Greedy Algorithms

Basics

- Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step
- For many optimization problems, using dynamic programming to determine the best choices is overkill;
- Simpler, more efficient algorithms will do
- Greedy algorithm always makes the choice that looks best at the moment

Basics

- Makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution
- do not always yield optimal solutions, but for many problems they do
- quite powerful and works well for a wide range of problems
- minimum–spanning–tree algorithms, Dijkstra’s algorithm

An activity–selection problem

- Scheduling several competing activities that require exclusive use of a common resource
- Goal – Selecting a maximum–size set of mutually compatible activities
- a set $S = \{a_1, a_2,\ldots,a_n\}$ of n proposed activities that wish to use a resource, such as a lecture hall, which can serve only one activity at a time

An activity–selection problem

- Each activity a_i has a start time s i and a finish time f_i, where $0 \leq s_i < fi < \infty$
- If selected, activity ai takes place during the half–open time interval $[s_i, f_i)$
- Activities a_i and a_j are compatible if the intervals $[s_i, f_i)$ and $[s_j , f_j)$ do not overlap
- That is, a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$

An activity–selection problem

- Each activity a_i has a start time s i and a finish time f_i, where $0 \leq s_i < fi < \infty$
- If selected, activity ai takes place during the half–open time interval $[s_i, f_i)$
- Activities a_i and a_j are compatible if the intervals $[s_i, f_i)$ and $[s_j , f_j)$ do not overlap
- That is, a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$

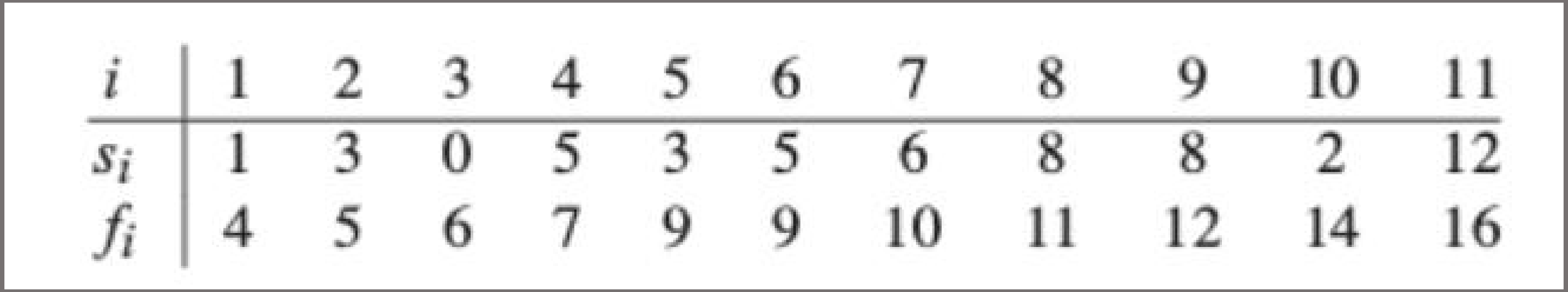
An activity–selection problem

- We wish to select a maximum–size subset of mutually compatible activities
- We assume that the activities are sorted in monotonically increasing order of finish time:

$$f_1 \le f_2 \le f_3 \le \cdots \le f_{n-1} \le f_n \ .$$

Example

- Consider the following set S 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

- For this example, the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities
- not a maximum subset
- subset $\{a_1, a_4, a_8, a_{11}\}$ is larger
- another largest subset is $\{a_2, a_4, a_9, a_{11}\}$

Optimal substructure of the activity-selection problem

- S_{ij} – Set of activities that start after activity a_i finishes and that finish before activity a_j starts
- Suppose that we wish to find a maximum set of mutually compatible activities in S_{ij}, and suppose further that such a maximum set is A_{ij} , which includes some activity a_k

Optimal substructure of the activity–selection problem

- By including a_k in an optimal solution, we are with two subproblems:

(i) Finding mutually compatible activities in set S_{ik} (activities that start after activity a_i finishes and that finish before activity a_k starts)

(ii) Finding mutually compatible activities in the set S_{kj} (activities that start after activity a_k finishes and that finish before activity a_j starts).

Optimal substructure of the activity–selection problem

- Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$, so that A_{ik} contains the activities in A_{ij} that finish before a_k starts and A_{kj} contains the activities in A_{ij} that start after a_k finishes.
- Thus, we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, and so the maximum–size set A_{ij} of mutually compatible activities in S_{ij} consists of $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ activities.

Optimal substructure of the activity–selection problem

- This way of characterizing optimal substructure suggests that we might solve the activity–selection problem by dynamic programming
- If we denote the size of an optimal solution for the set S_{ij} by c[i,j] , then we would have the recurrence

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

Optimal substructure of the activity–selection problem

- to examine all activities in S_{ij} to find which one to choose, so that

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

- We could then develop a recursive algorithm and memoize it, or we could work bottom–up and fill in table entries as we go along.
- But we would be overlooking another important characteristic of the activity–selection problem that we can use to great advantage.

Greedy choice for activity–selection problem

- If we could choose an activity to add to our optimal solution without having to first solve all the subproblems
- Could save us from having to consider all the choices inherent in recurrence
- Consider only one choice: the greedy choice

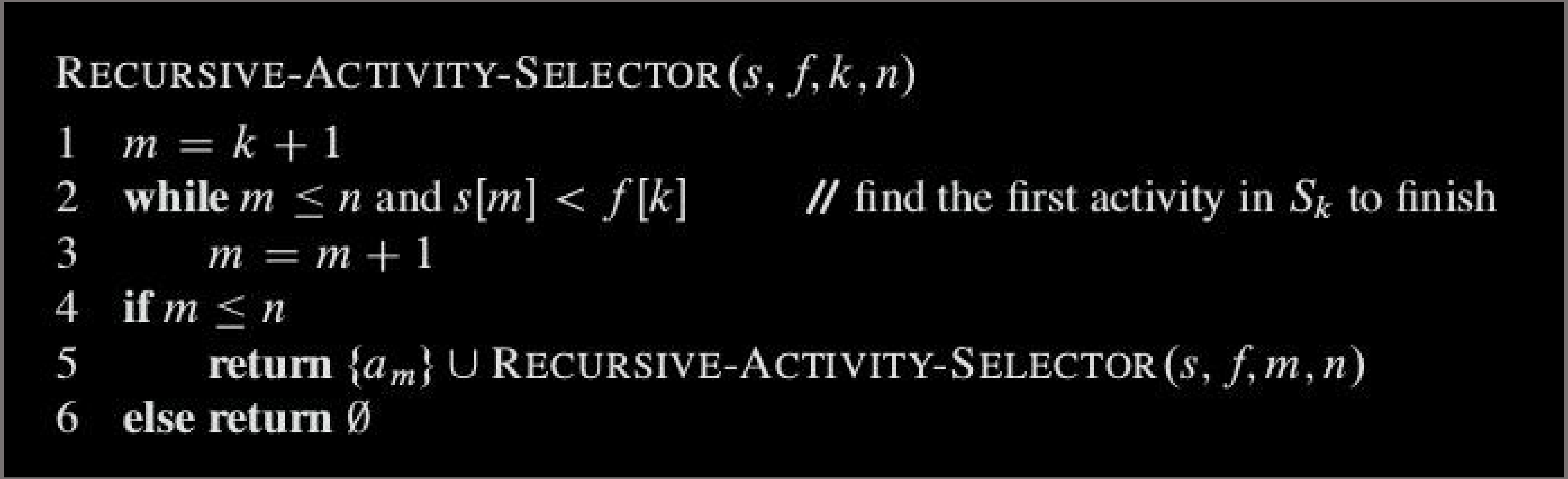
Making the Greedy Choice

- Our intuition tells us to choose the activity in S with the earliest finish time, since that would leave the resource available for as many of the activities that follow it as possible
- If more than one activity in S has earliest finish time, then we can choose any such activity
- Since activities are sorted in monotonically increasing order by finish time, the greedy choice is activity a_1 .

Making the Greedy Choice

- If we make the greedy choice, we have only one remaining subproblem to solve:
- finding activities that start after a_1 finishes
- f_1 is the earliest finish time of any activity, and therefore no activity can have a finish time less than or equal to s_1
- Here 'k' in optimal substructure is 1, thus all activities that are compatible with activity a_1 must start after a_1 finishes

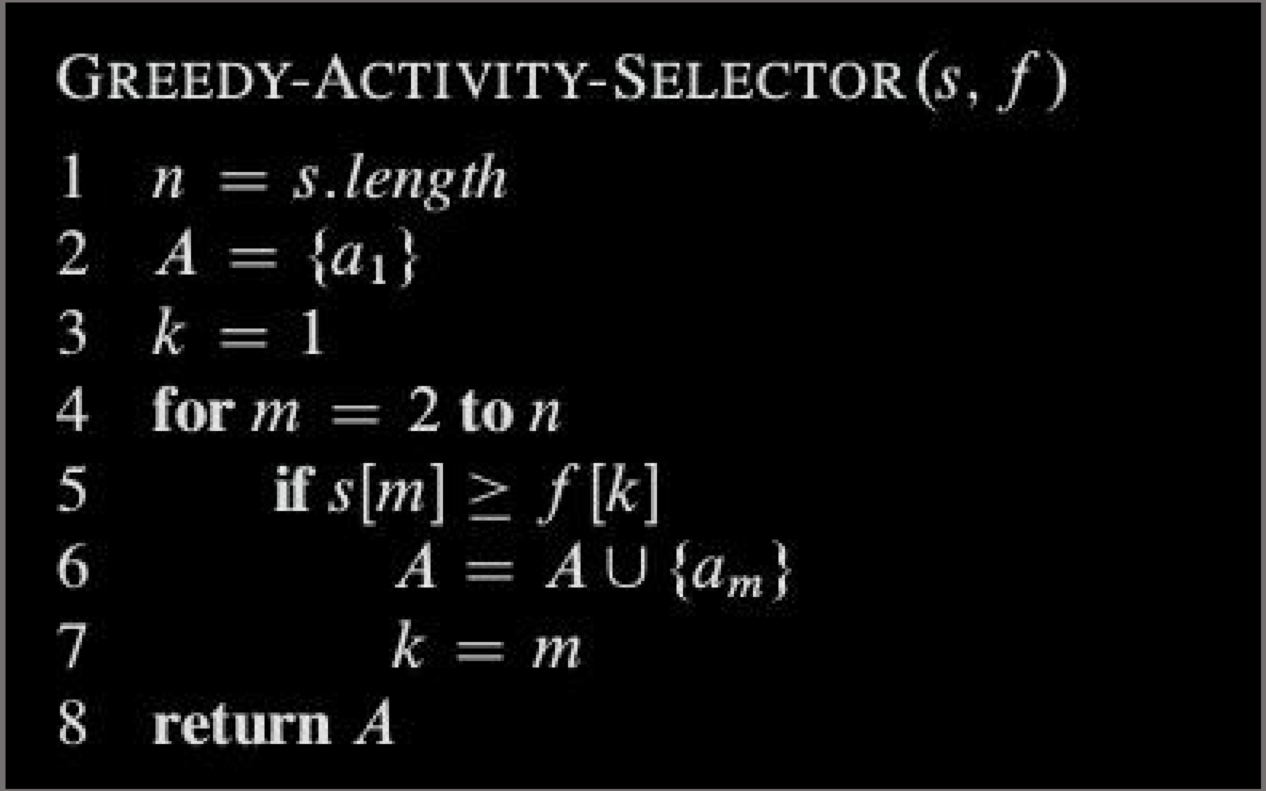
Recursive Greedy Algorithm



```
RECURSIVE-ACTIVITY-SELECTOR$(s, f, k, n)$
  $m = k + 1$
  while $m \le n$ and $s[m] < f[k]$      // find the first activity in $S_k$ to finish
      $m = m + 1$
  if $m \le n$
      return $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR$(s, f, m, n)$
  else return $\emptyset$
```

- Initial call, which solves the entire problem, is RECURSIVE –ACTIVITY –S ELECTOR (s, f, 0, n)
- Running time is θ (n)

Iterative Greedy Algorithm



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

- Initial call, which solves the entire problem, is GREEDY–ACTIVITY–S ELECTOR (s, f)
- Running time is θ (n)

Proof for Intitution

Theorem 16.1

- Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time
- Then a_m is included in some maximum–size subset of mutually compatible activities of S_k

Proof for Intitution

- Let A_k be a maximum–size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the earliest finish time
- If $a_j = a_m$, we are done, since we have shown that a_m is in some maximum–size subset of mutually compatible activities of S_k .
- If $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k but substituting a_m for a_j

Proof for Intitution

- The activities in A'_k are disjoint, which follows because the activities in A_k are disjoint, a_j is the first activity in A_k to finish, and $f_m \leq f_j$
- Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum–size subset of mutually compatible activities of S_k, and it includes a_m

Elements of the Greedy Strategy

- Obtains an optimal solution to a problem by making a sequence of choices
- At each decision point, 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

Greedy Strategy followed for Activity Selection Problem

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Show that if we make the greedy choice, then only one subproblem remains
4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur in either order.)
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm

Elements of the Greedy Strategy

- In the activity–selection problem, we first defined the subproblems S_{ij}, where both i and j varied
- We then found that if we always made the greedy choice, we could restrict the subproblems to be of the form S_k
- Alternatively, we could have fashioned our optimal substructure with a greedy choice in mind, so that the choice leaves just one subproblem to solve

Elements of the Greedy Strategy

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.

Elements of the Greedy Strategy

3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Question

- Will a greedy algorithm will always solve a particular optimization problem
- No way works all the time
- Two key ingredients:
  - Greedy–choice property
  - Optimal substructure
- If we can demonstrate that the problem has these properties, then we are well on the way to developing a greedy algorithm for it

DP Vs Greedy

- In DP, we make a choice at each step, but the choice usually depends on the solutions to subproblems
- We may follow bottom up approach
- In a greedy, we make whatever choice seems best at the moment and then solve the subproblem that remains
- Choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems

DP Vs Greedy

- DP – Solves subproblems before making the first choice
- Greedy – Makes its first choice before solving any subproblems
- Greedy strategy usually progresses in a top–down fashion, making one greedy choice after another, reducing each given problem instance to a smaller one

DP Vs Greedy

- We must prove that a greedy choice at each step yields a globally optimal solution
- Typically, as in the case of Theorem 16.1, the proof examines a globally optimal solution to some subproblem
- It then shows how to modify the solution to substitute the greedy choice for some other choice, resulting in one similar, but smaller, subproblem

Efficient Greedy Choice

- In the activity–selection problem, assuming that we had already sorted the activities in monotonically increasing order of finish times, we needed to examine each activity just once.
- By preprocessing the input or by using an appropriate data structure (often a priority queue)
- We often can make greedy choices quickly, thus yielding an efficient algorithm.

Optimal substructure

- Problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems
- This property is a key ingredient of assessing the applicability of DP as well as greedy algorithms
- As an example, recall demonstration in Section 16.1 that if an optimal solution to subproblem S_{ij} includes an activity a_k, then it must also contain optimal solutions to subproblems S_{ik} and S_{kj}

Optimal substructure

- Based on this observation of optimal substructure, we were able to devise the recurrence (16.2) that described the value of an optimal solution
- a more direct approach regarding optimal substructure when applying it to greedy algorithms
- As mentioned above, we have the luxury of assuming that we arrived at a subproblem by having made the greedy choice in the original problem.

Optimal substructure

- All we really need to do is argue that an optimal solution to the subproblem, combined with the greedy choice already made, yields an optimal solution to the original problem
- This scheme implicitly uses induction on the subproblems to prove that making the greedy choice at every step produces an optimal solution

Decision Greedy vs Dynamic Programming

- Both greedy and dynamic–programming strategies exploit optimal substructure
- Might be tempted to generate a dynamic–programming solution to a problem when a greedy solution suffices or,
- Conversely, you might mistakenly think that a greedy solution works when in fact a dynamic–programming solution is required

0–1 knapsack problem

- A thief robbing a store finds n items – must either take it or leave it behind – cannot take a fractional amount of an item
- The ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers
- 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?

Fractional knapsack problem

- Setup is same, but thief can take fractions of items, rather than having to make a binary (0–1) choice for each item.
- You can think of an item in the 0–1 knapsack problem as being like a gold ingot
- an item in the fractional knapsack problem as more like gold dust

Optimal–substructure property for knapsack problem

- For the 0–1 problem, consider the most valuable load that weighs at most W pounds.
- If we remove item j from this load, the remaining load must be the most valuable load weighing at most $W - w_j$ that the thief can take from the $n - 1$ original items excluding j

Optimal–substructure property for knapsack problem

- For the fractional problem, consider that if we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ that the thief can take from the $n - 1$ original items plus $w_j - w$ pounds of item j

Solution Statergy

- Although the problems are similar:

- Fractional knapsack problem – Solved by a greedy strategy

- but we cannot solve the 0–1 problem by such a strategy

Greedy Algorithm for Fractional knapsack problem

- Compute the value per pound v_i / w_i for each item
- Begin by taking as much as possible of the item with the greatest value per pound
- If the supply of that item is exhausted then take as much as possible of the item with the next greatest value per pound, and so forth, until he reaches his weight limit W

Greedy Algorithm for Fractional knapsack problem

- Thus, by sorting the items by value per pound, the greedy algorithm runs in O(n lg n)time
- But this will not work for 0/1 knapsack problem

Illustration

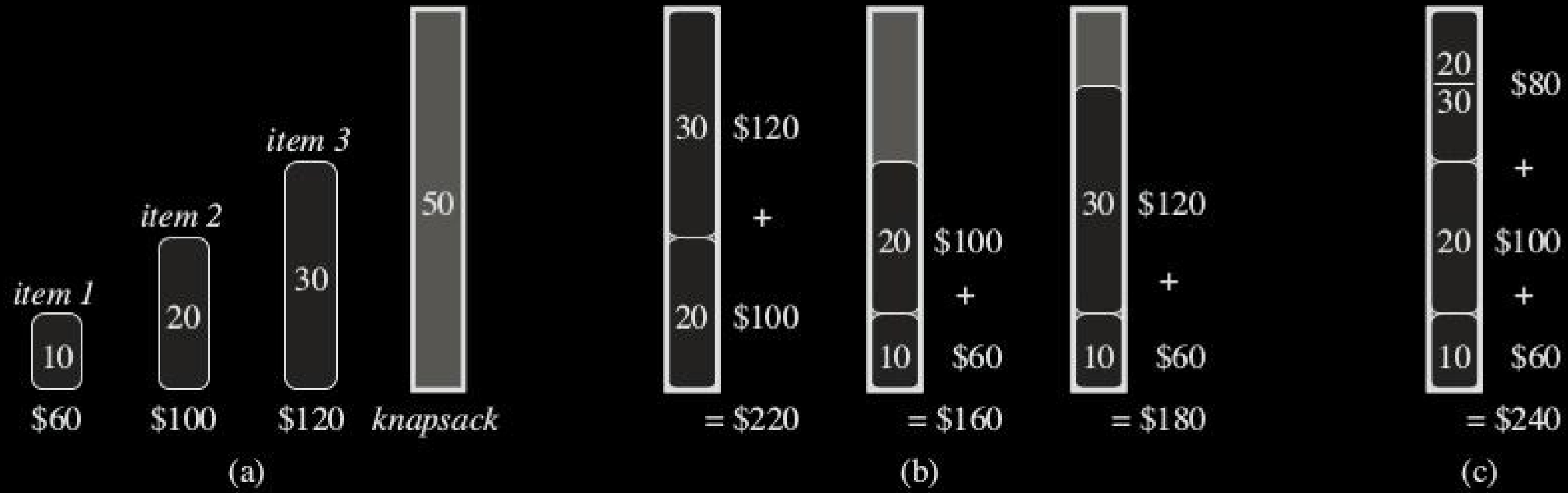


Figure 16.2 An example showing that the greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though 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.

Illustration

- This example has 3 items and a knapsack that can hold 50 pounds.
- Item 1 weighs 10 pounds and is worth 60 dollars
- Item 2 weighs 20 pounds and is worth 100 dollars
- Item 3 weighs 30 pounds and is worth 120 dollars
- Thus, the value per pound of item 1 – 6 dollars, item 2 (5 dollars) or item 3 (4 dollars)