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

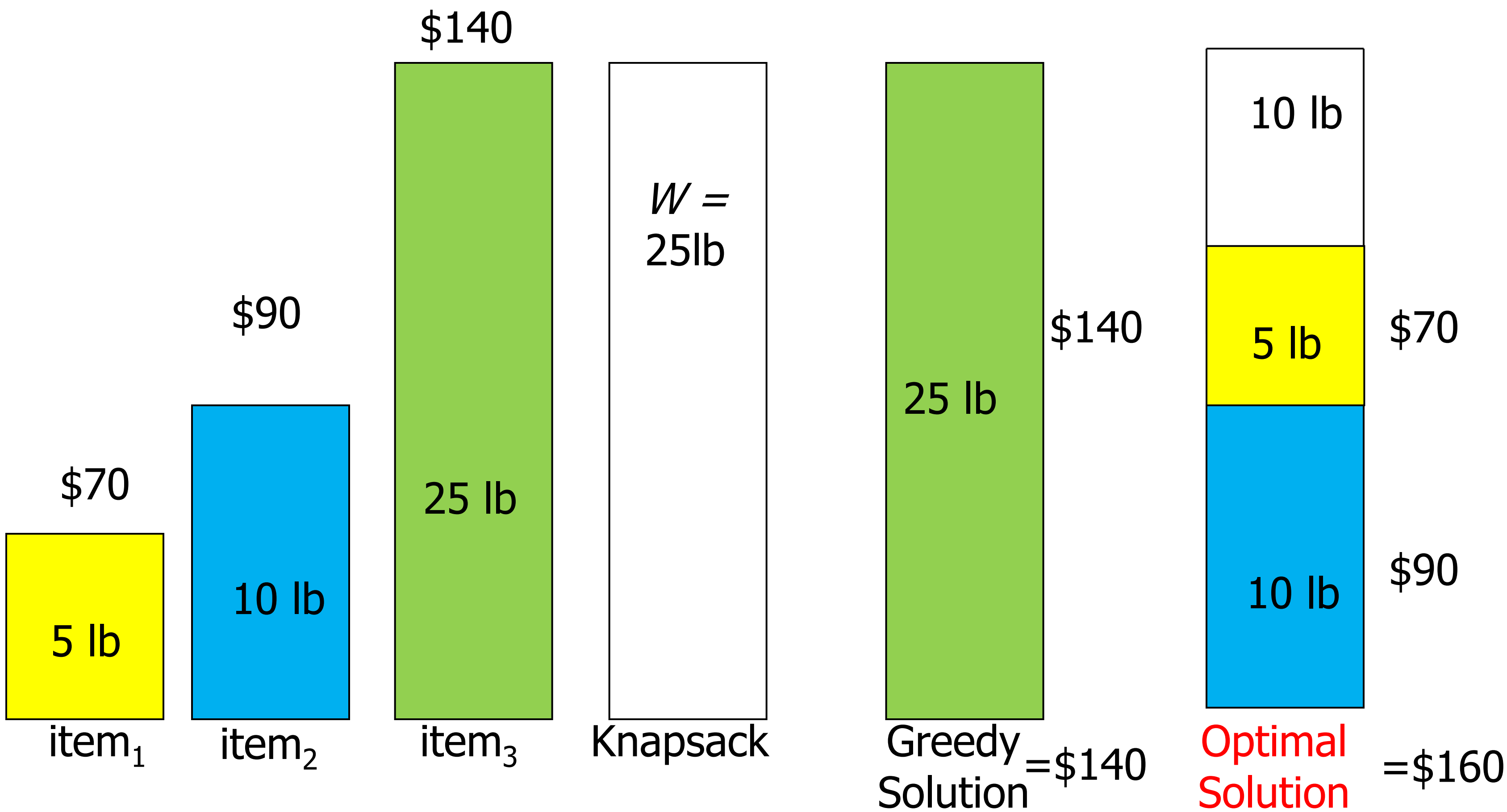
Determine a subset *T* of { 1, 2, …, *n* } that satisfies the following:

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

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

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

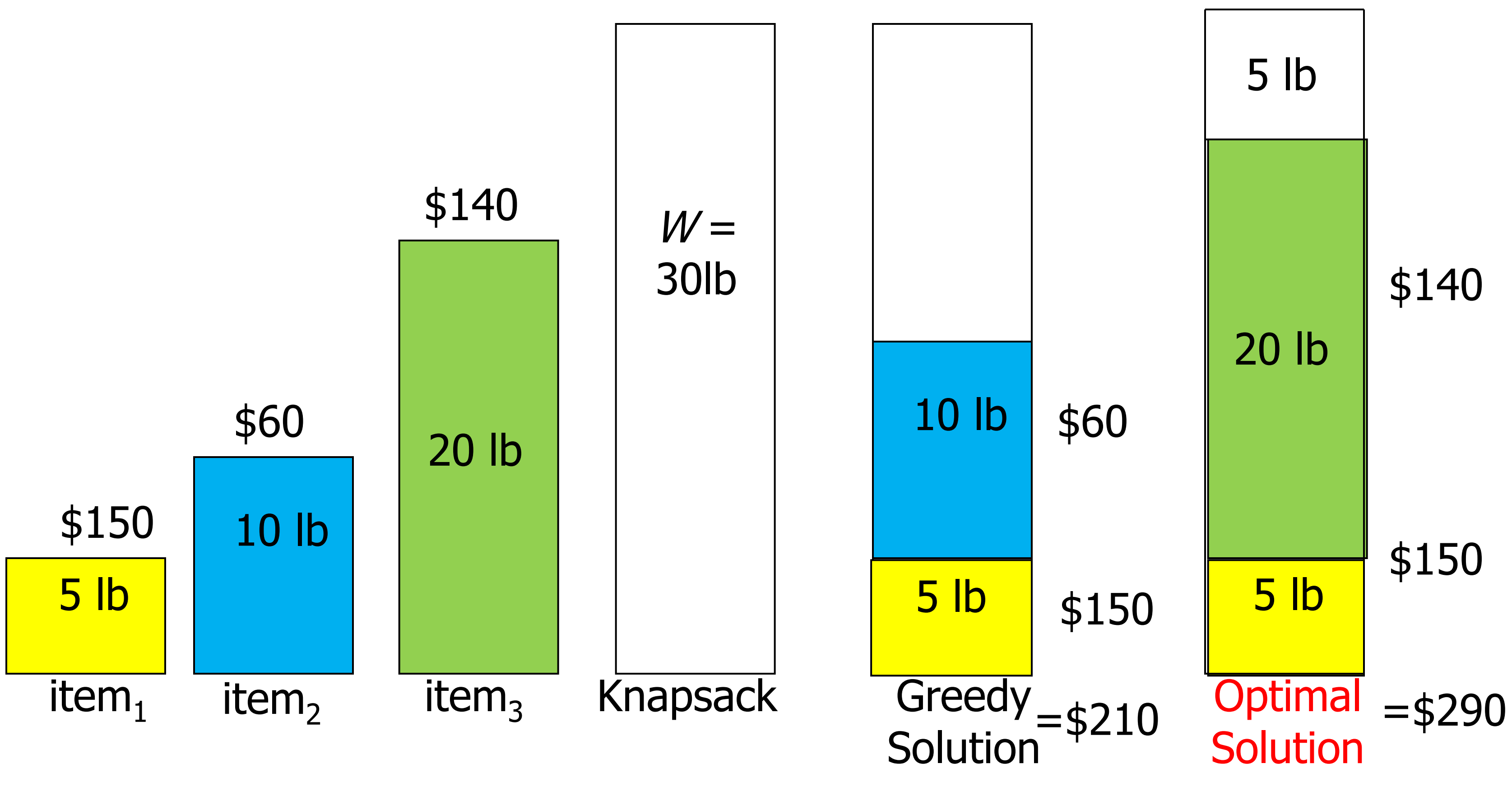
$S = \{ (item_1 , 5, \$70), (item_2 ,10, \$90), (item_3, 25, \$140) \}$



Greedy 2: Selection criteria: *Minimum weight* item

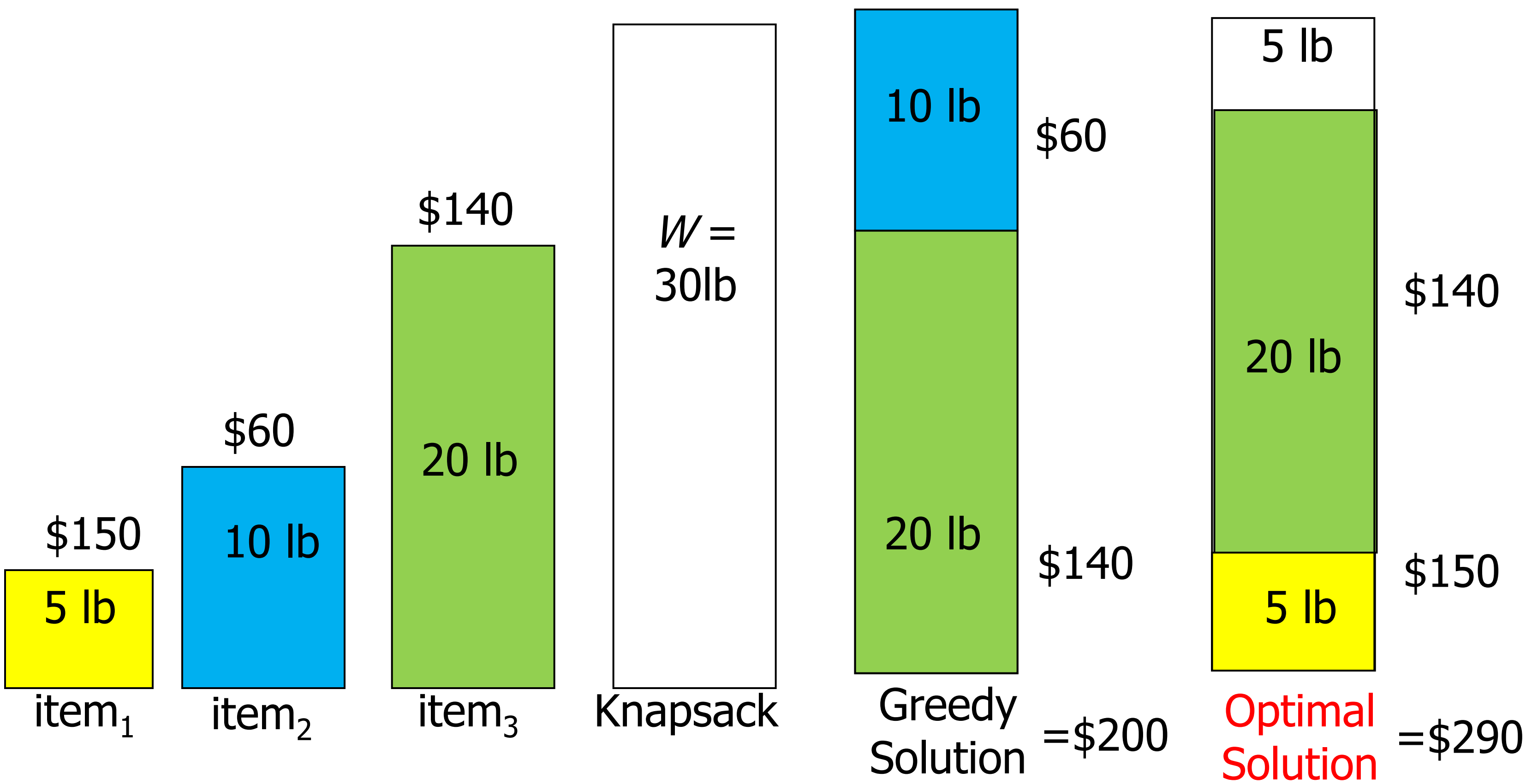
Counter Example:

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



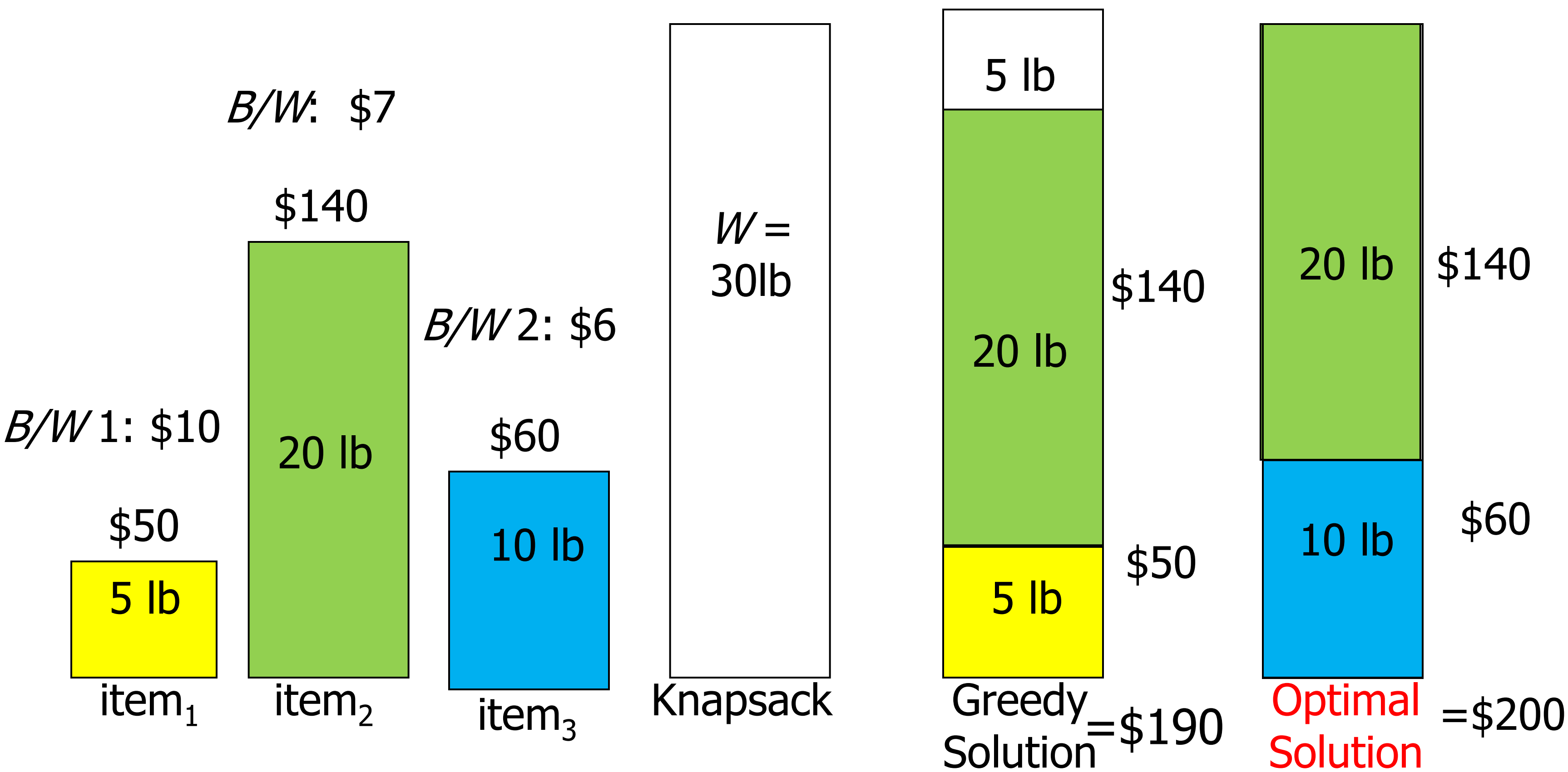
Greedy 3: Selection criteria: *Maximum weight* item
Counter Example:

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



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?

$W = 9$

$w_A = 2$	$b_A = \$40$
$w_B = \pi$	$b_B = \$50$
$w_C = 1.98$	$b_C = \$100$
$w_D = 5$	$b_D = \$95$
$w_E = 3$	$b_E = \$30$

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

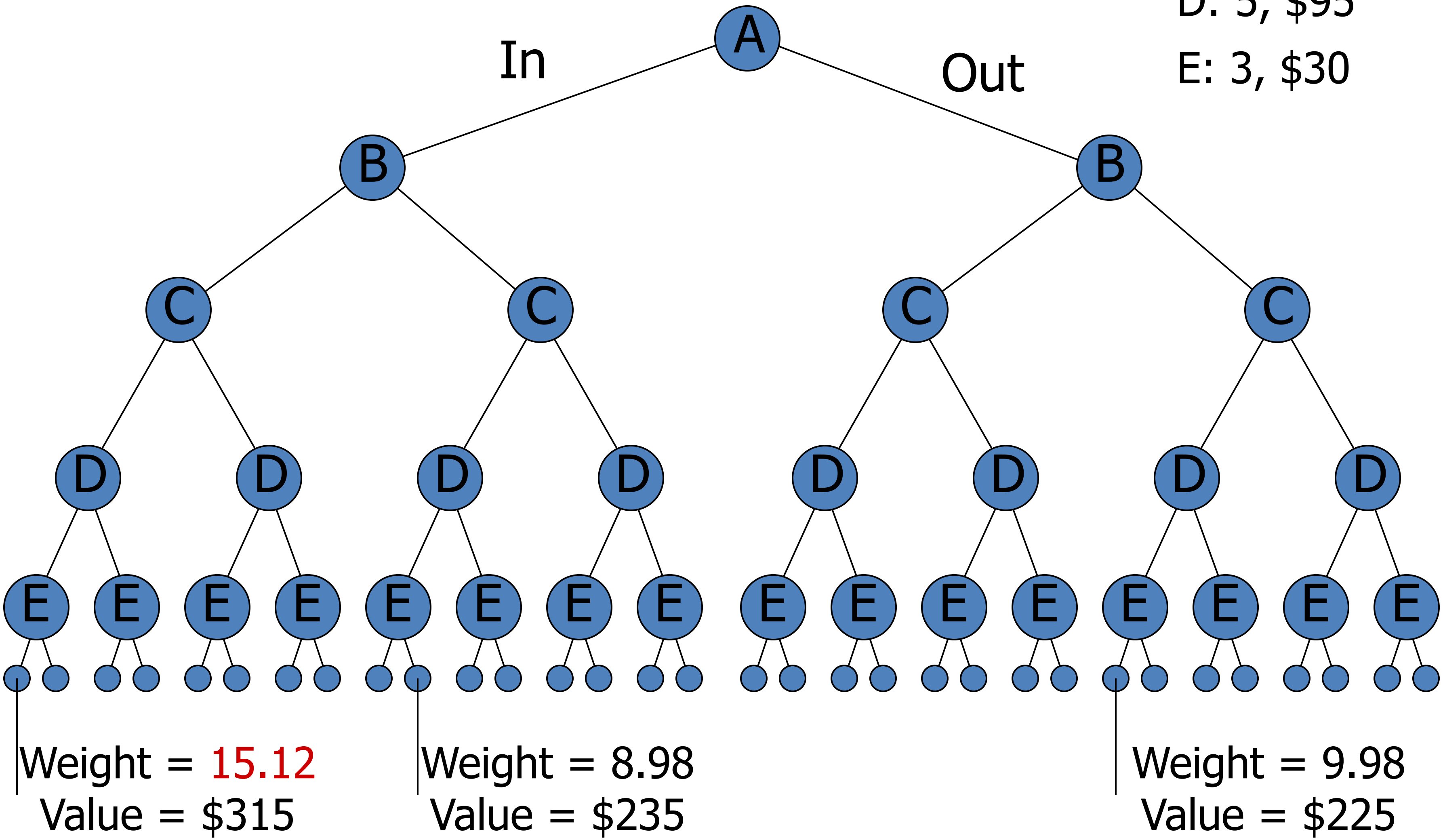
A: 2, $40

B: π, $50

C: 1.98, $100

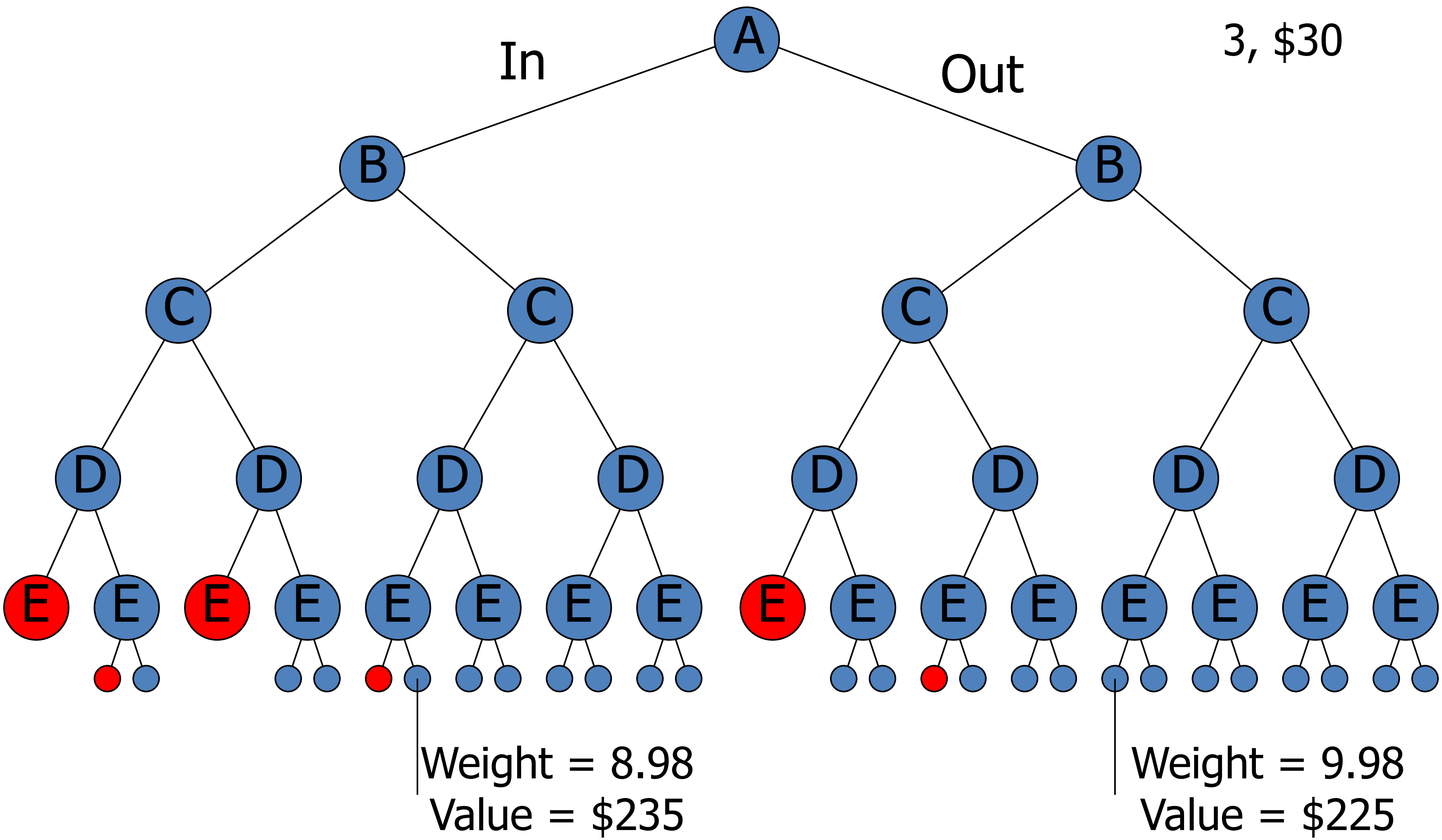
D: 5, $95

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*

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

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

$\min\{(\boldsymbol{W} - totweight), w_k\}$, 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***

$\boldsymbol{x_i} \leftarrow 0$

$\boldsymbol{v_i} \leftarrow \boldsymbol{b_i} / \boldsymbol{w_i}$ {value}

$\boldsymbol{w} \leftarrow 0$ {current total weight}

while $\boldsymbol{w} < \boldsymbol{W}$

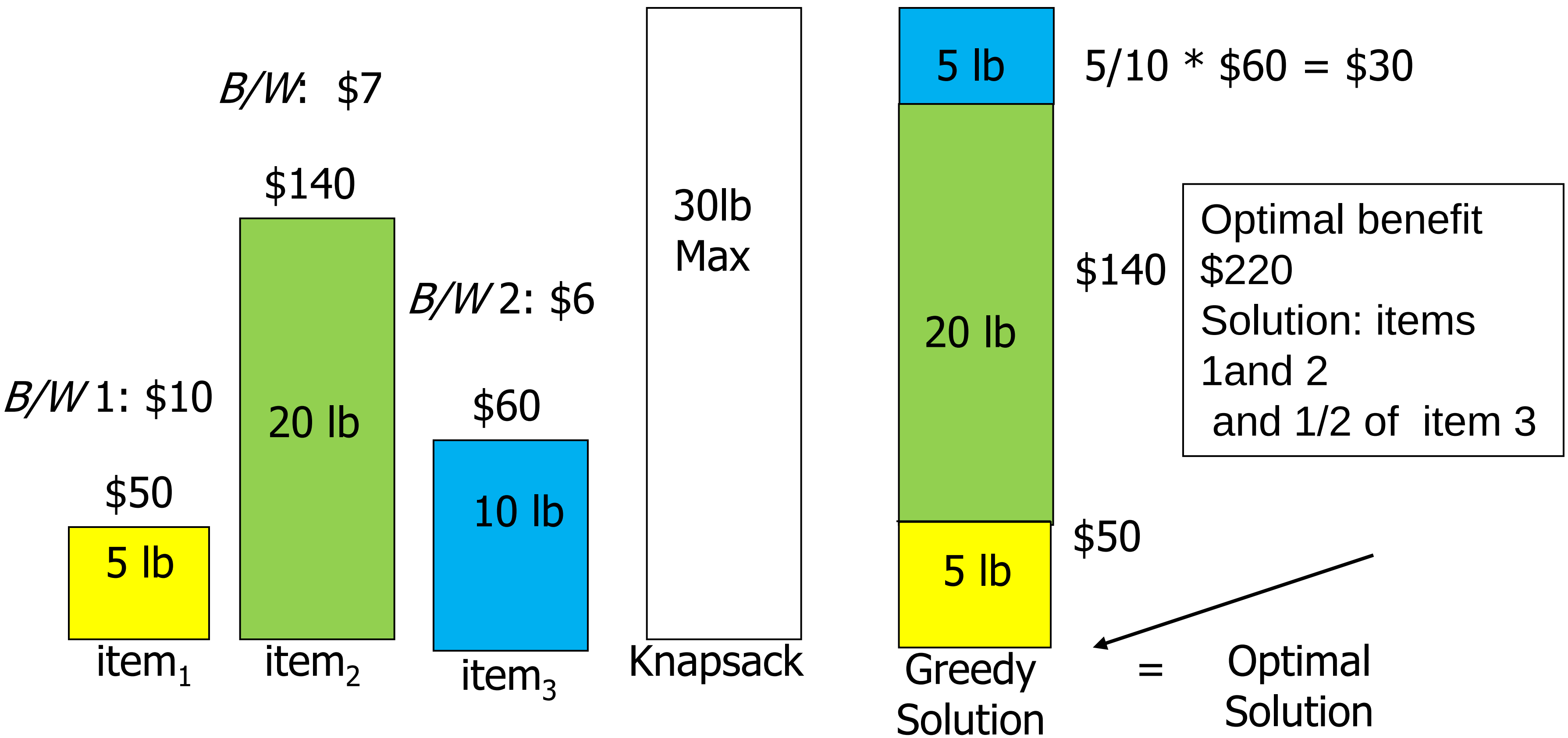
remove item $\boldsymbol{i}$ with highest $\boldsymbol{v_i}$

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

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

Example of applying the optimal greedy algorithm for Fractional Knapsack Problem

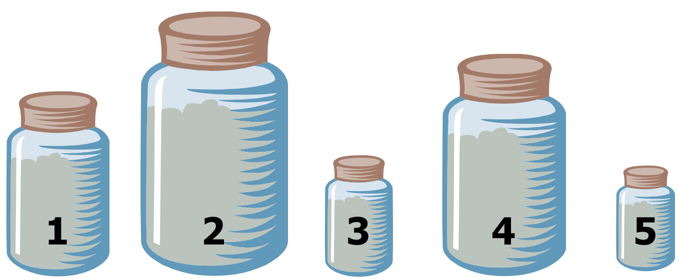
$S = \{ (item_1 , 5, \$50), (item_2, 20, \$140) (item_3 ,10, \$60), \}$



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.

Items:



	1	2	3	4	5
Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit:	\$12	\$32	\$40	\$30	\$50
Value: (\$ per ml)	3	4	20	5	50

"knapsack"

10 ml

Solution:
- 1 ml of 5
- 2 ml of 3
- 6 ml of 4
- 1 ml of 2

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 , .., 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 is 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 ≤ 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

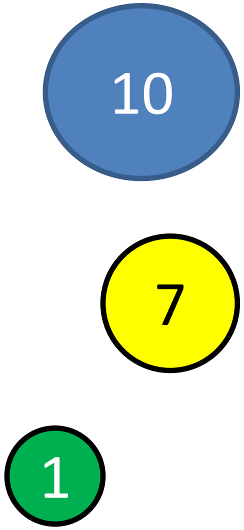
- For coin denominations of 25, 10, 5, 1
  - The greedy choice property is not violated

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

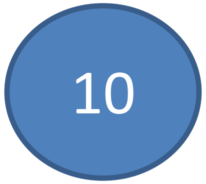
Coin Change Fail

- Step 1: A = 15

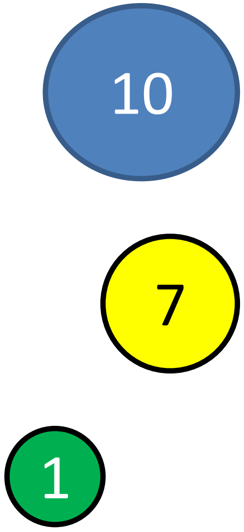


Coin Change Fail

- Step 1: A = 15

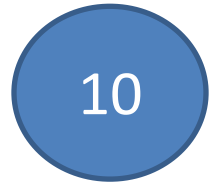


- Step2: (15-10) = 5



Coin Change Fail

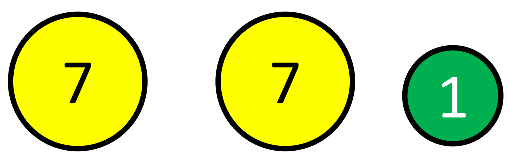
- Step 1: A = 15

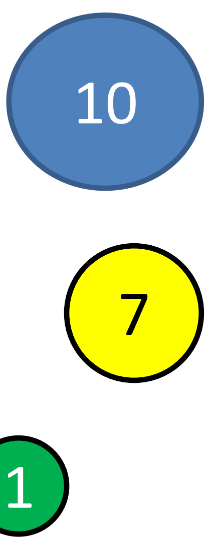


- Step2: (15-10) = 5

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%.

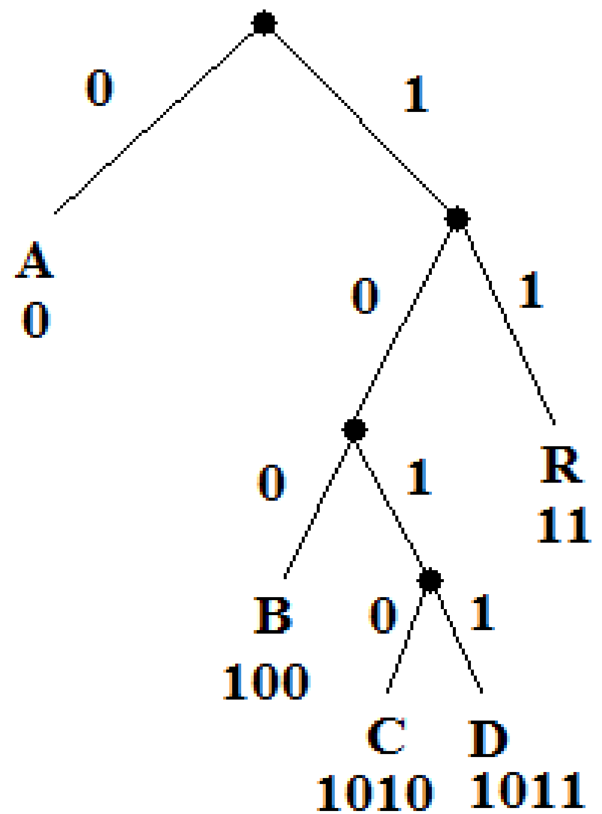
However…

- 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

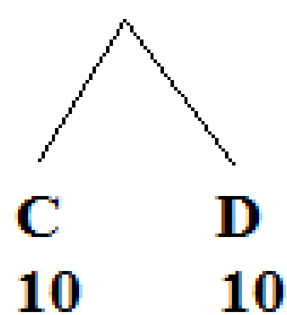
- This is a good sign that we have to use a bottom-up manner to build the optimal code!

- Huffman’s idea is based on a greedy approach, using the previous notices.

Constructing a Huffman Code

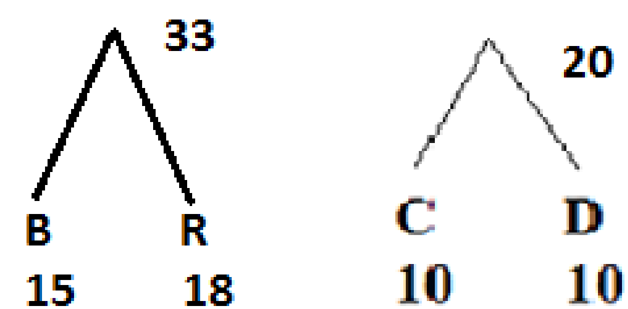
- Assume that frequencies of symbols are

  A: 50 B: 15 C: 10 D: 10 R: 18
- Smallest numbers are 10 and 10 (C and D)



Constructing a Huffman Code

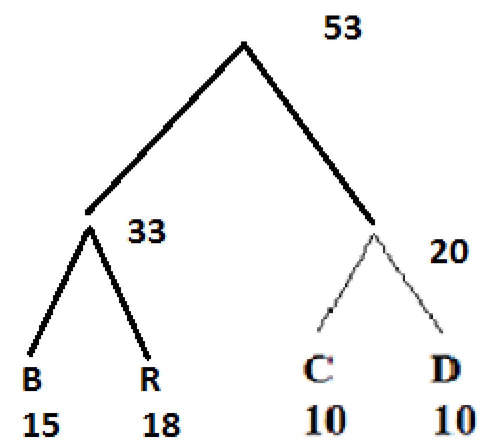
- Now Assume that frequencies of symbols are
  A: 50 B: 15 C+D: 20 R: 18
- C and D have already been used, and the new node above them (call it C+D) has value 20
- The smallest values are B + R



Constructing a Huffman Code

- Now Assume that frequencies of symbols are
  A: 50 B+R: 33 C+D: 20

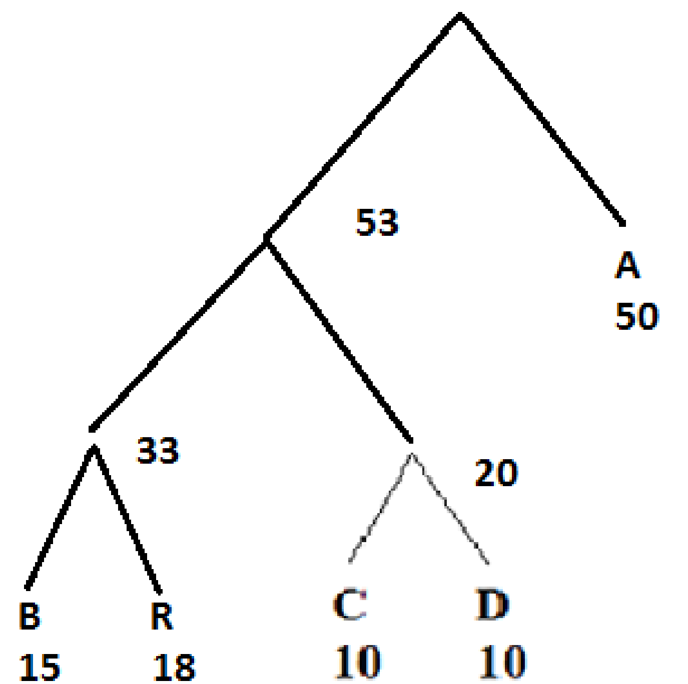
- The smallest values are
  (B + R)+(C+D)=53



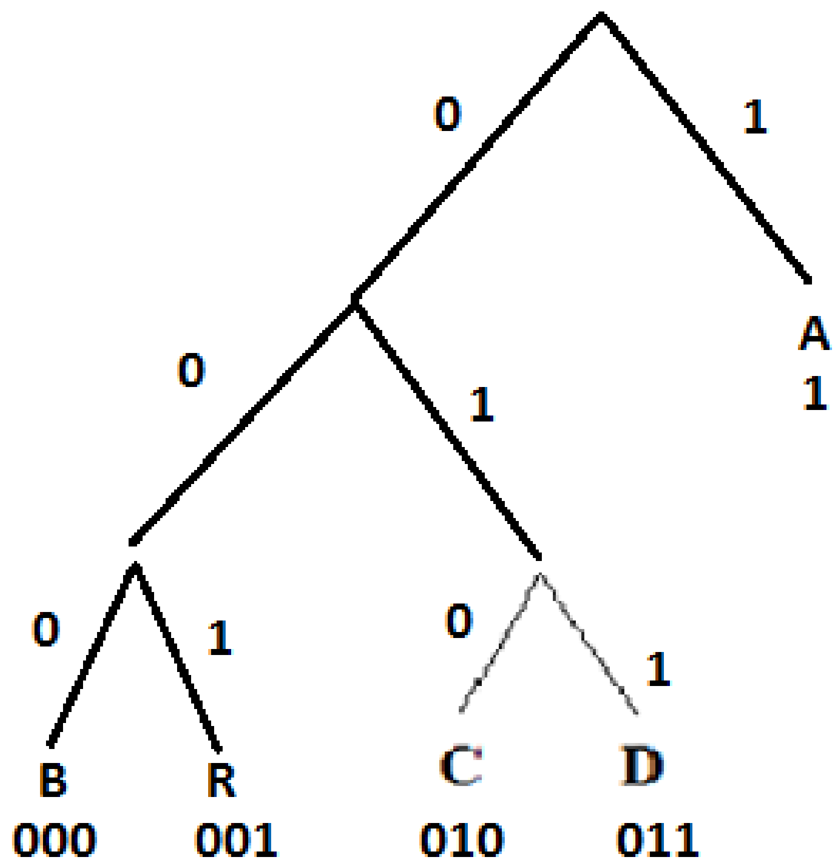
Constructing a Huffman Code

- 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

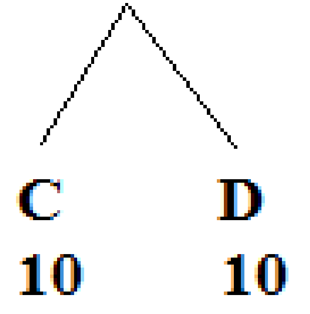


Constructing a Huffman Code

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)

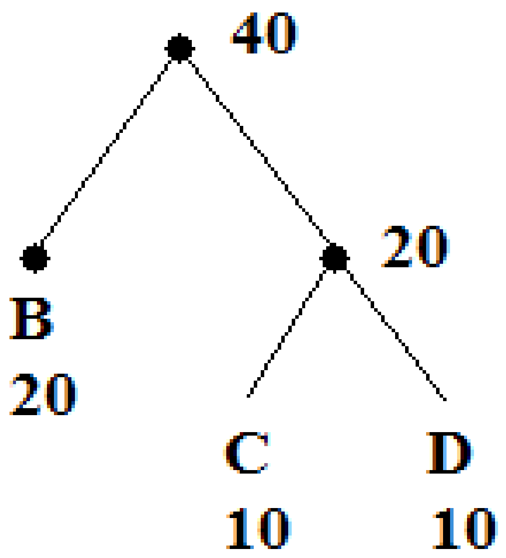


Constructing a Huffman Code

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

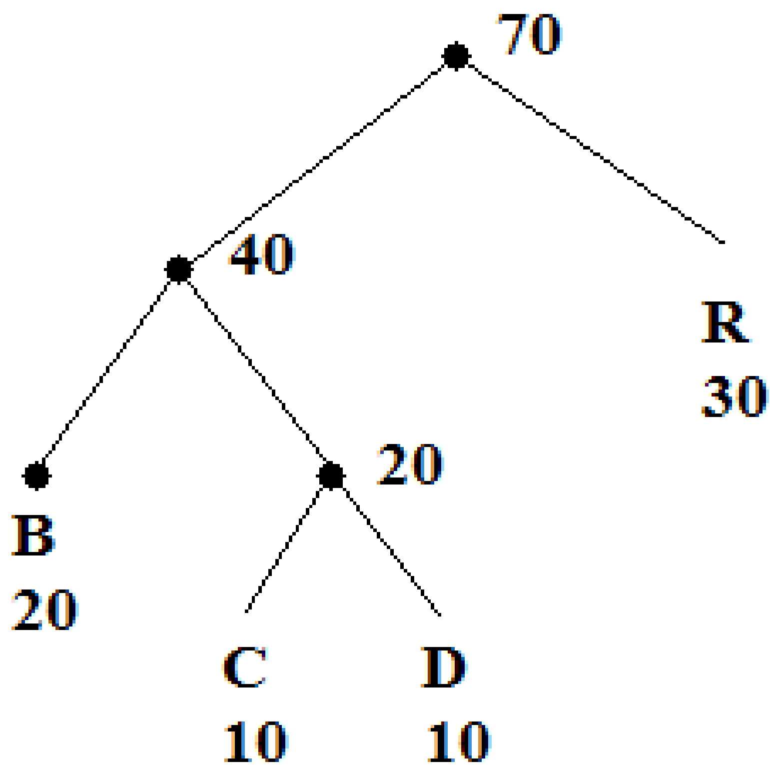


Constructing a Huffman Code

Assume that frequencies of symbols are

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

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

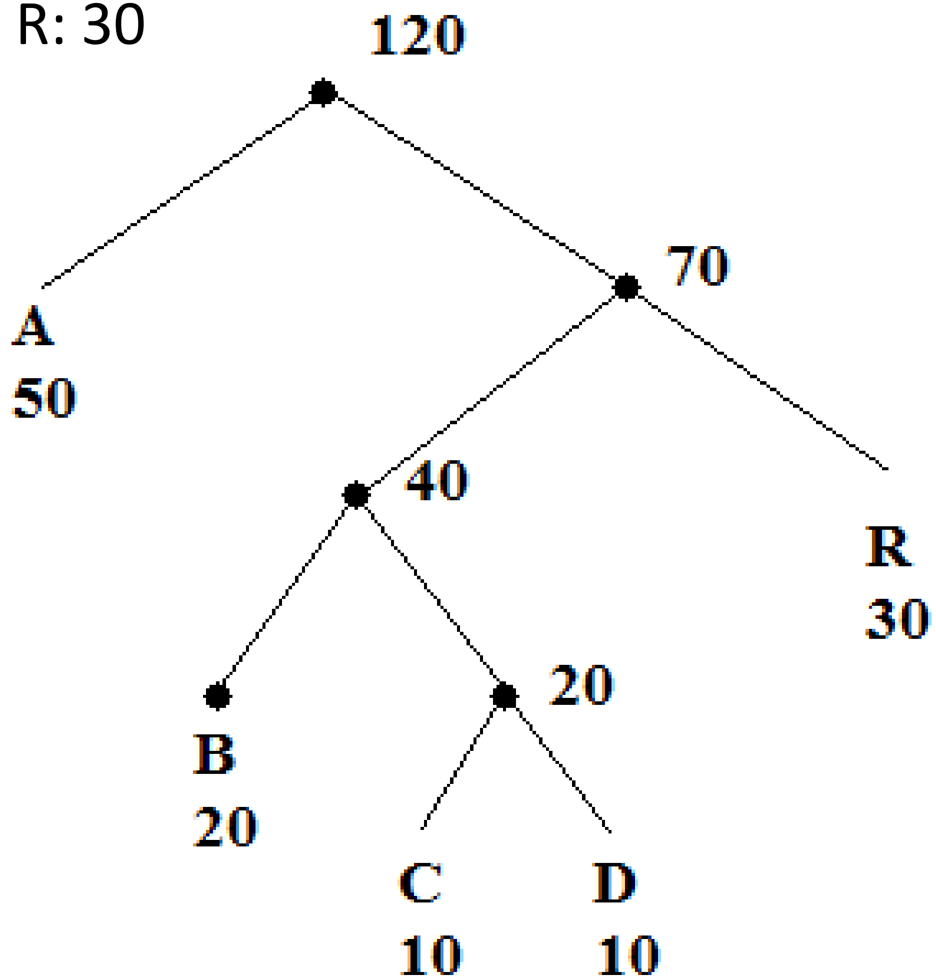


Constructing a Huffman Code

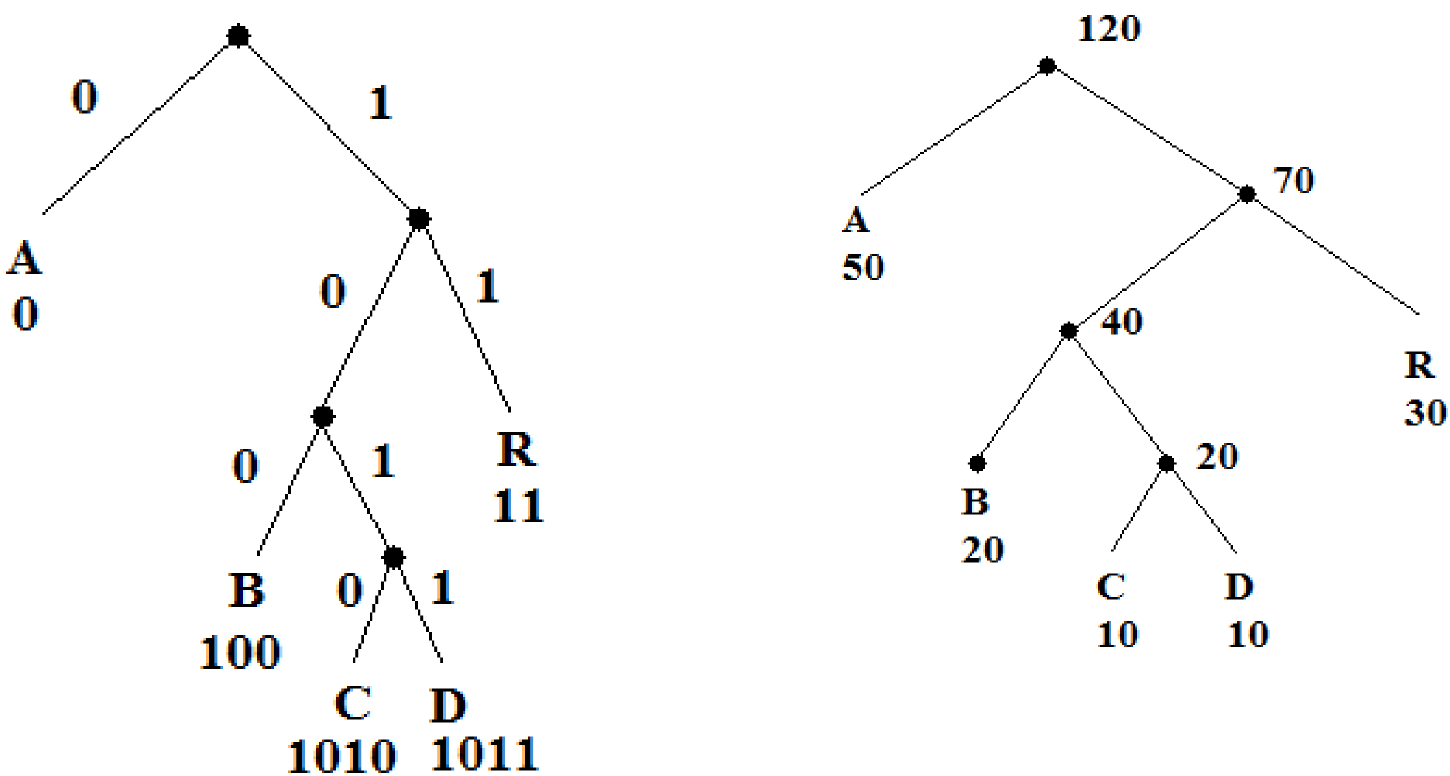
Assume that frequencies of symbols are

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

Finally



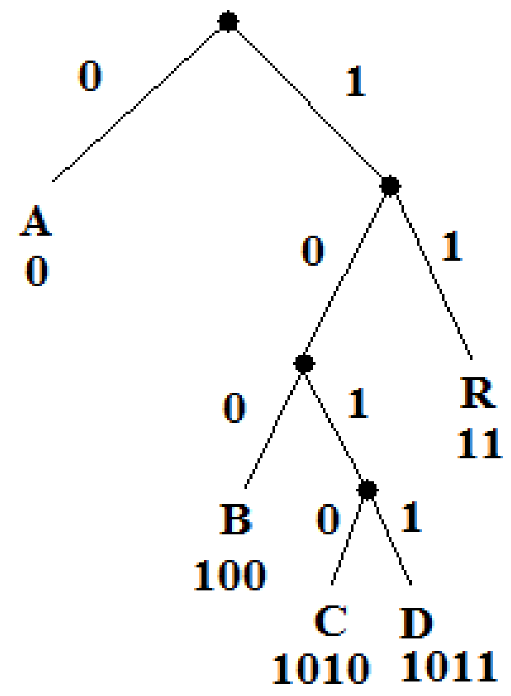
Constructing a Huffman Code



Decode the tree

- Suppose we have the Following code: 10001011

- What is the decode result?

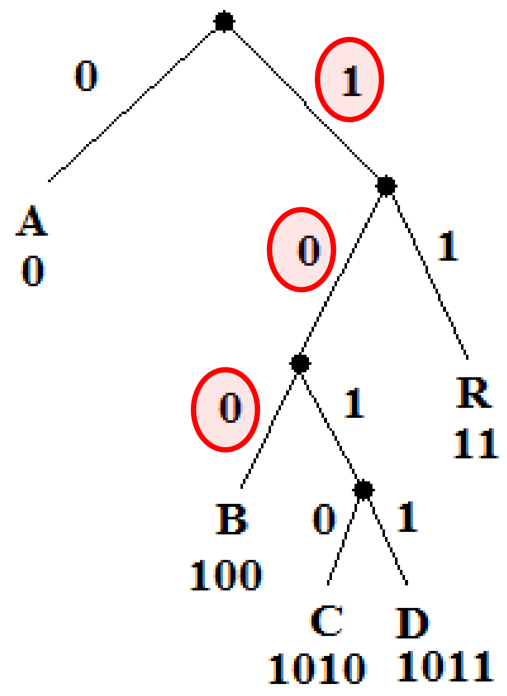


Decode the tree

- Suppose we have the Following code: **10**01011

- What is the decode result?

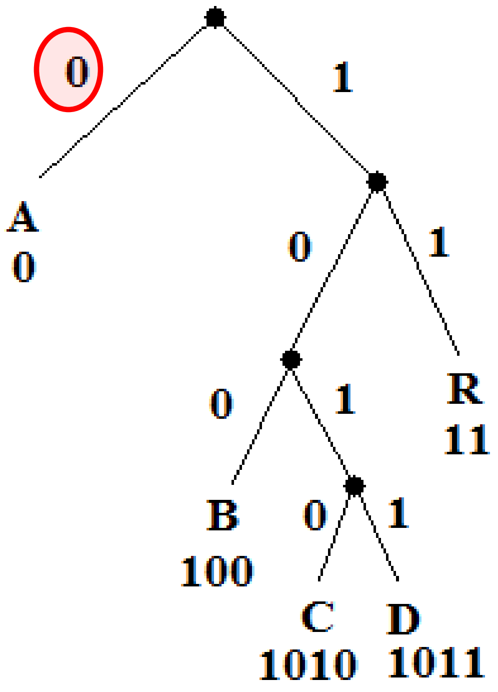
B



Decode the tree

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

BA

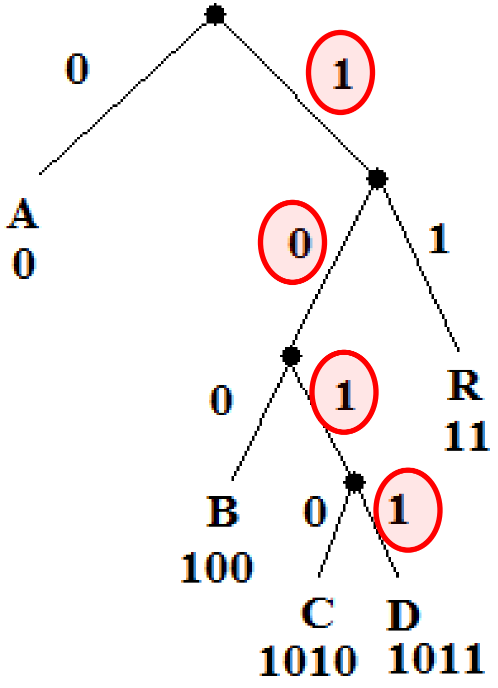


Decode the tree

- Suppose we have the Following code: 10001011

- What is the decode result?

BAD

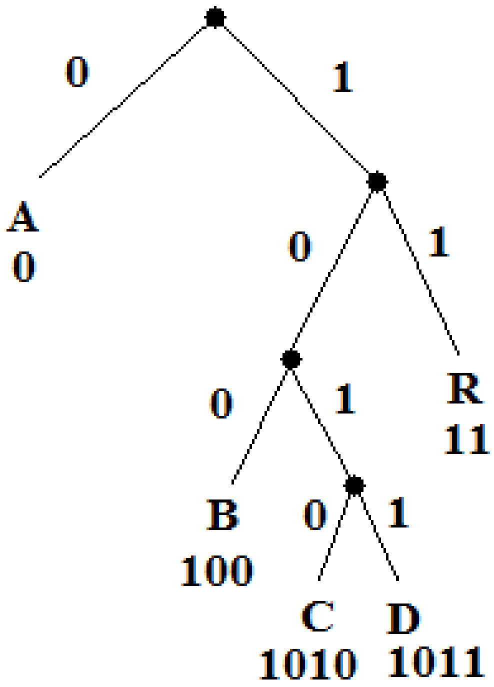


Decode the tree

- Suppose we have the Following code: 10001011

- What is the decode result?

BAD



Greedy Algorithms

- The Greedy Algorithm Techniques
- Knapsack Problem
- Huffman Codes
- Scheduling

Scheduling Problems

There are many variations of the scheduling problem.

- Activity: Goal maximize the number of activities
- Machine/Task Scheduling: Goal minimize the number of machines needed to complete all Tasks with start/finish constraints.
- Job Scheduling: Goal minimize the total time it takes to complete all jobs on a set of machines.
- And more ….

An Activity Scheduling Problem

Input: A set of activities S = {$a_1, \ldots, a_n$}

- Each activity has start time and a finish time: $a_i = (s_i, f_i)$
- Two activities are compatible if and only if their interval does not overlap

Output: a maximum-size subset of mutually compatible activities

The Activity Scheduling Problem

Here are a set of start and finish times

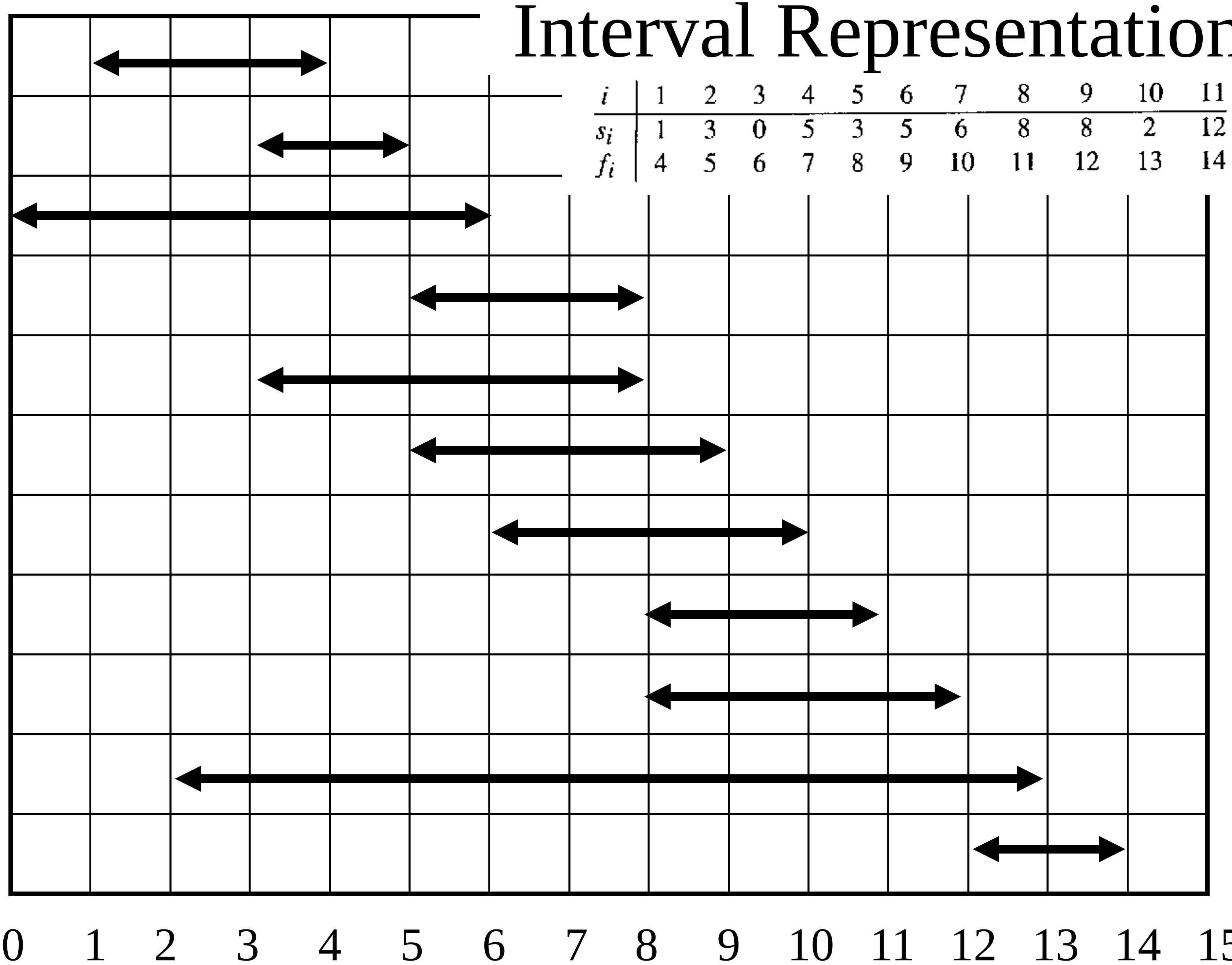
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

What is the maximum number of activities that can be completed?

- $\{a_3, a_9, a_{11}\}$ can be completed
- But so can $\{a_1, a_4, a_8, a_{11}\}$ which is a larger set
- But it is not unique, consider $\{a_2, a_4, a_9, a_{11}\}$

Interval Representation

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	8	9	10	11	12	13	14



Activity Scheduling: Greedy Algorithms

Greedy. Consider activities in some natural order.
Take each activity provided it's compatible with the ones already taken.

- [Earliest start time] Consider activities in ascending order of s_j.
- [Earliest finish time] Consider activities in ascending order of f_j.
- [Shortest interval] Consider activities in ascending order of $f_j - s_j$.
- [Fewest conflicts] For each activity j, count the number of conflicting activities c_j. Schedule in ascending order of c_j.

Greedy Algorithms are not always Optimal

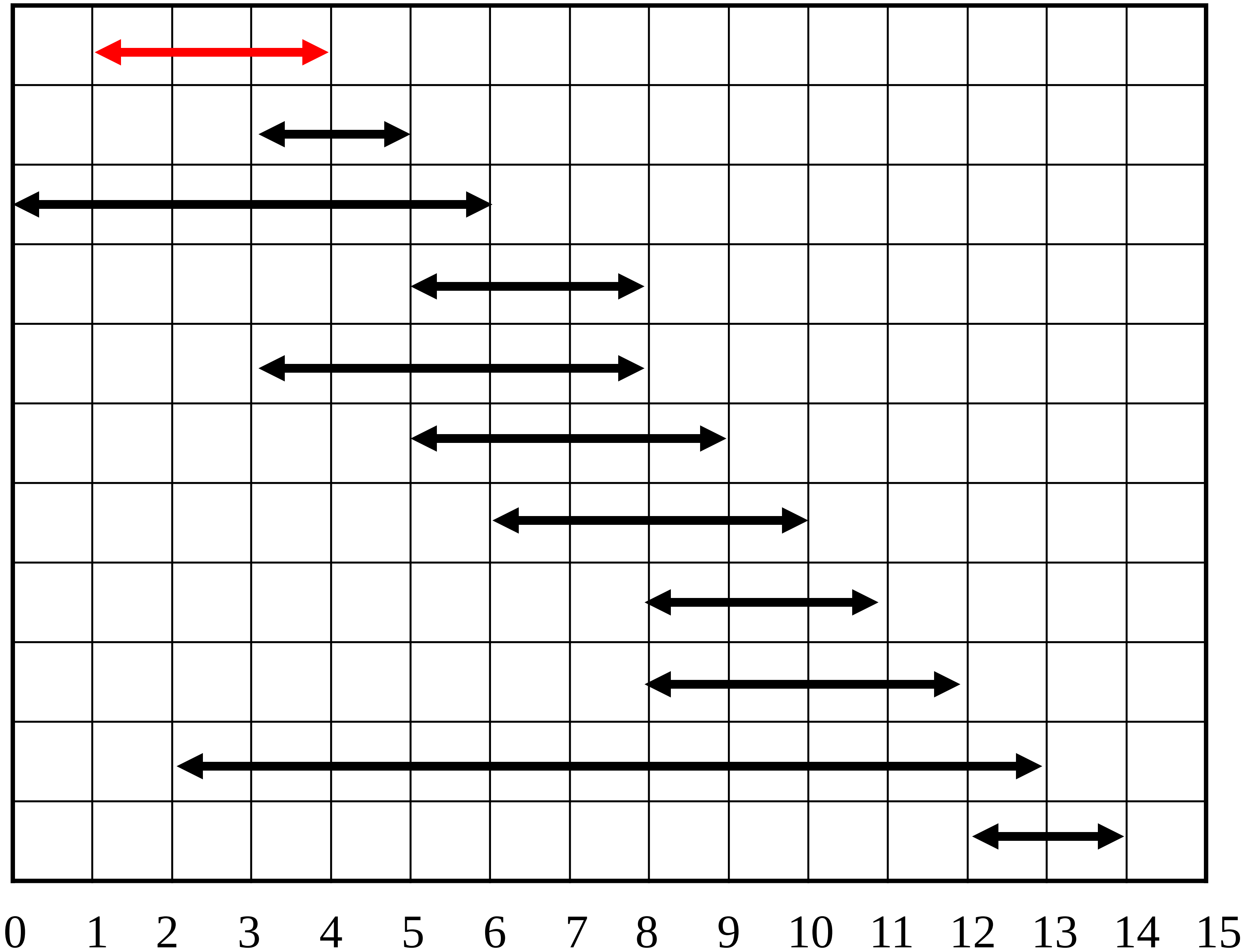
Counterexample for earliest start time

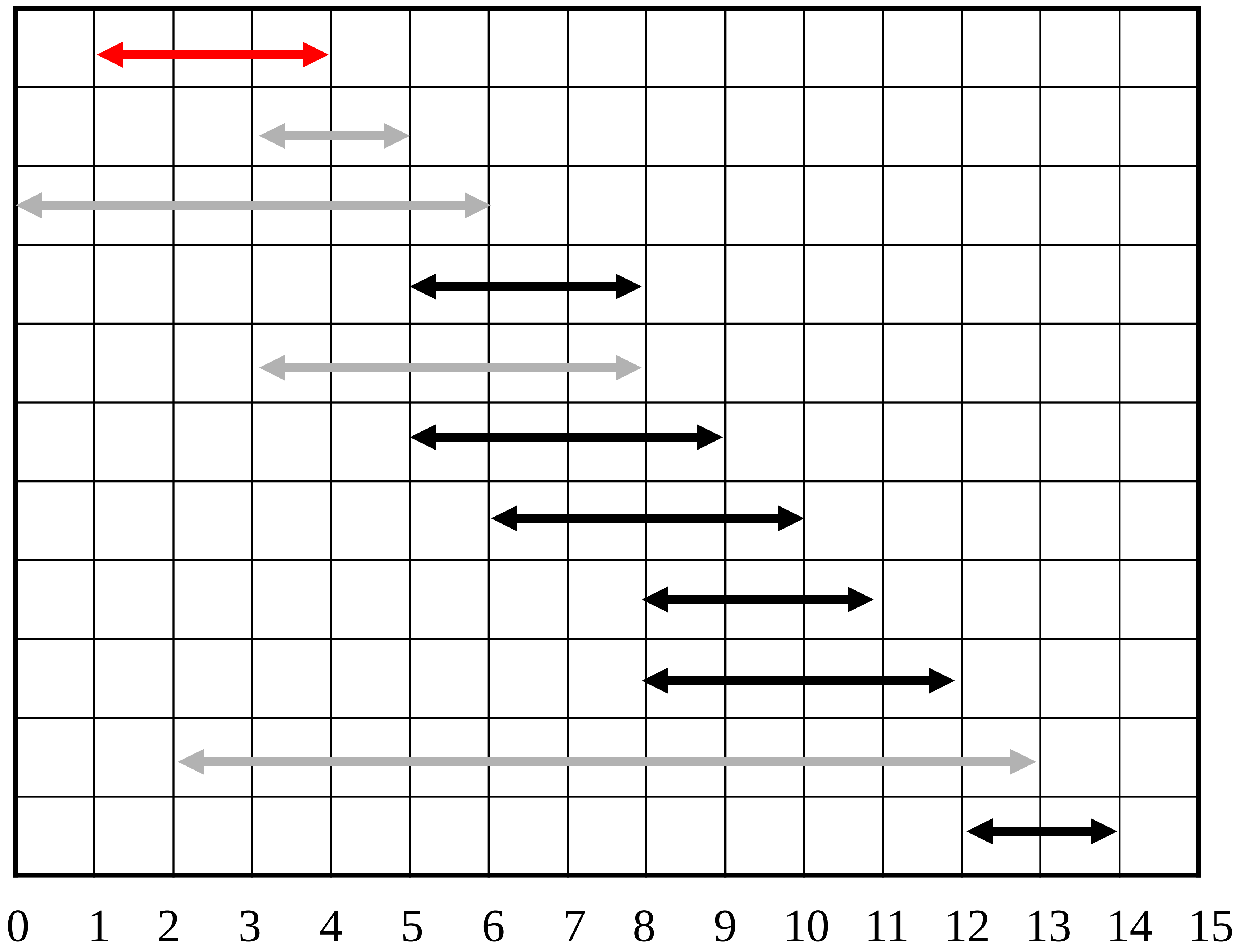
Counterexample for shortest interval

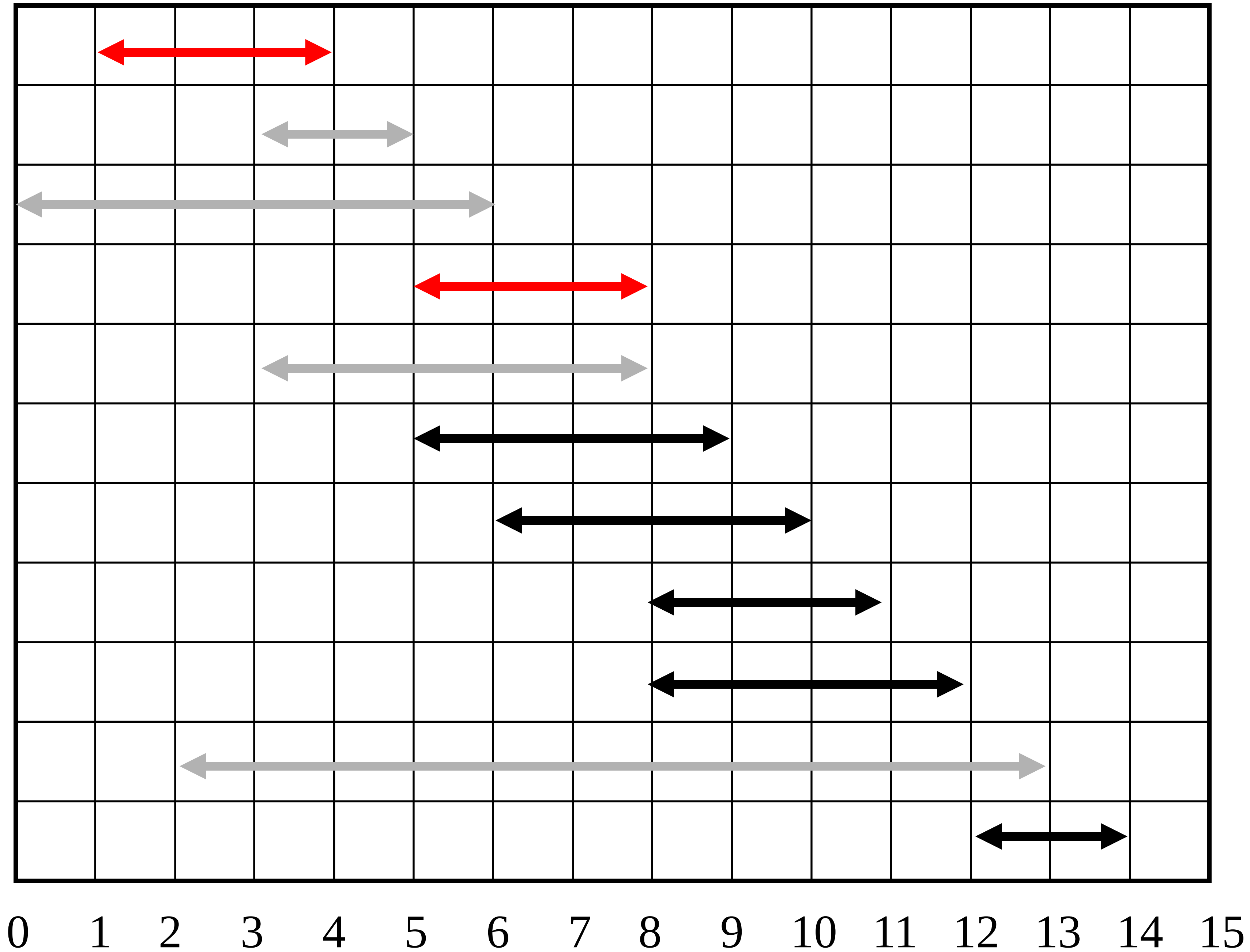
Counterexample for fewest conflicts

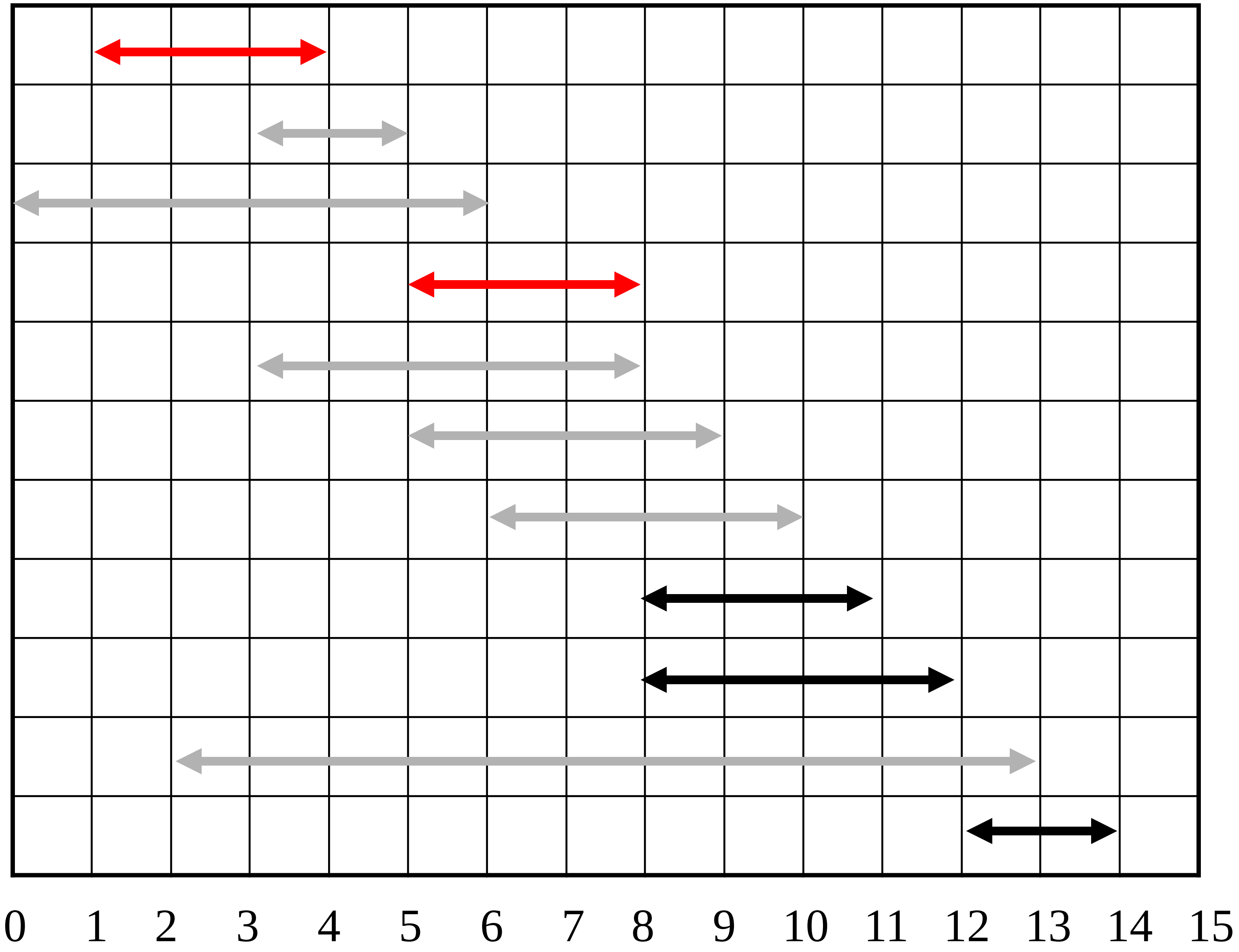
Earliest Finish Greedy Strategy

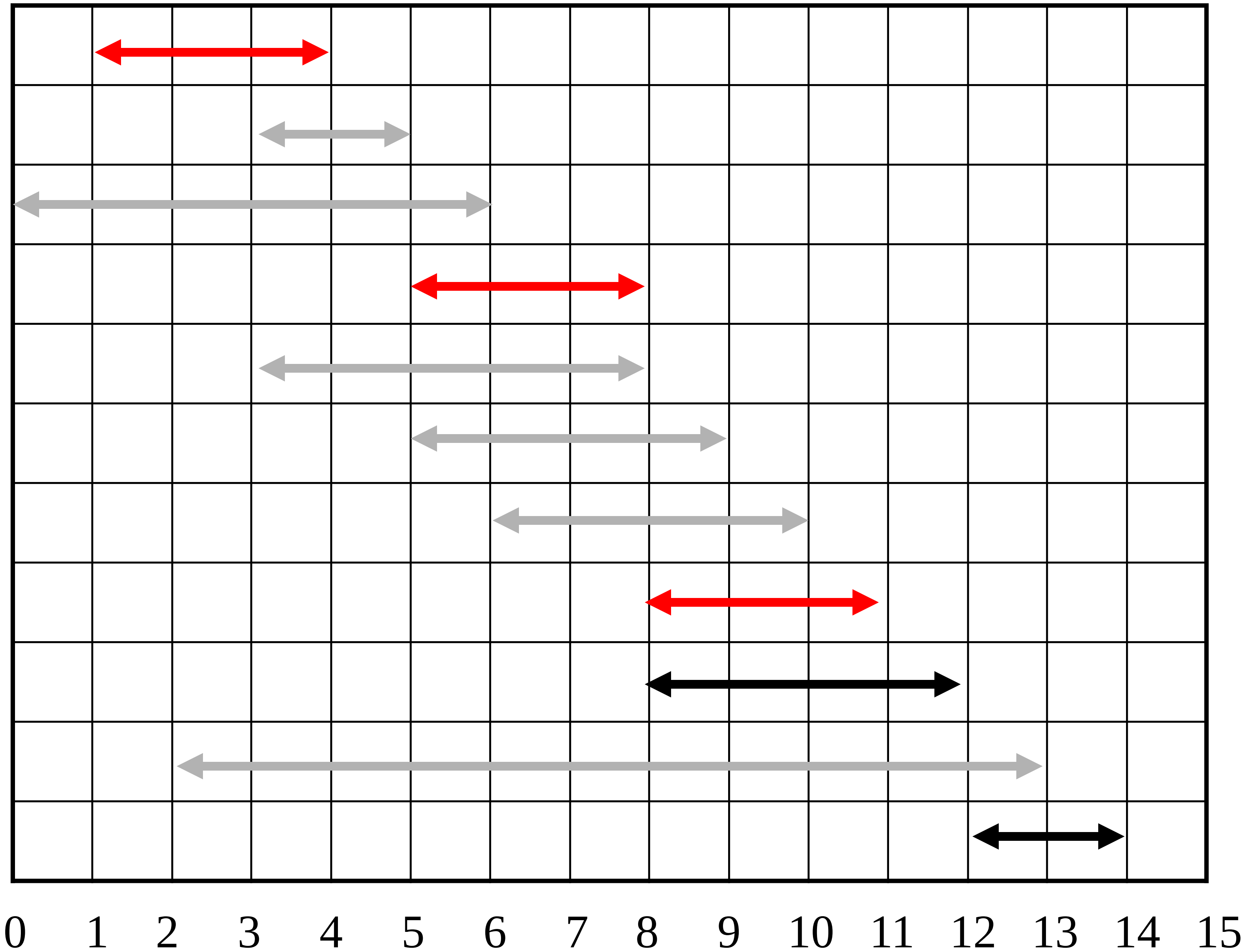
- Select the activity with the earliest finish
- Eliminate the activities that could not be scheduled
- Repeat!
- Greedy in the sense that it leaves as much opportunity as possible for the remaining activities to be scheduled
- The greedy choice is the one that maximizes the amount of unscheduled time remaining

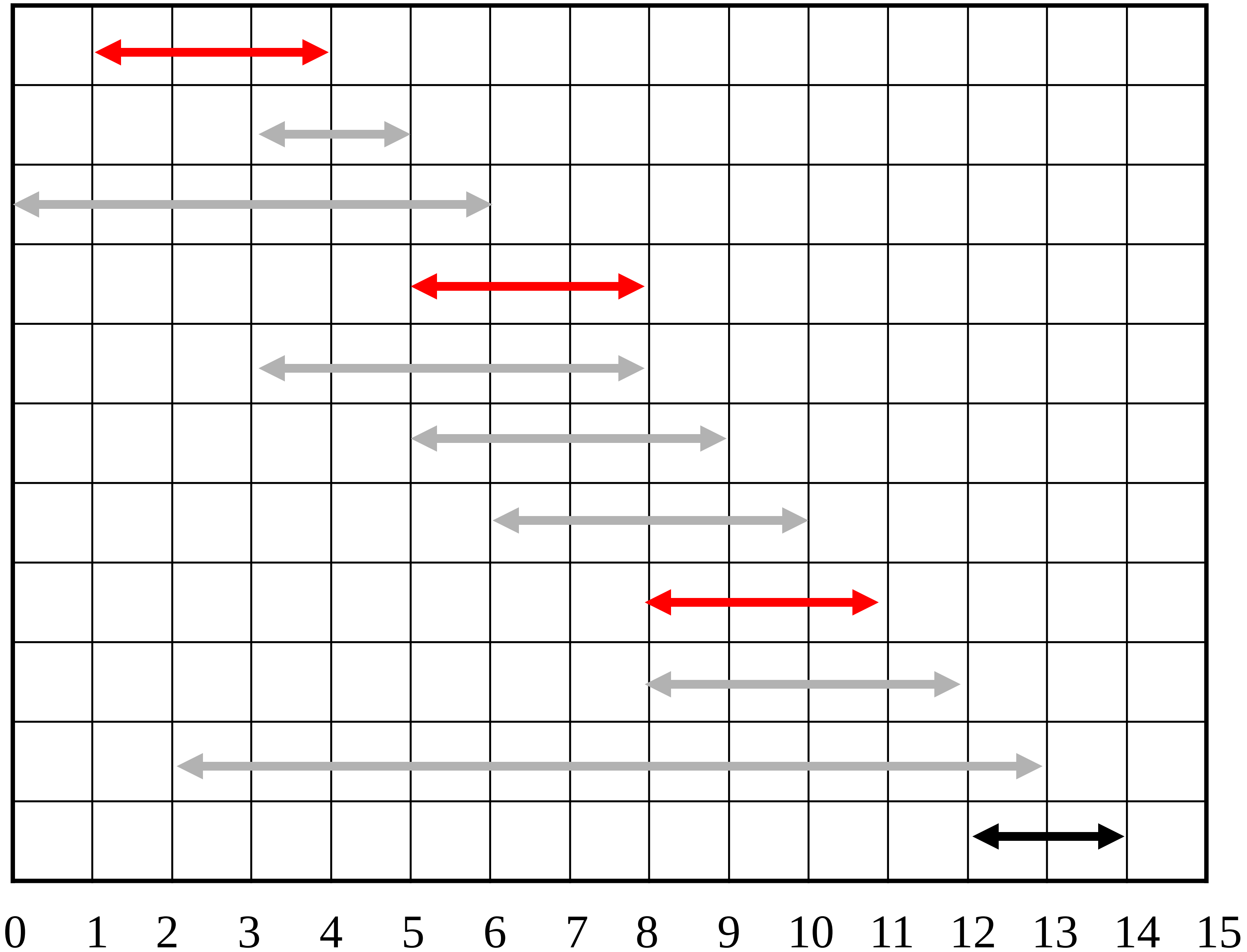

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

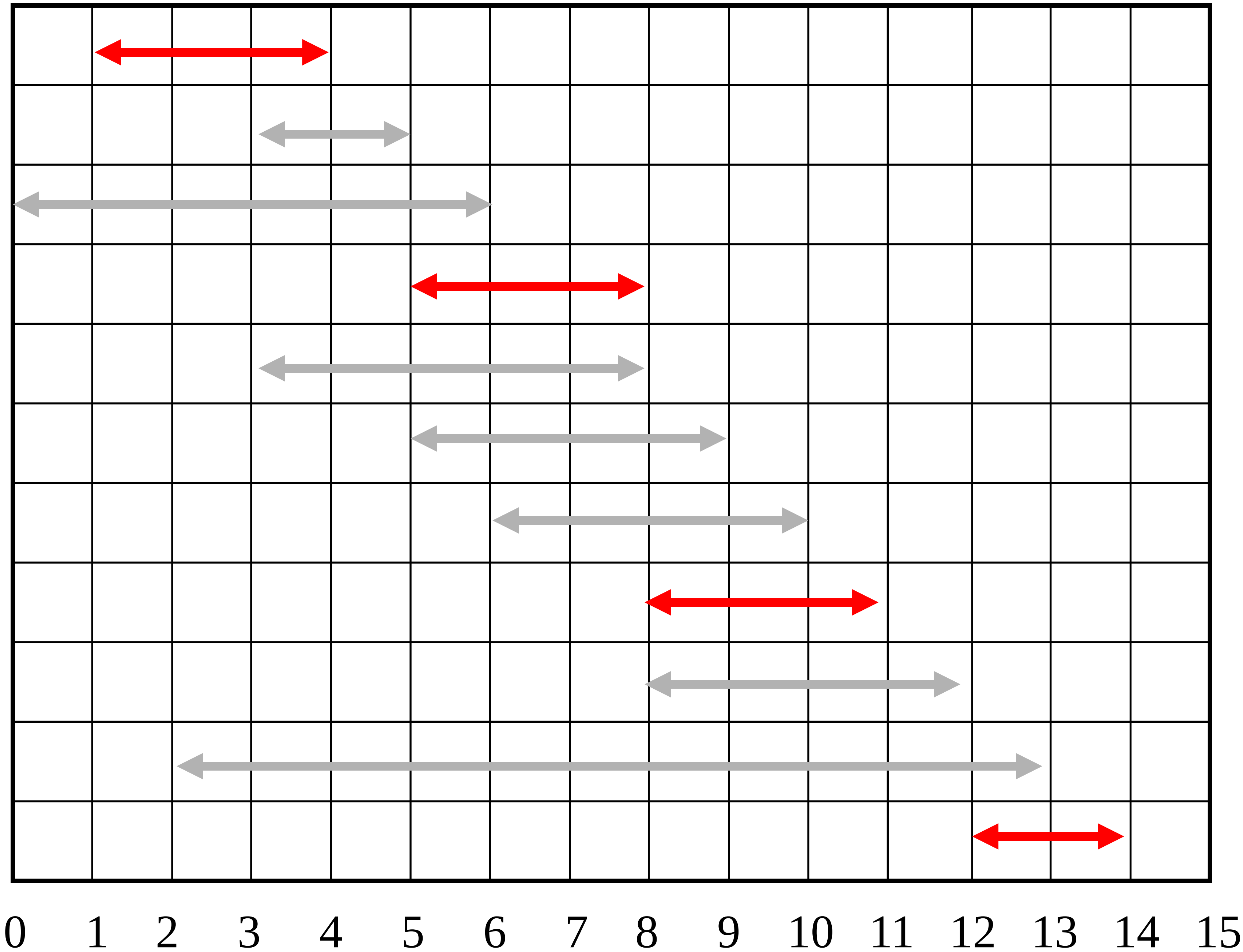

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Assuming activities are sorted by finish 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

Why this Algorithm is Optimal?

We will show that this algorithm uses the following properties

- The problem has the optimal substructure property
- The algorithm satisfies the greedy-choice property

Thus, it is Optimal

Greedy-Choice Property

- Show there is an optimal solution that begins with a greedy choice (with activity 1, which as the earliest finish time)
- Suppose $A \subseteq S$ in an optimal solution
  - Order the activities in A by finish time. The first activity in A is k
    - If k = 1, the schedule A begins with a greedy choice
    - If $k \neq 1$, show that there is an optimal solution B to S that begins with the greedy choice, activity 1
  - Let $B = A - \{k\} \cup \{1\}$
    - $f_1 \leq f_k \rightarrow$ activities in B are disjoint (compatible)
    - B has the same number of activities as A
    - Thus, B is optimal

Greedy-Choice Property

- A globally optimal solution can be arrived at by making a locally optimal (greedy) choice
  - Make whatever choice seems best at the moment and then solve the sub-problem arising after the choice is made
  - The 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 sub-problems
- Of course, we must prove that a greedy choice at each step yields a globally optimal solution

Elements of Greedy Strategy

- An greedy algorithm makes a sequence of choices, each of the choices that seems best at the moment is chosen
  - NOT always produce an optimal solution
- Two ingredients that are exhibited by most problems that lend themselves to a greedy strategy
  - Greedy-choice property
  - Optimal substructure

Optimal Substructures

A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to sub-problems

Optimal Substructures

Once the greedy choice of activity 1 is made, the problem reduces to finding an optimal solution for the activity-selection problem over those activities in S that are compatible with activity 1

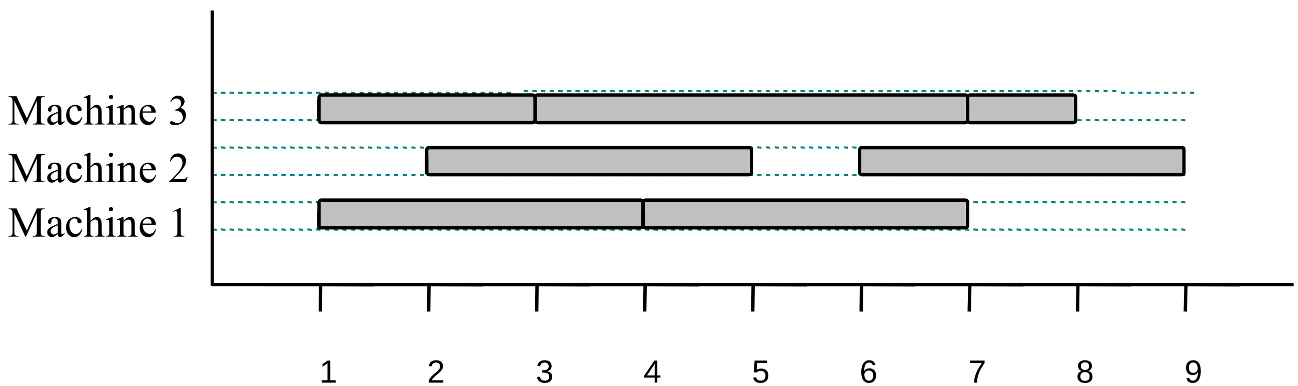
- If A is optimal to S, then $A' = A - \{1\}$ is optimal to $S' = \{i \in S: s_i \geq f_1\}$
- If we could find a solution B' to S' with more activities than A', adding activity 1 to B' would yield a solution B to S with more activities than A ➔ contradicting the optimality of A

After each greedy choice is made, we are left with an optimization problem of the same form as the original problem

- By induction on the number of choices made, making the greedy choice at every step produces an optimal solution

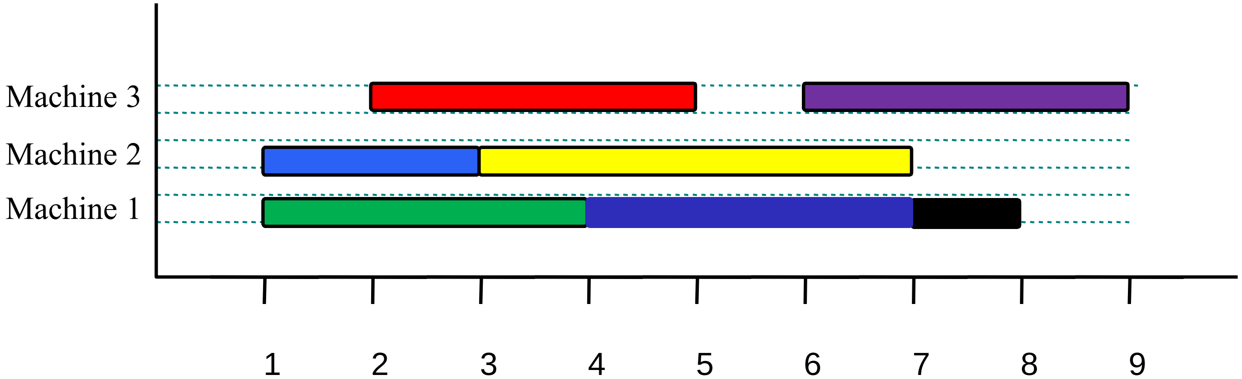
Machine Scheduling with start times

- Given: a set T of n tasks, each having:
  - A start time, s_i
  - A finish time, f_i (where $s_i < f_i$)
- Goal: Perform all the tasks using a minimum number of "machines."



Example

- Given: a set T of n=7 tasks, each having:
  - A start time, s_i
  - A finish time, f_i (where $s_i < f_i$)
  - [1,4], **[1,3]**, [2,5], [3,7], **[4,7]**, [6,9], [7,8] (ordered by start)
- Goal: Perform all tasks on min. number of machines



Machine Scheduling Algorithm

- **Greedy choice**: consider tasks by their start time and use as few machines as possible with this order.
  - Run time: Θ(n log n).
- **Correctness:** Suppose there is a better schedule.
  - We can use k-1 machines
  - The algorithm uses k
  - Let i be first task scheduled on machine k
  - Task i must conflict with k-1 other tasks
  - K mutually conflict tasks
  - But that means there is no non-conflicting schedule using k-1 machines

Algorithm TaskSchedule(*T*)

    Input: set ***T*** of tasks w/ start time s_i and finish time f_i

    Output: non-conflicting schedule with minimum number of machines

    $\boldsymbol{m} \leftarrow 0$ {no. of machines}

    while ***T is not empty***

        remove task i w/ smallest s_i

        if ***there's a machine j for*** **i then**

            schedule i on machine j

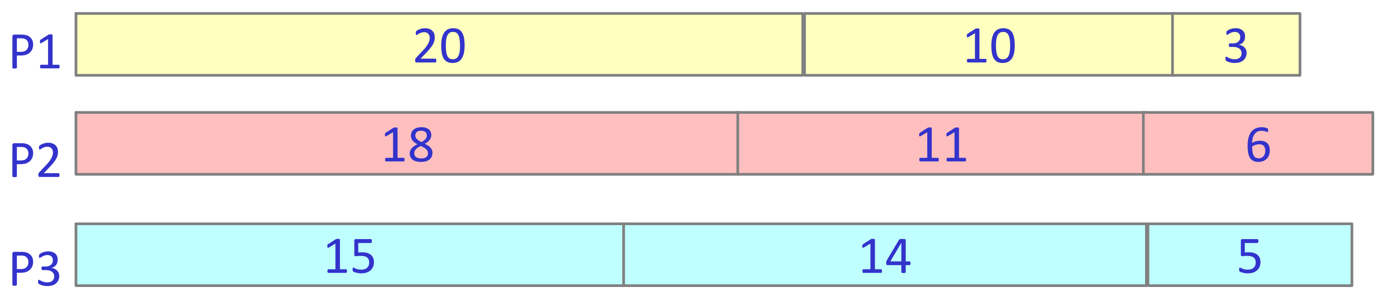
        else

            $\boldsymbol{m} \leftarrow \boldsymbol{m} + 1$

            schedule i on machine m

Job Scheduling Problem

- There is no specified start times only durations.
- You have to run nine jobs, with running times of 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes
- You have three processors on which you can run these jobs
- You decide to do the longest-running jobs first, on whatever processor is available

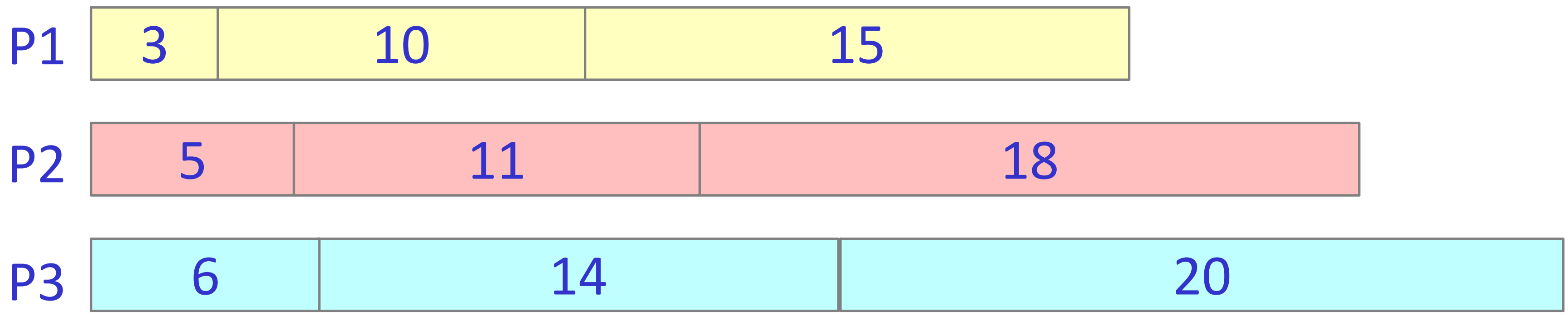


Time to completion: 18 + 11 + 6 = 35 minutes

This solution isn't bad, but we might be able to do better

Another approach

- What would be the result if you ran the *shortest* job first?
- Again, the running times are 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes

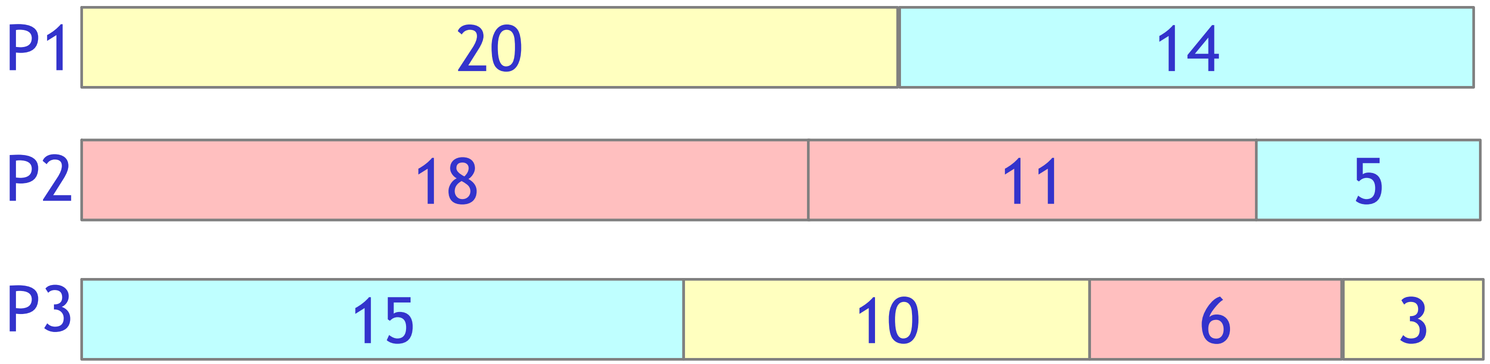


That wasn’t such a good idea; time to completion is now
6 + 14 + 20 = 40 minutes

Note, however, that the greedy algorithm itself is fast

- All we had to do at each stage was pick the minimum or maximum

An optimum solution



- This solution is clearly optimal (why?)
- Clearly, there are other optimal solutions (why?)
- How do we find such a solution?
  - One way: Try all possible assignments of jobs to processors
  - Unfortunately, this approach can take exponential time