Greedy Algorithm

Kuan-Yu Chen (陳冠宇)

2019/06/12 @ TR-310-1, NTUST

Review

- For many optimization problems, using dynamic programming to determine the best choices is overkill
  - More efficient algorithms will do

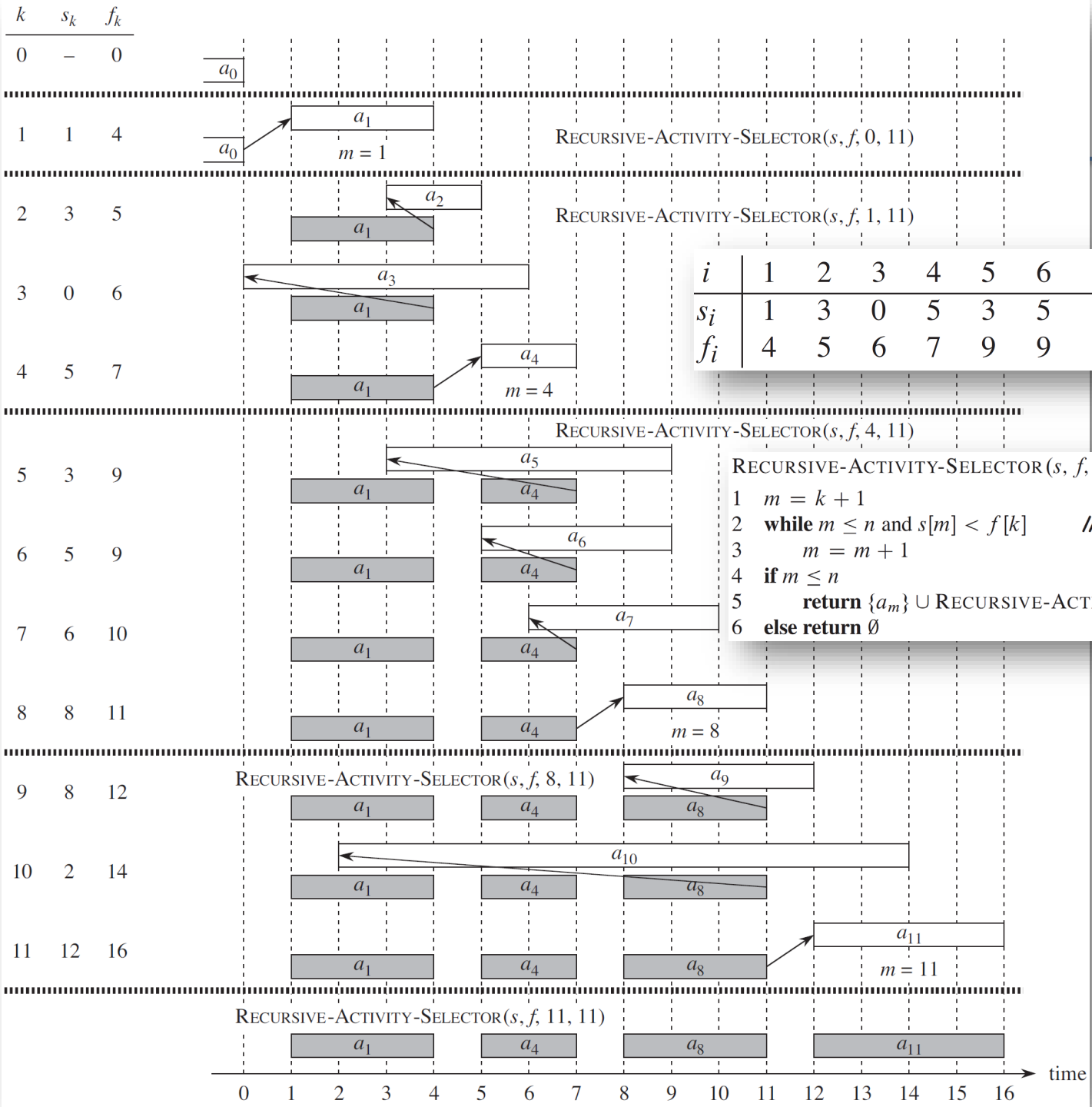
- A ***greedy algorithm*** always makes the choice that looks best at the moment
  - It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution
  - **Greedy algorithms do not always yield optimal solutions**, but for many problems they do

Recursive Greedy Algorithm

- The recursive greedy algorithm
  - It takes the start and finish times of the activities, i.e., s and f
  - The index k that defines the subproblem S_k it is to solve
    - $S_k = \{a_i \in S : s_i \geq f_k\}$
  - The size n of the original problem

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

  - In order to start, we add the fictitious activity a_0 with $f_0 = 0$, so that subproblem S_0 is the entire set of activities S
    - The initial call, which solves the entire problem, is RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)



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

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

Iterative Greedy Algorithm

- We can convert our recursive procedure to an iterative one
  - The procedure RECURSIVE-ACTIVITY-SELECTOR is almost "tail recursive"
  - In fact, some compilers for certain programming languages perform this task automatically

- The procedure GREEDY-ACTIVITY-SELECTOR assumes that the input activities are ordered by monotonically increasing finish time

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

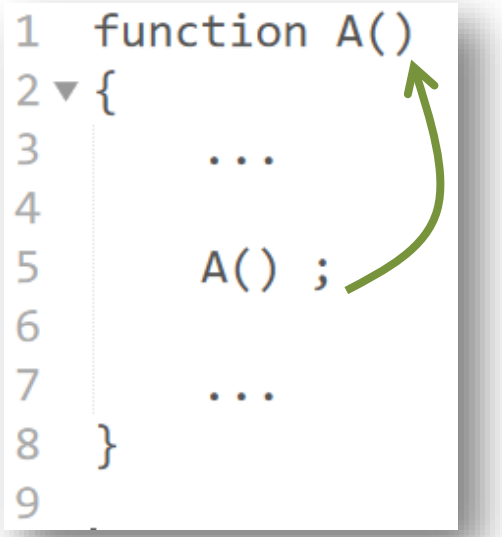
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

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

```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```

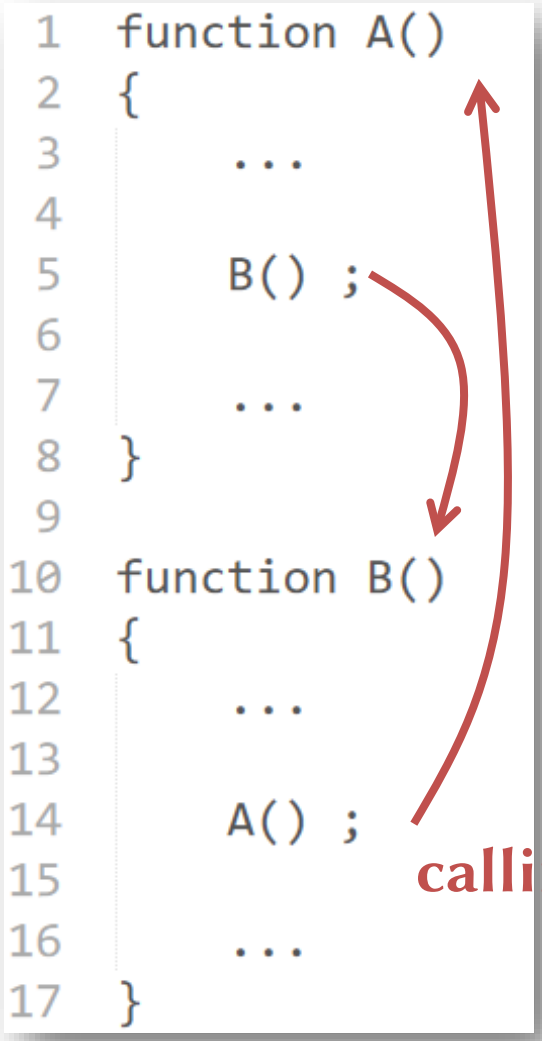
Appendix

Direct Recursion



```
1  function A()
2  {
3      ...
4
5      A() ;
6
7      ...
8  }
9
```

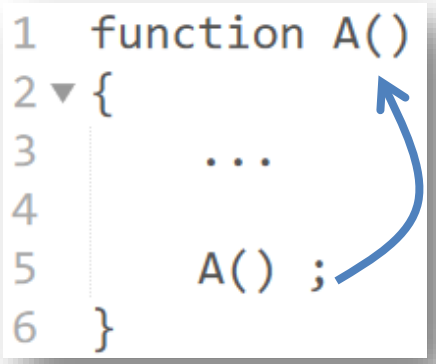
Indirect Recursion



```
1  function A()
2  {
3      ...
4
5      B() ;
6
7      ...
8  }
9
10 function B()
11 {
12     ...
13
14     A() ;
15
16     ...
17 }
```

calling cycle

Tail Recursion



```
1  function A()
2  {
3      ...
4
5      A() ;
6  }
```

Property

- At each decision point, the greedy algorithm makes choice that seems best at the moment
  - This heuristic strategy does not always produce an optimal solution
  - But as we saw in the activity-selection problem, sometimes it does!

- How can we tell whether a greedy algorithm will solve a particular optimization problem?
  - **Greedy-choice property** and **optimal substructure** are the two key ingredients
  - If we can demonstrate that the problem has these properties, then we are well on the way to developing a greedy algorithm for it

Two Key Ingredients

- Greedy-choice property
  - We can assemble a globally optimal solution by making locally optimal (greedy) choices
    - In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems

- Optimal substructure
  - A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems

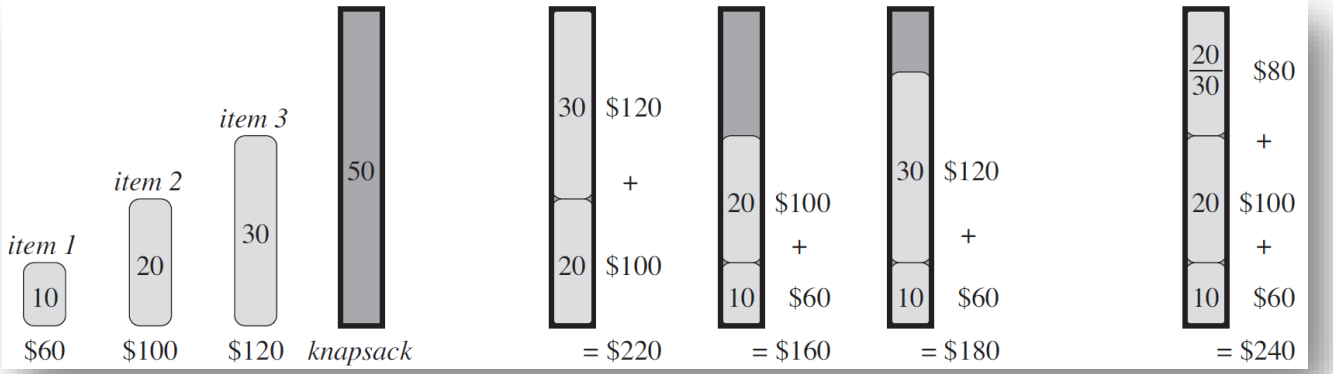
DP & Greedy Algorithm.

- In dynamic programming, we make a choice at each step, but the choice usually depends on the solutions to subproblems
  - Typically in a bottom-up manner

- In a greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblem that remains
  - The choice may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems

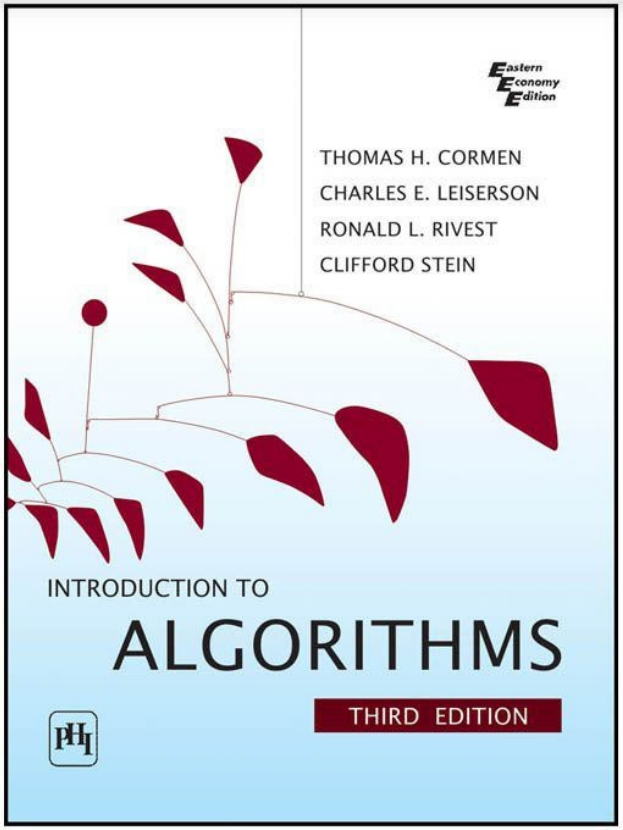
DP & Greedy Algorithm..

- Let us investigate two variants of a classical optimization problem
  - The **0-1 knapsack problem**
    - A thief robbing a store finds n items. The i^{th} item is worth i dollars and weighs w_i pounds, where i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W. Which items should he take?
  - The **fractional knapsack problem**
    - The setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item



In This Semester.

In This Semester..



http://nlp.csie.ntust.edu.tw/sws2019/

http://nlp.csie.ntust.edu.tw/sws2019/

Questions?

kychen@mail.ntust.edu.tw