

Greedy Algorithms

- Knapsack
- Coin Change
- Huffman Code
- Scheduling

Optimization Problems

- Optimization problem: a problem of finding the best solution from all feasible solutions.
- Two common techniques:
 - Greedy Algorithms
 - Dynamic Programming (global)

Elements of Greedy Strategy

- ***Greedy-choice property***: A global optimal solution can be arrived at by making locally optimal (greedy) choices
- ***Optimal substructure***: an optimal solution to the problem contains within it optimal solutions to sub-problems

Greedy Algorithms

A greedy algorithm works in phases. At each phase:

- You take the best you can get right now, without regard for future consequences
- You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

Greedy algorithms typically consist of

- A set of ***candidate solutions***
- ***Function*** that checks if the candidates are ***feasible***
- ***Selection function*** indicating at a given time which is the most **promising candidate** not yet used
- ***Objective function*** giving the value of a solution; this is the function we are trying to optimize

Analysis

- The selection function is usually based on the objective function; they may be identical. But, often there are several plausible ones.
- At every step, the procedure chooses the best candidate, without worrying about the future. It never changes its mind: once a candidate is included in the solution, it is there for good; once a candidate is excluded, it's never considered again.
- Greedy algorithms do NOT always yield optimal solutions, but for many problems they do.

Greedy vs DP

- Greedy and Dynamic Programming are methods for solving optimization problems.
- Greedy algorithms are usually more efficient than DP solutions.
- However, often you need to use dynamic programming since the optimal solution cannot be guaranteed by a greedy algorithm.
- DP provides efficient solutions for some problems for which a brute force approach would be very slow.
- To use Dynamic Programming we need only show that the principle of optimality applies to the problem.

Examples of Greedy Algorithms

- Knapsack
- Coin Change
- Data compression
 - Huffman coding
- Scheduling
 - Activity Selection
 - Task Scheduling
 - Minimizing time in system
 - Deadline scheduling
- Graph Algorithms
 - Breath First Search (shortest path 4 un-weighted graph)
 - Dijkstra's (shortest path) Algorithm
 - Minimum Spanning Trees

The 0/1 Knapsack problem

- Given a knapsack with weight $W > 0$.
- A set S of n items with weights $w_i > 0$ and benefits $b_i > 0$ for $i = 1, \dots, n$.
- $S = \{ (item_1, w_1, b_1), (item_2, w_2, b_2), \dots, (item_n, w_n, b_n) \}$
- Find a subset of the items which does not exceed the weight W of the knapsack and maximizes the benefit.

0/1 Knapsack problem

Determine a subset T of $\{ 1, 2, \dots, n \}$ that satisfies the following:

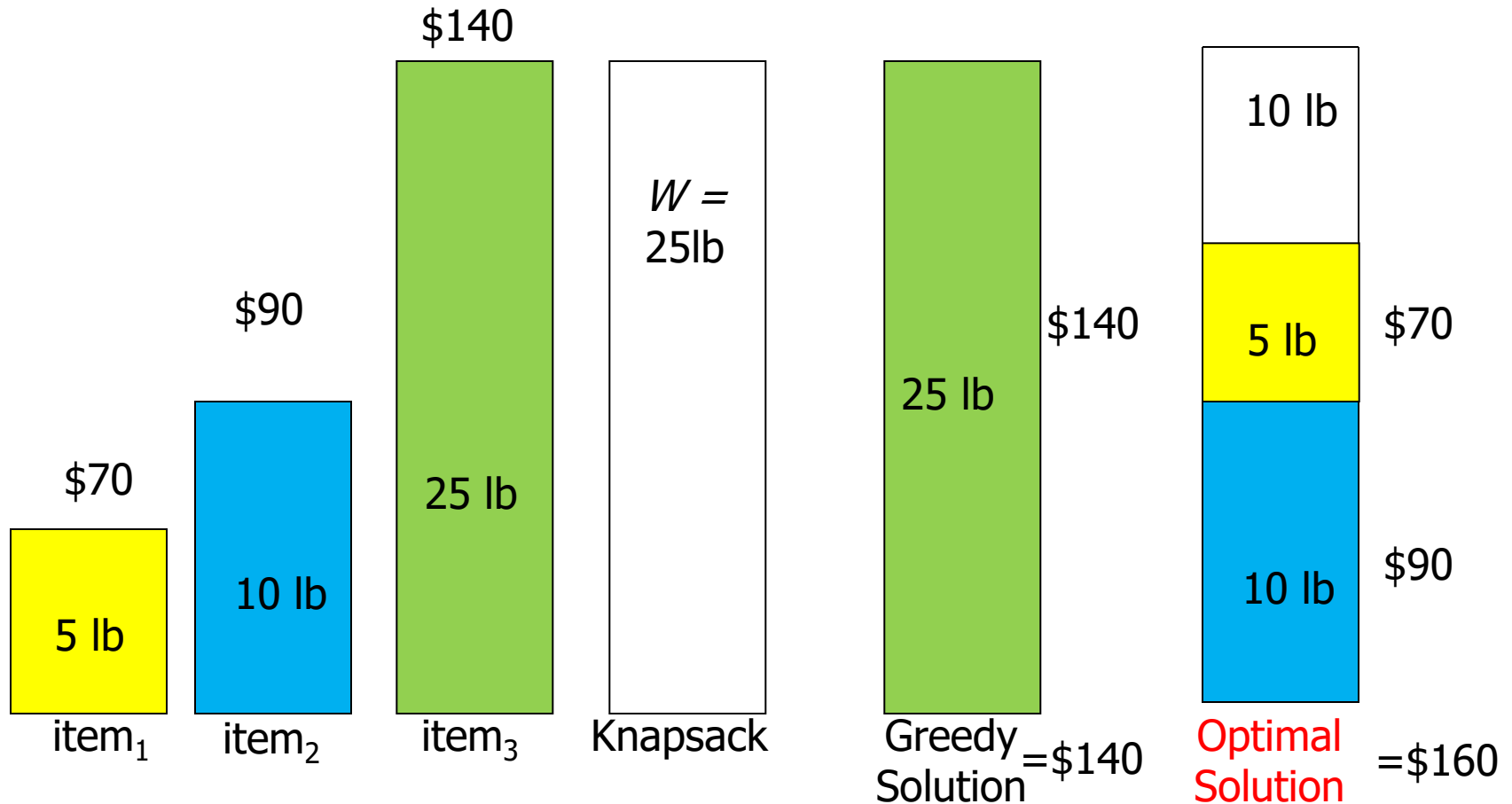
$$\max \sum_{i \in T} b_i \text{ where } \sum_{i \in T} w_i \leq W$$

In 0/1 knapsack a specific item is either selected or not

Greedy 1: Selection criteria: *Maximum beneficial* item.

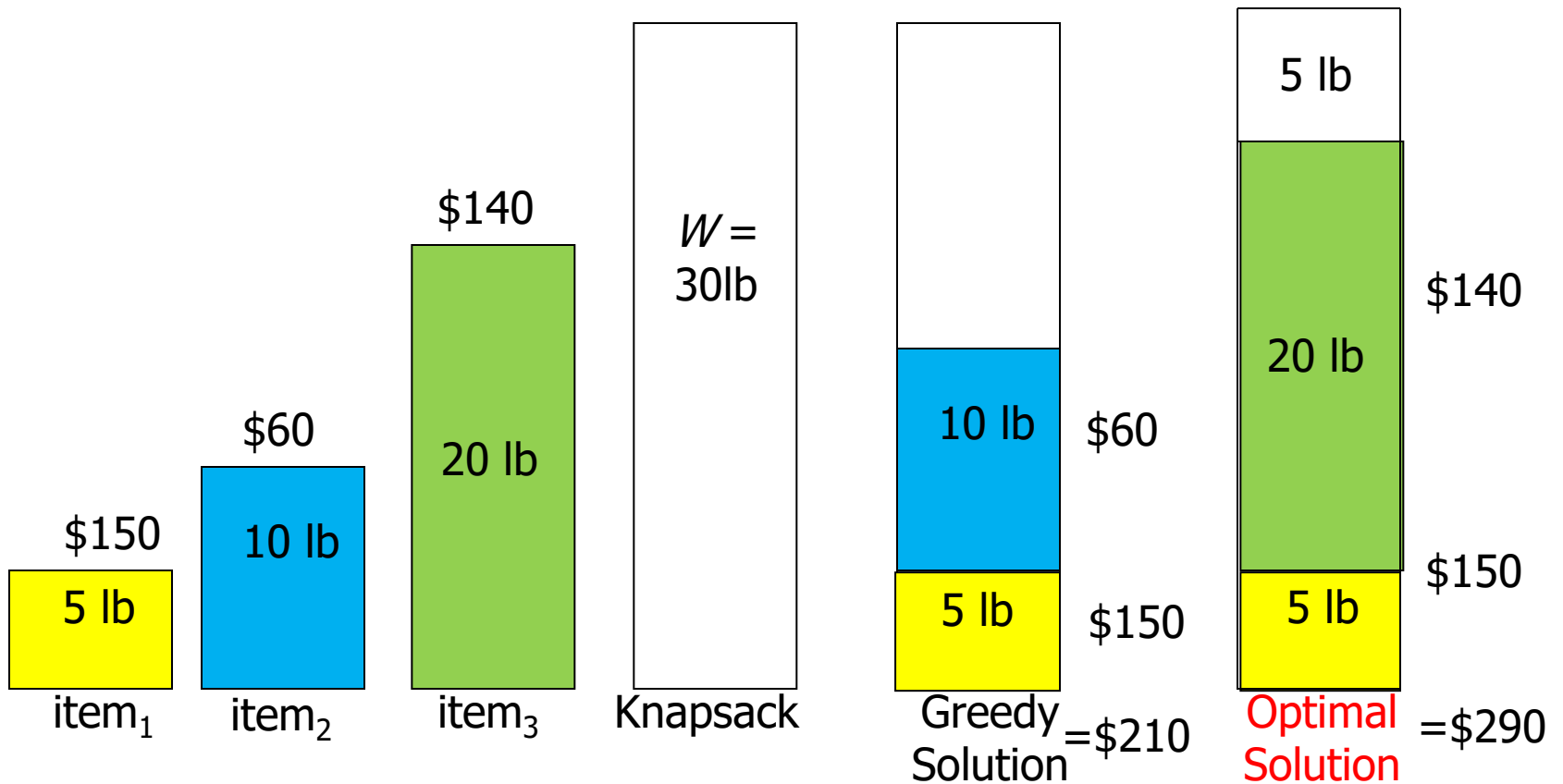
Counter Example:

$$S = \{ (item_1, 5, \$70), (item_2, 10, \$90), (item_3, 25, \$140) \}$$



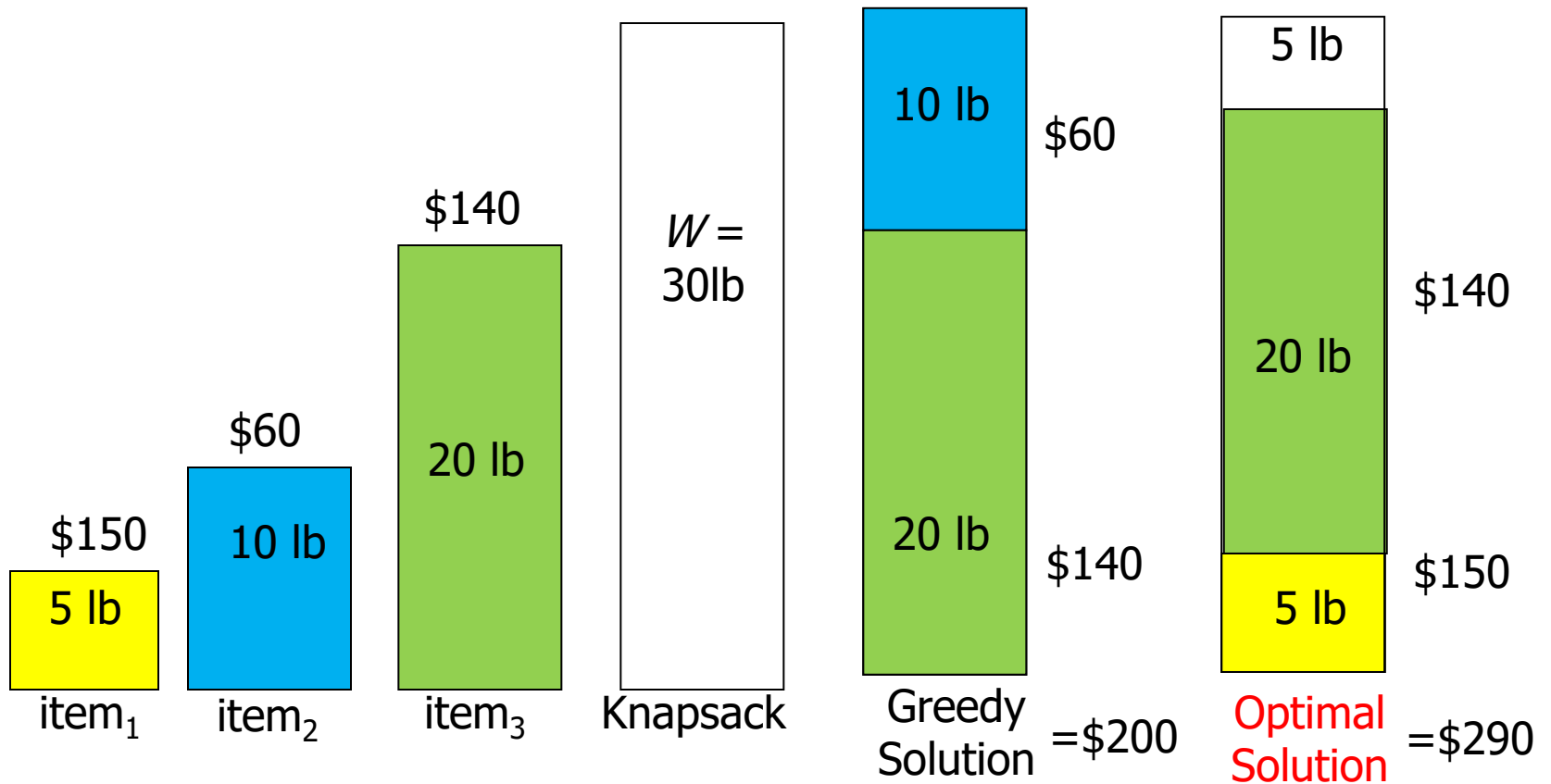
Greedy 2: Selection criteria: *Minimum weight* item
Counter Example:

$$S = \{ (item_1, 5, \$150), (item_2, 10, \$60), (item_3, 20, \$140) \}$$



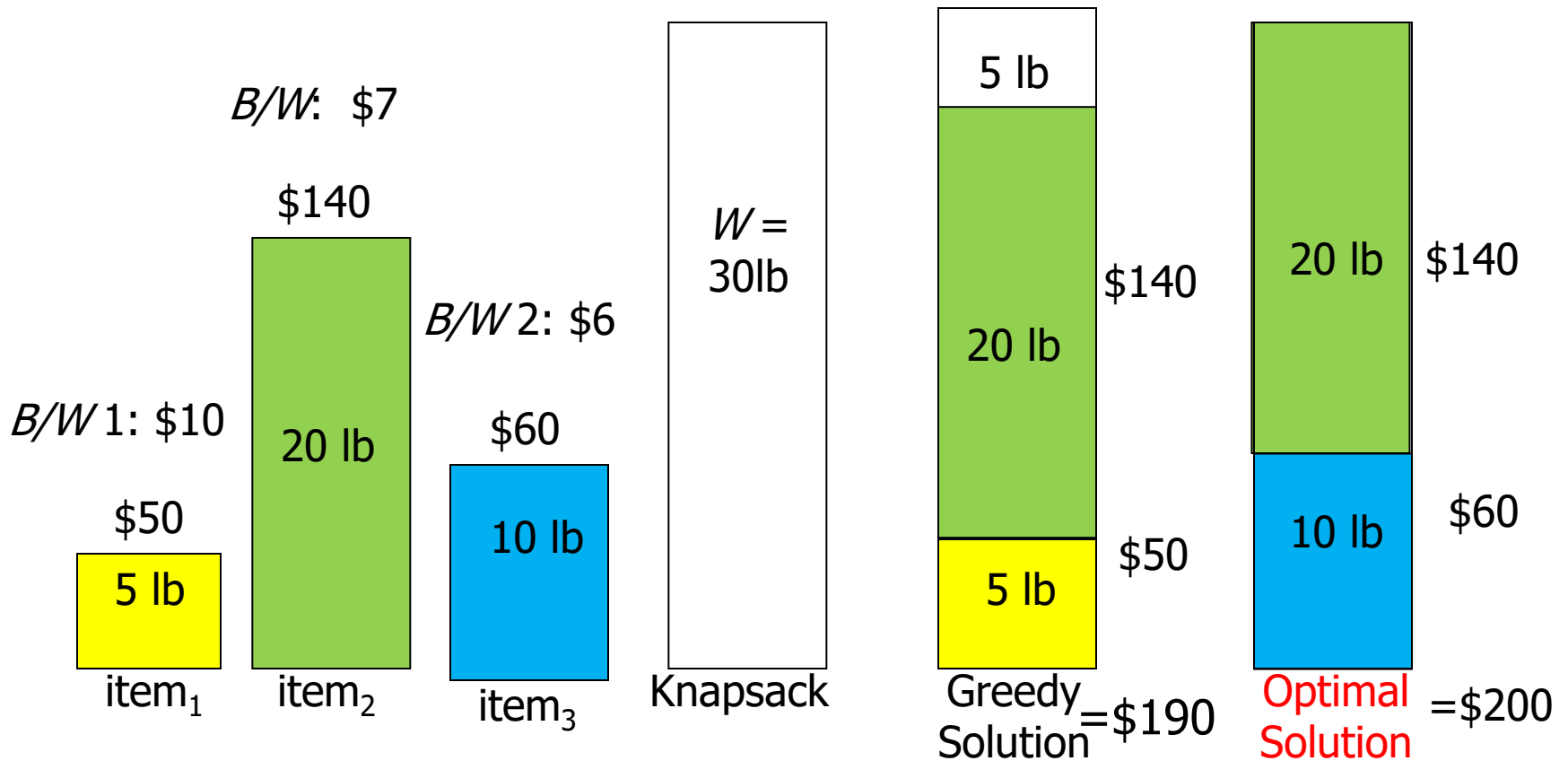
Greedy 3: Selection criteria: *Maximum weight* item
Counter Example:

$$S = \{ (item_1, 5, \$150), (item_2, 10, \$60), (item_3, 20, \$140) \}$$



Greedy 4: Selection criteria: *Maximum benefit per unit item* Counter Example

$$S = \{ (item_1, 5, \$50), (item_2, 20, \$140), (item_3, 10, \$60), \}$$



0-1 Knapsack: Greedy Does Not Work

- Can use DP but it is pseudo-polynomial $O(Wn)$
- What DP doesn't work?

Situation where dynamic programming does not work

What if our problem can't be described with integers?

$$W = 9$$

$$w_A = 2 \quad b_A = \$40$$

$$w_B = \pi \quad b_B = \$50$$

$$w_C = 1.98 \quad b_C = \$100$$

$$w_D = 5 \quad b_D = \$95$$

$$w_E = 3 \quad b_E = \$30$$

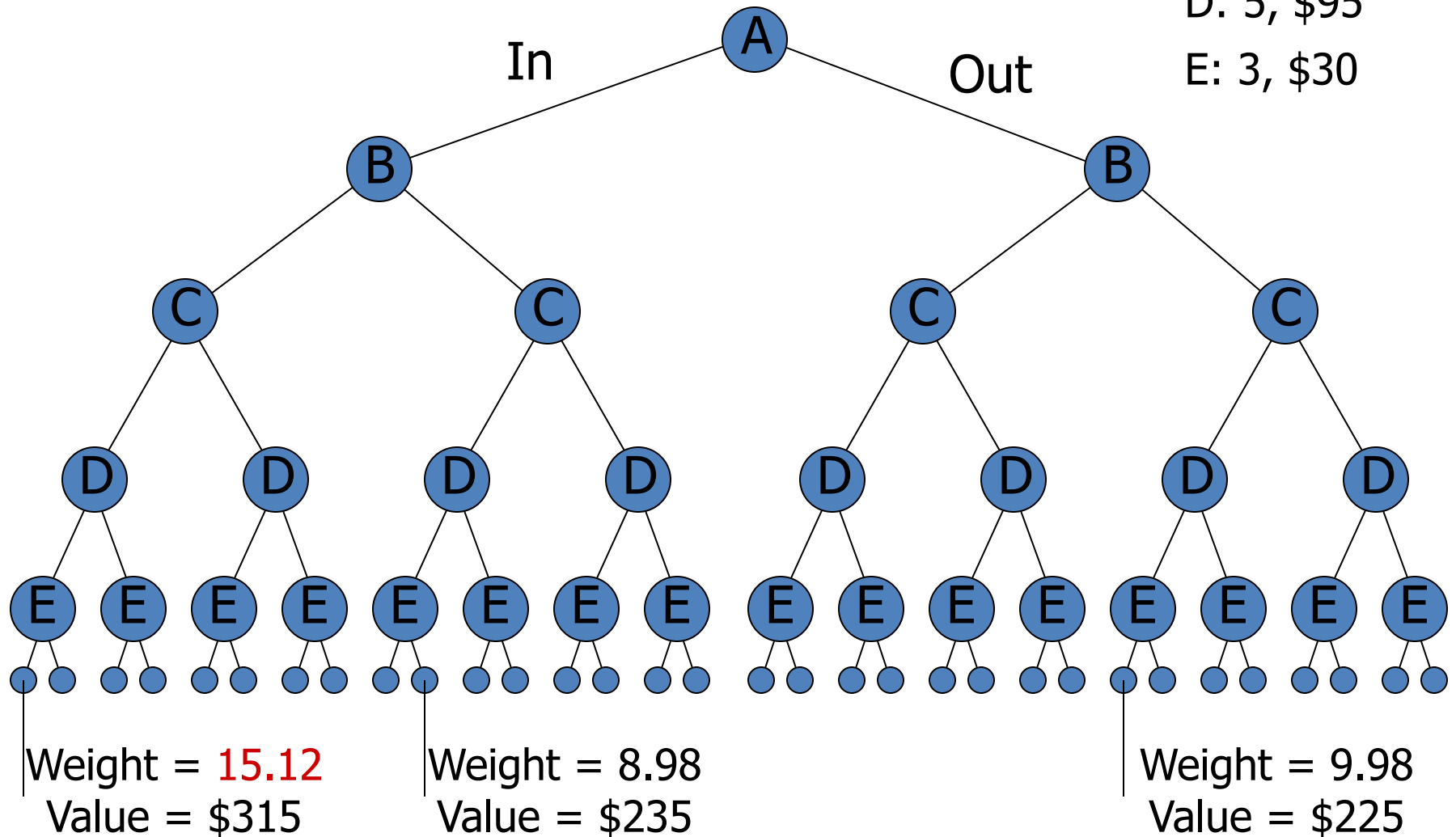
We have to resort to brute force....

Brute Force

- Generate all possible solutions
 - With n items, there are 2^n solutions to be generated
 - Check each to see if they satisfy the constraint
 - Save maximum solution that satisfies constraint
- Can be represented as a tree

Brute Force: Branching

A: 2, \$40
B: π , \$50
C: 1.98, \$100
D: 5, \$95
E: 3, \$30



Backtracking

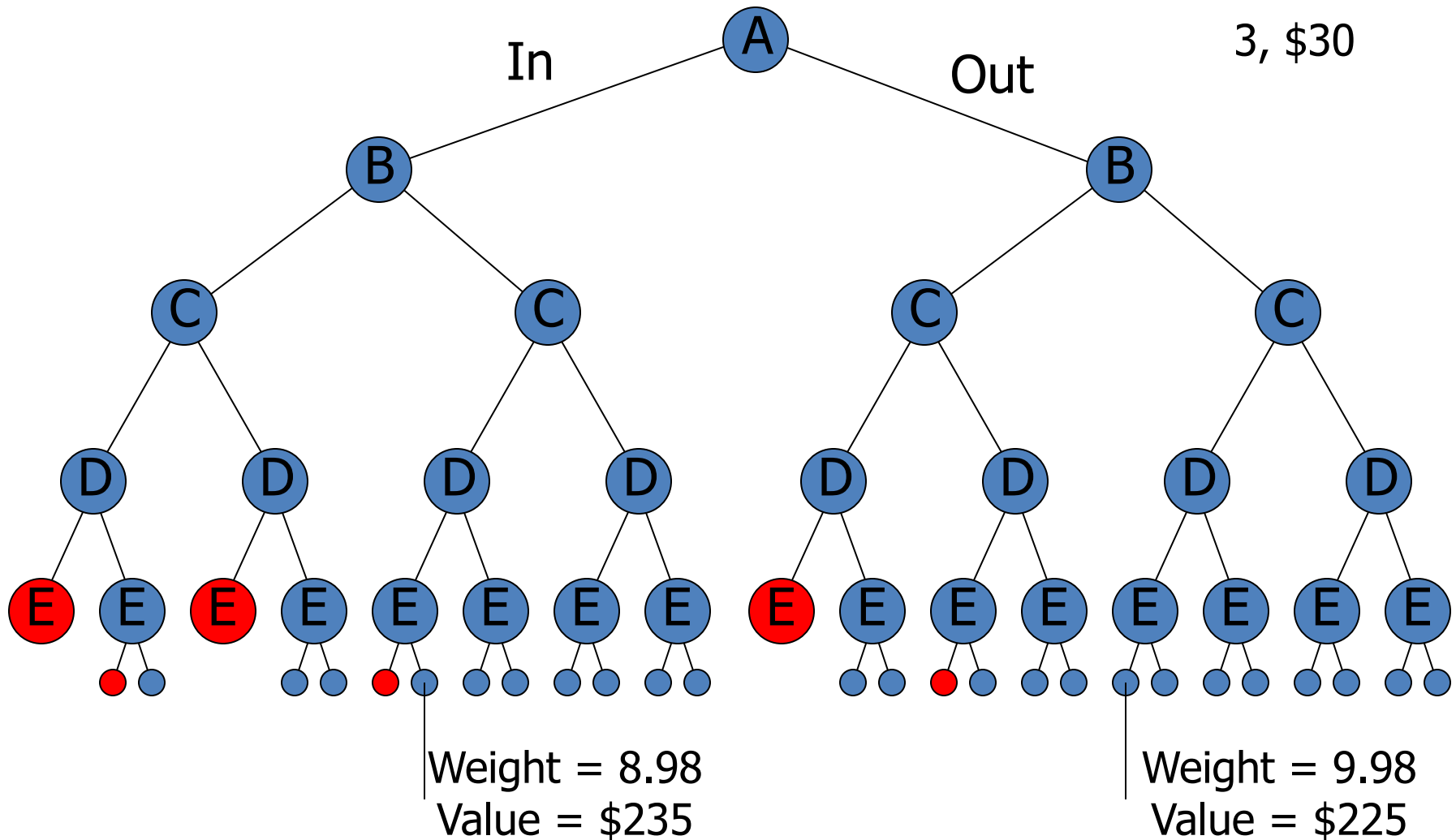
2, \$40

π , \$50

1.98, \$100

5, \$95

3, \$30



Fractional Knapsack

Let k be the index of the last item included in the knapsack. We may be able to include the whole or only a fraction of item k

$$\text{Without item } k \text{ totweight} = \sum_{i=1}^{k-1} w_i$$

$$FWK = \sum_{i=1}^{k-1} b_i + \min\{(\mathbf{W} - \text{totweight}), w_k\} \times (b_k / w_k)$$

$\min\{(\mathbf{W} - \text{totweight}), w_k\}$, means that we either take the whole of item k when the knapsack can include the item without violating the constraint, or we fill the knapsack by a fraction of item

A Greedy Algorithm for Fractional Knapsack

In this problem a fraction of any item may be chosen

The greedy algorithm uses the *maximum benefit per unit* selection criteria

1. Calculate $v_i = b_i / w_i$ for $1 \leq i \leq n$ $\Theta(n)$
2. Sort items in decreasing b_i / w_i . $\Theta(n \lg n)$
3. *Add items to knapsack (starting at the first) until there are no more items, or until the capacity W is exceeded.*

If knapsack is not yet full, fill knapsack with a fraction of next unselected item. $\Theta(n)$

Running time: $\Theta(n \lg n)$

The Fractional Knapsack Algorithm

- Greedy choice: Keep taking item with highest **value** (benefit to weight ratio)
 - Use a heap-based priority queue to store the items, then the time complexity is $O(n \log n)$.

Algorithm *FKnapsack*(S, W)

Input: set S of items w/ benefit b_i and weight w_i ; max. weight W

Output: amount x_i of each item i to maximize benefit with weight at most W

for each item i in S

$x_i \leftarrow 0$

$v_i \leftarrow b_i / w_i$ {value}

$w \leftarrow 0$ {current total weight}

while $w < W$

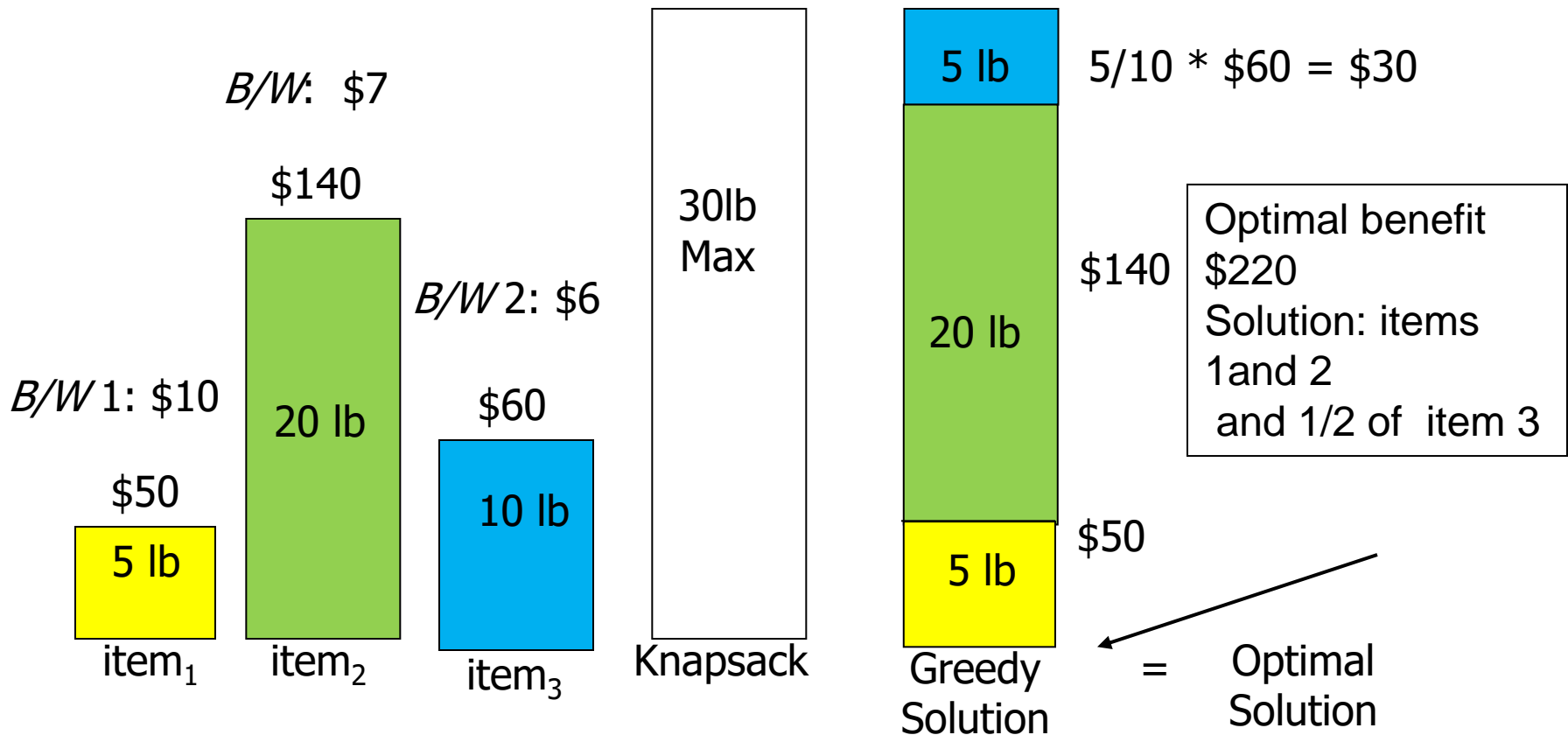
remove item i with highest v_i

$x_i \leftarrow \min\{w_i, W - w\}$

$w \leftarrow w + \min\{w_i, W - w\}$

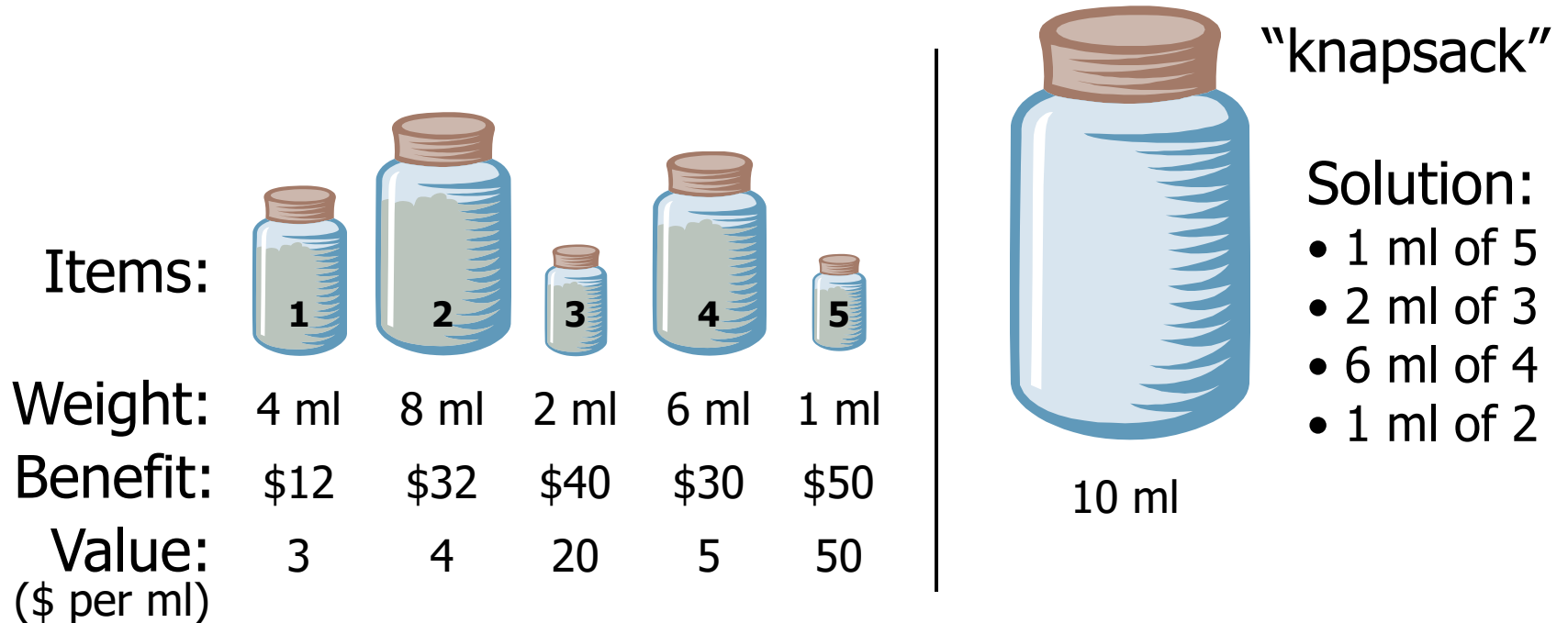
Example of applying the optimal greedy algorithm for Fractional Knapsack Problem

$$S = \{ (item_1, 5, \$50), (item_2, 20, \$140), (item_3, 10, \$60), \}$$



Greedy Knapsack

- Given: A set S of n items, with each item i having
 - b_i - a positive benefit
 - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W .



Fractional Knapsack has greedy choice property

That is, if b_i/w_i is the maximum ratio, then there exists an optimal solution that contains item x_i up to the extent of $\min\{w_i, W\}$.

Proof (by contradiction): Assume that there does not exist an optimal solution that contains x_i . Let $O = \{x_j, \dots, x_k\}$ be an optimal solution that does not contain x_i . Let x_t be the item with maximum weight w_t in O .

1) If $w_t \geq w_i$, then replace w_i amount of x_t by w_i amount of x_i . This will either increase the value of the solution if $b_i/w_i > b_t/w_t$ or be an alternative maximum solution if $b_i/w_i = b_t/w_t$.

2) If $w_t < w_i$, then

a) Let S be a subset of items in O whose total weight is greater than w_i . Replacing w_i of this total weight by w_i of x_i will improve the value of the solution.

Fractional Knapsack has greedy choice property

b) If no such set S exists then the sum of the weights of all items in $O = W \leq w_i$. Replace all the items in O by W units of x_i and the solution will improve (or leading to an alternative solution containing x_i).

Therefore we have shown that adding item x_i to O will improve the solution or lead to an alternative maximum solution.

Coin Change

- Coin changing problem (informal):
 - Given certain amount of change: A
 - The denominations of coins are: 25, 10, 5, 1
 - How to use the fewest coins to make this change?
- $A = 25q + 10d + 5n + p$, what are the q , d , n , and p , minimizing $(q+d+n+p)$
- Can you design an algorithm to solve this problem?



Coin changing problem


- Greedy choice
 - Choose as many of the largest coins available.
- Optimal substructure
 - After the greedy choice, assuming the greedy choice is correct, can we get the optimal solution from a subproblem.
 - Given $A = 63$ cents
 - Assuming we have chosen $2 * 25 = 50$
 - Is two quarters + optimal **coin**(63-50) the optimal solution of 63 cents?

Coin Change

- Step 1: $A = 63$




Coin Change

- Step 1: $A = 63$, $q = 2$ 





Coin Change

- Step 1: $A = 63$, $q = 2$ 
- Step 2: $(63 - 50) = 13$





Coin Change

- Step 1: $A = 63$, $q = 2$ 
- Step 2: $(63 - 50) = 13$, $d = 1$ 





Coin Change

- Step 1: $A = 63$, $q = 2$ 
- Step 2: $(63-50) = 13$, $d = 1$ 
- Step 3: $(13-10) = 3$






Coin Change




- Step 1: $A = 63$, $q = 2$ 
- Step 2: $(63-50) = 13$, $d = 1$ 
- Step 3: $(13-10) = 3$



Coin Change

- Step 1: $A = 63$, $q = 2$ 
- Step 2: $(63-50) = 13$, $d = 1$ 
- Step 3: $(13-10) = 3$, $p = 3$ 

Coin Change

- Step 1: $A = 63$, $q = 2$ 
- Step 2: $(63-50) = 13$, $d = 1$ 
- Step 3: $(13-10) = 3$, $p = 3$ 
- Number of coins = 6

Coin Change

- Step 1: $A = 63$, $q = 2$



- Step 2: $(63-50) = 13$, $d = 1$



- Step 3: $(13-10) = 3$, $p = 3$



- Number of coins = 6
- For coin denominations of 25, 10, 5, 1
 - The greedy choice property is not violated

A failure of the Greedy Algorithm

- Suppose in a fictional monetary system, we have 1 cent, 7 cent, and 10 cent coins
- The greedy algorithm results in a solution, but not in an optimal solution

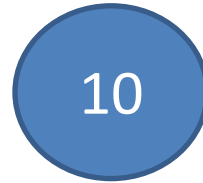
Coin Change Fail

- Step 1: $A = 15$



Coin Change Fail

- Step 1: $A = 15$



- Step2: $(15-10) = 5$



Coin Change Fail

- Step 1: $A = 15$

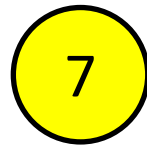


- Step 2: $(15 - 10) = 5$



This is six coins

The optimal solution is three coins



Huffman Codes

Text Compression (Zip)

- On a computer: changing the representation of a file so that it takes less space to store or/and less time to transmit.
- Original file can be reconstructed exactly from the compressed representation
- Very effective technique for compressing data, saving 20% - 90%.

First Approach

- Consider the word **ABRACADABRA**
- How can we write this string in a most economical way?
- Since it has 5 letters, we would need 3 bits to represent each character. For example.
 - A = 000
 - B = 001
 - C = 010
 - D = 011
 - R = 100
- Since there are 11 letters in ABRACADABRA it requires 33 bits.
- Is there a better way?

Of Course!!

- Magic word: ABRACADABRA
- LET $A = 0$
 $B = 100$
 $C = 1010$
 $D = 1011$
 $R = 11$
- Thus, ABRACADABRA = 01001101010010110100110
- So 11 letters demand 23 bits < 33 bits, an improvement of about 30%.

However...

- There are some concerns...
- Suppose we have
 - A-> 01
 - B-> 0101
- If we have 010101, is this AB? BA? Or AAA?
- Therefore: **prefix codes**, no codeword is a prefix of another codeword, is necessary

Prefix Codes

- Any prefix code can be represented by a full binary tree
- Each leaf stores a symbol.
- Each node has two children – left branch means 0, right means 1.
- codeword = path from the root to the leaf
interpreting suitably the left and right branches

For Example

A = 0

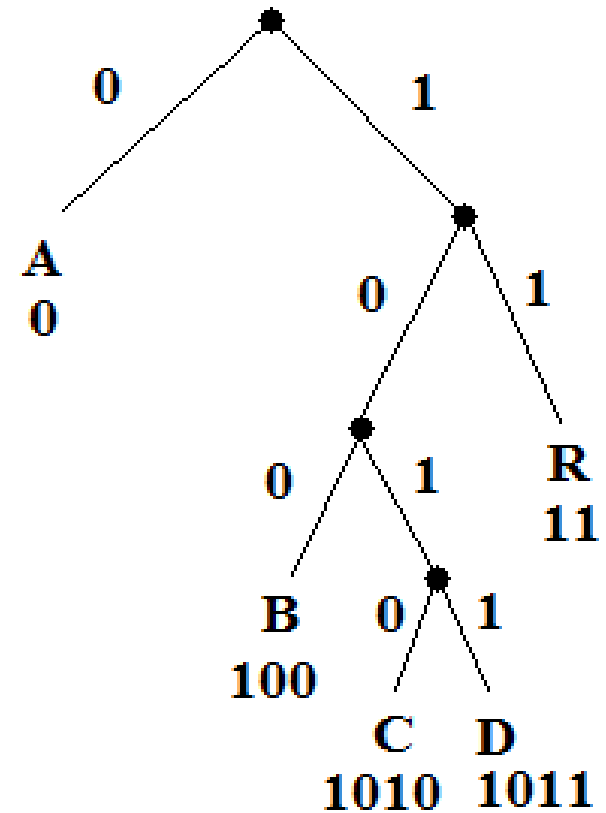
B = 100

C = 1010

D = 1011

R = 11

Decoding is unique and simple!

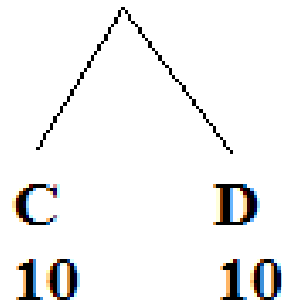


How do we find the optimal coding tree?

- It is clear that the two symbols with the smallest frequencies must be at the bottom of the optimal tree, as children of the lowest internal node
- This is a good sign that we have to use a bottom-up manner to build the optimal code!
- Huffman's idea is based on a **greedy** approach, using the previous notices.

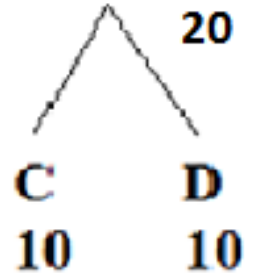
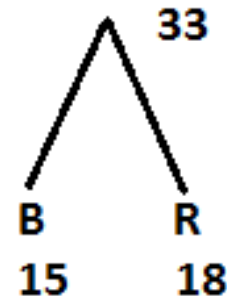
Constructing a Huffman Code

- Assume that frequencies of symbols are
A: 50 B: 15 C: 10 D: 10 R: 18
- Smallest numbers are 10 and 10 (C and D)



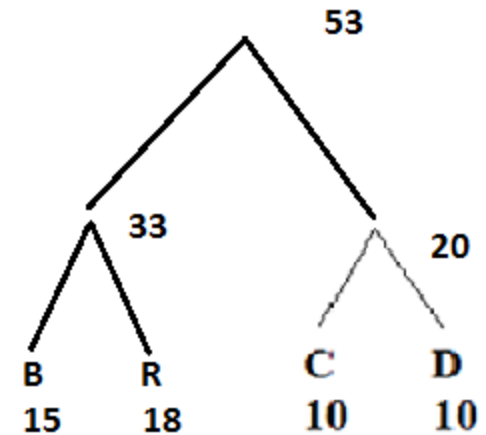
Constructing a Huffman Code

- Now Assume that frequencies of symbols are
A: 50 B: 15 C+D: 20 R: 18
- C and D have already been used, and
the new node above them (call it C+D)
has value 20
- The smallest values are B + R



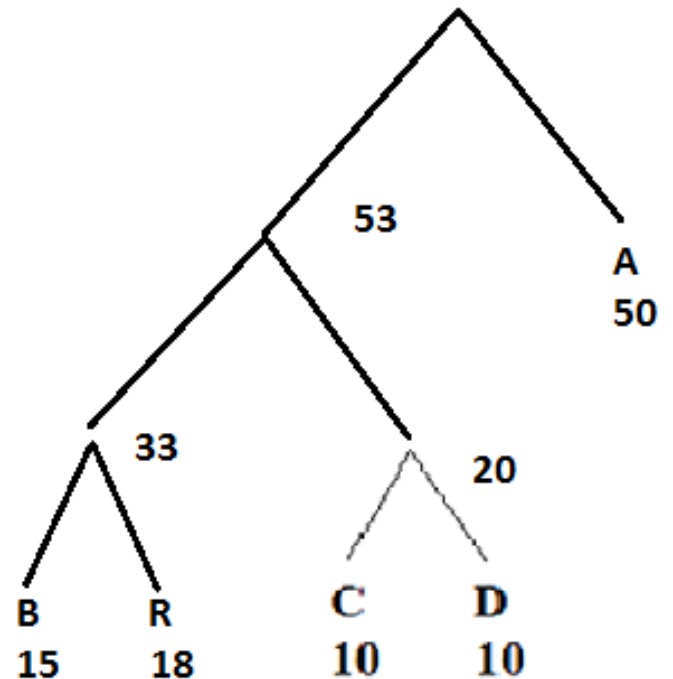
Constructing a Huffman Code

- Now Assume that frequencies of symbols are
A: 50 B+R: 33 C+D: 20
- The smallest values are
 $(B + R) + (C + D) = 53$

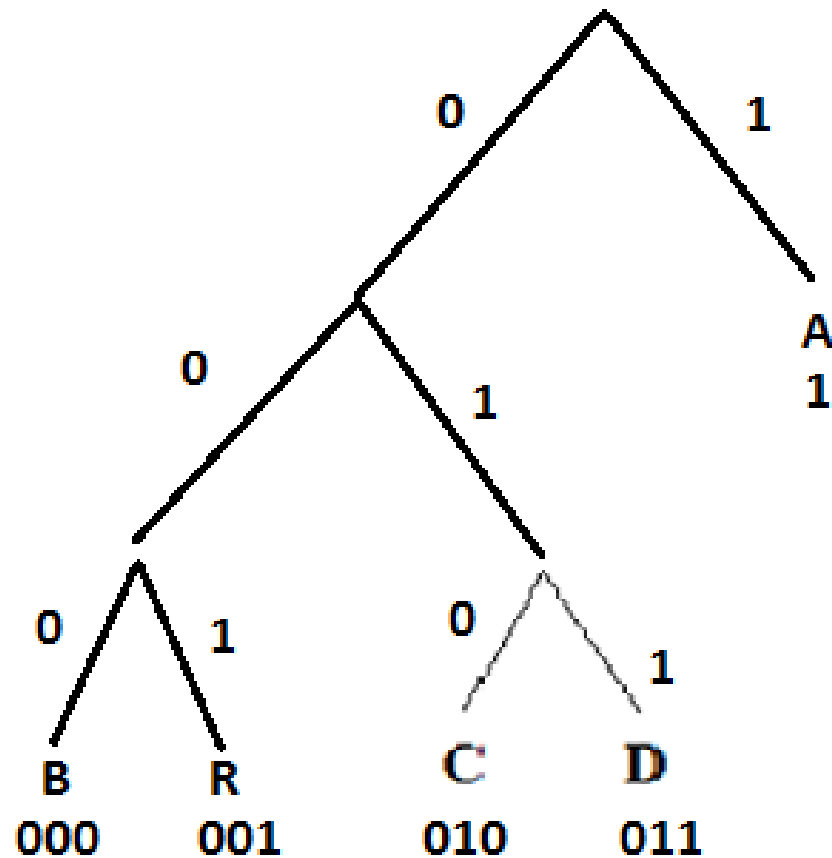


Constructing a Huffman Code

- Now Assume that frequencies of symbols are
A: 50 (B+R) + (C+D): 53
- The smallest values are
 $A + ((B + R) + (C + D)) = 103$



Constructing a Huffman Code

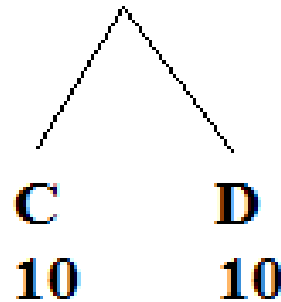


Constructing a Huffman Code

Assume that frequencies of symbols are

A: 50 B: 20 C: 10 D: 10 R: 30

Smallest numbers are 10 and 10 (C and D)

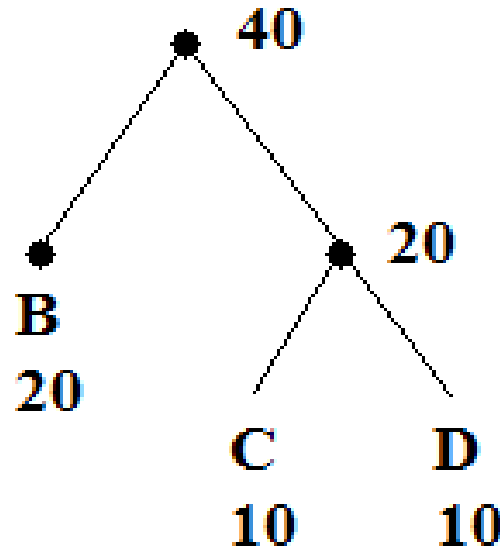


Constructing a Huffman Code

Assume that frequencies of symbols are

A: 50 B: 20 C: 10 D: 10 R: 30

- C and D have already been used, and the new node above them (call it C+D) has value 20
- The smallest values are B, C+D

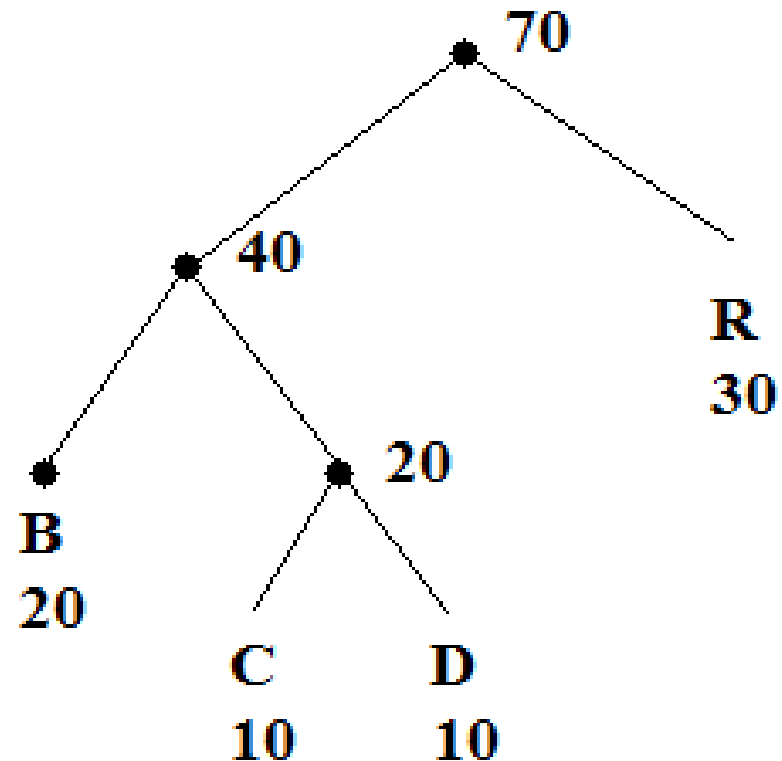


Constructing a Huffman Code

Assume that frequencies of symbols are

A: 50 B: 20 C: 10 D: 10 R: 30

Next, B+C+D (40) and R (30)

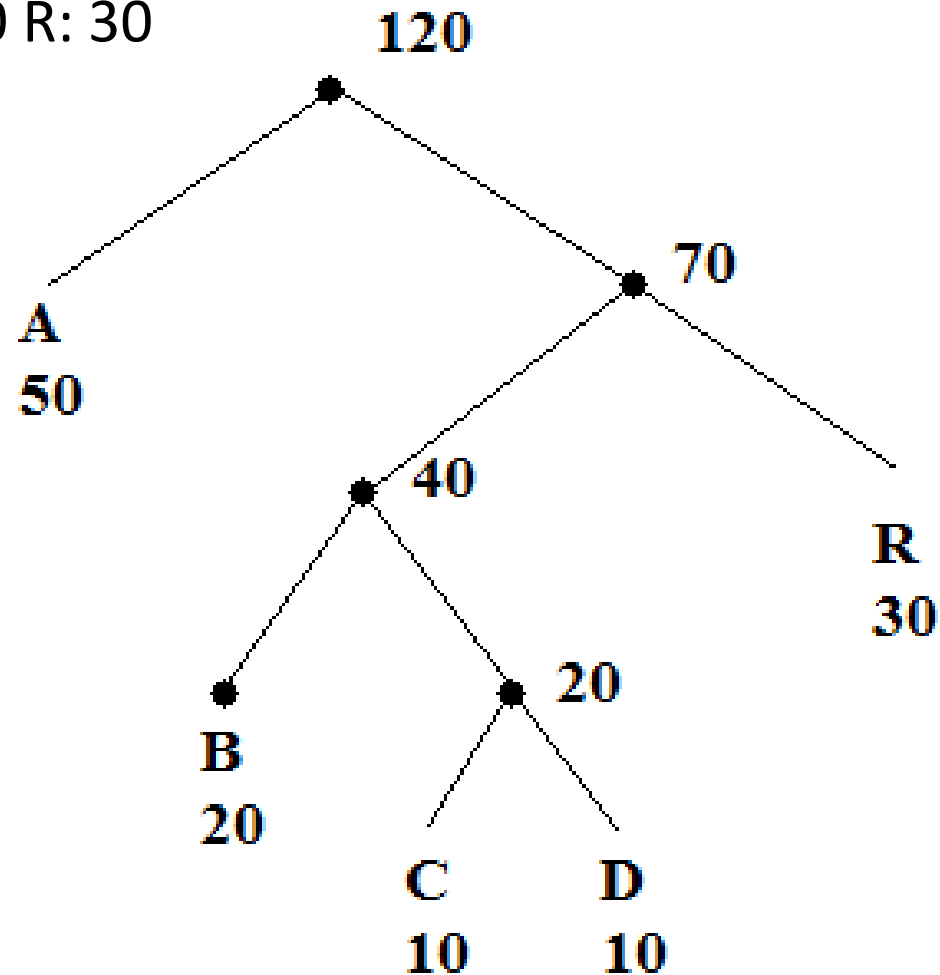


Constructing a Huffman Code

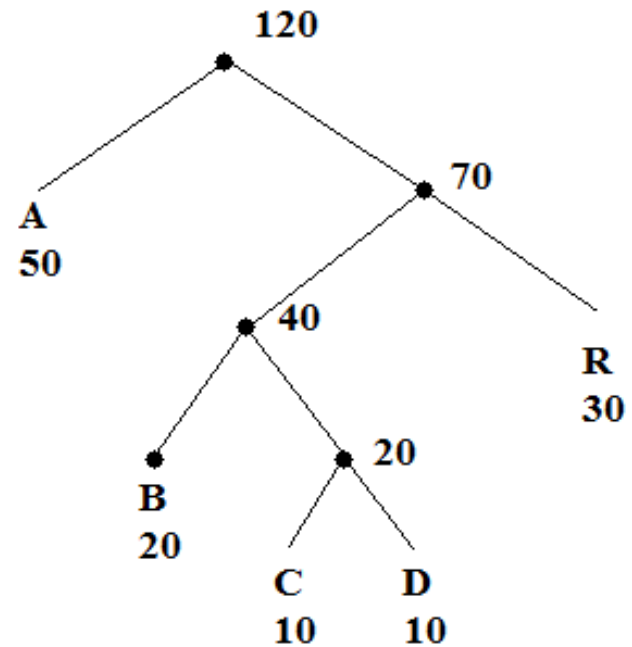
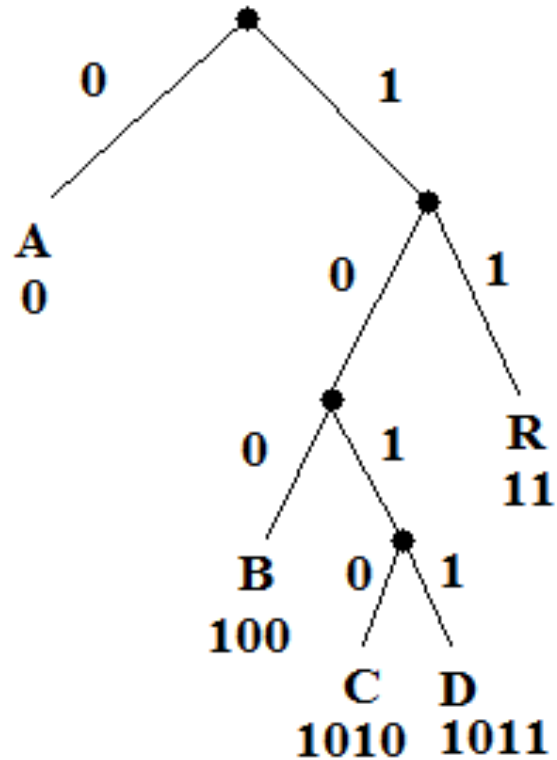
Assume that frequencies of symbols are

A: 50 B: 20 C: 10 D: 10 R: 30

Finally

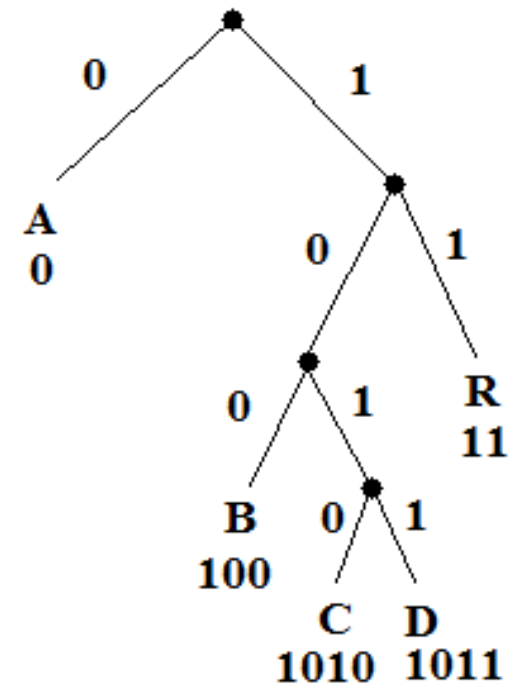


Constructing a Huffman Code



Decode the tree

- Suppose we have the
Following code:
10001011
- What is the decode
result?

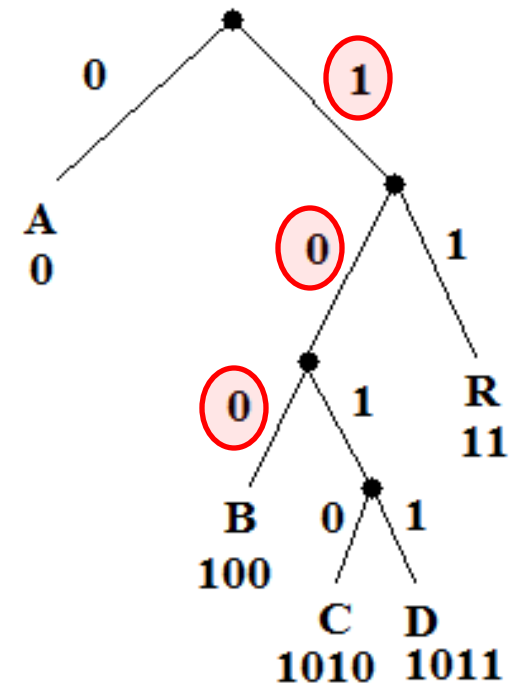


Decode the tree

- Suppose we have the
Following code:

10001011

- What is the decode
result? **B**



Decode the tree

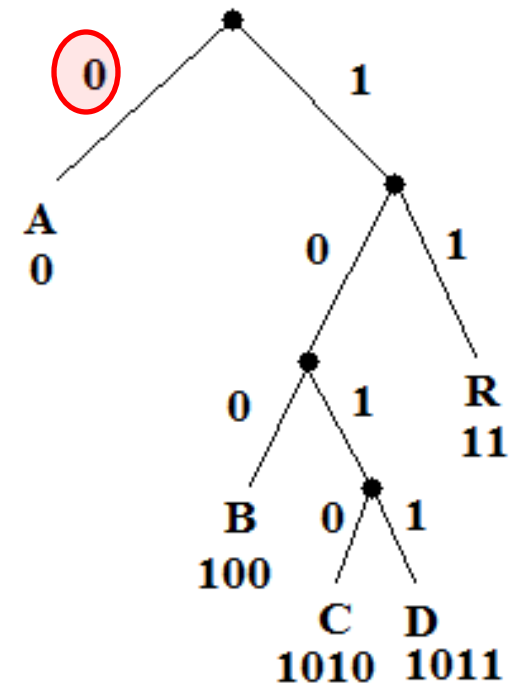
- Suppose we have the

Following code:

100**0**1011

- What is the decode result?

BA



Decode the tree

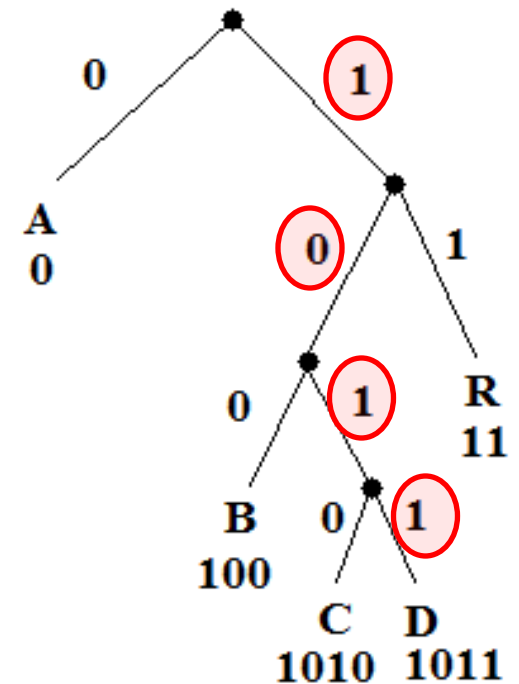
- Suppose we have the

Following code:

1000**1011**

- What is the decode result?

BAD



Decode the tree

- Suppose we have the
Following code:
10001011
- What is the decode
result? **BAD**

