일 비싼 것) = '3,7')이 된다.  
하지만 결과적으로 global optimum을 (최댓값)을 얻는 것이 아니라..  
local optimum (극댓값)을 얻는 문제가 된다.

② 물론 8인치짜리 rod에서는 greedy approach에서 global optimum을  
얻을 수 있지만.. local optimum이 나올 수 있는 경우의 수가 존재하므로  
greedy algorithm을 사용할 수 없다.

∴ Dynamic Programming을 통해 솔루션을 놓쳐야 한다!

## 2. knapsack problem (배낭문제)

⇒ 도둑이 배낭을 끼고 보석가게에 들어왔다. 훔친 보석의 총 무게가 용량 W를 넘어가면 짖어진다.



## <(Greedy Approach)>

- 단위 무게당 가격이 높으면 가방에 집어 넣는다 (최적 선택 - 현 상황에서)

위 그림에서 (a) item 1, 2, 3을 봄 때 dp를 적용하면 (b)의 첫번째 경우 = \$ 2200이 되는 global optimum을 얻을 수 있다.

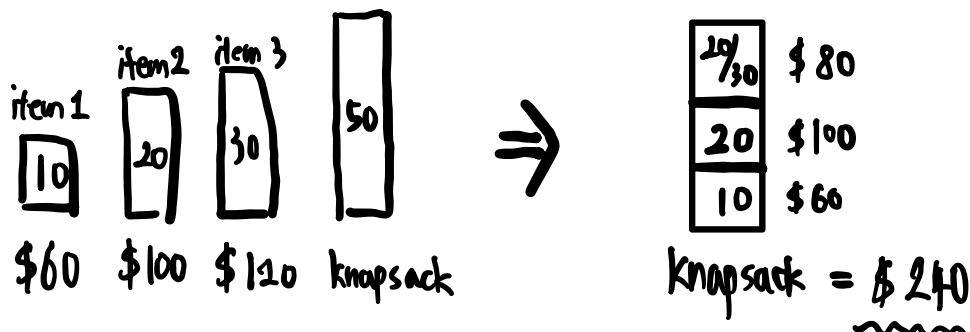
Greedy Algorithm을 사용하면  
(item 1: 1g당 \$6 이므로 (b)의 두번째 그림 = \$1600)  
item 2: 1g당 \$5  
item 3: 1g당 \$4

될 것이다. 결국 local optimum이 구해지는 결과이므로 풀 수 없게 된다.

그렇다면, 어떤 문제가 Greedy Algorithm으로 풀릴까?

### 3. Fractional knapsack problem

⇒ 해당 문제는 2번쩨의 기본적인 설정은 같다. 하지만! 도둑이 각 아이템을 가져갈지 말지 결정하는 예민법적인 선택만 하는 것이 아닌, 아이템을 조개어서 가져갈 수 있도록 한다. 아이템을 조개어 봄 수 있기 때문에, 단위 무게당 가격이 높은 아이템을 넣을 수 있다. 위 경우, global optimum을 얻을 수 있어서 greedy algorithm을 사용할 수 있다! 또한, DP와 비교해서 수행 속도가 빠르다!



불충분한 시간에서 빠르게 험의적인 편의를 못 할 때 빠르게 사용하는 간편한



\* Heuristic한 전략은 항상 최적의 solution을 제공하지 못하지만, 행동 선택 문제 (activity-selection problem) 을 해결할 때 성공할 수 있다. (greedy-choice property 를 생각하자! )

optimal substructure

## ① Greedy-choice property

⇒ 우리가 선택을 할 때, sub problem을 고려하지 않고 초기 문제에서 가장 최선으로 보이는 선택 (locally optimal) 을 버리는 것이다.

∴ 이전의 선택이 그 이후의 선택에 영향을 주지 않는다.

## ② Optimal substructure

⇒ 전체 문제에 대한 optimal solution이 하위 문제들에 대한 optimal solution을 포함하는 경우, 전체问题是 optimal substructure를 갖게 된다. (greedy 뿐 아니라 DP에서도..)

위 두 조건이 성립해야.. greedy algorithm 적용이 가능하다.

### <example>

대표적인 Greedy 문제인 Activity-selection에 대해 살펴보자.

- n개의 회의 시간이 존재
- i 번째 회의는 각각 시작시간과 종료시간이 존재
- 회의실은 하나를 사용해야 되므로 회의를 최적화하여 최대 회의 개수를 만들면 global optimum이 된다.

이 문제를 상호배타적인 험김성 (mutually compatible) 한 activity의 부분집합을 알고자 한다.

↳ 서로 겹치지 않고 진행하는 회의 (위 문제의 경우)

이 회의가 끝나는 시간의 순서가 오름차순으로 정렬되었다고 가정하자!

또한, 끝나자마자 (안 중복하는 시간 내에서) 바로 회의가 시작할 수 있다고 가정.

$s_i$ : 시작시간  
 $f_i$ : 끝나는시간  $\Rightarrow [s_i, f_i)$

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

$\Rightarrow$  제일 일찍 회의가 끝나는 1번 회의 다음, 바로  $f_i$ 보다 큰  $s_i$ 가 나올 때 까지 찾는다. 최종적으로  $1 \rightarrow 4 \rightarrow 8 \rightarrow 11$  이 되겠다.

위를 식으로 정리하면,  $S_k = \{a_i \in S : s_i \geq f_{i-1}\}$

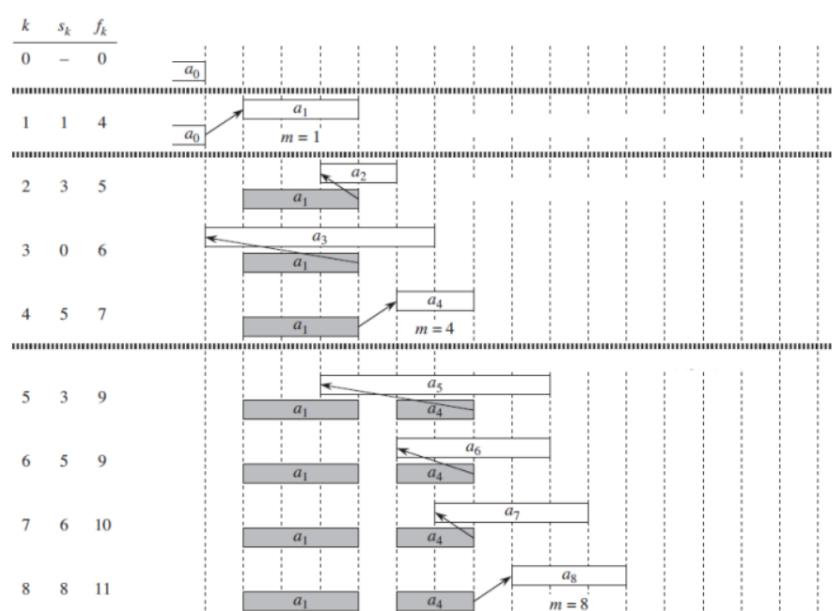
$a$ 는 회의 (activity),  $S$ 는  $a_k$ 가 끝난 후의 시작되는 활동의 집합이라 하자.

( $S_0 = 0$ 번 activity가 끝난 후에 시작하는 모든 activity의 합집합)

$$= a_1 + \underbrace{S_1}_{\downarrow} \quad (S_0 \text{ 중에서 제일 빨리 끝나는 1번 회의 선택})$$

1번 회의가 끝난 후 모두 회의의 합집합 (substructure)

$$\underbrace{a_4 + S_4}_{\downarrow}$$



Non-empty인 하위 문제  $S_k$ 를 고려하고,  $a_m$ 가 가장 빠르게 마치는 시간을 갖도록 하는  $S_k$  안의 activity가 되도록 한다.  $a_m$ 은  $S_k$ 의 상호 배타적으로 양립 가능한 활동들로 이루어진 최대 크기의 부분집합 안에 포함되도록 한다.



Greedy Algorithm은 전형적으로 아래한 Top-down을 기준다.

하위 문제를 선택한 후에, 하위 문제를 해결하는 방식이다.

(Greedy를 쓸지 고민된다면... 그냥 DP를 쓰자...)