Greedy Algorithms

Knapsack problem

There are two versions of the problem:

1. "0-1 knapsack problem"
   - Items are indivisible; you either take an item or not. Some special instances can be solved with *dynamic programming*
2. "Fractional knapsack problem"
   - Items are divisible: you can take any fraction of an item

The Knapsack Problem

The classic Knapsack problem is:

A thief breaks into a store and wants to fill his knapsack of capacity K with goods of as much value as possible.

Decision version: Does there exist a collection of items that fits into his knapsack and whose total value is >= W?

0-1 Knapsack problem

1. 0-1 Knapsack Problem:

A thief robbing a store finds n items.

i^{th} item: worth v_i dollars

w_i pounds

W, w_i, v_i are integers.

He can carry at most W pounds.

0-1 Knapsack problem

- Given a knapsack with maximum capacity *W*, and a set *S* consisting of *n* items
- Each item *i* has some weight w_i and benefit value b_i (all w_i and *W* are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem

- Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \le W$$

- The problem is called a "*0-1*" problem, because each item must be entirely accepted or rejected.

0-1 Knapsack problem: brute-force approach

A straightforward algorithm:

- Since there are n items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with maximum value and with total weight less or equal to W
- Running time will be $O(2^n)$

Knapsack Problems

2. Fractional Knapsack Problem:

A thief robbing a store finds n items.

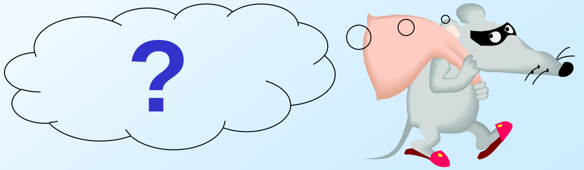
i^{th} item: worth v_i dollars

w_i pounds

W, w_i, v_i are integers.

He can carry at most W pounds.

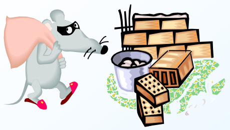
He can take fractions of items.



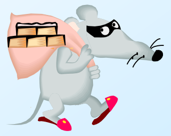
Knapsack Problems

Dynamic Programming Solution

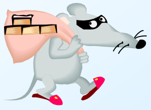
Both problems exhibit the optimal-substructure property:



Consider the most valuable load that weighs at most W pounds.



If j^{th} item is removed from his load,



the remaining load must be the most valuable load weighting at most $W-w_j$ that he can take from the n-1 original items excluding j.

=> Can be solved by dynamic programming

Knapsack Problems

Dynamic Programming Solution

Example: 0-1 Knapsack Problem

Suppose there are n=100 ingots:
30 Gold bars: each $10000, 8 pounds **(most expensive)**
20 Silver bars : each $2000, 3 pound per piece
50 Copper bars : each $500, 5 pound per piece

Then, the most valuable load for to fill W pounds
= The most valuable way among the followings:

(1) take 1 **gold** bar + the most valuable way to fill **W-8** pounds from **29** gold bars , 20 silver bars and 50 copper bars

(2) take 1 **silver** bar + the most valuable way to fill **W-3** pounds from 30 gold bars , **19** silver bars and 50 copper bars

(3) take 1 **copper** ingot + the most valuable way to fill **W-5** pounds from 30 gold bars , 20 silver bars and **49** copper bars

Knapsack Problems

Dynamic Programming Solution

Example: Fractional Knapsack Problem

Suppose there are totally n = 100 pounds of metal dust:
30 pounds Gold dust: each pound $10000 **(most expensive)**
20 pounds Silver dust: each pound $2000
50 pounds Copper dust: each pound $500

Then, the most valuable way to fill a capacity of W pounds
= The most valuable way among the followings:

(1) take 1 pound of **gold** + the most valuable way to fill **W-1** pounds from **29** pounds of gold, 20 pounds of silver, 50 pounds of copper

(2) take 1 pound of **silver** + the most valuable way to fill **W-1** pounds from 30 pounds of gold, **19** pounds of silver, 50 pounds of copper

(3) take 1 pound **copper** + the most valuable way to fill pounds from pounds of gold, pounds of silver, pounds of copper

Knapsack Problems

By Greedy Strategy

Both problems are similar. But Fractional Knapsack Problem can be solved in a greedy strategy.

Step 1. Compute the value per pound for each item
Eg. gold dust: $10000 per pound **(most expensive)**
Silver dust: $2000 per pound
Copper dust: $500 per pound

Step 2. Take as much as possible of the most expensive (ie. Gold dust)

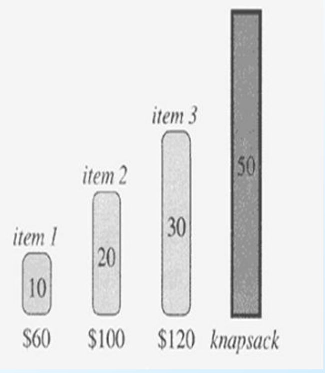
Step 3. If the supply of that item is exhausted (ie. no more gold) and he can still carry more, he takes as much as possible of the item that is next most expensive and so forth until he can't carry any more.

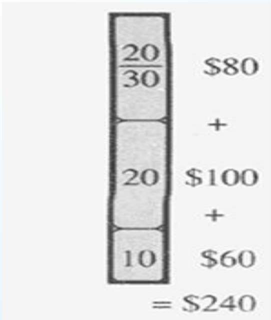
Greedy Algorithm for Fractional Knapsack problem

- Fractional knapsack can be solvable by the greedy strategy
  - Compute the value per pound v_i/w_i for each item
  - Obeying a greedy strategy, take as much as possible of the item with the greatest value per pound.
  - If the supply of that item is exhausted and there is still more room, take as much as possible of the item with the next value per pound, and so forth until there is no more room
  - **O(n lg n) (we need to sort the items by value per pound)**

O-1 knapsack is harder

- 0-1 knapsack **cannot** be solved by the greedy strategy
  - Unable to fill the knapsack to capacity, and the empty space lowers the effective value per pound of the packing
  - Dynamic Programming





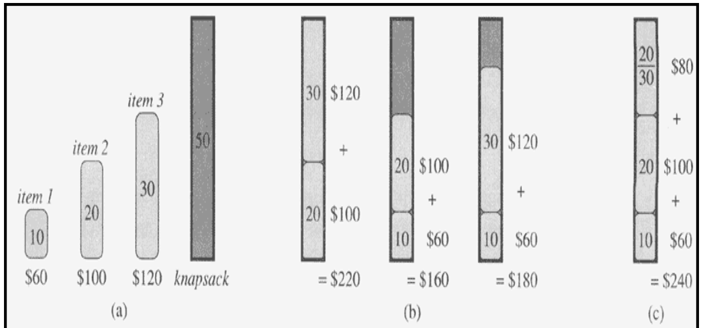


Figure 16.2 The greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

Greedy Algorithm Design

- At each step, we quickly make a choice that currently looks best.
  --A local optimal (greedy) choice.
- Greedy choice can be made first before solving further sub-problems.
- Top-down approach
- Usually faster, simpler

Greedy Algorithms

Techniques for solving optimization problems:

1. Dynamic Programming
2. Greedy Algorithms (“Greedy Strategy”)

For some optimization problems

- Greedy Strategy is simpler and more efficient.
- Dynamic Programming is “overkill”

Activity-Selection Problem

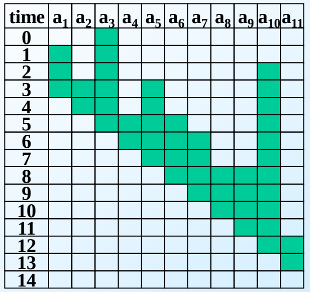
For a set of proposed activities that wish to use a lecture hall, select **a maximum-size subset of "compatible activities".**

- Set of activities: $S=\{a_1,a_2,\ldots a_n\}$
- Duration of activity a_i: [$start\_time_i$, $finish\_time_i$]
- Activities ***sorted in increasing order of finish time***:

i	**1**	2	3	4	5	6	7	8	9	10	11
$start\_time_i$	**1**	3	0	5	3	5	6	8	8	2	12
$finish\_time_i$	**4**	5	6	7	8	9	10	11	12	13	14

Activity-Selection Problem

i	1	2	3	4	5	6	7	8	9	10	11
$start\_time_i$	1	3	0	5	3	5	6	8	8	2	12
$finish\_time_i$	4	5	6	7	8	9	10	11	12	13	14



Compatible activities:
$\{a_3, a_9, a_{11}\}$,
$\{a_1,a_4,a_8,a_{11}\}$,
$\{a_2,a_4,a_9,a_{11}\}$

Activity-Selection Problem

Dynamic Programming Solution (Step 1)

Step 1. Characterize the structure of an optimal solution.

S:

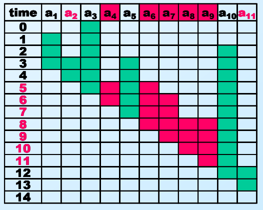
i	1	2	3	4	5	6	7	8	9	10	11(=n)
$start\_time_i$	1	3	0	5	3	5	6	8	8	2	12
$finish\_time_i$	4	5	6	7	8	9	10	11	12	13	14

Let $S_{i,j}$ be the set of activities that

start after a_i finishes and

finish before a_j starts.

eg. $S_{2,11}=$



Definition:

$S_{ij}=\{a_k \in S: finish\_time_i \leq start\_time_k < finish\_time_k \leq start\_time_j\}$

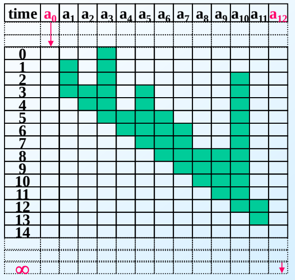
Activity-Selection Problem

Dynamic Programming Solution (Step 1)

Add fictitious activities: a_0 and a_{n+1}:

S:

i	0	1	2	3	4	5	6	7	8	9	10	11(=n)	12
$start\_time_i$		1	3	0	5	3	5	6	8	8	2	12	∞
$finish\_time_i$	0	4	5	6	7	8	9	10	11	12	13	14	



ie. $S_{0,n+1}$
$=\{a_1,a_2,a_3,a_4,a_5,a_6,a_7,a_8,a_9,a_{10},a_{11}\}$
$= S$

Note: If $i>=j$ then $S_{i,j}=\emptyset$

Activity-Selection Problem

Dynamic Programming Solution (Step 1)

The problem:

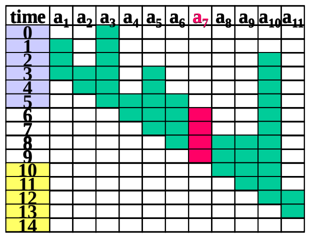
For a set of proposed activities that wish to use a lecture hall, select a maximum-size subset of "compatible activities = Select a maximum-size subset of compatible activities from $S_{0,n+1}$.

Substructure:

Suppose a solution to $S_{i,j}$ includes activity a_k, then,2 subproblems are generated: $S_{i,k}$, $S_{k,j}$

The maximum-size subset $A_{i,j}$ of compatible activities is:

$A_{i,j}=A_{i,k} \cup \{a_k\} \cup A_{k,j}$



Suppose a solution to $S_{0,n+1}$ contains a_7, then, 2 subproblems are generated: $S_{0,7}$ and $S_{7,n+1}$

Activity-Selection Problem

Dynamic Programming Solution (Step 2)

Step 2. Recursively define an optimal solution

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

If i>=j, then $S_{i,j}=\emptyset$, ie. c[i,j]=0.

$$c(i,j) = \begin{cases} 0 & \text{if } S_{i,j}=\emptyset \\ \text{Max}_{i<k<j}\ \{c[i,k] + c[k,j] + 1\} & \text{if } S_{i,j}\neq\emptyset \end{cases}$$

Step 3. Compute the value of an optimal solution in a bottom-up fashion

Step 4. Construct an optimal solution from computed information.

Activity-Selection Problem

Greedy Strategy Solution

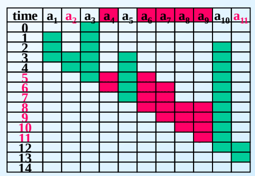
$$c(i,j) = \begin{cases} 0 & \text{if } S_{i,j}=\emptyset \\ \text{Max}_{i<k<j}\ \{c[i,k]+c[k,j]+1\} & \text{if } S_{i,j}\neq\emptyset \end{cases}$$

Consider any nonempty sub-problem $S_{i,j}$, and let a_m be the activity in $S_{i,j}$ with the earliest finish time.

Then,

1. a_m is used in some maximum-size subset of compatible activities of $S_{i,j}$.
2. The sub-problem $S_{i,m}$ is empty, so that choosing a_m leaves the sub-problem $S_{m,j}$ as the only one that may be non-empty.

eg. $S_{2,11}=\{a_4,a_6,a_7,a_8,a_9\}$



Among $\{a_4,a_6,a_7,a_8,a_9\}$, a_4 will finish earliest

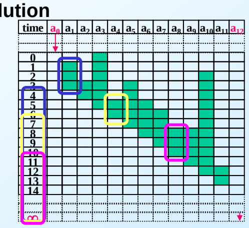
1. A_4 is used in the solution
2. After choosing A_4, there are 2 subproblems: $S_{2,4}$ and $S_{4,11}$. But $S_{2,4}$ is empty. Only $S_{4,11}$ remains as a subproblem.

Activity-Selection Problem

Greedy Strategy Solution

Hence, to solve the $S_{i,j}$:

1. Choose the activity a_m with the earliest finish time.
2. **Solution of $S_{i,j} = \{a_m\} \cup$ Solution of subproblem $S_{m,j}$**



That is,

To solve $S_{0,12}$, we select a_1 that will finish earliest, and solve for $S_{1,12}$.

To solve $S_{1,12}$, we select a_4 that will finish earliest, and solve for $S_{4,12}$.

To solve $S_{4,12}$, we select a_8 that will finish earliest, and solve for $S_{8,12}$.

…

Greedy Choices (Locally optimal choice)

To leave as much opportunity as possible for the remaining activities to be scheduled.

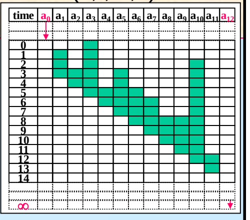
Solve the problem in a top-down fashion

- s➔ start time
- f➔finish time
- k➔ subproblem to solve
- n➔ size of the original problem

Activity-Selection Problem

Greedy Strategy Solution

```
Recursive-Activity-Selector(s,f,k,n)
m = k+1
   // Find first activity in Sk to finish
while m <=n and s[m] < f[k]
      m = m + 1
if m <=n
      return {am} U Recursive-Activity-Selector(s,f,m,n)
else return Ø
```



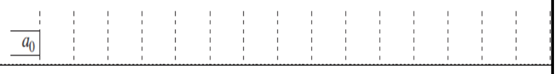
S (Set of activities)

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

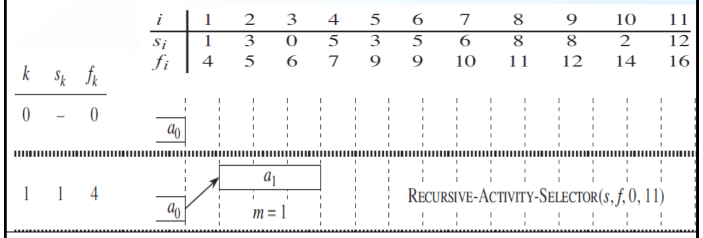
Recursive-Activity-Selector(s,f,0,11)

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

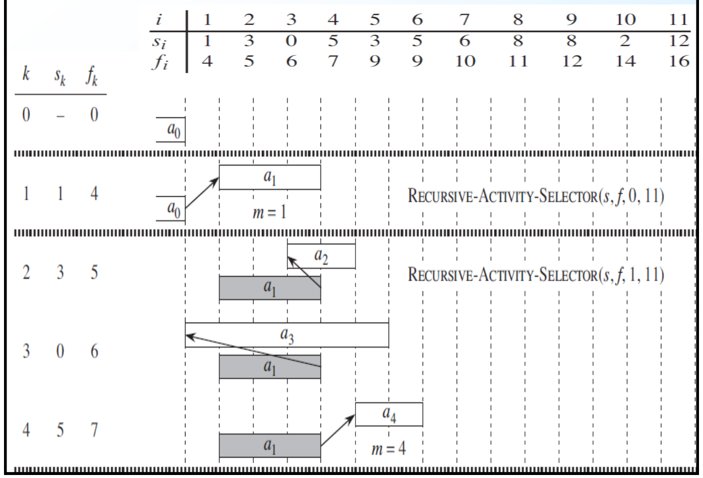
k	s_k	f_k
0	–	0

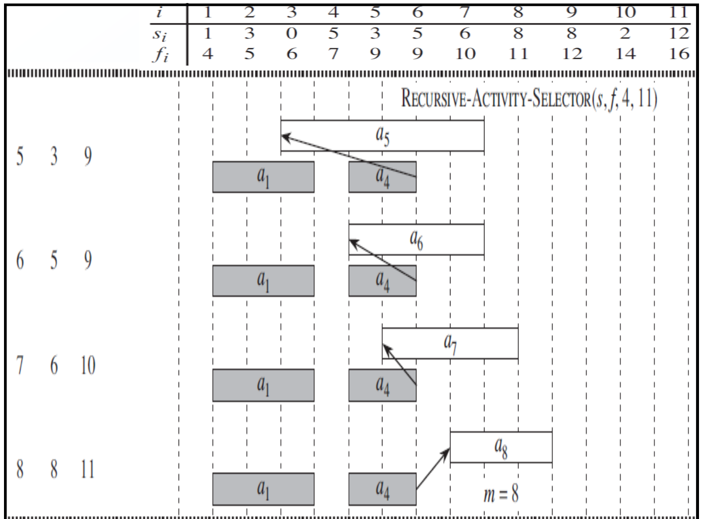


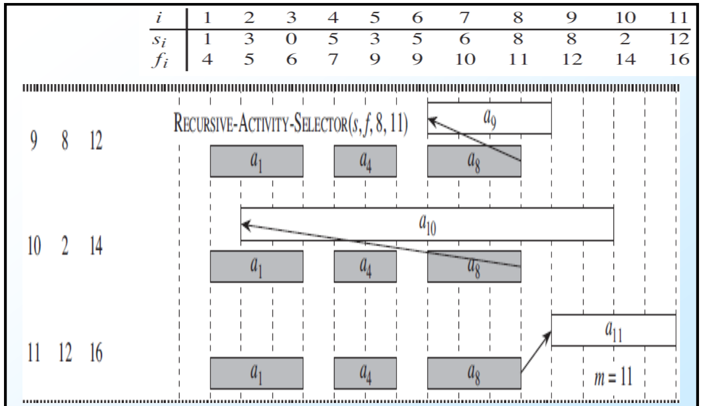
Recursive-Activity-Selector(s,f,0,11)



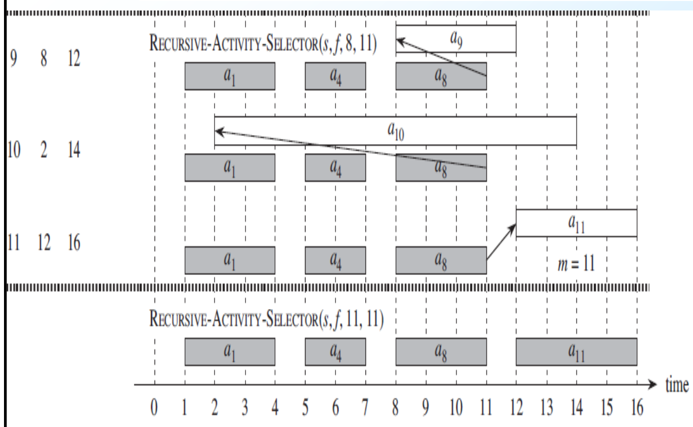
Recursive-Activity-Selector(s,f,0,11)







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

- The resulting set of selected activities is={a_1, a_4, a_8,a_{11}}

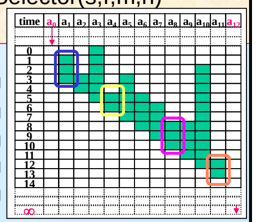
Activity-Selection Problem

Greedy Strategy Solution

```
Recursive-Activity-Selector(s,f,k,n)
    m = k+1
    // Find first activity in S_k  to finish
    while m <=n and s[m] < f[k]
        m = m + 1
    if m <=n
        return {a_m} U Recursive-Activity-Selector(s,f,m,n)
    else return Ø
```

Order of calls:

{1,4,8,11} Recursive-Activity-Selector(0,12)
{4,8,11} Recursive-Activity-Selector(1,12)
{8,11} Recursive-Activity-Selector(4,12)
{11} Recursive-Activity-Selector(8,12)
Ø Recursive-Activity-Selector(11,12)

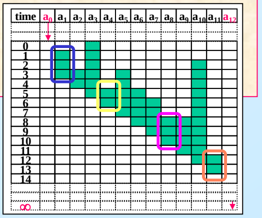


- Recursive-Activity-Selector=O(n)
- Since each activity is examined exactly once

Activity-Selection Problem

Greedy Strategy Solution

```
Iterative-Activity-Selector(s,f)
   n=s.length
   A = {a1}
   k=1
   for m = 2 to n
       if s[m]>=f[k]  //start_time_m>=finish_time_last_selected
           A = A U {a_m}
               k(last_selected) = m
   return A
```



Activity-Selection Problem

Greedy Strategy Solution

For both Recursive-Activity-Selector and Iterative-Activity-Selector,

Running times are $\Theta(n)$

Reason: each a_m are examined once.