Advanced Algorithms Analysis and Design

By

Nazir Ahmad Zafar

Lecture No 25

Greedy Algorithms

Today Covered

- Activity Selection Problem
  - Example
  - Recursive algorithm
  - Iterative Algorithm
- Fractional Knapsack Problem
  - Problem Analysis
  - Greedy Approach for Fractional Knapsack
- Coin Change Making Problem
  - Analysis
  - Greedy Algorithm

Theorem: Why This Solution is Optimal?

Statement:

Consider any nonempty subproblem S_{ij}, and let a_m be the activity in S_{ij} with the earliest finish time:

$$f_m = \min \{f_k : a_k \in S_{ij}\} \text{, then}$$

1. Activity a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.
2. The subproblem S_{im} is empty, so that choosing a_m leaves the subproblem S_{mj} as the only one that may be nonempty.

Note:

After proving these properties, it is guaranteed that the greedy solution to this problem does exist.

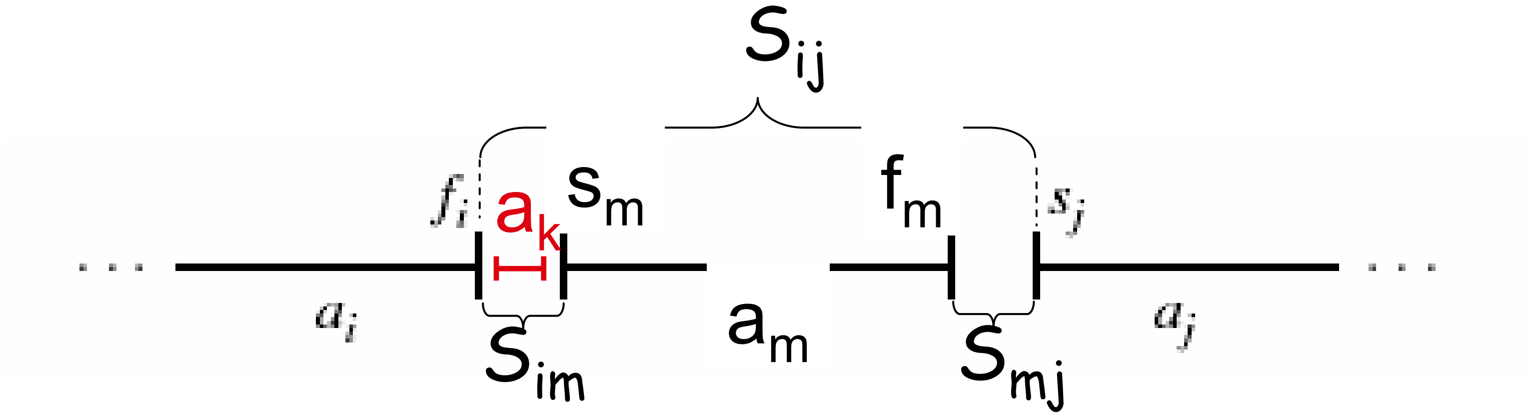
Theorem

Proof (Part B)

First we prove second part because it is bit simpler

- Suppose that S_{im} is nonempty
- It means there is some activity a_k such that:

  $f_i \le s_k < f_k \le s_m < f_m. \Rightarrow f_k < f_m.$
- Then a_k is also in S_{ij} and it has an earlier finish time than a_m, which contradicts our choice of a_m. Hence S_{im} is empty, proved



Theorem

Part A

- To prove first part, suppose that A_{ij} is a maximum-size subset of mutually compatible activities of S_{ij},
- Order A_{ij} monotonic increasing order of finish time
- Let a_k be the first activity in A_{ij}.

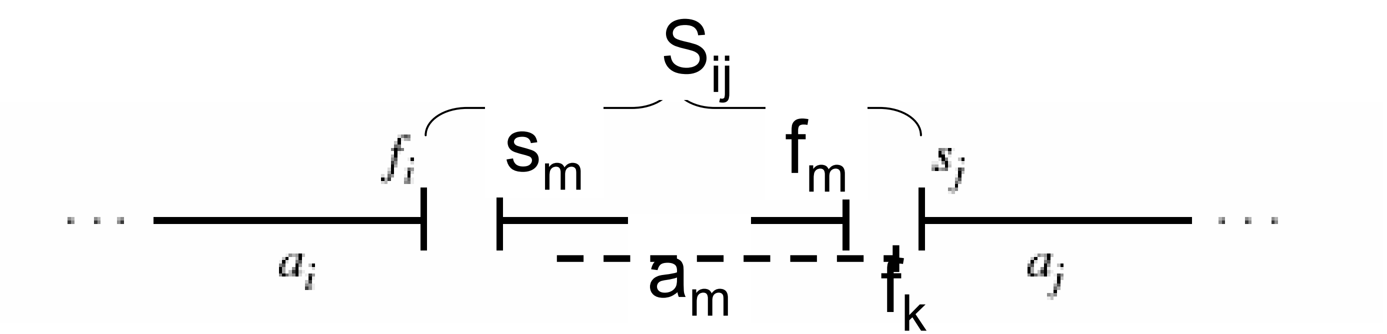
Case 1

- If $a_k = a_m$, then we are done, since we have shown that a_m is used in some maximal subset of mutually compatible activities of S_{ij}.

Theorem

Case 2

- If $a_k \neq a_m$, then we construct the subset
  $A'_{ij} = A_{ij} \setminus \{a_k\} \cup \{a_m\}$
- Since activities in A_{ij} are disjoint, so is true for A'_{ij}.
- *As* a_k is first activity in A_{ij} to finish, and $f_m \leq f_k$.
- Noting that A'_{ij} has same number of activities as A_{ij}
- We see that A'_{ij} is a maximal subset of mutually compatible activities of S_{ij} that includes a_m.
- Hence proves the theorem.



Why is this Theorem Useful?

	Dynamic programming	Using the theorem
Number of subproblems in the optimal solution	2 subproblems: S_{ik}, S_{kj}	1 subproblem: S_{mj} $S_{im} = \varnothing$
Number of choices to consider	$j - i - 1$ choices	1 choice: the activity with the earliest finish time in S_{ij}

- Making the greedy choice i.e., the activity with the earliest finish time in S_{ij}
  - Reduce the number of subproblems and choices
  - Solved each subproblem in a top-down fashion
- Only one subproblem left to solve.

A Recursive Greedy Algorithm

Recursive-Activity-Selector (s, f, i, j)

$m \leftarrow i + 1$

while $m < j$ and $s_m < f_i$ $\quad \triangleright$ Find the first activity in S_{ij}.

  do $m \leftarrow m + 1$

if $m < j$

 then

  return $\{a_m\} \cup$ Recursive-Activity-Selector (s, f, m, j)

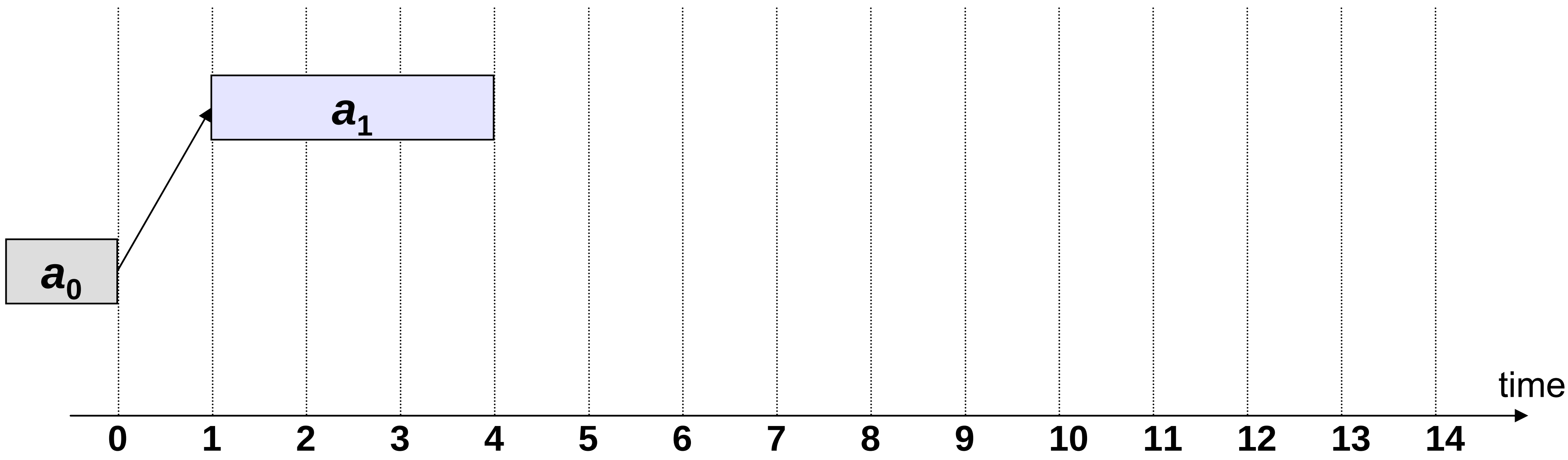
 else return $\varnothing$

Example: A Recursive Greedy Algorithm

i	0	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	0	4	5	6	7	8	9	10	11	12	13	14

For the Recursive Greedy Algorithm, the set *S* of activities is sorted in increasing order of finish time

A Recursive Greedy Algorithm



$i = 0$,

$j = n + 1 = 12$

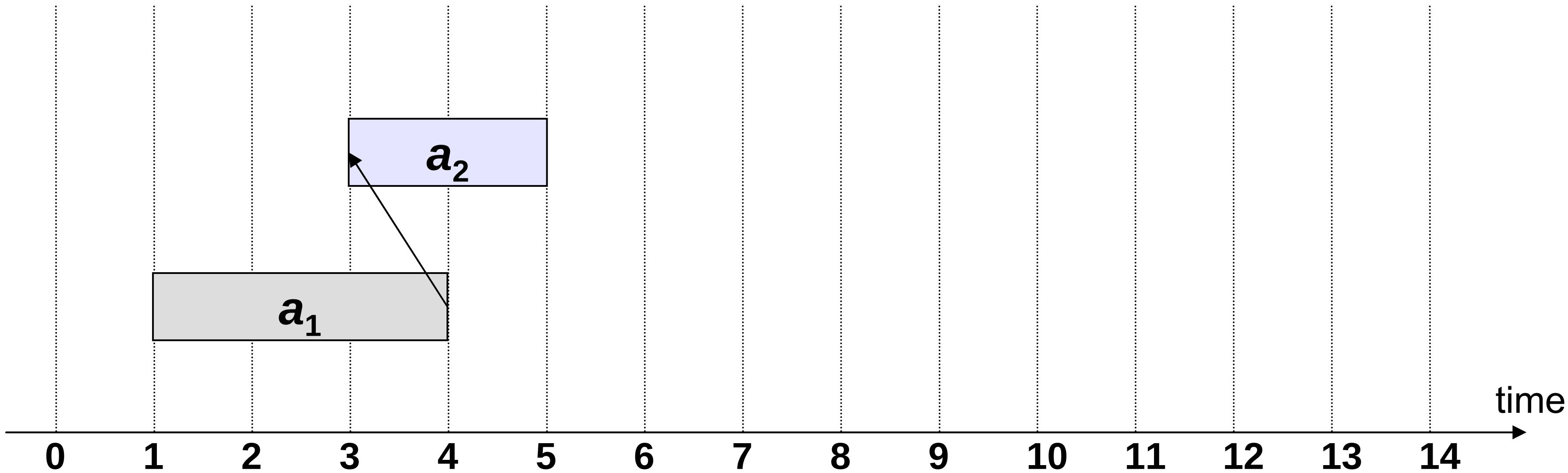
$m \leftarrow i + 1 \leftarrow 0 + 1 = 1$

$m < j$ (1 < 12) and $s_1 < f_0$ (But 1>0)

if $m < j$ (1 < 12)

return $\{a_1\} \cup$ Recursive-Activity-Selector (s, f, 1,12)

A Recursive Greedy Algorithm



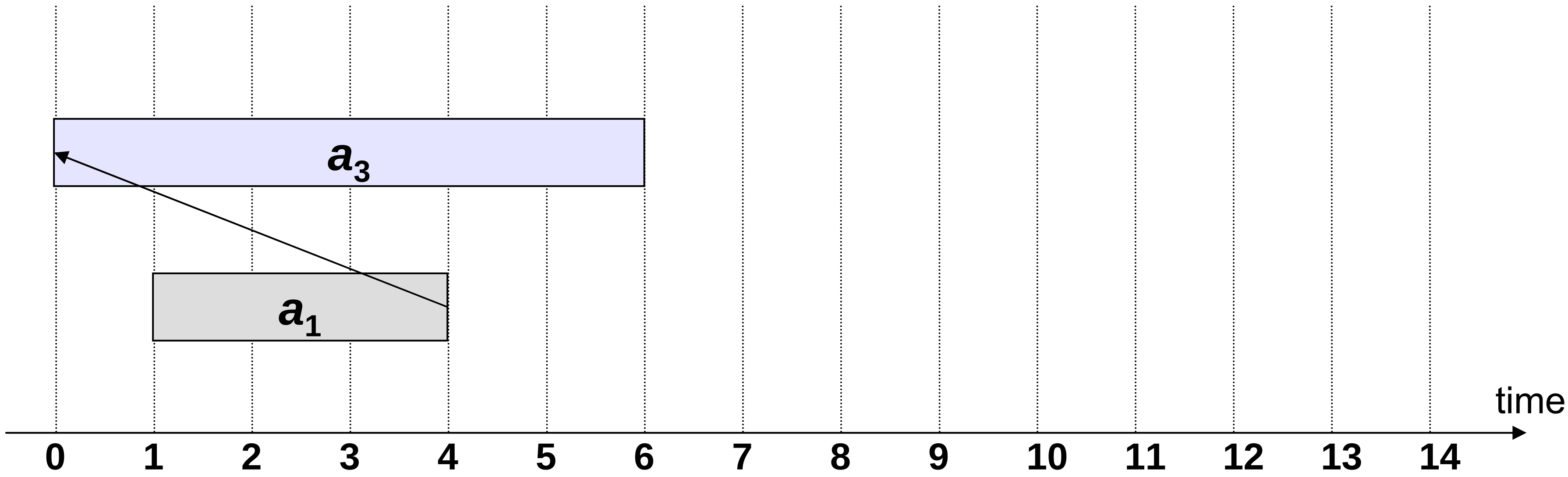
$i = 1,$

$m \leftarrow i + 1 \leftarrow 1 + 1 = 2$

$m < j$ (2 < 12) and $s_2 < f_1$ (3 < 4)

$m \leftarrow m + 1 \leftarrow 2 + 1 = 3$

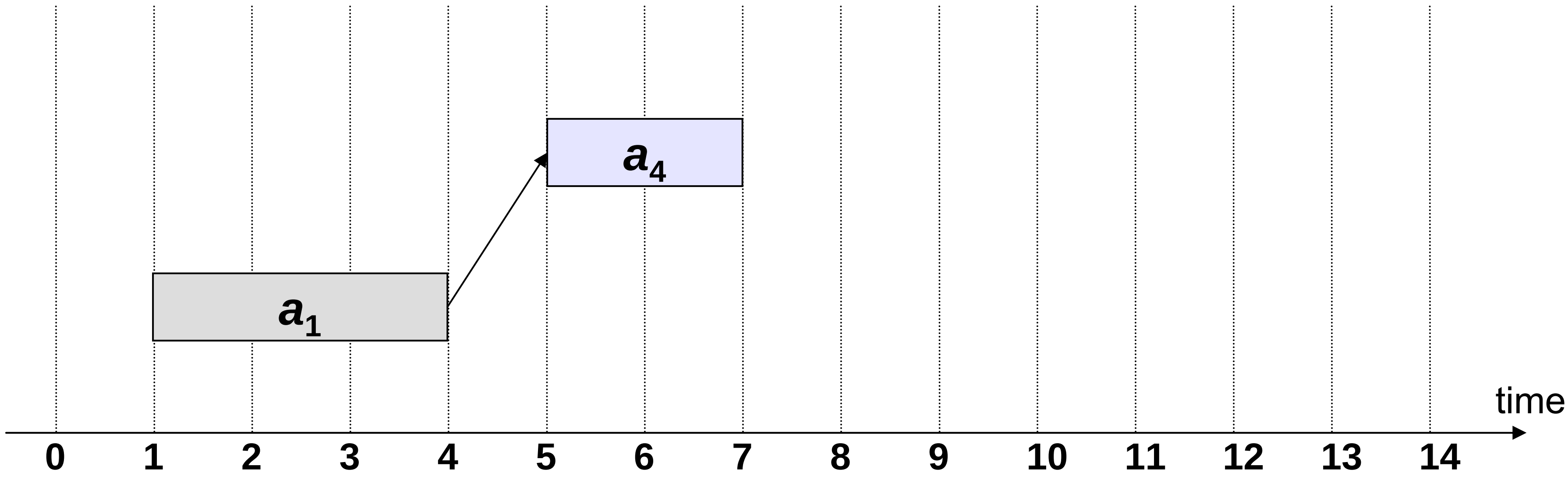
A Recursive Greedy Algorithm



$m < j$ (3 < 12) and $s_3 < f_1$ (0 < 4)

$m \leftarrow m + 1 \leftarrow 3 + 1 = 4$

A Recursive Greedy Algorithm

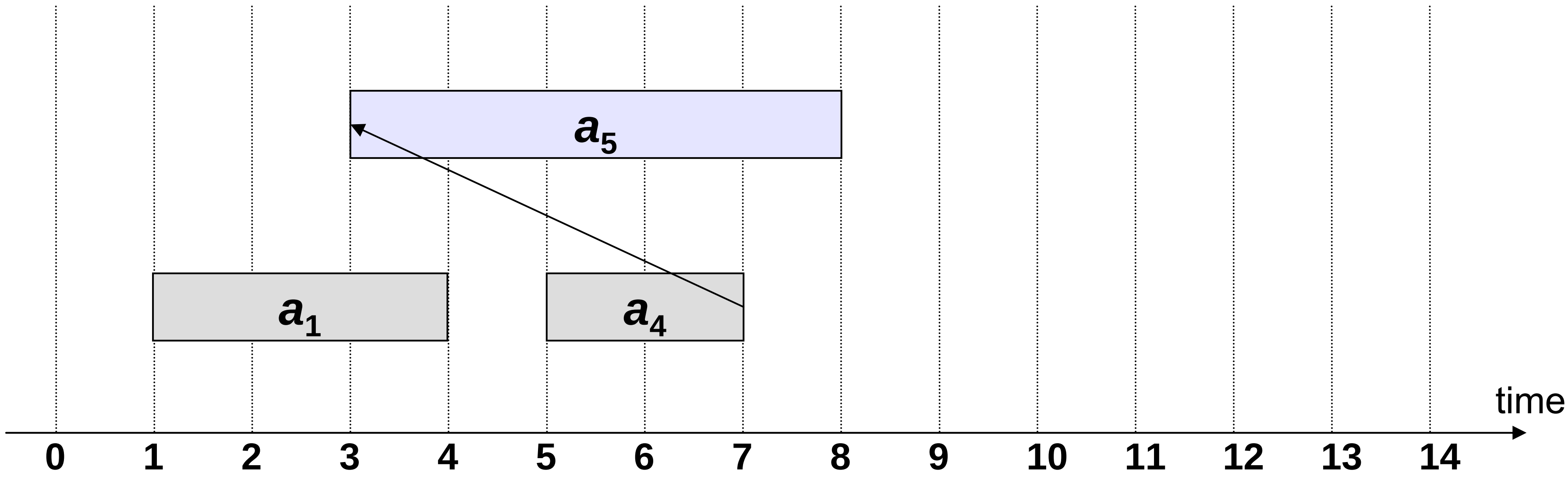


$m < j$ (4 < 12) and $s_4 < f_1$ (But 5 > 4)

if $m < j$ (4 < 12)

return $\{a_4\} \cup$ Recursive-Activity-Selector(s, f, 4,12)

A Recursive Greedy Algorithm



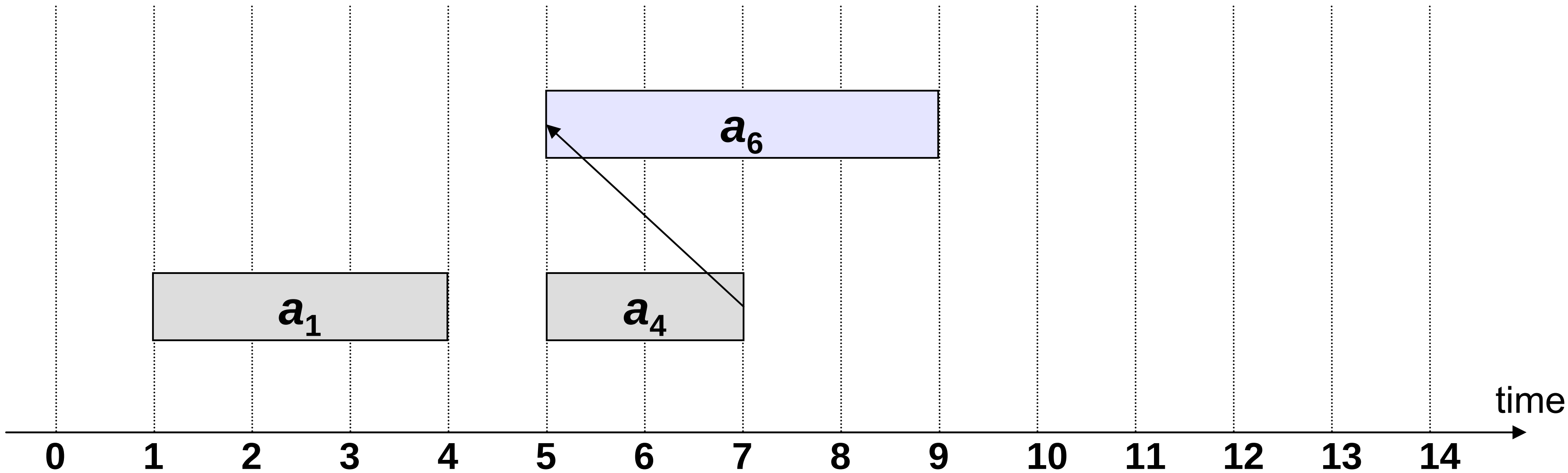
$i = 4,$

$m \leftarrow i + 1 \leftarrow 4 + 1 = 5$

$m < j$ $(5 < 12)$ and $s_5 < f_4$ $(3 < 7)$

$m \leftarrow m + 1 \leftarrow 5 + 1 = 6$

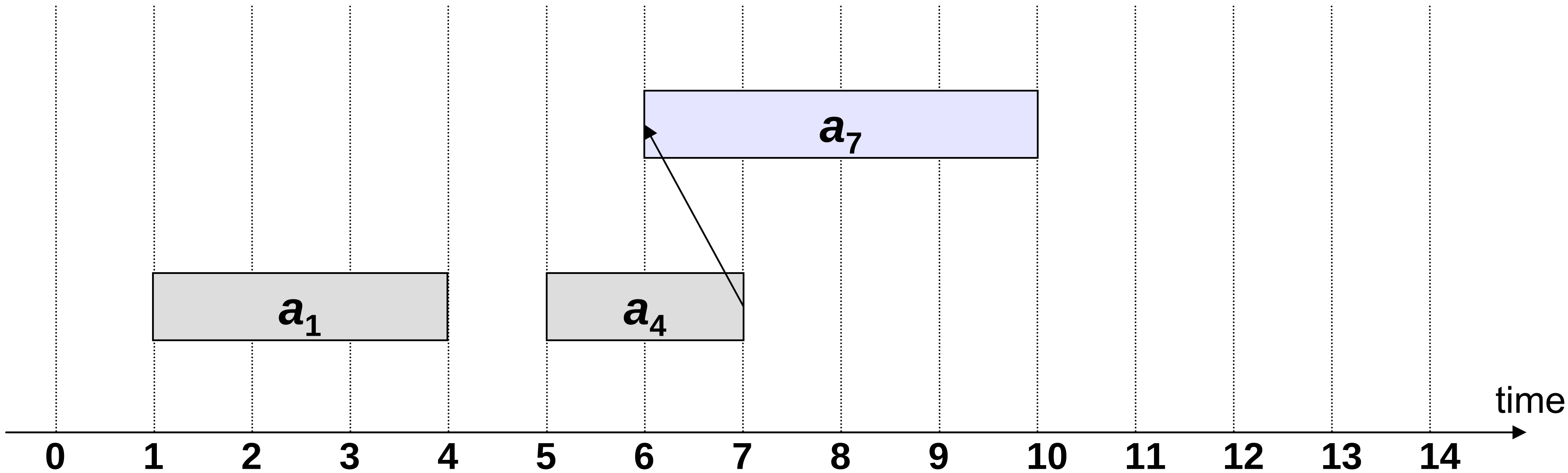
A Recursive Greedy Algorithm



$m < j$ (6 < 12) and $s_6 < f_4$ (5 < 7)

$m \leftarrow m + 1 \leftarrow 6 + 1 = 7$

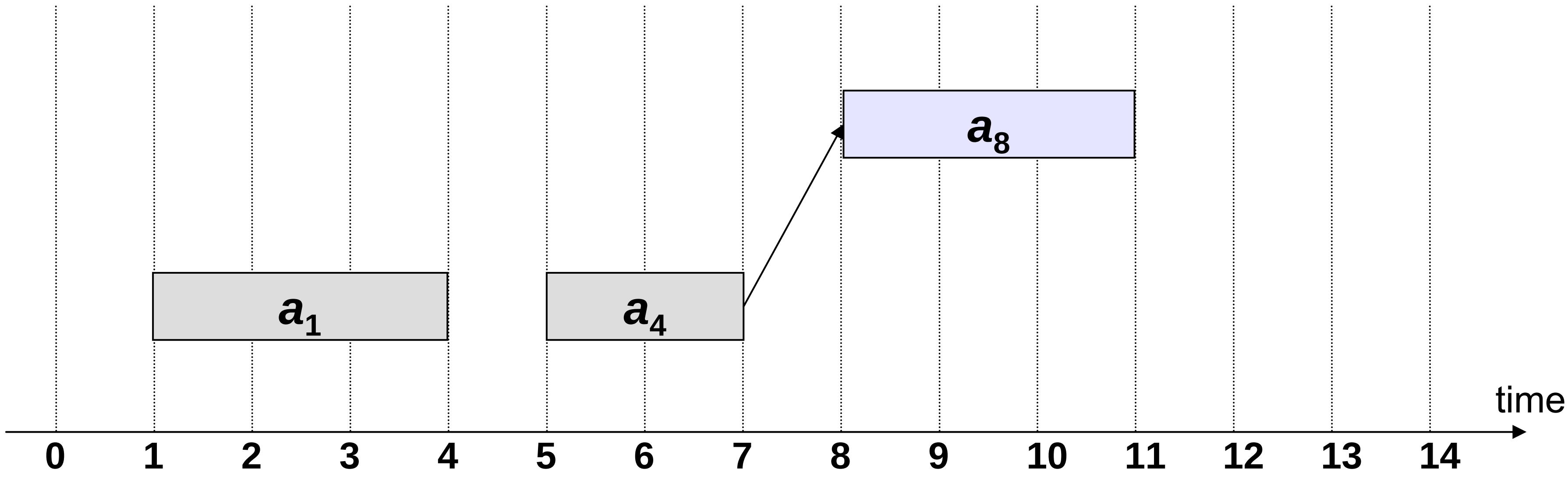
A Recursive Greedy Algorithm



$$m < j \ (7 < 12) \text{ and } s_7 < f_4 \ (6 < 7)$$

$$m \leftarrow m + 1 \leftarrow 7 + 1 = 8$$

A Recursive Greedy Algorithm

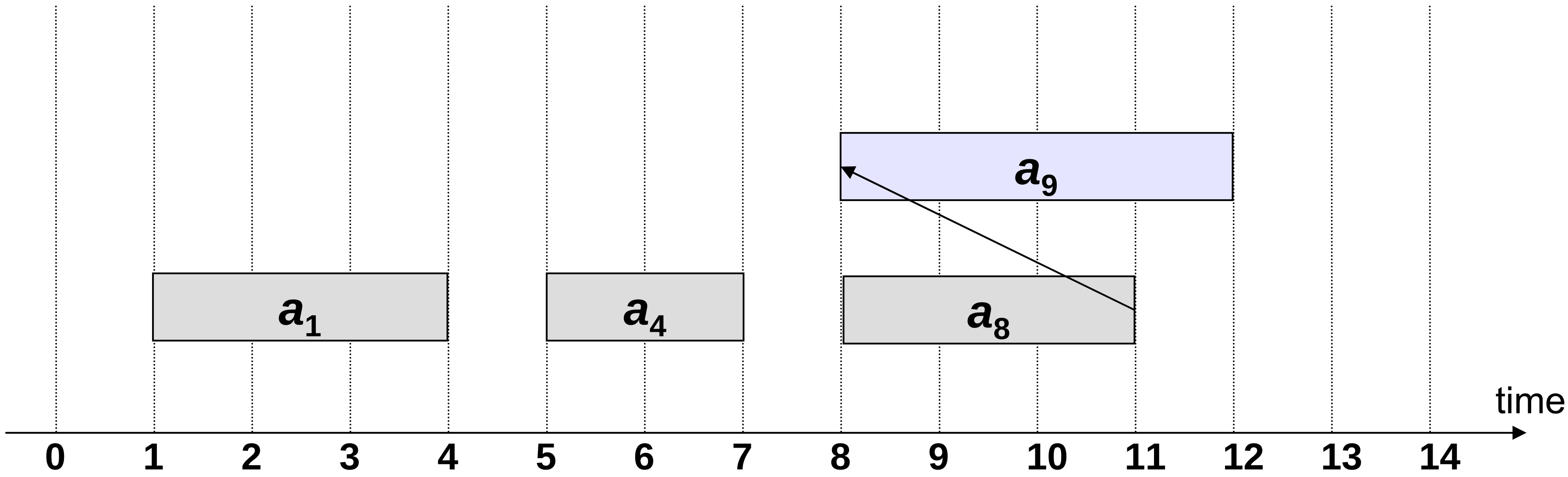


$m < j$ (8 < 12) and $s_8 < f_1$ (But 8 > 7)

if $m < j$ (8 < 12)

return $\{a_8\} \cup$ Recursive-Activity-Selector $(s, f, 8, 12)$

A Recursive Greedy Algorithm



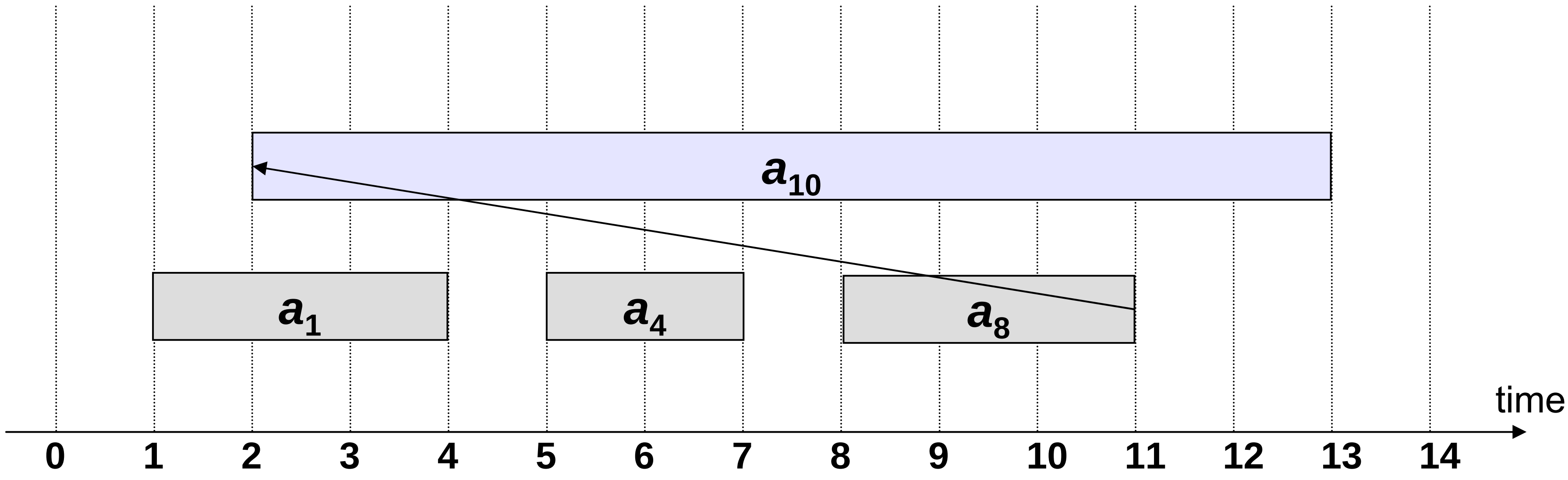
$i = 8,$

$m \leftarrow i + 1 \leftarrow 8 + 1 = 9$

$m < j$ (9 < 12) and $s_9 < f_8$ (8 < 11)

$m \leftarrow m + 1 \leftarrow 9 + 1 = 10$

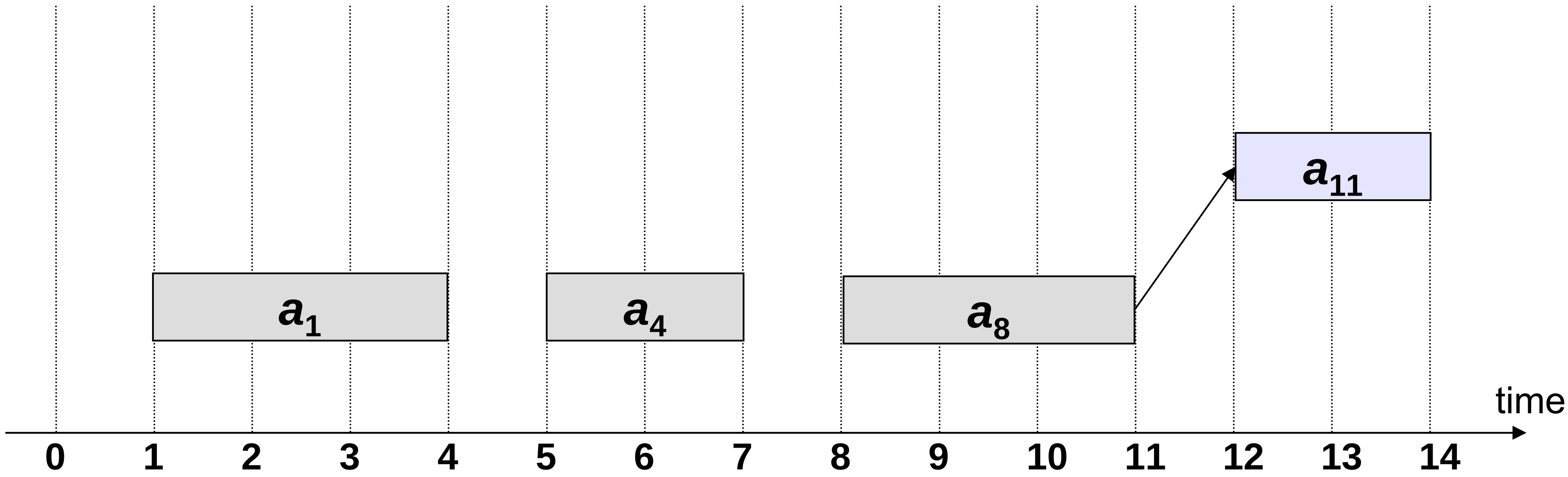
A Recursive Greedy Algorithm



$$m < j \ (10 < 12) \text{ and } s_{10} < f_8 \ (2 < 11)$$

$$m \leftarrow m + 1 \leftarrow 10 + 1 = 11$$

A Recursive Greedy Algorithm

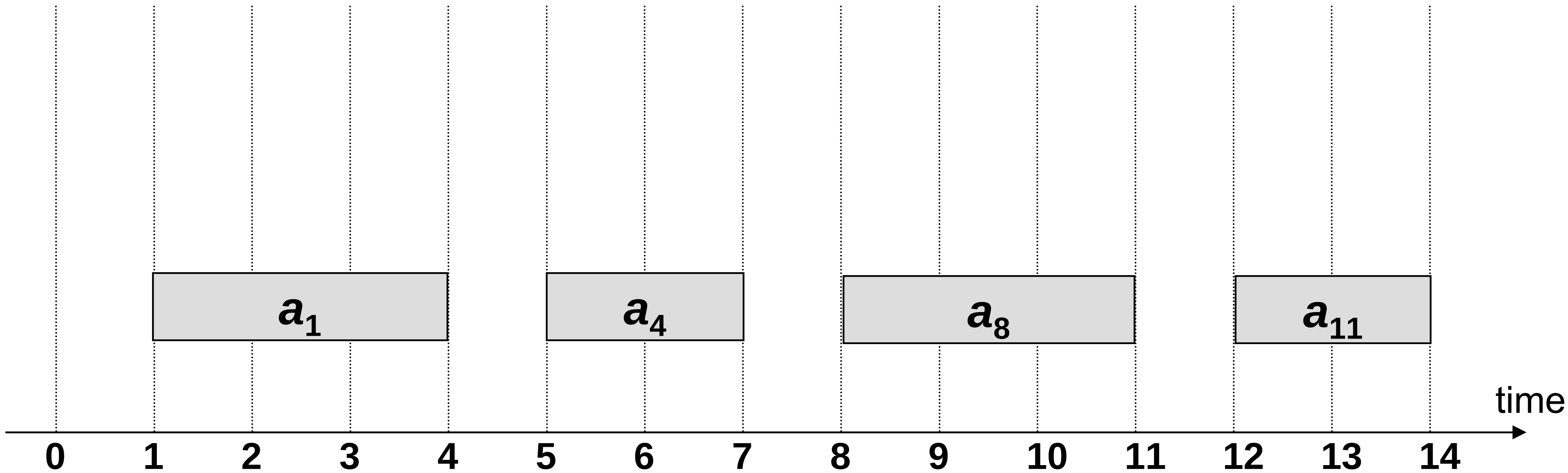


$m < j$ (11 < 12) and $s_{11} < f_8$ (But 12 > 11)

if $m < j$ (11 < 12)

return $\{a_{11}\} \cup$ Recursive-Activity-Selector (s, f, 11,12)

A Recursive Greedy Algorithm



$i = 11$,

$m \leftarrow i + 1 \leftarrow 11 + 1 = 12$

$m < j$ (But $12 = 12$)

Iterative-Activity-Selector (s, f)

$n \leftarrow length[s]$

$A \leftarrow \{a_1\}$

$i \leftarrow 1$

for m $\leftarrow$ 2 to n

    do if $s_m \geq f_i$

        then A $\leftarrow A \cup \{a_m\}$

            $i \leftarrow m$

return A

Summary

- A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices.
- For each decision point in the algorithm, the choice that seems best at the moment is chosen at that time.
- This strategy does not always produce an optimal solution, but as we saw in the activity-selection problem, sometimes it does.
- Now we give a sequence of steps designing an optimal solution of using greedy approach

Summary: Steps Designing Greedy Algorithms

We went through the following steps in the above problem:

1. Determine the suboptimal structure of the problem.
2. Develop a recursive solution.
3. Prove that at any stage of the recursion, one of the optimal choices is the greedy choice. Thus, it is always safe to make the greedy choice.
4. Show that all but one of the sub-problems induced by having made the greedy choice are empty.
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert this recursive algorithm to an iterative one.

Checks in Designing Greedy Algorithms

- In the beneath every greedy algorithm, there is almost always a dynamic programming solution.

  How can one tell if a greedy algorithm will solve a particular optimization problem?
- There is no way in general, but there are two key ingredients
  - greedy choice property and
  - optimal sub-structure
- If we can demonstrate that the problem has these properties, then we are well on the way to developing a greedy algorithm for it.

The Knapsack Problem

- **The 0-1 Knapsack Problem**
  - A thief robbing a store finds n items: i-th item worth v_i and weight w_i, where v_i and w_i integers
  - The thief can only carry weight W in his knapsack
  - Items must be taken entirely or left behind
  - Which items should the thief take to maximize the value of his load?
- **The Fractional Knapsack Problem**
  - Similar to 0-1 can be solved by greedy approach
  - In this case, the thief can take fractions of items.

Greedy Fails in 0-1 knapsack problem

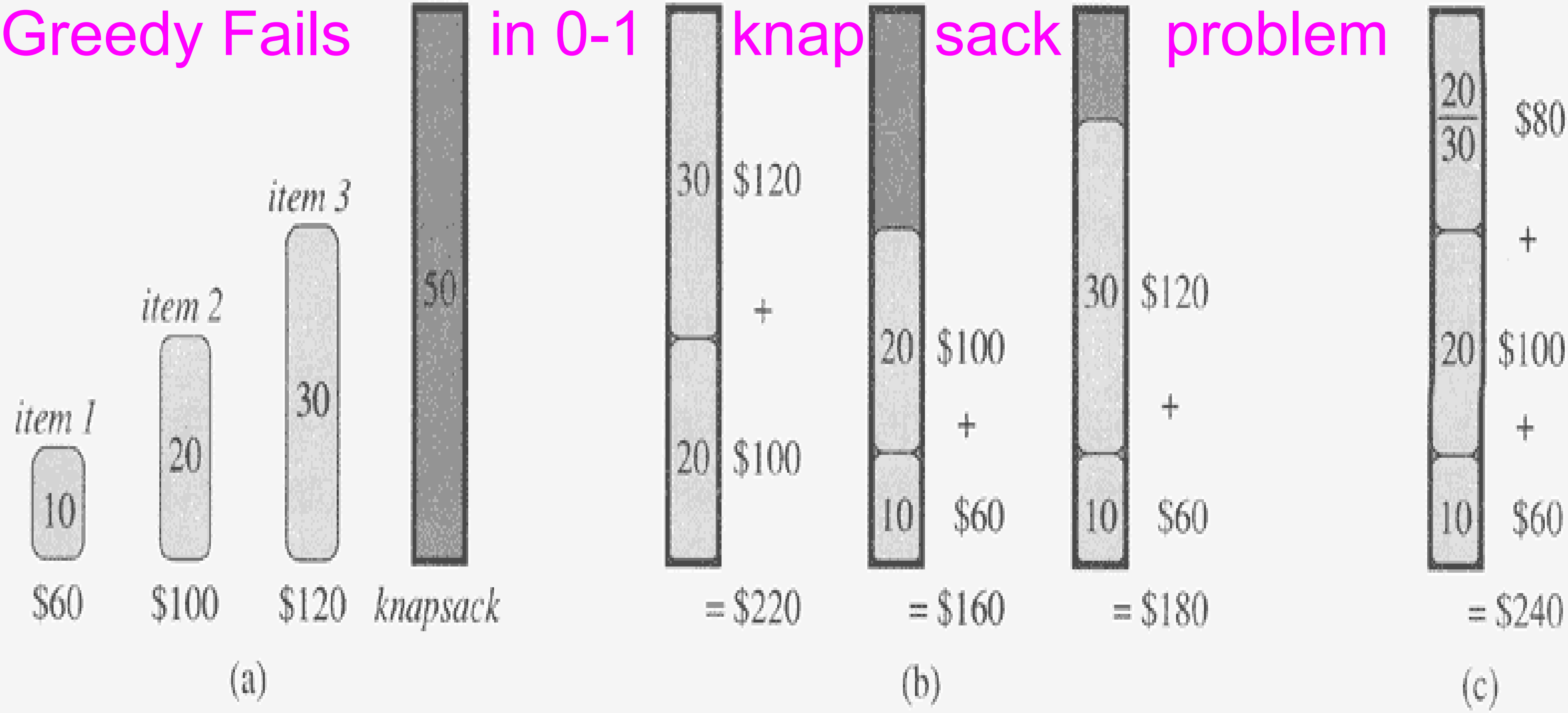


Figure 16.2 The greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest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

Developing Algorithm: Fractional Knapsack

- Pick the item with the maximum value per pound v_i/w_i
- If the supply of that element is exhausted and the thief can carry more then take as much as possible from the item with the next greatest value per pound
- Continue this process till knapsack is filled
- It is good to order items based on their value per pound

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$$

Algorithm: Fractional Knapsack Problem

Fractional-Knapsack (W, v[n], w[n])

1. While $w > 0$ and as long as there are items remaining
2. pick item with maximum v_i/w_i
3. $x_i \leftarrow \min(1, w/w_i)$
4. remove item i from list
5. $w \leftarrow w - x_i w_i$

- w the amount of space remaining in the knapsack ($w = W$)
- Running time: $\Theta(n)$ if items already ordered; else $\Theta(n \lg n)$

Making Change

Making Change

Someone comes to your store and makes a purchase of 98.67. He/she gives you 100. You want to give back change using the least number of coins.

- **INPUT:** The values of coins: $C_1, C_2, \ldots, C_k$, and an integer N. Assume that some coin has value 1.
- **GOAL:** To find a multi-set of coins S whose sum is N where the total number of coins is minimized.
- A greedy approach is to add the highest value coin possible.

Making Change

Greedy algorithm (C, N)

1. sort coins so $C1 \geq C2 \geq . . . \geq Ck$
2. $S = \Phi$;
3. Change = 0
4. i = 1 \\ Check for next coin
5. **while** Change $\neq$ N do \\ all most valuable coins
6. **if** Change + Ci $\leq$ N **then**
7. Change = Change + Ci
8. $S = S \cup \{Ci\}$
9. **else** *i* = *i+1*

Making Change

- In Pakistan, our currency notes are

  $C_1 = 5000$, $C_2 = 1000$, $C_3 = 500$, $C_4 = 100$, $C_5 = 50$, $C_6 = 20$, $C_7 = 10$
- Applying above greedy algorithm to $N = 13{,}660$, we get

  $S = \{C_1, C_1, C_2, C_2, C_2, C_3 , C_4 , C_5 , C_7\}$
- Does this algorithm always find an optimal solution? For Pakistani currency.
- It does but does not hold always

Dynamic Programming vs. Greedy Algorithms

- Dynamic programming
  - We make a choice at each step
  - The choice depends on solutions to subproblems
  - Bottom up solution, smaller to larger subproblems
- Greedy algorithm
  - Make the greedy choice and THEN
  - Solve subproblem arising after the choice is made
  - The choice we make may depend on previous choices, but not on solutions to subproblems
  - Top down solution, problems decrease in size

Conclusion

- Weaknesses of dynamic programming are discussed
- Approach of designing dynamic algorithms is used for design of greedy algorithms.
- Activity selection problem is discussed in detail.
- Best, at a moment, of the sub-problems in dynamic programming are selected. The other sub-problem is forced to become empty in activity selection problem
- Optimality and correctness is proved.
- Discussed why greedy algorithm are efficient.
- Some problems are discussed where Greedy algorithms do not work.

Conclusion

- 0-1 Knapsack problem discussed with greedy approach
- Fractional Knapsack problem analyzed and algorithm using greedy approach is given
- Two different versions of the Task Scheduling Problem are analyzed
- Task Scheduling linked with 0-1 Knapsack
- Coin change problem is discussed with greedy approach
- It is observed that all coin changing problems can not be solved using greedy approach
- Relationship between dynamic programming and greedy approach is reviewed