T&R Team of Algorithm Design
College of Computer Science and Engineering, CQU

Algorithm Analysis & Design

Introduction to Algorithm

16

GREEDY ALGORITHM

Locally optimal choice

Overview

- Like dynamic programming, used to solve optimization problems.
- Dynamic programming can be overkill; greedy algorithms tend to be easier to code
- Problems exhibit optimal substructure (like DP).
- Problems also exhibit the **greedy-choice** property.
  - When we have a choice to make, make the one that looks best *right now*.
  - Make a **locally optimal choice** in hope of getting a **globally optimal solution**.

Greedy Strategy

- The choice that seems best at the moment is the one we go with.
  - Prove that when there is a choice to make, one of the optimal choices is the greedy choice. Therefore, it's always safe to make the greedy choice.
  - Show that all but one of the subproblems resulting from the greedy choice are empty.

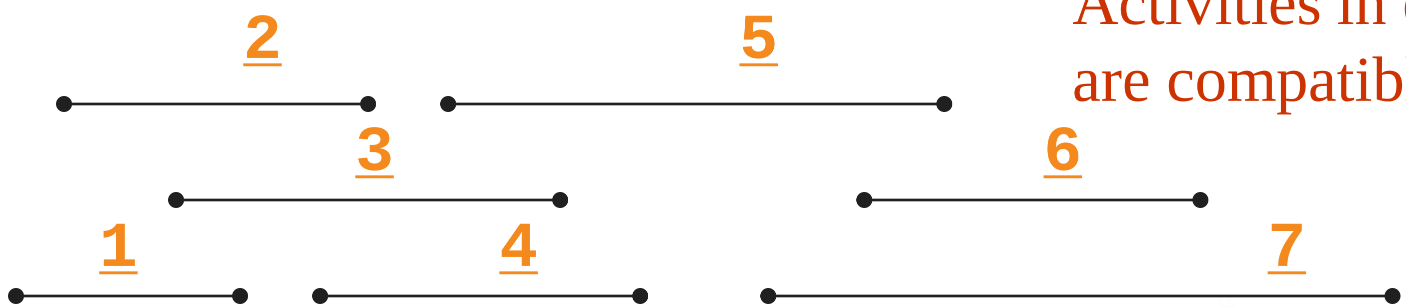
Activity-Selection Problem

- Problem: get your money's worth out of a festival
  - Buy a wristband that lets you onto any ride
  - Lots of rides, each starting and ending at different times
  - Your goal: ride as many rides as possible
    - Another, alternative goal that we don't solve here: maximize time spent on rides
- Welcome to the *activity selection problem*

Activity-selection Problem

- Input: Set *S* of *n* activities, $a_1, a_2, \ldots, a_n$.
  - s_i = start time of activity *i*.
  - f_i = finish time of activity *i*.
- Output: Subset A of maximum number of compatible activities.
  - Two activities are compatible, if their intervals don't overlap.

Example:



Activities in each line are compatible.

Optimal Substructure

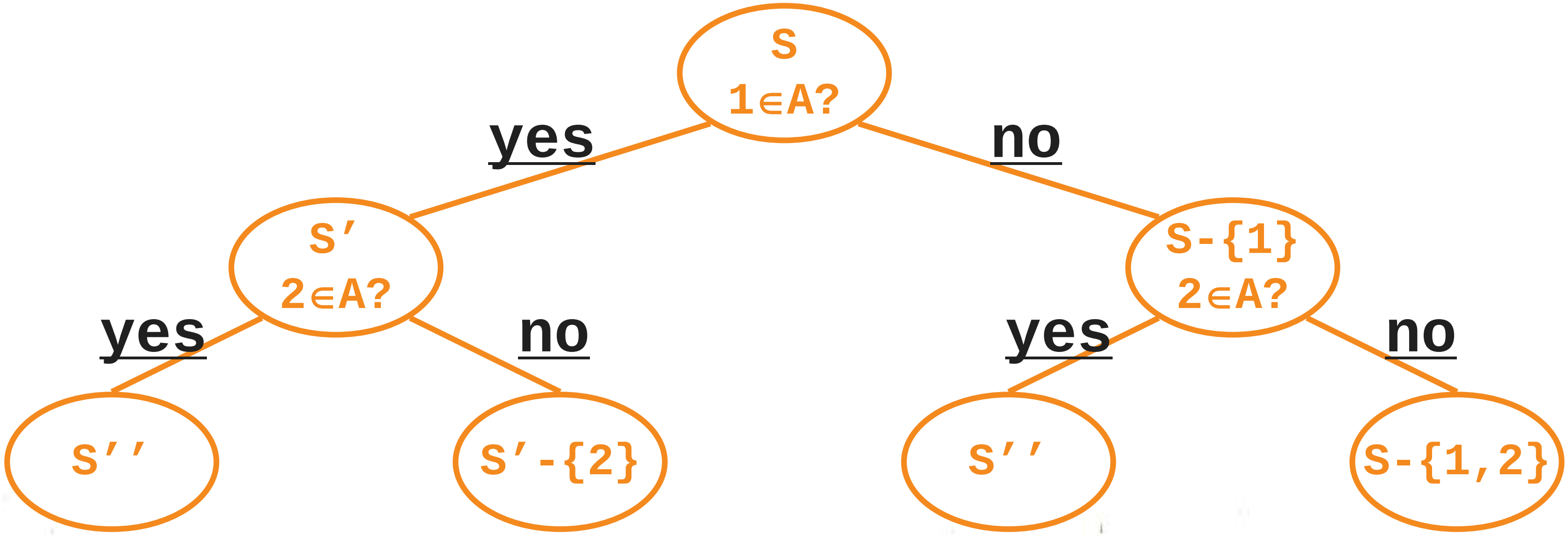
- Assume activities are sorted by finishing times.
  - $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Suppose an optimal solution includes activity a_k.
  - This generates two subproblems.
  - Selecting from $a_1, \ldots, a_{k-1}$, activities compatible with one another, and that finish before a_k starts (compatible with a_k).
  - Selecting from $a_{k+1}, \ldots, a_n$, activities compatible with one another, and that start after a_k finishes.
  - The solutions to the two subproblems must be optimal.
    - Prove using the cut-and-paste approach.

Optimal Substructure

- Assume activities are sorted by finishing times.
  - $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Suppose an optimal solution includes activity a_k.
  - This generates two subproblems.
  - Selecting from $a_1, \ldots, a_{k-1}$, activities compatible with one another, and that finish before a_k starts (compatible with a_k).
  - Selecting from $a_{k+1}, \ldots, a_n$, activities compatible with one another, and that start after a_k finishes.
  - The solutions to the two subproblems must be optimal.
    - Prove using the cut-and-paste approach.

Activity Selection: Repeated Subproblems

- Consider a recursive algorithm that tries all possible compatible subsets to find a maximal set, and notice repeated subproblems:



Recursive Solution

- Let S_{ij} = subset of activities in S that start after a_i finishes and finish before a_j starts.
- Subproblems: Selecting maximum number of mutually compatible activities from S_{ij}.
- Let $c[i, j]$ = size of maximum-size subset of mutually compatible activities in S_{ij}.

Recursive Solution:

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

Greedy Choice Property

- Dynamic programming? Memoize? Yes, but…
- Activity selection problem also exhibits the *greedy choice* property:
  - Locally optimal choice $\Rightarrow$ globally optimal sol'n
  - Them 16.1: if S is an activity selection problem sorted by finish time, then $\exists$ optimal solution $A \subseteq S$ such that $\{1\} \in A$
    - Sketch of proof: if $\exists$ optimal solution B that does not contain $\{1\}$, can always replace first activity in B with $\{1\}$ (*Why?*). Same number of activities, thus optimal.

Greedy-choice Property

- The problem also exhibits the greedy-choice property.
  - There is an optimal solution to the subproblem S_{ij}, that includes the activity with the smallest finish time in set S_{ij}.
  - Can be proved easily.
- Hence, there is an optimal solution to S that includes a_1.
- Therefore, make this greedy choice without solving subproblems first and evaluating them.
- Solve the subproblem that ensues as a result of making this greedy choice.
- Combine the greedy choice and the solution to the subproblem.

Recursive Algorithm

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

1. $m \leftarrow i+1$
2. **while** $m < j$ and $s_m < f_i$
3. **do** $m \leftarrow m+1$
4. **if** $m < j$
5. **then return** $\{a_m\} \cup$ Recursive-Activity-Selector(s, f, m, j)
6. else return ϕ

Initial Call: Recursive-Activity-Selector (s, f, 0, n+1)

Complexity: $\Theta(n)$

Straightforward to convert the algorithm to an iterative one.

Typical Steps

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
- Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.
- Make the greedy choice and **solve top-down**.
- May have to preprocess input to put it into greedy order.
  - Example: Sorting activities by finish time.

Activity Selection: A Greedy Algorithm

- So actual algorithm is simple:
  - Sort the activities by finish time
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes
  - Repeat until no more activities
- Intuition is even more simple:
  - Always pick the shortest ride available at the time

GREEDY-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

Elements of Greedy Algorithms

- Greedy-choice Property.
  - A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- Optimal Substructure.

Knapsack Problem

The Knapsack Problem

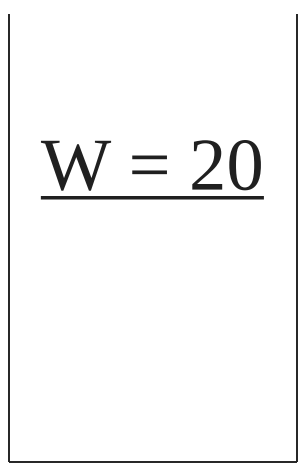
- The famous *knapsack problem*:
  - A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?

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 , b_i and *W* are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem: a picture

This is a knapsack
Max weight: W = 20



Items	Weight w_i	Benefit value b_i
	2	3
	3	4
	4	5
	5	8
	9	10

The Knapsack Problem

- More formally, the *0-1 knapsack problem*:
  - The thief must choose among n items, where the ith item worth v_i dollars and weighs w_i pounds
  - Carrying at most W pounds, maximize value
    - Note: assume v_i, w_i, and W are all integers
    - "0-1" b/c each item must be taken or left in entirety
- A variation, the *fractional knapsack problem*:
  - Thief can take fractions of items
  - Think of items in 0-1 problem as gold ingots, in fractional problem as buckets of gold dust

0-1 Knapsack problem

- Problem, in other words, is to find

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

- The problem is called a *"0-1"* problem, because each item must be entirely accepted or rejected.
- Just another version of this problem is the "*Fractional Knapsack Problem*", where we can take fractions of items.

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with 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 the most total value and with total weight less or equal to *W*
- Running time will be $O(2^n)$

0-1 Knapsack problem: brute-force approach

- Can we do better?
- Yes, with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Let's try this:

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for

$S_k = \{items\ labeled\ 1, 2, ..\ k\}$

Defining a Subproblem

Defining a Subproblem

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k *= {items labeled 1, 2, .. k}*

Defining a Subproblem

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k = *{items labeled 1, 2, .. k}*

- This is a valid subproblem definition.

Defining a Subproblem

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k = *{items labeled 1, 2, .. k}*

- This is a valid subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?

Defining a Subproblem

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k = *{items labeled 1, 2, .. k}*

- This is a valid subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we <u>can't</u> do that.

  Explanation follows....

Defining a Subproblem

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

S_4

S_5

Defining a Subproblem

$w_1=2$ $b_1=3$	$w_2=4$ $b_2=5$	$w_3=5$ $b_3=8$	$w_4=3$ $b_4=4$	

S_4

S_5

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

Defining a Subproblem

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_4 = 3$ $b_4 = 4$	

Max weight: W = 20

For S_4:

Total weight: 14;

total benefit: 20

S_5

S_4

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

Defining a Subproblem

$w_1=2$ $b_1=3$	$w_2=4$ $b_2=5$	$w_3=5$ $b_3=8$	$w_4=3$ $b_4=4$	

Max weight: W = 20

For S_4:

Total weight: 14;

total benefit: 20

$w_1=2$ $b_1=3$	$w_2=4$ $b_2=5$	$w_3=5$ $b_3=8$	$w_4=9$ $b_4=10$

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

S_4

S_5

Defining a Subproblem

$w_1=2$ $b_1=3$	$w_2=4$ $b_2=5$	$w_3=5$ $b_3=8$	$w_4=3$ $b_4=4$	

Max weight: W = 20

For S_4:

Total weight: 14;

total benefit: 20

S_5

S_4

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

$w_1=2$ $b_1=3$	$w_2=4$ $b_2=5$	$w_3=5$ $b_3=8$	$w_4=9$ $b_4=10$

For S_5:

Total weight: 20

total benefit: 26

Defining a Subproblem

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_4 = 3$ $b_4 = 4$	

Max weight: W = 20

For S_4:

Total weight: 14;

total benefit: 20

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_4 = 9$ $b_4 = 10$

For S_5:

Total weight: 20

total benefit: 26

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

S_4

S_5

Solution for S_4 is not part of the solution for S_5!!!

Defining a Subproblem

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_4 = 3$ $b_4 = 4$	

?

Max weight: W = 20

For S_4:

Total weight: 14;

total benefit: 20

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_4 = 9$ $b_4 = 10$

For S_5:

Total weight: 20

total benefit: 26

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

S_4

S_5

Solution for S_4 is not part of the solution for S_5!!!

Defining a Subproblem (continued)

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is flawed and we need another one!
- Let's add another parameter: *w*, which will represent the exact weight for each subset of items
- The subproblem then will be to compute *B[k,w]*

Recursive Formula for subproblems

- Recursive formula for subproblems:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- It means, that the best subset of S_k that has total weight w is one of the two:

1) the best subset of S_{k-1} that has total weight w, **or**

2) the best subset of S_{k-1} that has total weight $w-w_k$ plus the item k

Recursive Formula

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

Recursive Formula

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight *w,* either contains item *k* or not.

Recursive Formula

$$B[k,w]=\begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\, B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.
- First case: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be > w, which is unacceptable

Recursive Formula

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight *w,* either contains item *k* or not.
- First case: $w_k > w$. Item *k* can't be part of the solution, since if it was, the total weight would be > *w*, which is unacceptable
- Second case: $w_k <= w$. Then the item *k* can be in the solution, and we choose the case with greater value

The Knapsack Problem
And Optimal Substructure

The Knapsack Problem And Optimal Substructure

- Both variations exhibit optimal substructure

The Knapsack Problem And Optimal Substructure

- Both variations exhibit optimal substructure
- To show this for the 0-1 problem, consider the most valuable load weighing at most *W* pounds

The Knapsack Problem And Optimal Substructure

- Both variations exhibit optimal substructure
- To show this for the 0-1 problem, consider the most valuable load weighing at most *W* pounds
  - *If we remove item j from the load, what do we know about the remaining load?*

The Knapsack Problem And Optimal Substructure

- Both variations exhibit optimal substructure
- To show this for the 0-1 problem, consider the most valuable load weighing at most *W* pounds
  - *If we remove item j from the load, what do we know about the remaining load?*
  - A: remainder must be the most valuable load weighing at most $W - w_j$ that thief could take from museum, excluding item j

Solving The Knapsack Problem

Solving The Knapsack Problem

- The optimal solution to the fractional knapsack problem can be found with a greedy algorithm
  - *How?*

Solving The Knapsack Problem

- The optimal solution to the fractional knapsack problem can be found with a greedy algorithm
  - *How?*
- The optimal solution to the 0-1 problem cannot be found with the same greedy strategy
  - Greedy strategy: take in order of dollars/pound
  - Example: 3 items weighing 10, 20, and 30 pounds, knapsack can hold 50 pounds
    - *Suppose item 2 is worth \$100. Assign values to the other items so that the greedy strategy will fail*

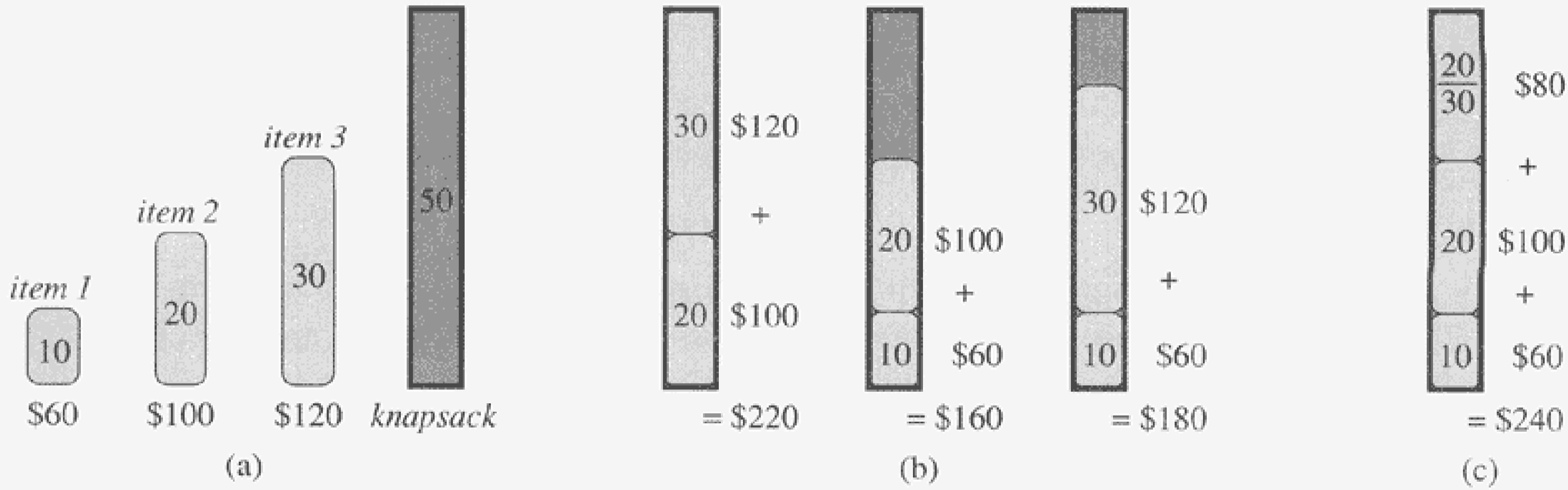


Figure 16.2 The greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest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

The Knapsack Problem: Greedy Vs. Dynamic

- The fractional problem can be solved greedily
- The 0-1 problem cannot be solved with a greedy approach
  - As you have seen, however, it can be solved with dynamic programming

0-1 Knapsack Algorithm

0-1 Knapsack Algorithm

for w = 0 to W

0-1 Knapsack Algorithm

```
for w = 0 to W
    B[0,w] = 0
```

0-1 Knapsack Algorithm

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
```

0-1 Knapsack Algorithm

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
    B[i,0] = 0
```

0-1 Knapsack Algorithm

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
    B[i,0] = 0
    for w = 0 to W
```

0-1 Knapsack Algorithm

```
for w = 0 to W
   B[0,w] = 0
for i = 0 to n
   B[i,0] = 0
   for w = 0 to W
        if wi <= w // item i can be part of the solution
```

0-1 Knapsack Algorithm

```
for w = 0 to W
   B[0,w] = 0
for i = 0 to n
   B[i,0] = 0
   for w = 0 to W
         if w_i <= w // item i can be part of the solution
                  if b_i + B[i-1,w-w_i] > B[i-1,w]
```

0-1 Knapsack Algorithm

for w = 0 to W

  B[0,w] = 0

for i = 0 to n

  B[i,0] = 0

  for w = 0 to W

    if w_i <= w // item i can be part of the solution

      if b_i + B[i-1,w-w_i] > B[i-1,w]

        B[i,w] = b_i + B[i-1,w- w_i]

0-1 Knapsack Algorithm

for w = 0 to W

B[0,w] = 0

for i = 0 to n

B[i,0] = 0

for w = 0 to W

if w_i <= w // item i can be part of the solution

if b_i + B[i-1,w-w_i] > B[i-1,w]

B[i,w] = b_i + B[i-1,w- w_i]

else

0-1 Knapsack Algorithm

for w = 0 to W

  $B[0,w] = 0$

for i = 0 to n

  $B[i,0] = 0$

  for w = 0 to W

    if w_i <= w // item i can be part of the solution

      if $b_i + B[i-1,w-w_i] > B[i-1,w]$

        $B[i,w] = b_i + B[i-1,w- w_i]$

      else

        $B[i,w] = B[i-1,w]$

0-1 Knapsack Algorithm

```
for w = 0 to W
   B[0,w] = 0
for i = 0 to n
   B[i,0] = 0
   for w = 0 to W
        if w_i <= w // item i can be part of the solution
              if b_i + B[i-1,w-w_i] > B[i-1,w]
                    B[i,w] = b_i + B[i-1,w- w_i]
              else
                    B[i,w] = B[i-1,w]
        else B[i,w] = B[i-1,w]  // w_i > w
```

Running time

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
    B[i,0] = 0
    for w = 0 to W
        < the rest of the code >
```

Running time

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
    B[i,0] = 0
    for w = 0 to W
        < the rest of the code >
```

What is the running time of this algorithm?

Running time

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
    B[i,0] = 0
    for w = 0 to W
        < the rest of the code >
```

What is the running time of this algorithm?

O(n*W)

Running time

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
    B[i,0] = 0
    for w = 0 to W
        < the rest of the code >
```

O(W)

What is the running time of this algorithm?

O(n*W)

Running time

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
    B[i,0] = 0
    for w = 0 to W
        < the rest of the code >
```

O(W)

O(W)

What is the running time of this algorithm?

O(n*W)

Running time

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
    B[i,0] = 0
    for w = 0 to W
        < the rest of the code >
```

O(W)

Repeat *n* times

O(W)

What is the running time of this algorithm?

O(n*W)

Running time

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
    B[i,0] = 0
    for w = 0 to W
        < the rest of the code >
```

O(W)

Repeat *n* times

O(W)

What is the running time of this algorithm?

O(n*W)

Remember that the brute-force algorithm takes $O(2^n)$

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Example (2)

W \ i	0	1	2	3	4
0					
1					
2					
3					
4					
5					

for w = 0 to W

    B[0,w] = 0

Example (2)

W \ i	0	1	2	3	4
0	0				
1					
2					
3					
4					
5					

for w = 0 to W

    B[0,w] = 0

Example (2)

W \ i	0	1	2	3	4
0	0				
1	0				
2					
3					
4					
5					

for w = 0 to W

    B[0,w] = 0

Example (2)

W \ i	0	1	2	3	4
0	0				
1	0				
2	0				
3					
4					
5					

for w = 0 to W

B[0,w] = 0

Example (2)

W \ i	0	1	2	3	4
0	0				
1	0				
2	0				
3	0				
4					
5					

for w = 0 to W
    B[0,w] = 0

Example (2)

W \ i	0	1	2	3	4
0	0				
1	0				
2	0				
3	0				
4	0				
5					

for w = 0 to W

    B[0,w] = 0

Example (2)

W \ i	0	1	2	3	4
0	0				
1	0				
2	0				
3	0				
4	0				
5	0				

for w = 0 to W

B[0,w] = 0

Example (3)

W \ i	0	1	2	3	4
0	0				
1	0				
2	0				
3	0				
4	0				
5	0				

```
for i = 0 to n
        B[i,0] = 0
```

Example (3)

W \ i	0	1	2	3	4
0	0	0			
1	0				
2	0				
3	0				
4	0				
5	0				

for i = 0 to n

    B[i,0] = 0

Example (3)

W \ i	0	1	2	3	4
0	0	0	0		
1	0				
2	0				
3	0				
4	0				
5	0				

for i = 0 to n

B[i,0] = 0

Example (3)

W \ i	0	1	2	3	4
0	0	0	0	0	
1	0				
2	0				
3	0				
4	0				
5	0				

for i = 0 to n

    B[i,0] = 0

Example (3)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				
3	0				
4	0				
5	0				

for i = 0 to n

    B[i,0] = 0

Example (4)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				
3	0				
4	0				
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i=-1$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (4)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				
3	0				
4	0				
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i=-1$

if w_i <= w // item i can be part of the solution

    if $b_i + B[i-1,w-w_i] > B[i-1,w]$

        $B[i,w] = b_i + B[i-1,w-w_i]$

    else

        B[i,w] = B[i-1,w]

else **B[i,w] = B[i-1,w]** // w_i > w

Example (4)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0 →	0			
2	0				
3	0				
4	0				
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i=-1$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (5)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0				
3	0				
4	0				
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i=0$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (5)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0				
3	0				
4	0				
5	0				

i=1

b_i=3

w_i=2

w=2

w-w_i =0

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (5)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	**3**			
3	0				
4	0				
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i=0$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (6)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0				
4	0				
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

if $w_i <= w$ // item i can be part of the solution

  if $b_i + B[i-1,w-w_i] > B[i-1,w]$

    $B[i,w] = b_i + B[i-1,w- w_i]$

  else

    $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Example (6)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0				
4	0				
5	0				

i=1

b_i=3

w_i=2

w=3

w-w_i=1

if w_i <= w // item i can be part of the solution

  if b_i + B[i-1,w-w_i] > B[i-1,w]

    B[i,w] = b_i + B[i-1,w- w_i]

  else

    B[i,w] = B[i-1,w]

else B[i,w] = B[i-1,w] // w_i > w

Example (6)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	**3**			
4	0				
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (7)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0				
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

if $w_i <= w$ // item i can be part of the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Example (7)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0				
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w
```

Example (7)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	**3**			
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

if $w_i <= w$ // item i can be part of the solution

  if $b_i + B[i-1,w-w_i] > B[i-1,w]$

    $B[i,w] = b_i + B[i-1,w-w_i]$

  else

    $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Example (8)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	3			
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=2$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (8)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	3			
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=2$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (8)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	3			
5	0	**3**			

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=2$

```
if wi <= w // item i can be part of the solution
    if bi + B[i-1,w-wi] > B[i-1,w]
        B[i,w] = bi + B[i-1,w- wi]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // wi > w
```

Example (9)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	3			
5	0	3			

i=2

b_i=4

w_i=3

w=1

w-w_i=-2

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (9)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	→		
2	0	3			
3	0	3			
4	0	3			
5	0	3			

i=2

b_i=4

w_i=3

w=1

$w-w_i$=-2

if w_i <= w // item i can be part of the solution
if b_i + B[i-1,w-w_i] > B[i-1,w]
B[i,w] = b_i + B[i-1,w- w_i]
else
B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

Example (9)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0 →	**0**		
2	0	3			
3	0	3			
4	0	3			
5	0	3			

i=2

b_i=4

w_i=3

w=1

w-w_i=-2

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (10)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3			
3	0	3			
4	0	3			
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (10)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3 →			
3	0	3			
4	0	3			
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (10)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3 →	**3**		
3	0	3			
4	0	3			
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (11)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3			
4	0	3			
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (11)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3			
4	0	3			
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (11)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	**4**		
4	0	3			
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

```
  if wi <= w // item i can be part of the solution
      if bi + B[i-1,w-wi] > B[i-1,w]
          B[i,w] = bi + B[i-1,w- wi]
      else
          B[i,w] = B[i-1,w]
  else B[i,w] = B[i-1,w]  // wi > w
```

Example (12)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3			
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w-w_i=1$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (12)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3			
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w-w_i=1$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (12)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	**4**		
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w-w_i=1$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (13)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	4		
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w-w_i=2$

if $w_i <= w$ // item i can be part of the solution

    if $b_i + B[i-1,w-w_i] > B[i-1,w]$

        $B[i,w] = b_i + B[i-1,w-w_i]$

    else

        $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Example (13)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	4		
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w-w_i=2$

if w_i <= w // item i can be part of the solution

    if b_i + B[i-1,w-w_i] > B[i-1,w]

        B[i,w] = b_i + B[i-1,w- w_i]

    else

        B[i,w] = B[i-1,w]

else B[i,w] = B[i-1,w] // w_i > w

Example (13)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	4		
5	0	3	7		

$i=2$
$b_i=4$
$w_i=3$
$w=5$
$w-w_i=2$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (14)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	4		
5	0	3	7		

i=3

b_i=5

w_i=4

w=1..3

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (14)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0 →		
2	0	3	3		
3	0	3	4		
4	0	3	4		
5	0	3	7		

i=3

b_i=5

w_i=4

w=1..3

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (14)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0 →	**0**	
2	0	3	3		
3	0	3	4		
4	0	3	4		
5	0	3	7		

i=3

b_i=5

w_i=4

w=1..3

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (14)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0 →	**0**	
2	0	3	3 →		
3	0	3	4		
4	0	3	4		
5	0	3	7		

$i=3$

$b_i=5$

$w_i=4$

$w=1..3$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (14)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	**0**	
2	0	3	3	**3**	
3	0	3	4		
4	0	3	4		
5	0	3	7		

$i=3$

$b_i=5$

$w_i=4$

$w=1..3$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (14)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0 →	**0**	
2	0	3	3 →	**3**	
3	0	3	4 →		
4	0	3	4		
5	0	3	7		

$i=3$

$b_i=5$

$w_i=4$

$w=1..3$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (14)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0 →	**0**	
2	0	3	3 →	**3**	
3	0	3	4 →	**4**	
4	0	3	4		
5	0	3	7		

$i=3$

$b_i=5$

$w_i=4$

$w=1..3$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (15)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4		
5	0	3	7		

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w- w_i=0$

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (15)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4		
5	0	3	7		

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w- w_i=0$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (15)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	**5**	
5	0	3	7		

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w-w_i=0$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (15)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7		

$i=3$

$b_i=5$

$w_i=4$

$w=5$

$w- w_i=1$

if $w_i <= w$ // item i can be part of the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // $w_i > w$

Example (15)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7 →		

$i=3$

$b_i=5$

$w_i=4$

$w=5$

$w-w_i=1$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (15)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7	→ 7	

$i=3$

$b_i=5$

$w_i=4$

$w=5$

$w- w_i=1$

```
if wi <= w // item i can be part of the solution
    if bi + B[i-1,w-wi] > B[i-1,w]
        B[i,w] = bi + B[i-1,w- wi]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // wi > w
```

Example (16)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7	7	

$i=3$
$b_i=5$
$w_i=4$
$w=1..4$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (16)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0 →	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7	7	

$i=3$

$b_i=5$

$w_i=4$

$w=1..4$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (16)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0 →	**0**
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7	7	

$i=3$

$b_i=5$

$w_i=4$

$w=1..4$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (16)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0 →	**0**
2	0	3	3	3 →	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7	7	

$i=3$

$b_i=5$

$w_i=4$

$w=1..4$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (16)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	**0**
2	0	3	3	3	**3**
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7	7	

i=3

b_i=5

w_i=4

w=1..4

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (16)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	**0**
2	0	3	3	3	**3**
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7	7	

$i=3$

$b_i=5$

$w_i=4$

$w=1..4$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (16)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0 →	**0**
2	0	3	3	3 →	**3**
3	0	3	4	4 →	**4**
4	0	3	4	5	
5	0	3	7	7	

$i=3$

$b_i=5$

$w_i=4$

$w=1..4$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (16)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0 →	**0**
2	0	3	3	3 →	**3**
3	0	3	4	4 →	**4**
4	0	3	4	5 →	
5	0	3	7	7	

$i=3$

$b_i=5$

$w_i=4$

$w=1..4$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (16)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0 →	**0**
2	0	3	3	3 →	**3**
3	0	3	4	4 →	**4**
4	0	3	4	5 →	**5**
5	0	3	7	7	

$i=3$
$b_i=5$
$w_i=4$
$w=1..4$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (17)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	3	3	3	3
3	0	3	4	4	4
4	0	3	4	5	5
5	0	3	7	7	

$i=3$

$b_i=5$

$w_i=4$

$w=5$

if $w_i \le w$ // item i can be part of the solution

if $b_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = b_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Example (17)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	3	3	3	3
3	0	3	4	4	4
4	0	3	4	5	5
5	0	3	7	7 →	

$i=3$
$b_i=5$
$w_i=4$
$w=5$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (17)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	3	3	3	3
3	0	3	4	4	4
4	0	3	4	5	5
5	0	3	7	7 →	**7**

$i=3$

$b_i=5$

$w_i=4$

$w=5$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Comments

- This algorithm only finds the max possible value that can be carried in the knapsack
- To know the items that make this maximum value, an addition to this algorithm is necessary
- Please see LCS algorithm from the previous lecture for the example how to extract this data from the table we built

Change-Making Problem

Change-Making Problem

Finding the number of ways of making changes for a particular amount of cents, *n, using a given* set of denominations C={c1…cd} (e.g, the US coin system: {1, 5, 10, 25, 50, 100})

– An example: n = 4,C = {1,2,3}, solutions: {1,1,1,1}, {1,1,2},{2,2},{1,3}.

Minimizing the number of coins returned for a particular quantity of change **(available coins {1, 5, 10, 25})**

– 30 Cents (solution: 25 + 5, two coins)

– 67 Cents ?

17 cents given denominations = {1, 2, 3, 4}?

Find the Fewest Coins: Casher's algorithm

- Given 30 cents, and coins {1, 5, 10, 25}
- Here is what a casher will do: always go with coins of highest value first
  - Choose the coin with highest value 25
    - 1 quarter
  - Now we have 5 cents left
    - 1 nickel

The solution is: 2 (one quarter + one nickel)

Greedy Algorithm Does not Always Give Optimal Solution to Coin Change Problem

- Coins = {1, 3, 4, 5}
- 7 cents = ?

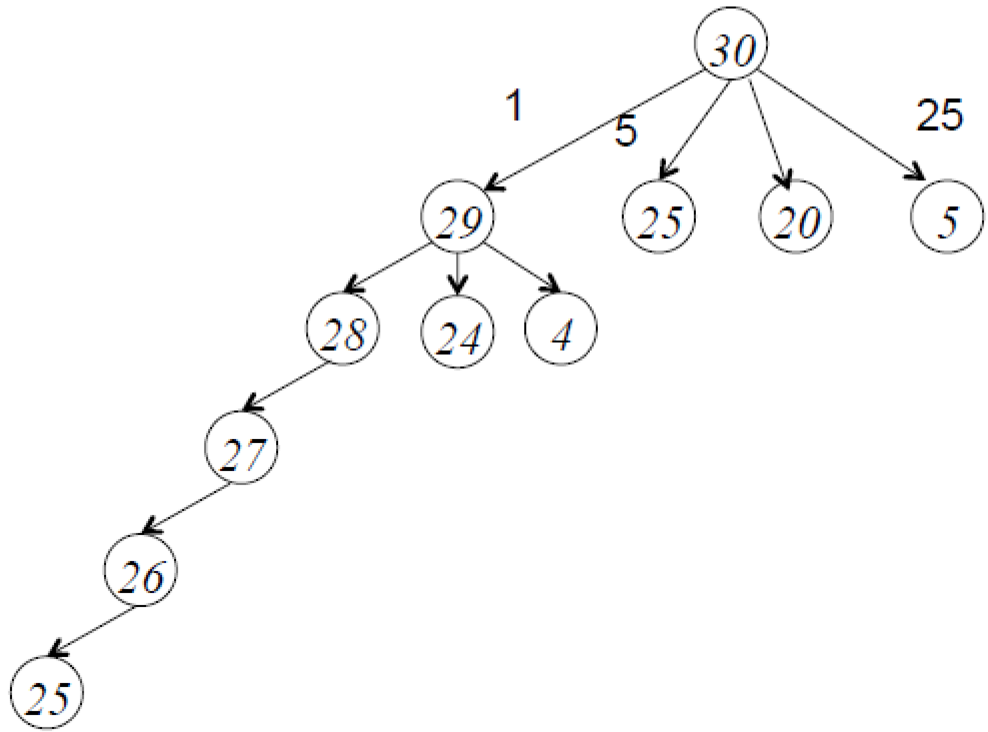
- Greedy solution:
  – 3 coins: one 5 + two 1

- Optimal solution:
  – 2 coins: one 3 + one 4

Find the Fewest Coins: Divide and Conquer

- 30 cents, given coins {1, 5, 10, 25, 50}, we need to calculate MinChange(30)
- Choose the smallest of the following:
  - 1 + MinChange(29) #give a penny
  - 1 + MinChange(25) #give a nickel
  - 1 + MinChange(10) #give a dime
  - 1 + MinChange(5) #give a quarter
- Do not know MinChange(29), MinChange(25), MinChange(10), MinChange(5)?

Recursive Algorithm Is Not Efficient



- It recalculates the optimal coin combination for a given amount of money repeatedly

Dynamic Programming

Problem: A country has coins with denominations

$$1 = d_1 < d_2 < \cdots < d_k.$$

You want to make change for n cents, using the smallest number of coins.

Example: U.S. coins

$$d_1 = 1 \quad d_2 = 5 \quad d_3 = 10 \quad d_4 = 25$$

Change for 37 cents – 1 quarter, 1 dime, 2 pennies.

What is the algorithm?

Change in another system

Suppose

$$d_1 = 1 \quad d_2 = 4 \quad d_3 = 5 \quad d_4 = 10$$

- Change for **7** cents – **5,1,1**
- Change for **8** cents – **4,4**

What can we do?

Dynamic Programming

Change in another system

Suppose

$$d_1 = 1 \;\; d_2 = 4 \;\; d_3 = 5 \;\; d_4 = 10$$

- Change for 7 cents – 5,1,1
- Change for 8 cents – 4,4

What can we do?

The answer is counterintuitive. To make change for n cents, we are going to figure out how to make change for every value $x < n$ first. We then build up the solution out of the solution for smaller values.

Dynamic Programming

Solution

We will only concentrate on computing the number of coins. We will later recreate the solution.

- Let $C[p]$ be the minimum number of coins needed to make change for p cents.
- Let x be the value of the first coin used in the optimal solution.
- Then $C[p] = 1 + C[p - x]$.

Problem: We don't know x.

Dynamic Programming

Solution

We will only concentrate on computing the number of coins. We will later recreate the solution.

- Let $C[p]$ be the minimum number of coins needed to make change for p cents.
- Let x be the value of the first coin used in the optimal solution.
- Then $C[p] = 1 + C[p - x]$.

Problem: We don't know x.

Answer: We will try all possible x and take the minimum.

$$C[p] = \begin{cases} \min_{i: d_i \leq p} \{C[p - d_i] + 1\} & \text{if } p > 0 \\ 0 & \text{if } p = 0 \end{cases}$$

Dynamic Programming Algorithm

```
DP-Change(n)
C[< 0] = ∞
C[0] = 0
for p = 2 to n
        do min = ∞
            for i = 1 to k
                do if (p ≥ d_i)
                    then if (C[p − d_i]) + 1 < min)
                        then min = C[p − d_i] + 1
                            coin = i

            C[p] = min
            S[p] = coin
```

Running Time: $O(nk)$

Greedy algorithm outputs optimal solutions for coin values 10, 5, 1

Proof:

Let N be the amount to be paid. Let the optimal solution be P=A*10 + B*5 + C. Clearly B≤1 (otherwise we can decrease B by 2 and increase A by 1, improving the solution). Similarly, C≤4.

Let the solution given by GreedyCoinChange be P=a*10 + b*5 + c. Clearly b≤1 (otherwise the algorithm would output 10 instead of 5). Similarly c≤4.

From 0≤ C≤4 and P=(2A+B)*5+C we have C=P mod 5.
Similarly c=P mod 5, and hence c=C. Let Q=(P-C)/5.
From 0≤ B≤ 1 and Q = 2A + B we have B=Q mod 2.
Similarly b=Q mod 2, and hence b=B.
Thus a=A, b=B, c=C, i.e., the solution given by GreedyCoinChange is optimal.

Homework

CLRS 16.1-2

CLRS 16.1-5

CLRS 16.2-7

Exercise in Class

Prove that Greedy algorithm outputs optimal solution for coin values 18,6,3,1

练习题

礼品分组

N个礼品，每个礼品的价格不一样。现要把所有礼品分组，每组的礼品数量不超过2个，且礼品总价格不超过C (C>0)，求分组的数目最少的分法。

练习题

堆积木

老师给每个小朋友分了些积木块，但每个小朋友手上的积木都不足以堆成想要的形状。现在你手上有一些积木，你可以全部交给某个小朋友让他有足够的积木堆成形状，堆完后再收回所有的积木。你最多可以让多少小朋友堆成积木。

练习题

PK赛

工科系男生寝室A和寝室B的人数都是N，为了争夺和文艺系女生寝室的联谊权，决定举行一场扳手劲的PK赛。比赛要进行N轮，每轮由双方寝室各派出一位男生参加，但每人只能比赛一次。假设寝室B的室长知道双方学生的实力，他如何安排寝室B学生的比赛顺序才能取得最多的胜利。

Huffman Codes

Data Compression

Q. Given a text that uses 32 symbols (26 different letters, space, and some punctuation characters), how can we encode this text in bits?

Q. Some symbols (e, t, a, o, i, n) are used far more often than others. How can we use this to reduce our encoding?

Q. How do we know when the next symbol begins?

Ex. c(a) = 01 What is 0101?

c(b) = 010

c(e) = 1

Data Compression

Q. Given a text that uses 32 symbols (26 different letters, space, and some punctuation characters), how can we encode this text in bits?
A. We can encode 32 different symbols using a fixed length of 5 bits per symbol. This is called fixed length encoding.

Q. Some symbols (e, t, a, o, i, n) are used far more often than others. How can we use this to reduce our encoding?
A. Encode these characters with fewer bits, and the others with more bits.

Q. How do we know when the next symbol begins?
A. Use a separation symbol (like the pause in Morse), or make sure that there is no ambiguity by ensuring that no code is a prefix of another one.

Ex. c(a) = 01 What is 0101?
c(b) = 010
c(e) = 1

Prefix Codes

Definition. A prefix code for a set S is a function c that maps each $x \in S$ to 1s and 0s in such a way that for $x,y \in S$, $x \neq y$, $c(x)$ is not a prefix of $c(y)$.

Ex. $c(a) = 11$

$c(e) = 01$

$c(k) = 001$

$c(l) = 10$

$c(u) = 000$

Q. What is the meaning of 1001000001 ?

Suppose frequencies are known in a text of 1G:

$f_a=0.4$, $f_e=0.2$, $f_k=0.2$, $f_l=0.1$, $f_u=0.1$

Q. What is the size of the encoded text?

Prefix Codes

Definition. A prefix code for a set S is a function c that maps each $x \in S$ to 1s and 0s in such a way that for $x,y \in S$, $x \neq y$, c(x) is not a prefix of c(y).

Ex. c(a) = 11
c(e) = 01
c(k) = 001
c(l) = 10
c(u) = 000

Q. What is the meaning of 1001000001 ?

A. "leuk"

Suppose frequencies are known in a text of 1G:

fa=0.4, fe=0.2, fk=0.2, fl=0.1, fu=0.1

Q. What is the size of the encoded text?

Prefix Codes

Definition. A prefix code for a set S is a function c that maps each $x \in S$ to 1s and 0s in such a way that for $x,y \in S$, $x \neq y$, c(x) is not a prefix of c(y).

Ex. c(a) = 11
c(e) = 01
c(k) = 001
c(l) = 10
c(u) = 000

Q. What is the meaning of 1001000001 ?
A. “leuk”

Suppose frequencies are known in a text of 1G:
fa=0.4, fe=0.2, fk=0.2, fl=0.1, fu=0.1
Q. What is the size of the encoded text?
A. 2*fa + 2*fe + 3*fk + 2*fl + 4*fu = 2.4G

Optimal Prefix Codes

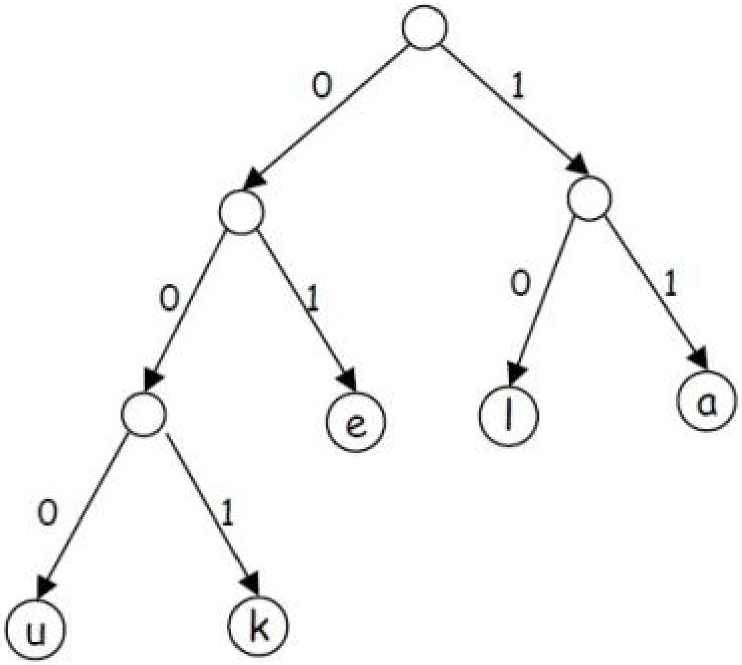
Definition. The average bits per letter of a prefix code c is the sum over all symbols of its frequency times the number of bits of its encoding:

$$ABL(c) = \sum_{x \in S} f_x \cdot |c(x)|$$

We would like to find a prefix code that is has the lowest possible average bits per letter.
Suppose we model a code in a binary tree…

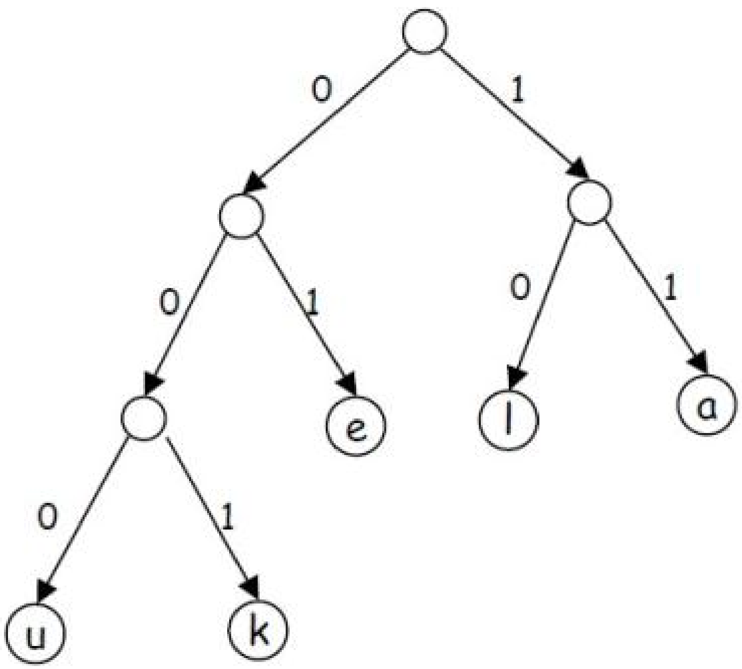
Representing Prefix Codes using Binary Trees

Ex. c(a) = 11
c(e) = 01
c(k) = 001
c(l) = 10
c(u) = 000



Representing Prefix Codes using Binary Trees

Ex. c(a) = 11
c(e) = 01
c(k) = 001
c(l) = 10
c(u) = 000



Q. How does the tree of a prefix code look?

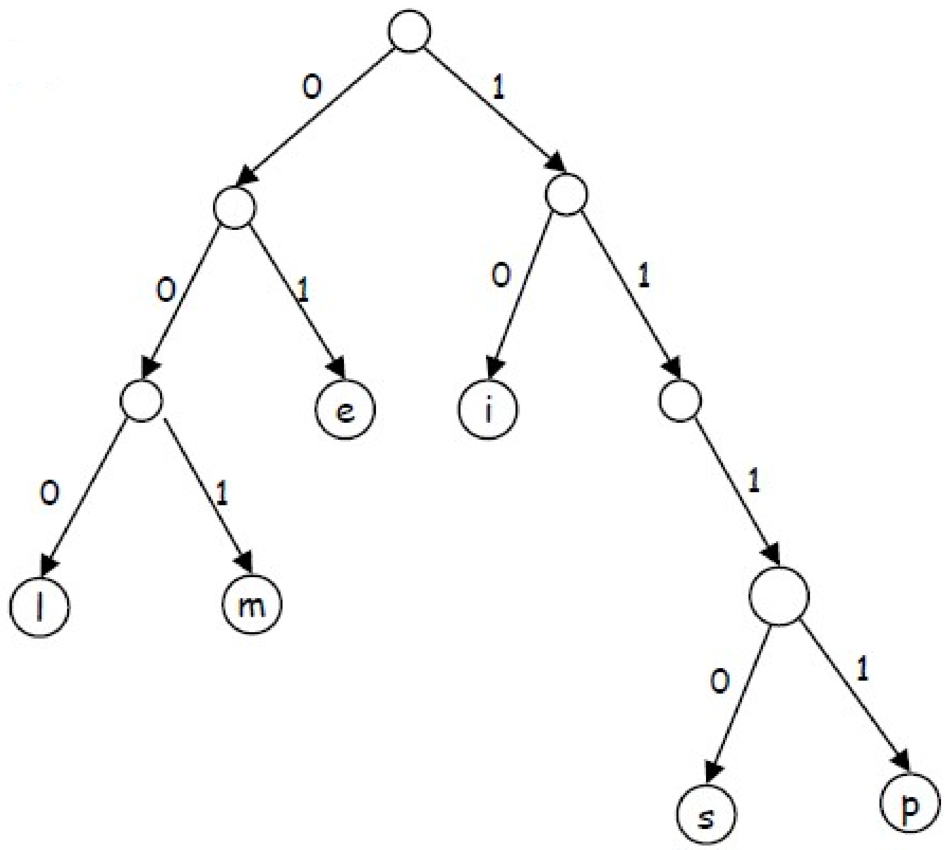
A. Only the leaves have a label.

Pf. An encoding of x is a prefix of an encoding of y if and only if the path of x is a prefix of the path of y.

Representing Prefix Codes using Binary Trees

Q. What is the meaning of 111010001111101000 ?
A. "simpel"

$$ABL(T) = \sum_{x \in S} f_x \cdot \text{depth}_T(x)$$

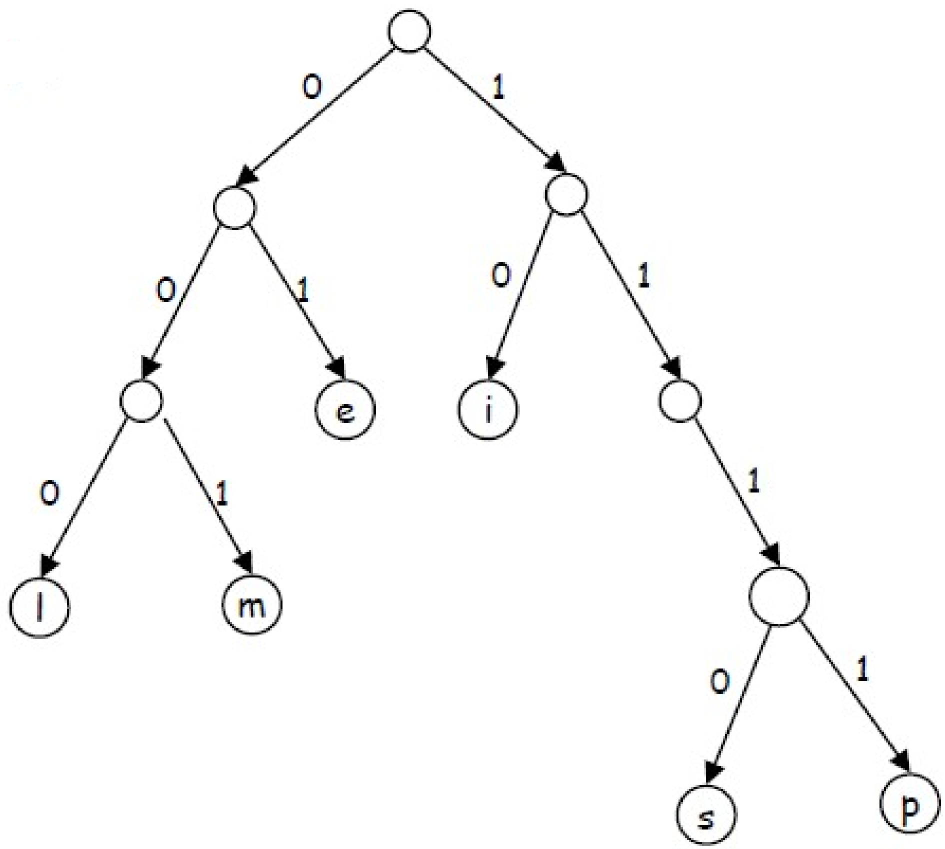


Representing Prefix Codes using Binary Trees

Q. What is the meaning of 111010001111101000 ?

A. “simpel”

$$ABL(T) = \sum_{x \in S} f_x \cdot \text{depth}_T(x)$$

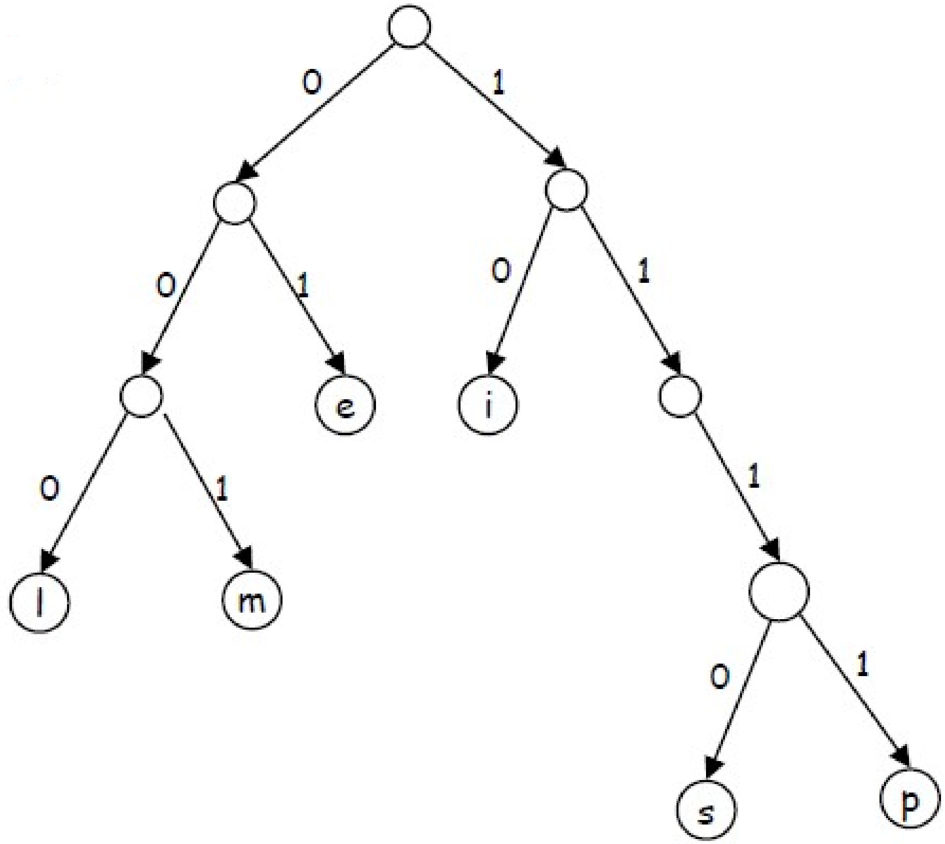


Q. How can this prefix code be made more efficient?

Representing Prefix Codes using Binary Trees

Q. What is the meaning of 111010001111101000 ?
A. "simpel"

$$ABL(T) = \sum_{x \in S} f_x \cdot \text{depth}_T(x)$$



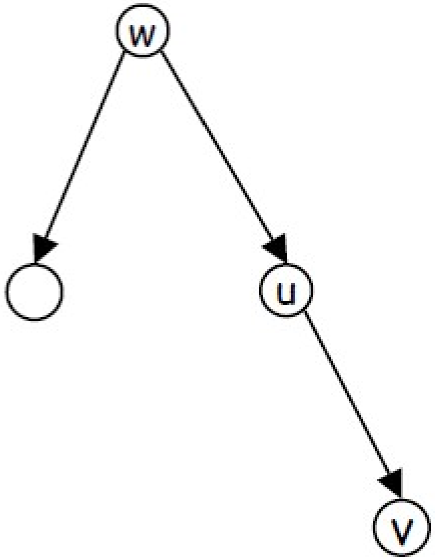
Q. How can this prefix code be made more efficient?

A. Change encoding of p and s to a shorter one.
This tree is now full.

Representing Prefix Codes using Binary Trees

Definition. A tree is full if every node that is not a leaf has two children.

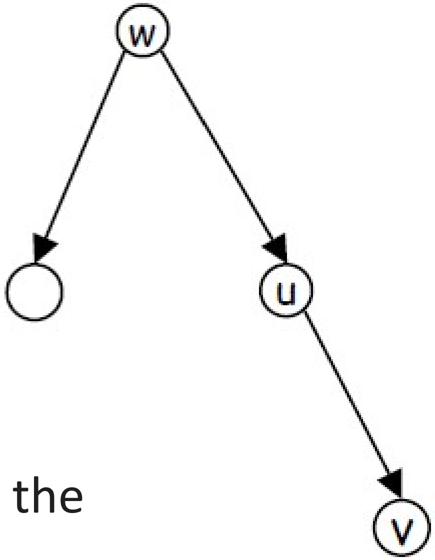
Claim. The binary tree corresponding to the optimal prefix code is full.



Representing Prefix Codes using Binary Trees

Definition. A tree is full if every node that is not a leaf has two children.

Claim. The binary tree corresponding to the optimal prefix code is full.
Pf. (by contradiction)
Suppose T is binary tree of optimal prefix code and is not full.
This means there is a node u with only one child v.

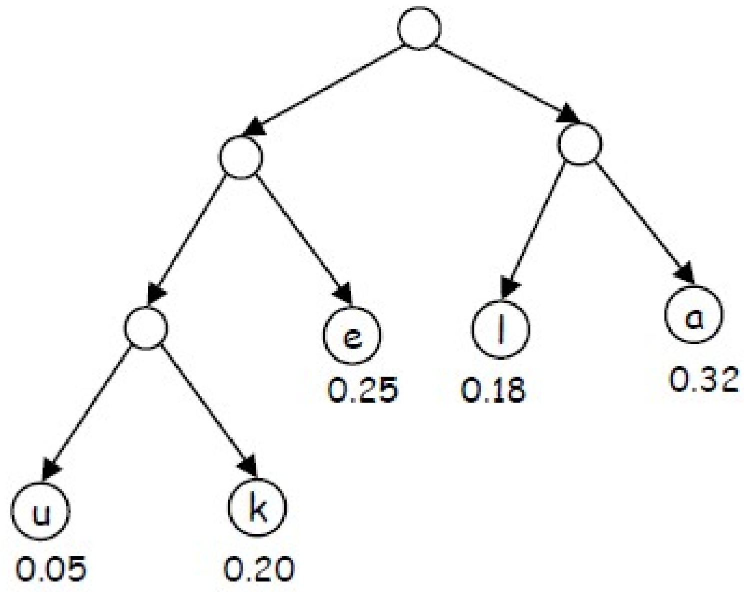


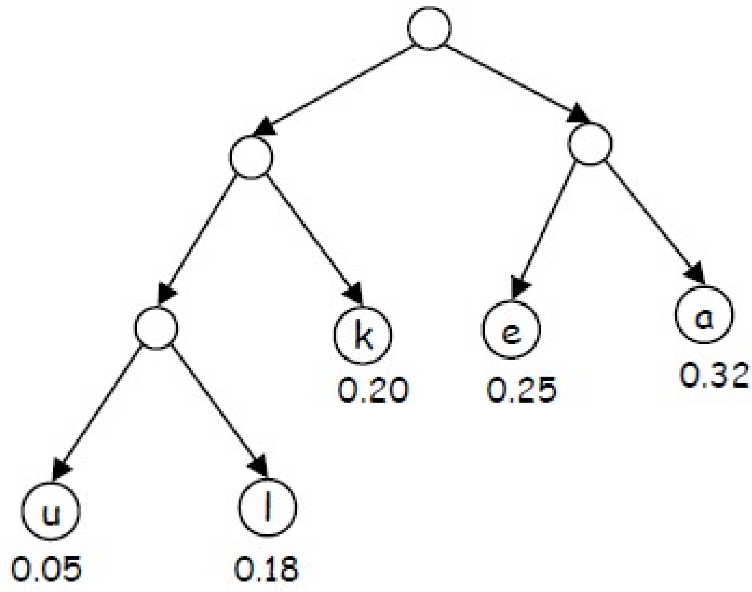
- Case 1: u is the root; delete u and use v as the root
- Case 2: u is not the root
  - let w be the parent of u
  - delete u and make v be a child of w in place of u
- In both cases the number of bits needed to encode any leaf in the subtree of v is decreased. The rest of the tree is not affected.
- Clearly this new tree T' has a smaller ABL than T. Contradiction.

Optimal Prefix Codes: False Start

Q. Where in the tree of an optimal prefix code should letters be placed with a high frequency?
A. Near the top.

Greedy template. Create tree top-down, split S into two sets S1 and S2 with (almost) equal frequencies. Recursively build tree for S1 and S2.
[Shannon-Fano, 1949] f_a=0.32, f_e=0.25, f_k=0.20, f_l=0.18, f_u=0.05





Optimal Prefix Codes: Huffman Encoding

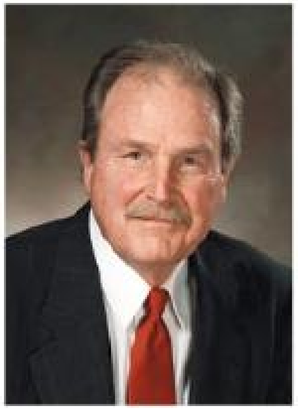
Observation. Lowest frequency items should be at the lowest level in tree of optimal prefix code.

Observation. For $n > 1$, the lowest level always contains at least two leaves.

Observation. The order in which items appear in a level does not matter.

Claim. There is an optimal prefix code with tree T^* where the two lowest-frequency letters are assigned to leaves that are siblings in T^*.

Greedy template. [Huffman, 1952] Create tree bottom-up.
Make two leaves for two lowest-frequency letters y and z.
Recursively build tree for the rest using a meta-letter for yz.



Optimal Prefix Codes: Huffman Encoding

```
Huffman(S) {
   if |S|=2 {
      return tree with root and 2 leaves
   } else {
      let y and z be lowest-frequency letters in S
      S′ = S
      remove y and z from S′
      insert new letter ω in S′ with fω=fy+fz
      T′ = Huffman(S′)
      T = add two children y and z to leaf ω from T′
      return T
   }
}
```

Q. What is the time complexity?
A. $T(n) = T(n-1) + O(n)$ so $O(n^2)$
Q. How to implement finding lowest-frequency letters efficiently?
A. Use priority queue for S: $T(n) = T(n-1) + O(\log n)$ so $O(n \log n)$

Huffman Encoding: Greedy Analysis

Claim. Huffman code for S achieves the minimum ABL of any prefix code.
Pf. by induction, based on optimality of T' (y and z removed, ω added)

Huffman Encoding: Greedy Analysis

Claim. Huffman code for S achieves the minimum ABL of any prefix code.
Pf. by induction, based on optimality of T' (y and z removed, ω added)

Claim. ABL(T')=ABL(T)-f_ω
Pf.

$$
\begin{aligned}
\mathrm{ABL}(T) &= \sum_{x\in S} f_x \cdot \mathrm{depth}_T(x) \\
&= f_y \cdot \mathrm{depth}_T(y) + f_z \cdot \mathrm{depth}_T(z) + \sum_{x\in S, x\neq y,z} f_x \cdot \mathrm{depth}_T(x) \\
&= \left(f_y + f_z\right) \cdot \left(1 + \mathrm{depth}_T(\omega)\right) + \sum_{x\in S, x\neq y,z} f_x \cdot \mathrm{depth}_T(x) \\
&= f_\omega \cdot \left(1 + \mathrm{depth}_T(\omega)\right) + \sum_{x\in S, x\neq y,z} f_x \cdot \mathrm{depth}_T(x) \\
&= f_\omega + \sum_{x\in S'} f_x \cdot \mathrm{depth}_{T'}(x) \\
&= f_\omega + \mathrm{ABL}(T')
\end{aligned}
$$

Huffman Encoding: Greedy Analysis

Claim. Huffman code for S achieves the minimum ABL of any prefix code.
Pf. (by induction)
Base: For n=2 there is no shorter code than root and two leaves.
Hypothesis: Suppose Huffman tree T' for S' of size n-1 with ω instead of y and z is optimal. (IH)
Step: (by contradiction)

- Idea of proof:

– Suppose other tree Z of size n is better.
– Delete lowest frequency items y and z from Z creating Z'
– Z' cannot be better than T' by IH.

Huffman Encoding: Greedy Analysis

Claim. Huffman code for S achieves the minimum ABL of any prefix code.

Pf. (by induction)

Base: For n=2 there is no shorter code than root and two leaves.

Hypothesis: Suppose Huffman tree T' for S' with ω instead of y and z is optimal. (IH)

Step: (by contradiction)

- Suppose Huffman tree T for S is not optimal.
- So there is some tree Z such that ABL(Z) < ABL(T).
- Then there is also a tree Z for which leaves y and z exist that are siblings and have the lowest frequency (see observation).
- Let Z' be Z with y and z deleted, and their former parent labeled ω.
- Similar T' is derived from S' in our algorithm.
- We know that ABL(Z')=ABL(Z)-fω, as well as ABL(T')=ABL(T)-fω.
- But also ABL(Z) < ABL(T), so ABL(Z') < ABL(T').
- Contradiction with IH.