Greedy Algorithms

- Knapsack
- Scheduling
- Huffman Code
- Coin Change

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

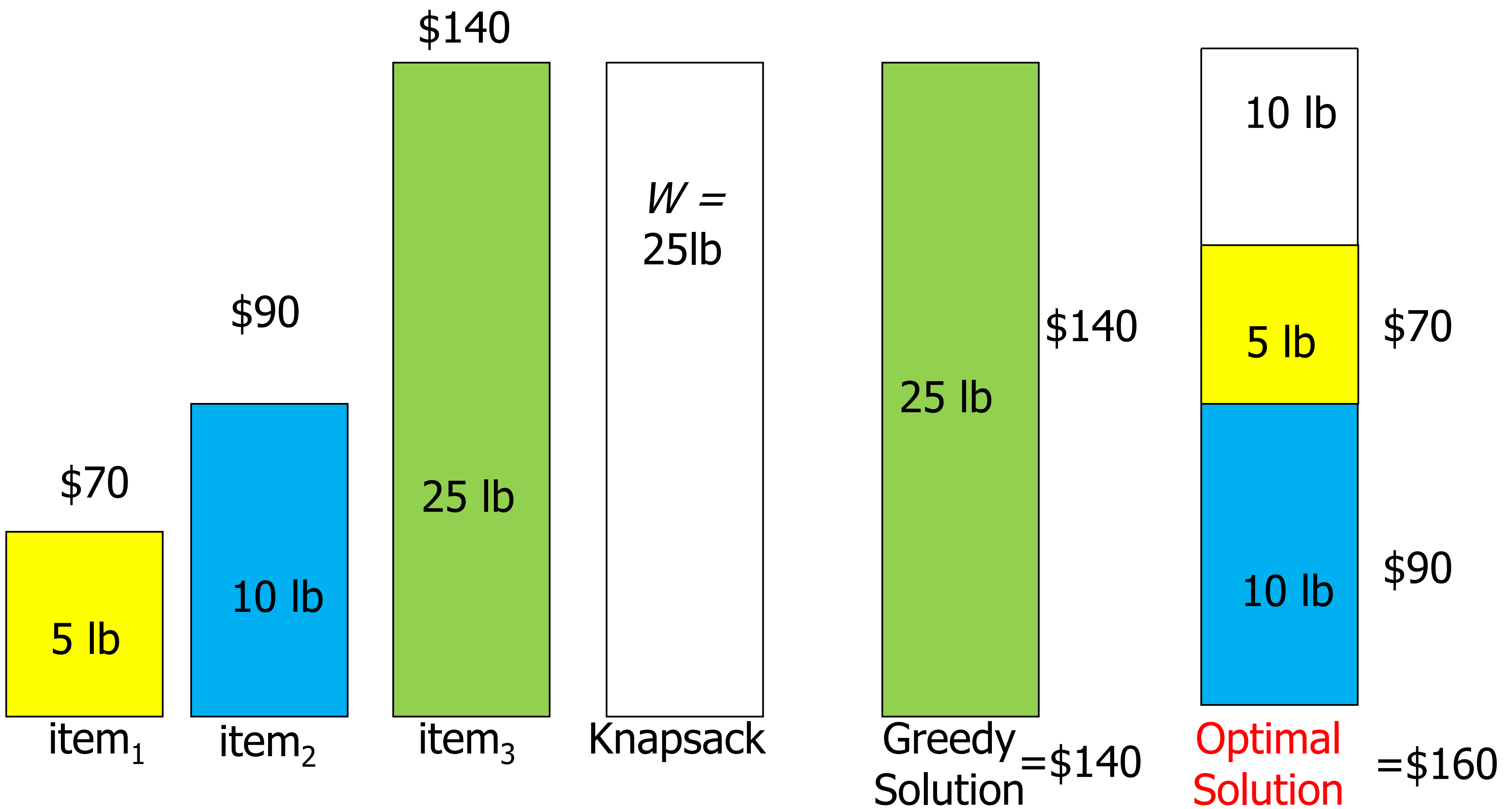
- Knapsack
- Data compression
  - Huffman coding
- Scheduling
  - Activity Selection
  - Task Scheduling
  - Minimizing time in system
  - Deadline scheduling
- Coin Change
- Graph Algorithms
  - Breath First Search (shortest path 4 un-weighted graph)
  - Dijkstra’s (shortest path) Algorithm
  - Minimum 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 values $v_i > 0$ for $i = 1,\ldots,n$.
- $S = \{ (item_1, w_1, v_1), (item_2, w_2, v_2), \ldots, (item_n, w_n, v_n) \}$
- Find a subset of the items which does not exceed the weight W of the knapsack and maximizes the value.

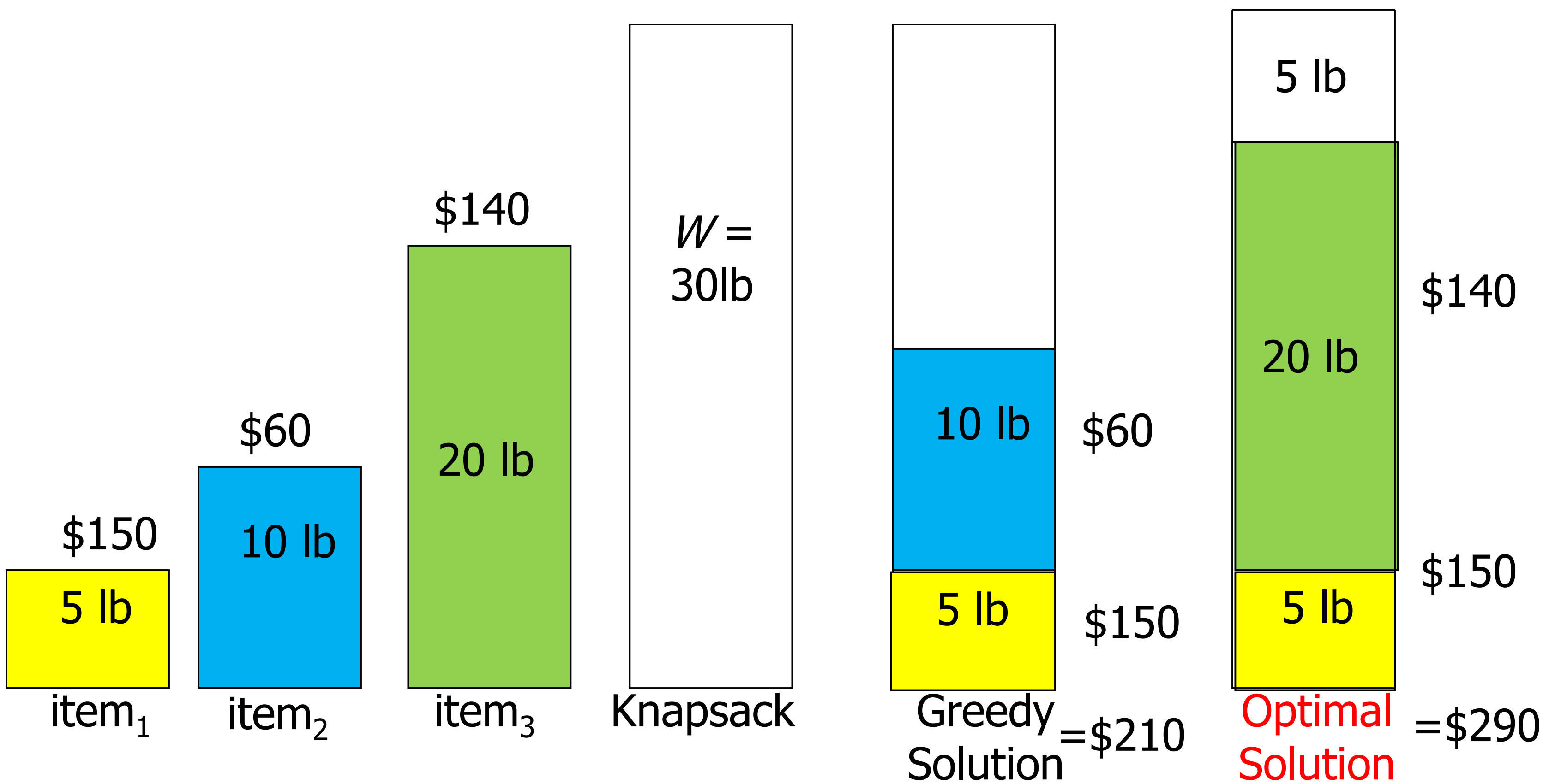
Greedy 1: Selection criteria: *Maximum valued* item.
Counter Example:

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



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

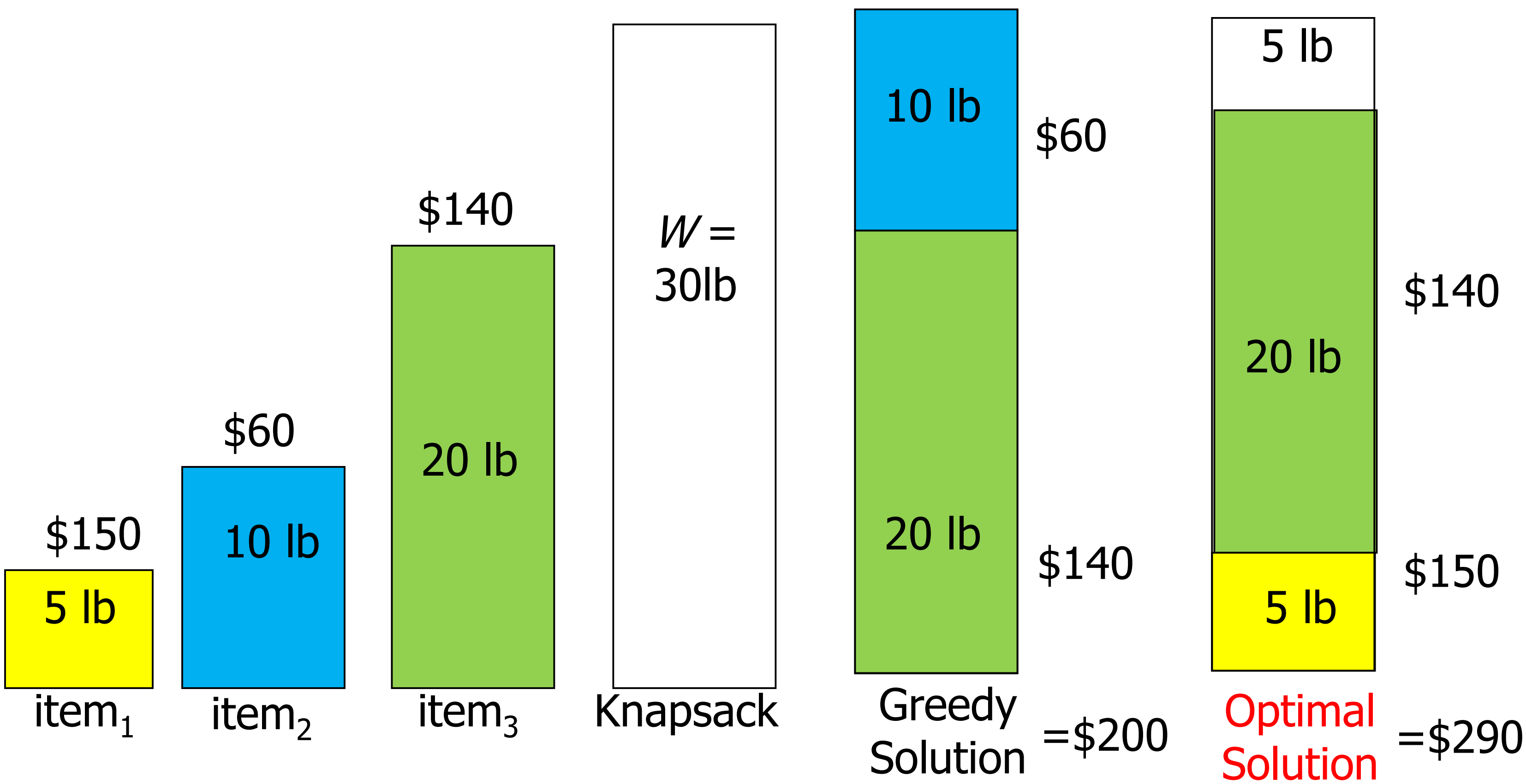
$S = \{ (item_1 , 5, \$150), (item_2 ,10, \$60), (item_3, 20, \$140) \}$



Greedy 3: Selection criteria: *Maximum weight* item

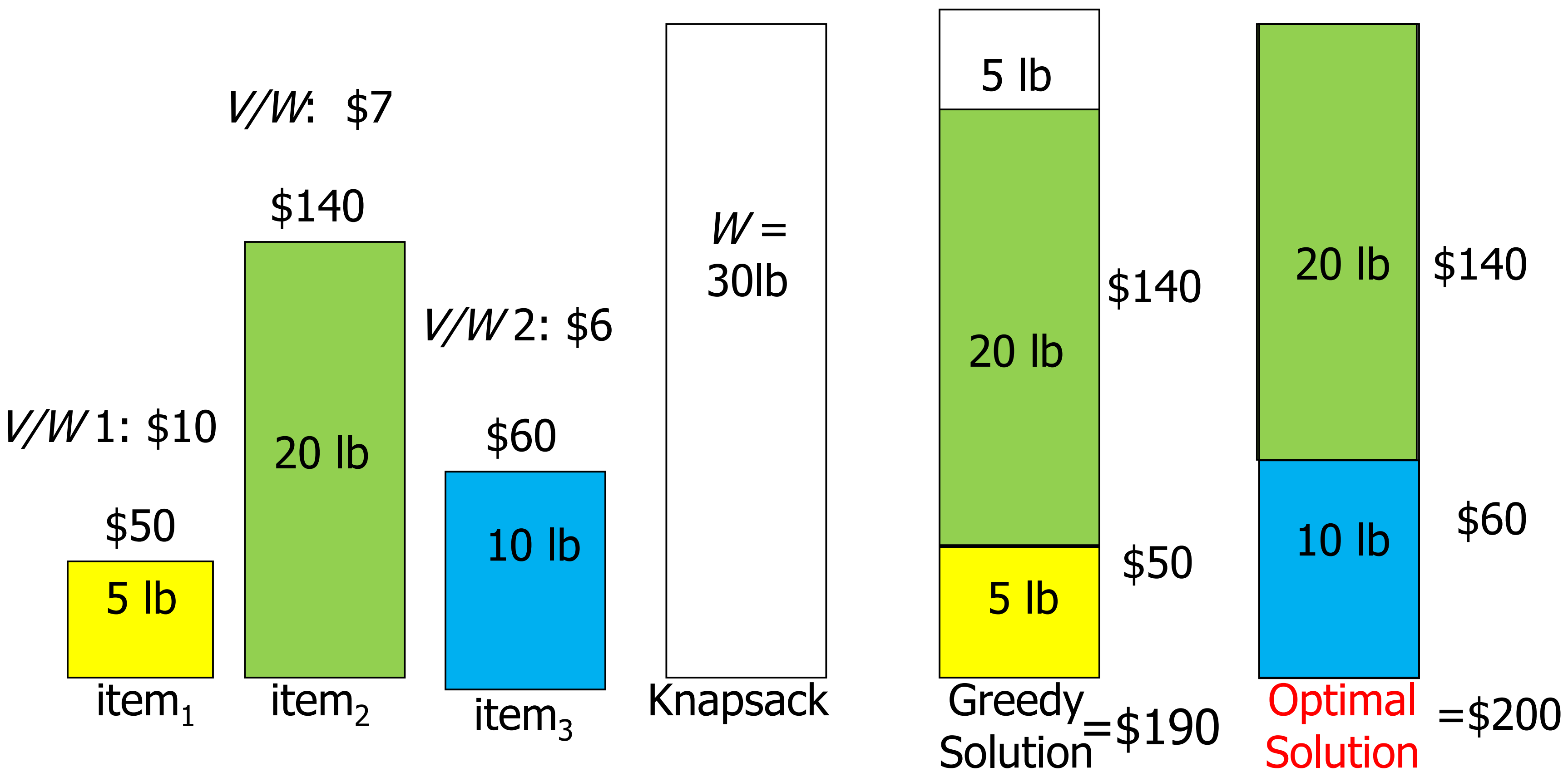
Counter Example:

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



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

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



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 = $\sum_{i=1}^{k-1} v_i$ + min{(***W*** - *totweight),*w_k } X (v_k / w_k)

min{(***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 ratio v_i / w_i for $1 \leq i \leq n$ $\Theta(n)$
2. Sort items in decreasing v_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** 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/ value v_i and weight w_i; max. weight W

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

for ***each item i in S***

$\quad x_i \leftarrow 0$

$\quad r_i \leftarrow v_i / w_i$ {ratio}

$w \leftarrow 0$ {current total weight}

while $w < W$

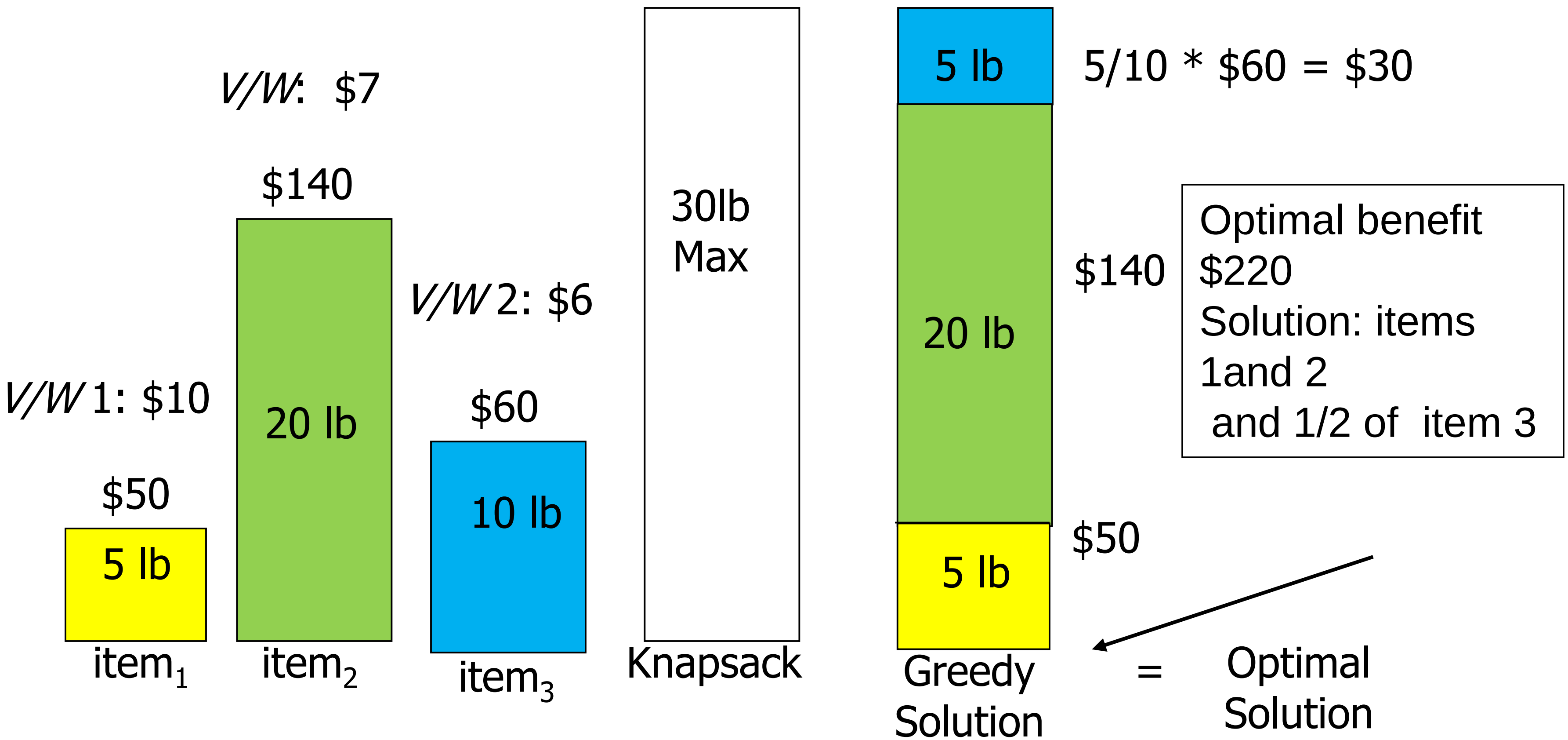
$\quad$ remove item i with highest r_i

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

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

Example of applying the optimal greedy algorithm for Fractional Knapsack Problem

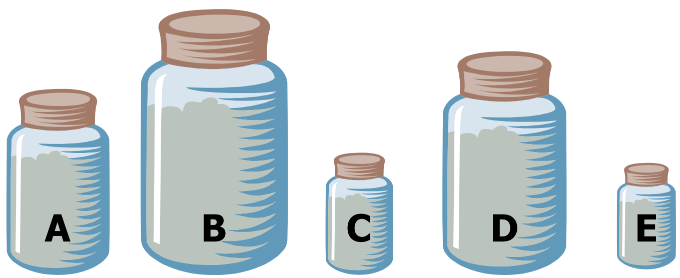
$S = \{ (item_1 , 5, \$50), (item_2, 20, \$140) (item_3 ,10, \$60), \}$



Fractional Knapsack

- Goal: Choose items with maximum total value but with weight at most W. You can now use fractions of items

Items:



	A	B	C	D	E
Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Value:	$24	$24	$30	$30	$7

"knapsack"

10 ml

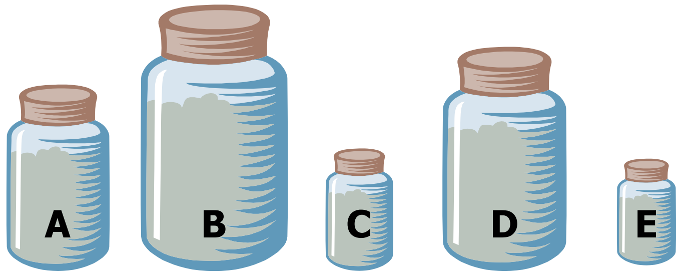
What is the maximum value?

a) 84
b) 76
c) 54
d) 67
e) None of the above

Fractional Knapsack

- Goal: Choose items with maximum total value but with weight at most W.

Items:



Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Value:	$24	$24	$30	$30	$7
Ratio: ($ per ml)	6	3	15	5	7

10 ml

"knapsack"

Solution:

- 2 ml of C - $30
- 1 ml of E - $7
- 4 ml of A -$24
- 3 ml of D - $15
- ________________

10 ---- $ 76

Fractional Knapsack has greedy choice property

That is, if v_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 $v_i/w_i > v_t/w_t$ or be an alternative maximum solution if $v_i/w_i = v_t/w_t$

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 = $W \leq w_i$. 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.

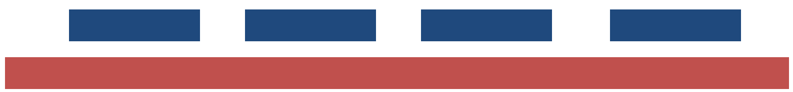
Activity Scheduling: Greedy Algorithms

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

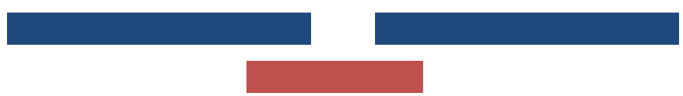
- [Earliest start time] Consider activities in ascending order of s_j.
- [Earliest finish time] Consider activities in ascending order of f_j.
- [Shortest interval] Consider activities in ascending order of $f_j - s_j$.
- [Fewest conflicts] For each activity j, count the number of conflicting activities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

Greedy Algorithms are not always Optimal

Counterexample for earliest start time



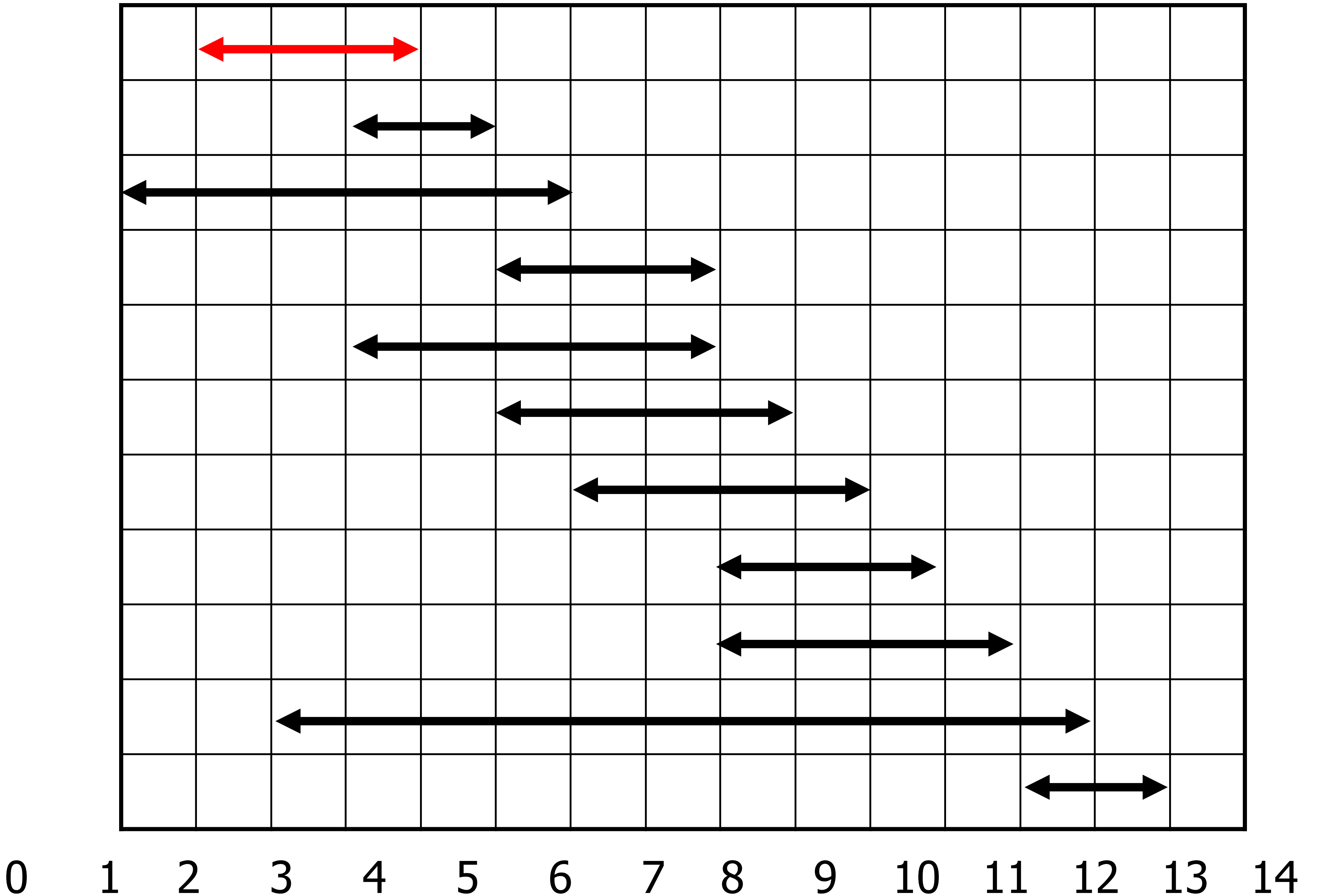
Counterexample for shortest interval

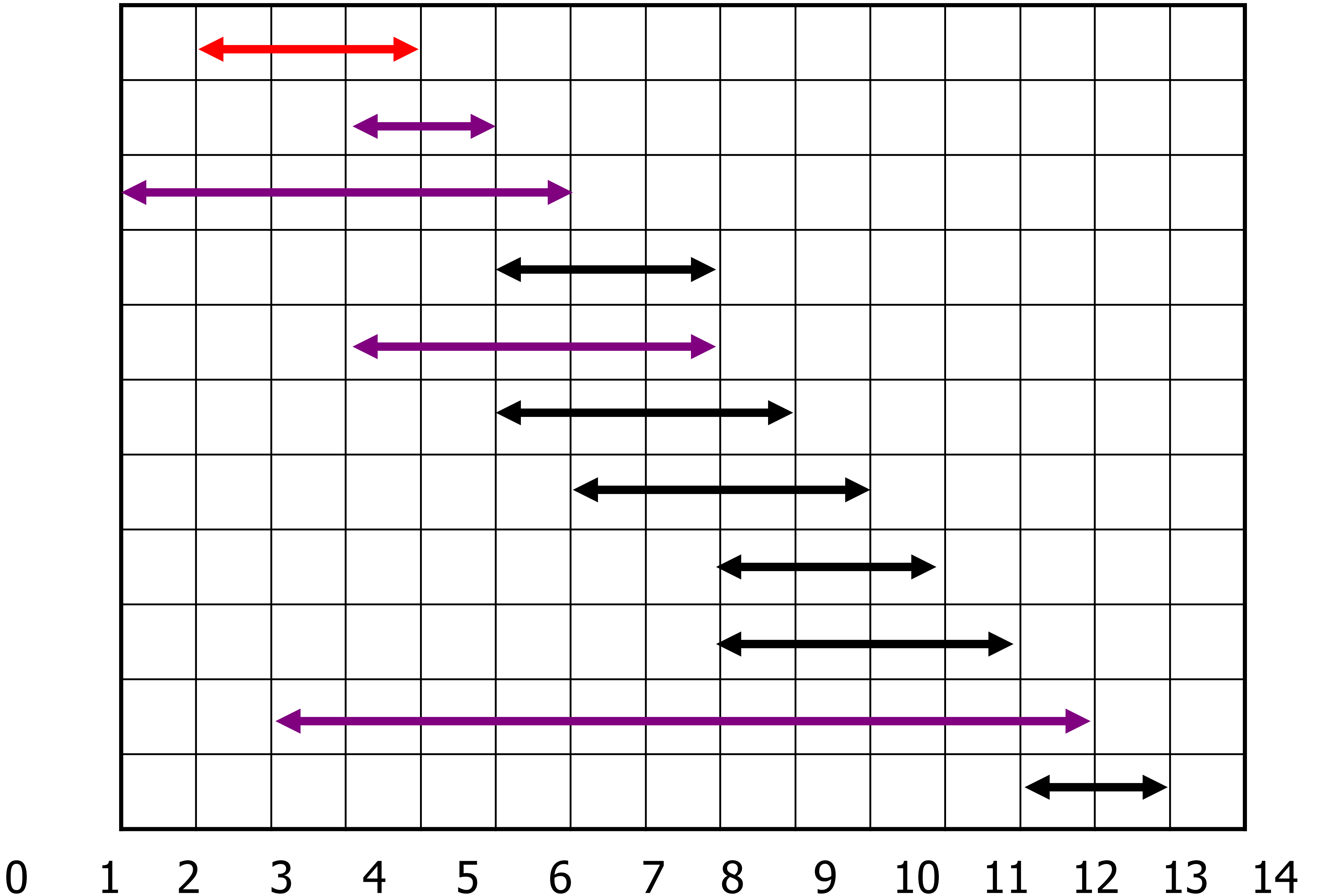


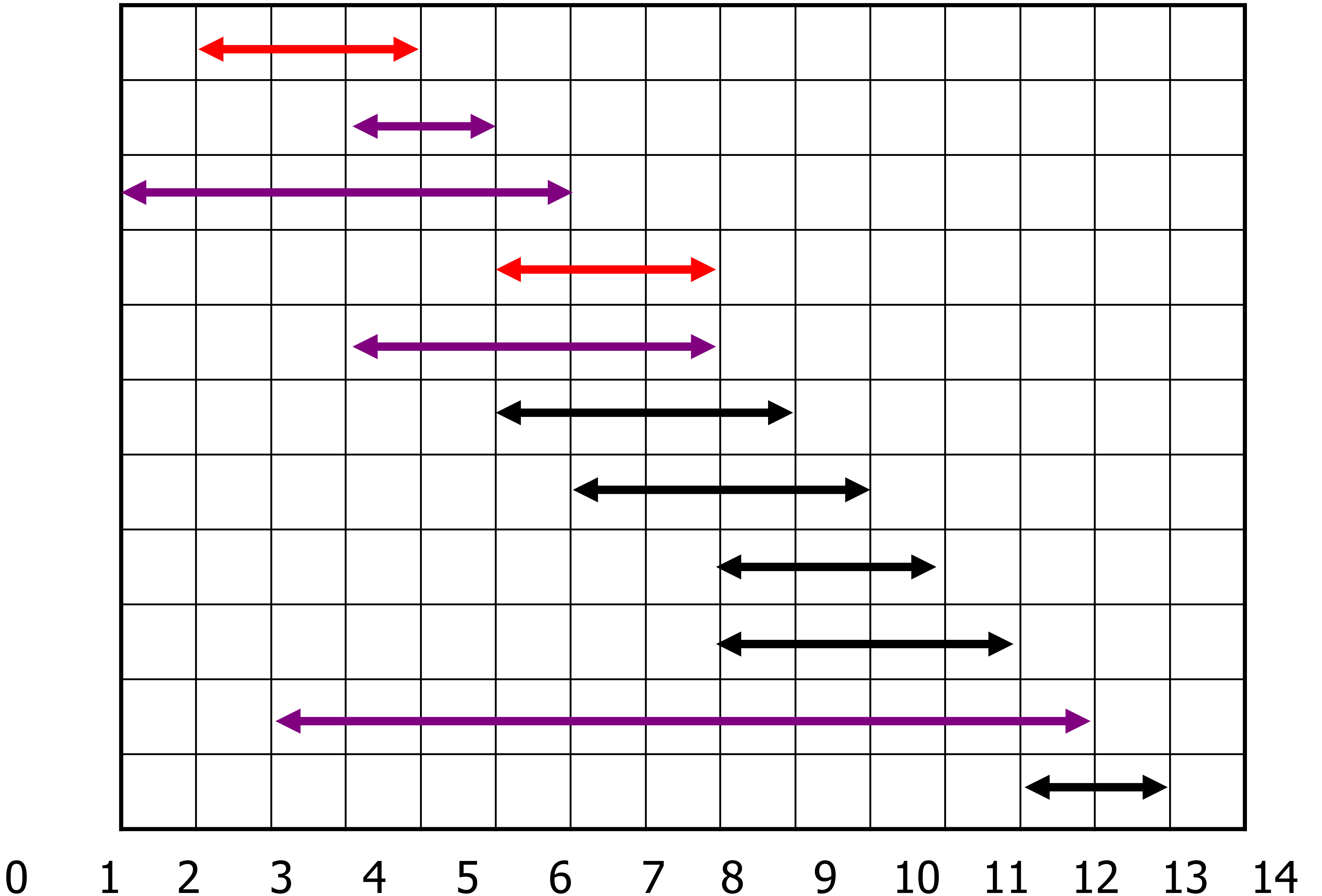
Counterexample for fewest conflicts

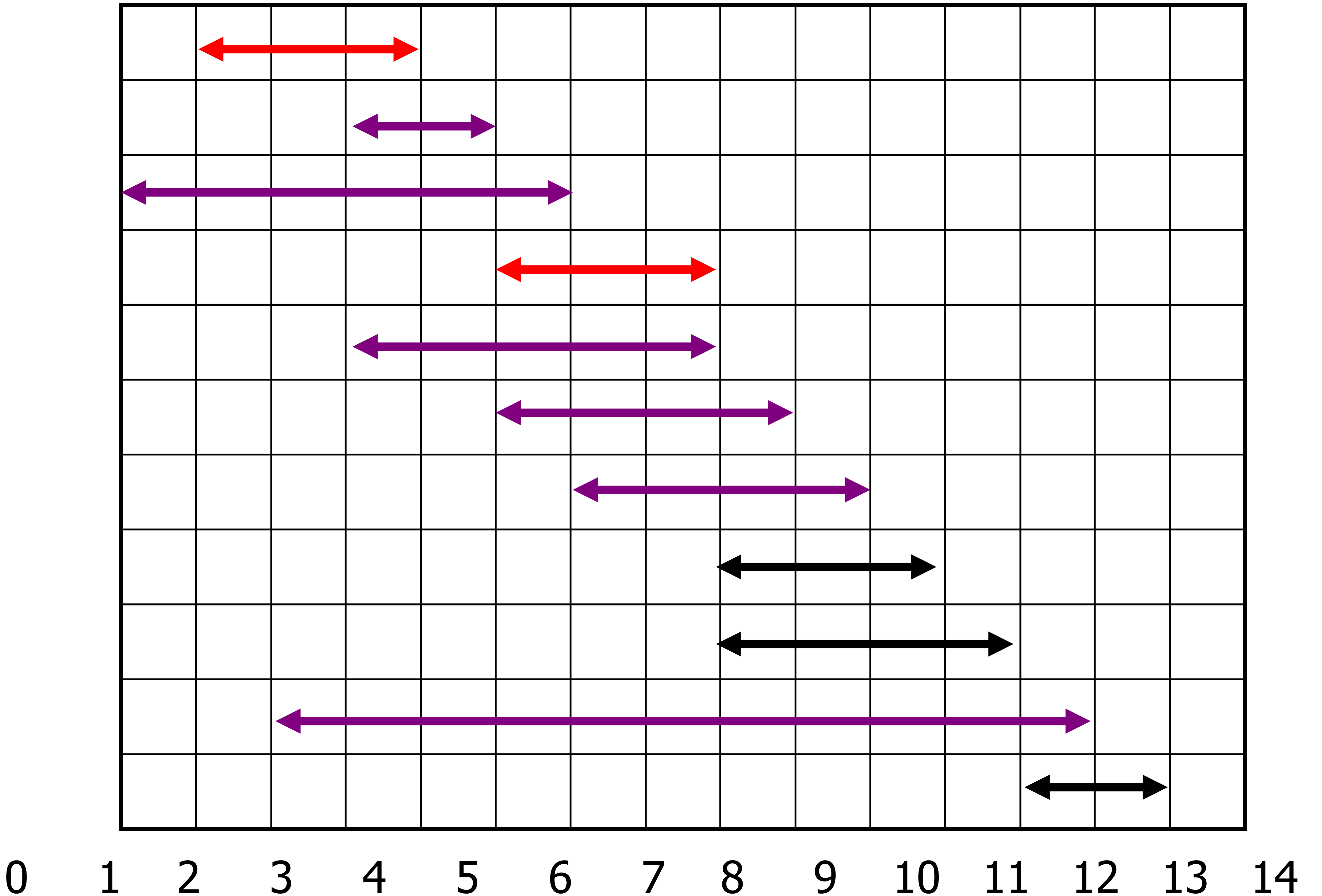
Earliest Finish Greedy Strategy

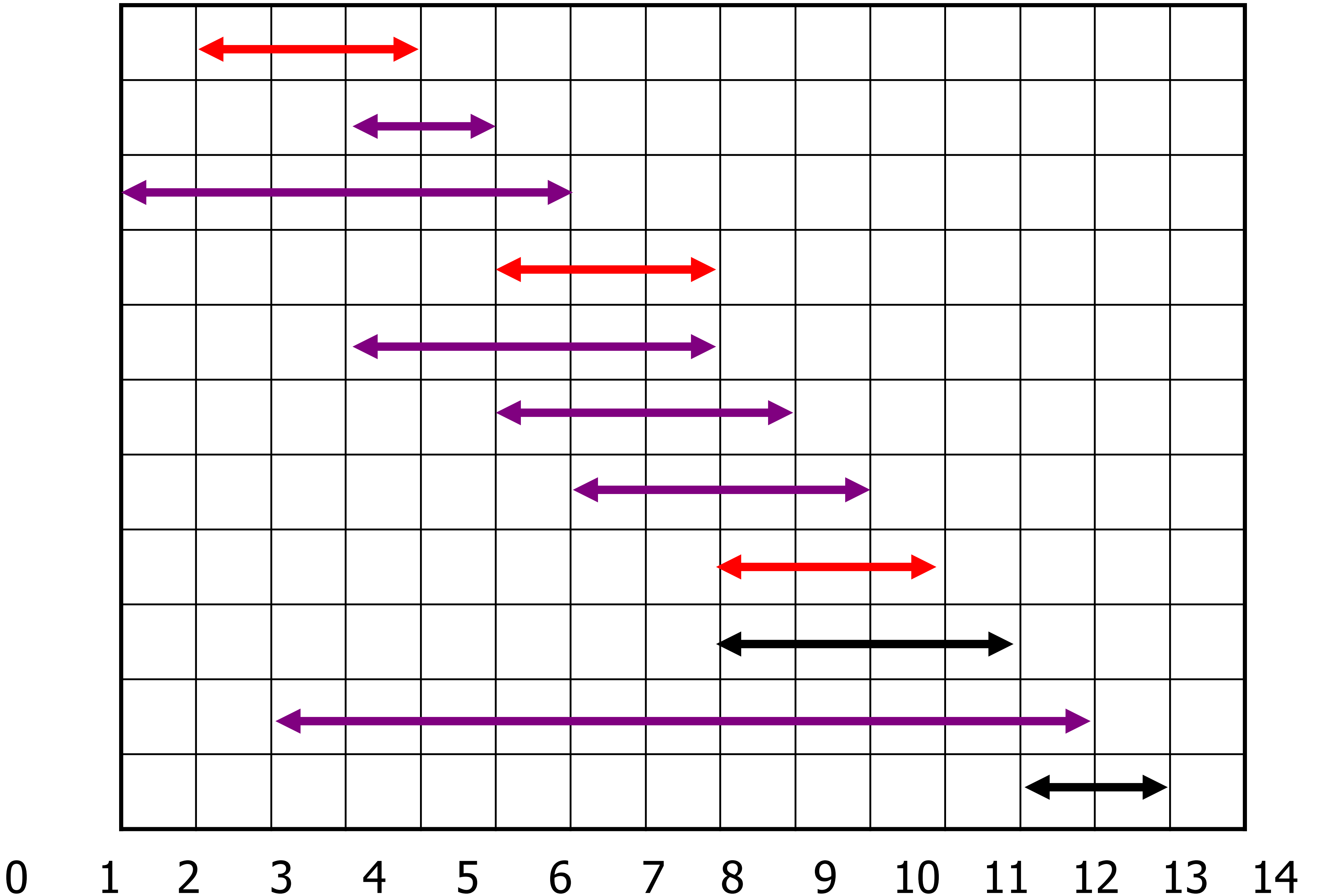
- Select the activity with the earliest finish
- Eliminate the activities that could not be scheduled
- Repeat!
- Greedy in the sense that it leaves as much opportunity as possible for the remaining activities to be scheduled
- The greedy choice is the one that maximizes the amount of unscheduled time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

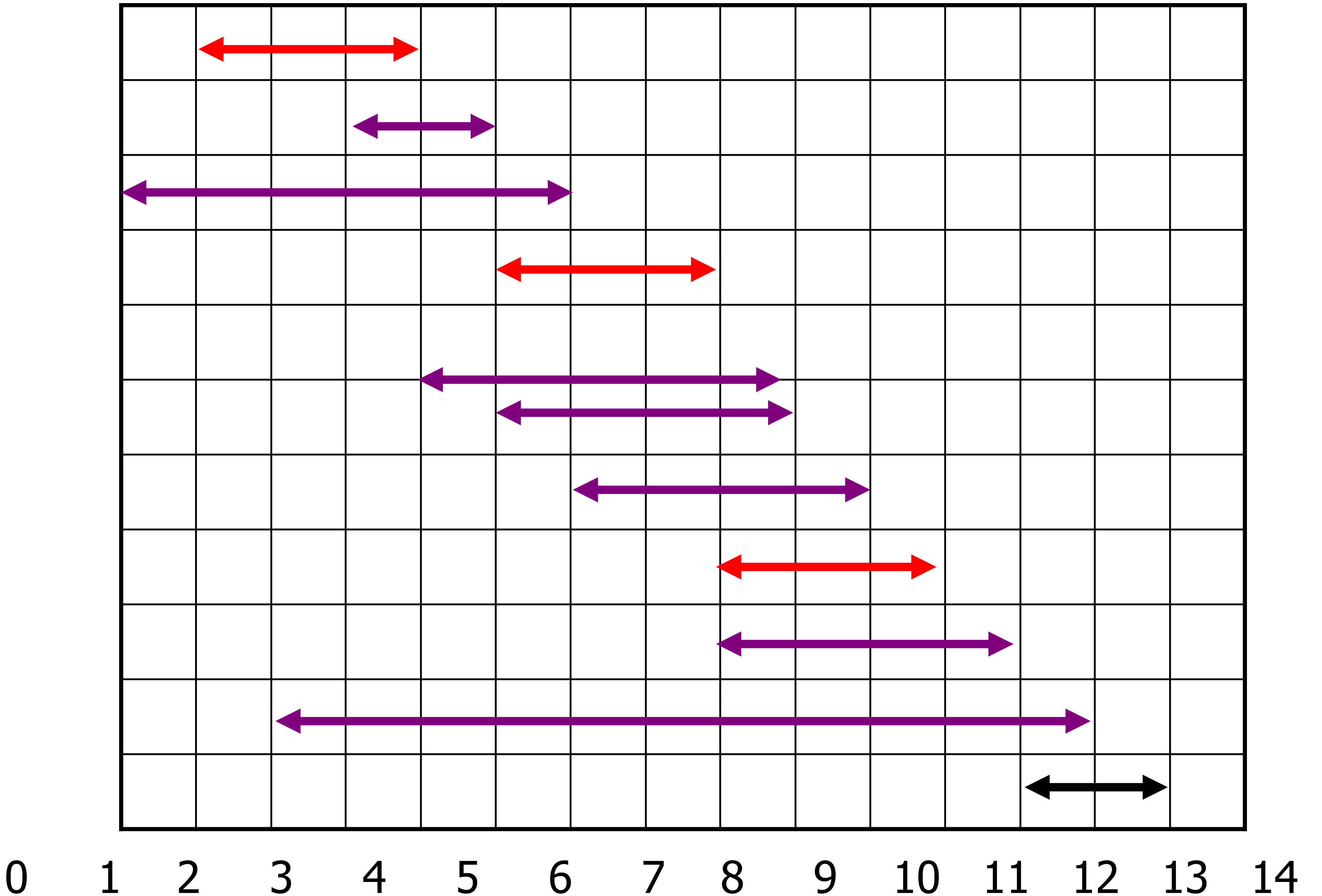

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

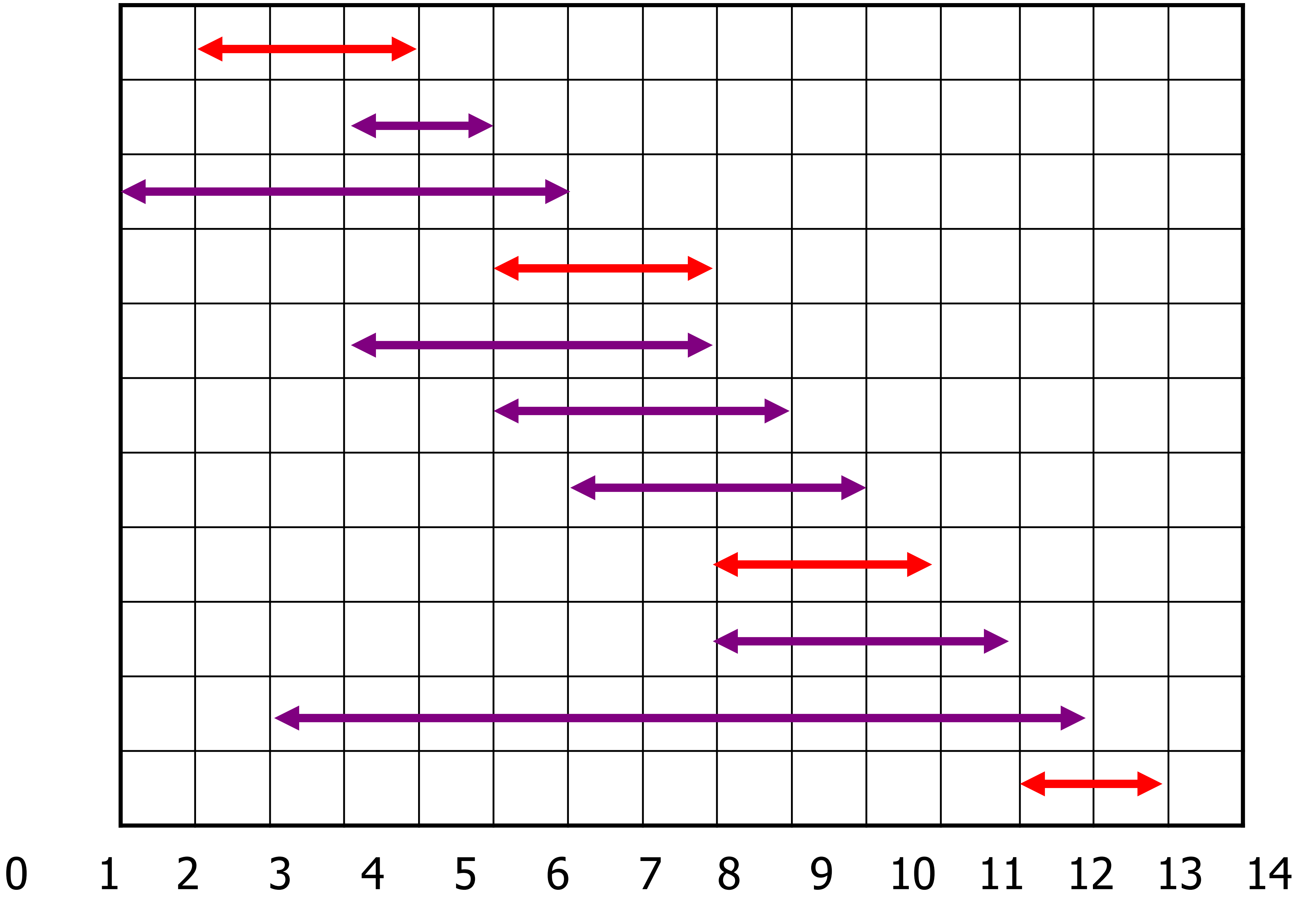

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14


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


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


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


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


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

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

a) Mergesort 3 faster
b) MergeSort faster
c) same

Given the activities below what is the maximum number of activities that can be scheduled?

Act #	1	2	3	4	5	6
Start	1	2	2	4	5	7
Finish	3	6	4	6	7	10

a) 1
b) 2
c) 3
d) 4
e) None of the above

Last-to-Start

Act #	1	2	3	4	5	6
Start	1	2	2	4	5	7
Finish	3	6	4	6	7	10

First-to-Finish

Act #	1	2	3	4	5	6
Start	1	2	2	4	5	7
Finish	3	6	4	6	7	10

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
  - Optimal substructure
  - Greedy-choice property

Optimal Substructures

Step 1: Characterize optimality

Without loss of generality, we will assume that the a's are sorted in non-decreasing order of finishing times, i.e. $f_1 \le f_2 \le \ldots \le f_n$.

Define the set S_{ij}

$S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$ as the subset of activities that can occur between the completion of a_i (f_i) and the start of a_j (s_j).

Note that $S_{ij} = \emptyset$ for $i \ge j$ since otherwise $f_i \le s_j < f_j \Rightarrow f_i < f_j$ which is a contradiction for $i \ge j$ by the assumption that the activities are in sorted order.

Optimal Substructures

Define the set S_{ij}
$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$ as the subset of activities that can occur between the completion of a_i (f_i) and the start of a_j (s_j).

Let A_{ij} be the *maximal* set of activities for S_{ij}. Using a "cut-and-paste" argument, if A_{ij} contains activity a_k then we can write
$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

where A_{ik} and A_{kj} must also be optimal (otherwise if we could find subsets with more activities that were still compatible with a_k then it would contradict the assumption that A_{ij} was optimal).

Step 2: Define the recursive solution (top-down)

Let c[i,j] = $|A_{ij}|$, then

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

i.e. compute *c[i,j]* for each *k* = *i*+1, ..., *j*-1 and select the max.

This would take exponential time!

Step 2: Define the recursive solution (top-down)

Let c[i,j] = $|A_{ij}|$, then

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

i.e. compute *c[i,j]* for each k = i+1, ..., j-1 and select the max.

Step 3: Compute the maximal set size (bottom-up)

Construct an nxn table which can be done in polynomial time since clearly for each *c[i,j]* we will examine no more than n subproblems giving an upper bound on the worst case of O(n^3).

BUT WE DON'T NEED TO DO ALL THAT WORK!

- Instead at each step we could simply select (*greedily*) the activity that finishes first and is compatible with the previous activities. Intuitively this choice leaves the most time for other future activities.

Greedy Algorithm Solution

- To use the greedy approach, we must *prove* that the greedy choice produces an optimal solution (although not necessarily the *only* solution).

Greedy Choice Property

To use the greedy approach, we must *prove* that the greedy choice produces an optimal solution (although not necessarily the *only* solution).

Let $S_k = \{a_i \in S_k : s_i \geq f_k\}$ be the set of activities that start after activity a_k finishes

Consider any non-empty subproblem S_k with activity a_m having the earliest finishing time, Then a_m included in some maximum-size subset of mutually compatible activities of S_k .

Greedy Choice Property

Note: If we make the greedy choice of a_1 then S_1 remains the only subproblem to solve. That is $S_{01} = \varnothing$.

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

then the following two conditions must hold

1. a_m is used in an optimal subset of S_{ij}

2. $S_{im} = \emptyset$ leaving S_{mj} as the only subproblem
meaning that the greedy solution produces an optimal solution.

Consider any non-empty subproblem S_k with activity a_m having the earliest finishing time, Then a_m included in some maximum-size subset of mutually compatible activities of S_k .

Let A_k be an optimal solution for S_k and a_j be the activity in A_k with the earliest finish time.

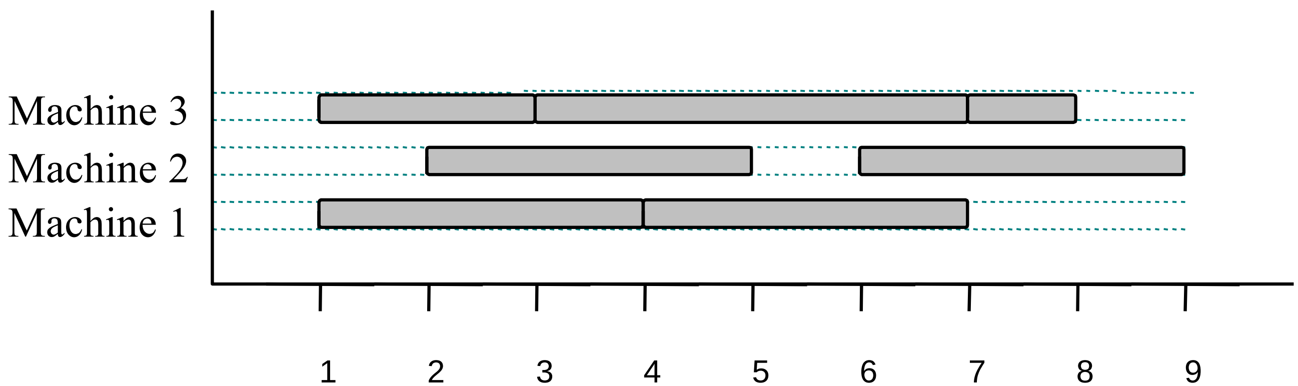
- If $a_j = a_m$ then the condition holds.
- If $a_j \neq a_m$ then construct $A_k' = A_k - \{a_j\} \cup \{a_m\}$.

Since $f_m \leq f_j \Rightarrow A_k'$ is still optimal.

The activities in are disjoint since the activites in are disjoint and $f_m \leq f_j$. Since $|A_k| = |A_k'|$, we conclude that A_k is a maximum-size subset of mutually compatible activates for S_k and include a_m.

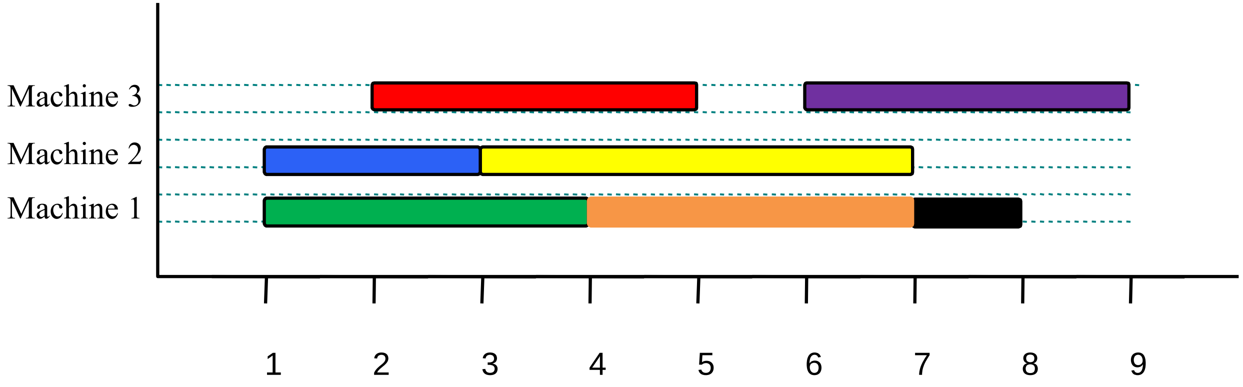
Machine Scheduling with start times

- 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: O(n log n) depending on data structure used.
- **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

$m \leftarrow 0$ {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

$m \leftarrow m + 1$

schedule i on machine m

To get O(log(n)) for each iteration, keep the machines in a **heap** using as key the latest finishing time assigned to that machine. This tells you when that machine will be free. When you select the next task i with start time s_i, and finishing time f_i check the min element of the machine heap say machine j with key fj indicate it is free at time fj

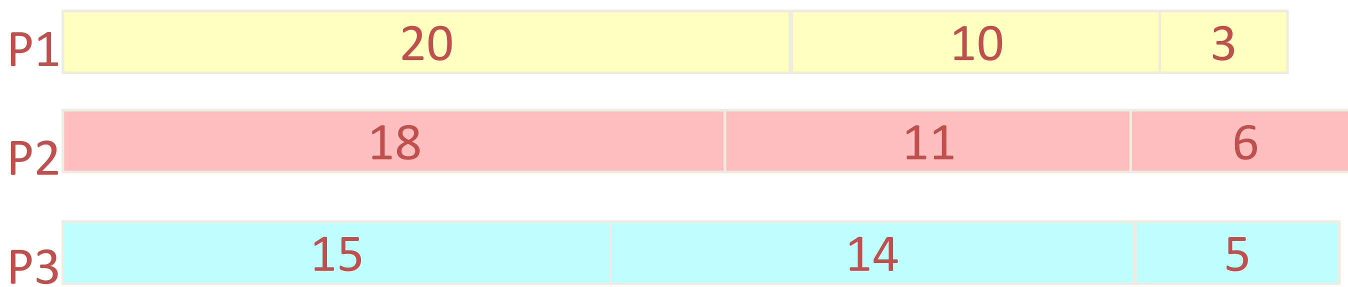
- If machine j is free at time s_i (s_i >= fj) then it is free forever starting at s_i. We can now
  1. Remove the machine j from the heap (removeMin), O(logn).
  2. Assign the current job to the removed machine j, Θ(1).
  3. Now machine j is free at f_i and we re-insert it into the heap, O(logn).

- If machine j is not free at s_i, then no machine is free at s_i since this was the minimum free time, so
  1. Increase m generating a new machine, Θ(1).
  2. Assign i to the new machine m, Θ(1).
  3. Insert machine m (which has key fi) into the heap, O(logn).

Note there are n iterations if n jobs that need to be scheduled on machines so O(nlgn) overall running time.

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

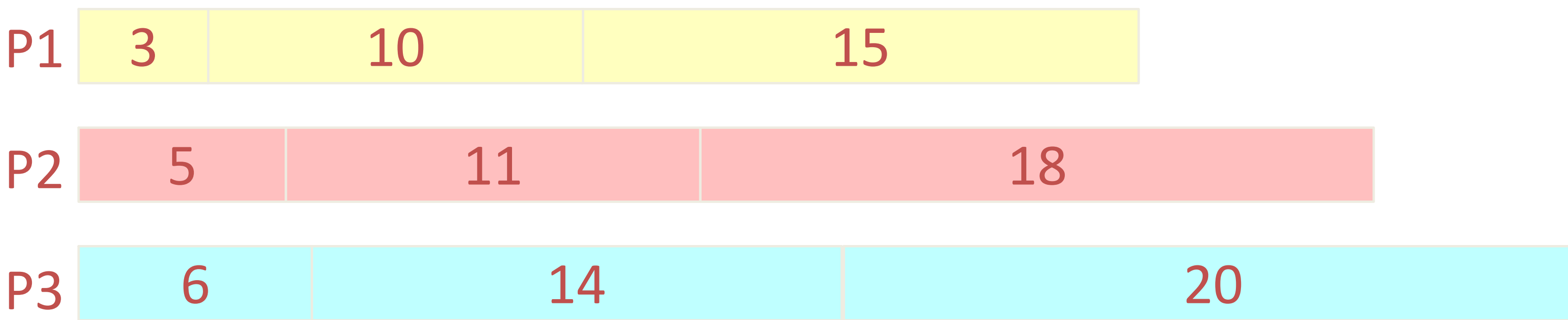


Time to completion: 18 + 11 + 6 = 35 minutes

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

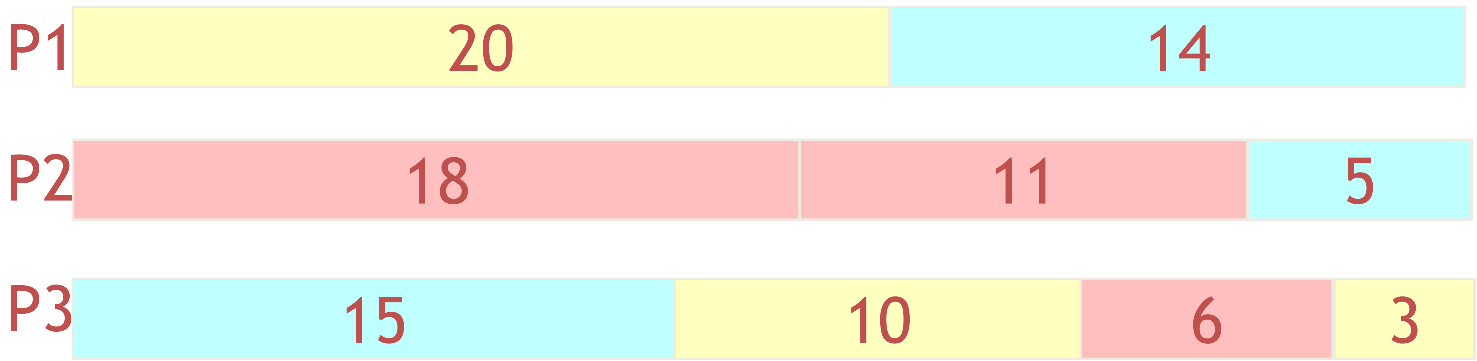


That wasn't such a good idea; time to completion is now
$6 + 14 + 20 = 40$ minutes

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

Announcements

- Quiz 4 – Due Sunday
- HW 4 – Due next Tuesday
- Finish Greedy Algorithms
- Questions on HW 4
- Weeks 6 & 7 Graph Algorithms

Huffman Codes

Text Compression (Zip)

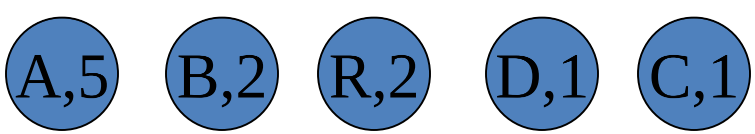
- On a computer: changing the representation of a file so that it takes less space to store or/and less time to transmit.
- Original file can be reconstructed exactly from the compressed representation

- Very effective technique for compressing data, saving 20% - 90%.

Huffman Coding

- As an example, lets take the string:

  "ABRACADABRA"
- We first to a frequency count of the characters:
  - A:5, B:2, C:1, D:1, R:2
- Next we use a Greedy algorithm to build up a Huffman Tree
  - We start with nodes for each character

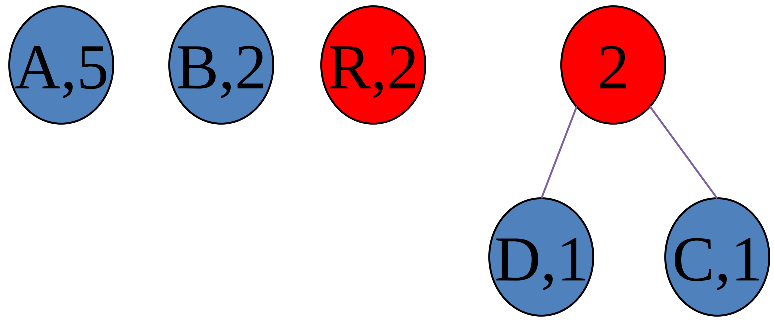


Huffman Coding

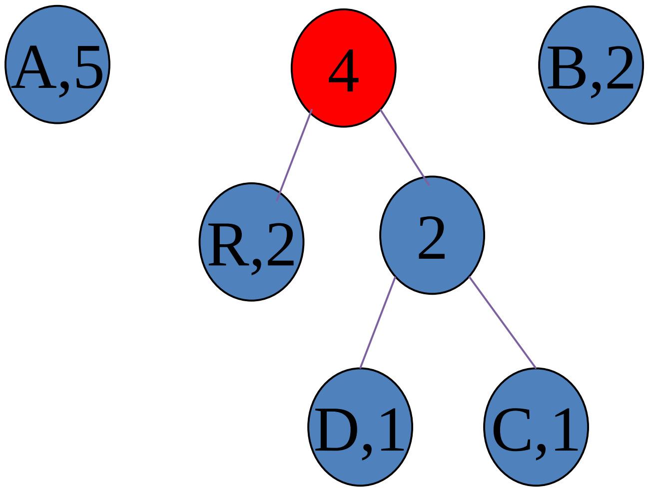
- We then pick the nodes with the smallest frequency and combine them together to form a new node
  - The selection of these nodes is the Greedy part
- The two selected nodes are removed from the set, but replace by the combined node
- This continues until we have only 1 node left in the set

Huffman Coding

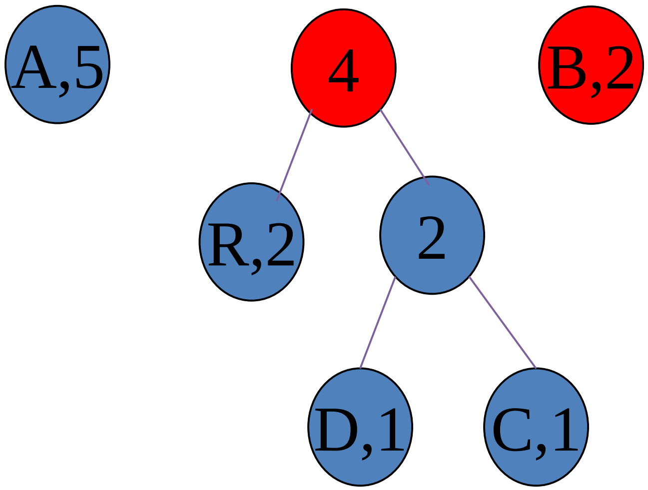
Huffman Coding



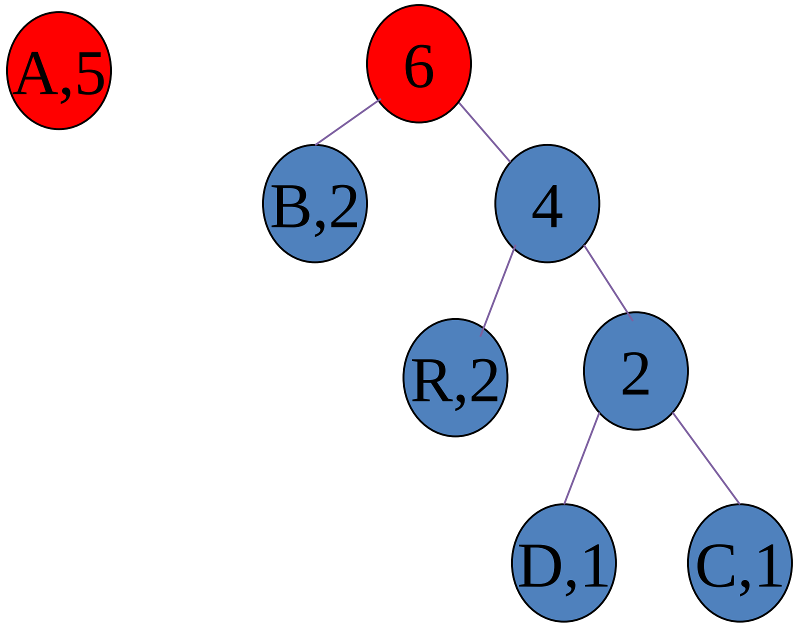
Huffman Coding



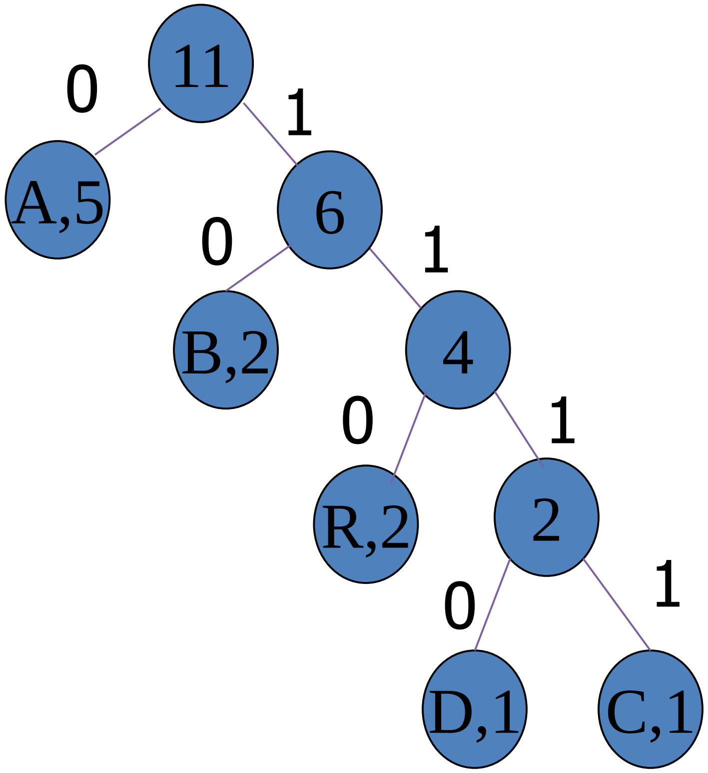
Huffman Coding



Huffman Coding



Huffman Coding



A = 0
B = 10
R = 110
D = 1110
C = 1111

Code word

ABRACADABRA

01011001111011100101100 = 23 bits << 33 bits

A = 0
B = 10
R = 110
D = 1110
C = 1111

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

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.

Huffman codes

- Idea: build tree bottom-up and make greedy choice (what is it?)
  - joining two symbols commits to a bit to distinguish them; which should we choose?

```
build heap H with frequencies as keys
for i = 1 to n-1
      allocate new node z
      z.left = x = EXTRACT-MIN(H)
      z.right = y = EXTRACT-MIN(H)
      z. freq = x.freq + y.freq; INSERT(H, z)
return EXTRACT-MIN(H)
```

Running time

- Build-Min-Heap O(n)
- Each Extract-Min O(lgn) for a total of O(nlgn)

- Overall O(nlgn)

Given the frequency table below:

T	A	M	P	O
.29	.36	.10	.05	.20

What is the code for A ?

a) 0
b) 1
c) 01
d) 10
e) None of the above

Given the frequency table below:

T	A	M	P	O
.29	.36	.10	.05	.20

What is the code for A ?

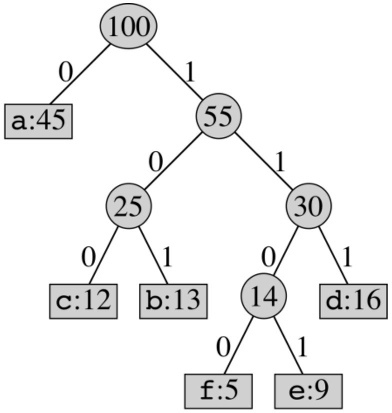
a) 0
b) 1
c) 01
d) 10
e) None of the above

Another Example

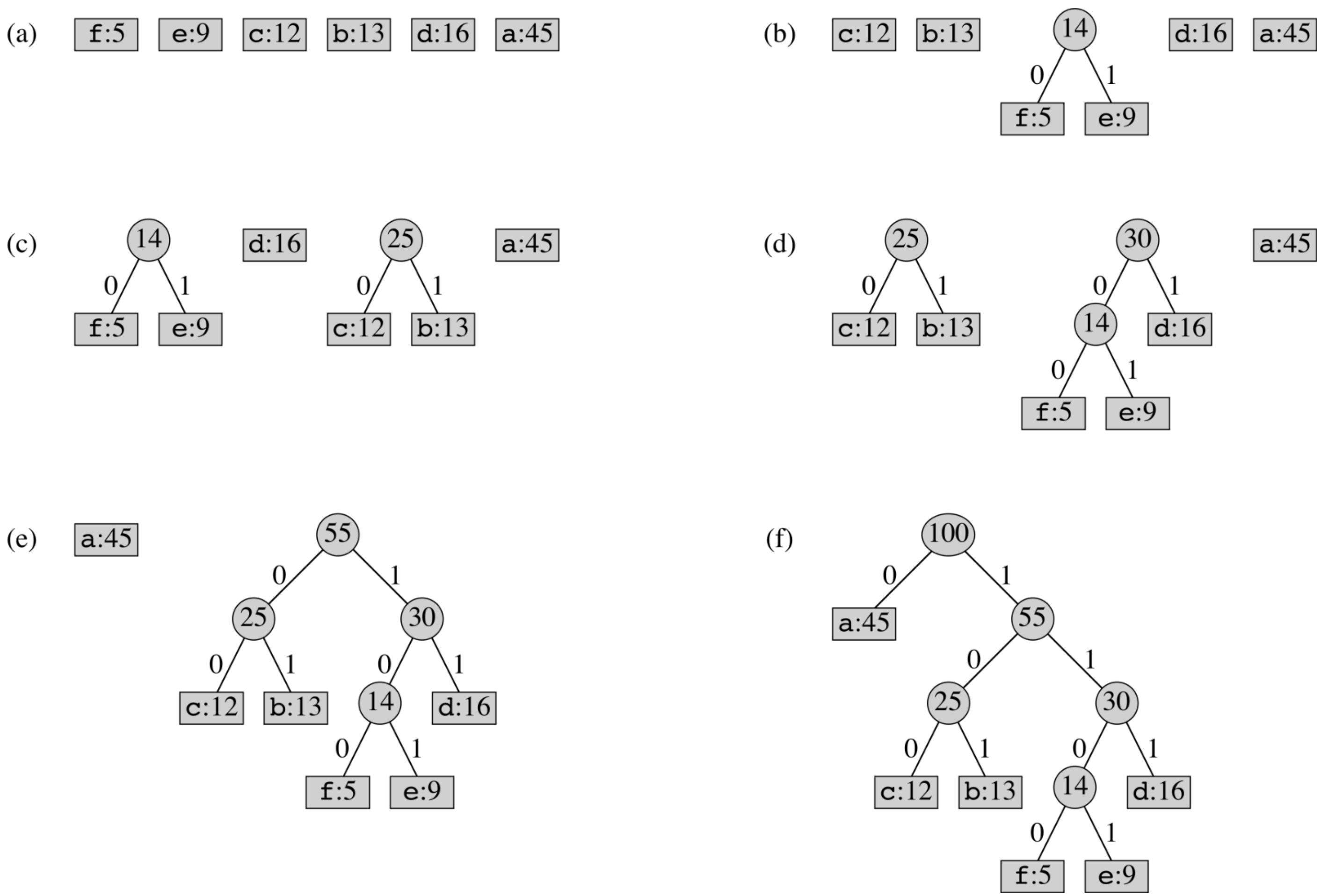
Example:

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

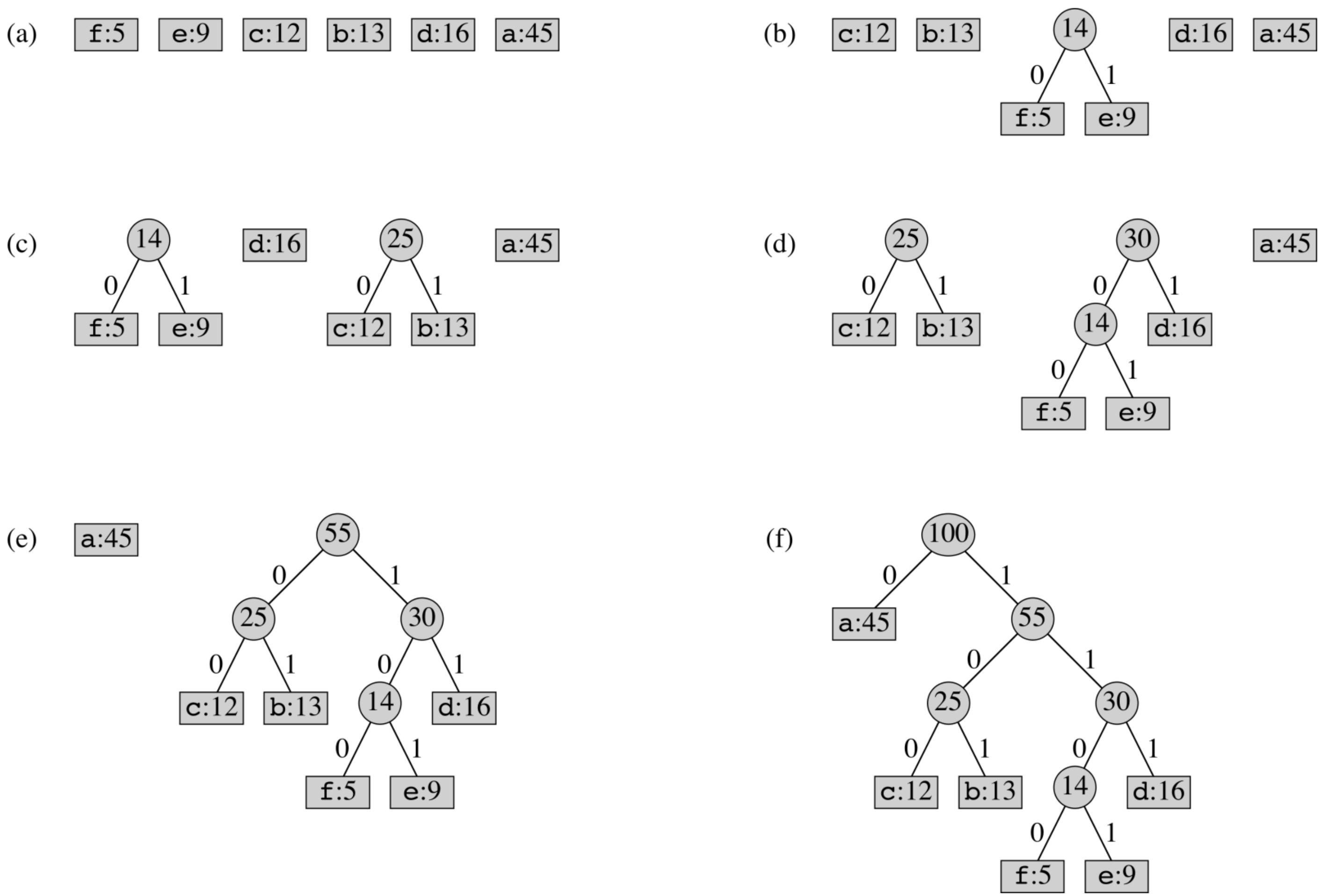
- can represent prefix-free encoding scheme by binary tree T:
- Problem: given frequencies, construct optimal tree (prefix-free encoding scheme)



Huffman example from CLRS



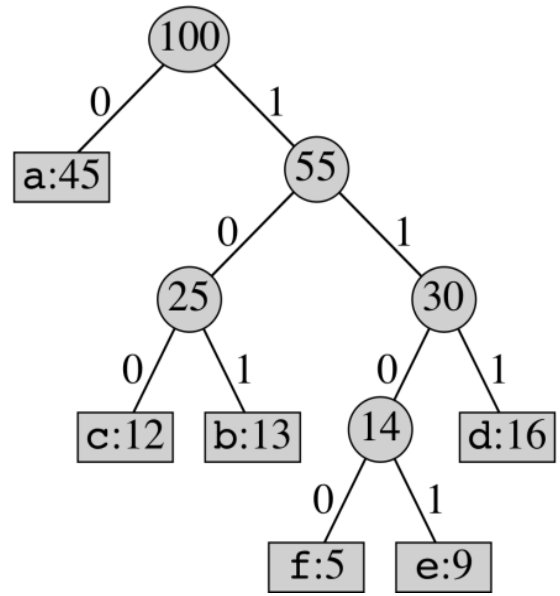
Huffman example from CLRS



Decoding

- Suppose we have the Following code: 1010111

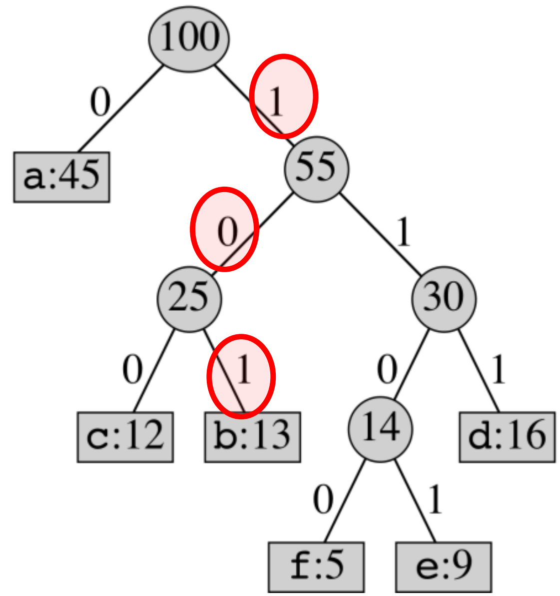
- What is the decode result?



Decoding

- Suppose we have the Following code: 1010111

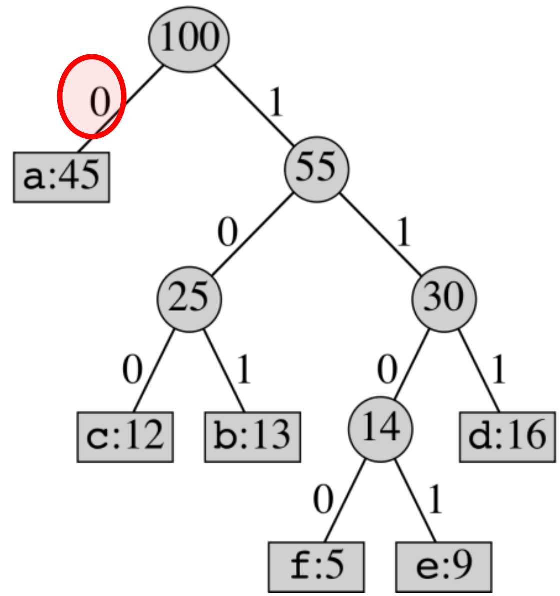
- What is the decode result? b



Decoding

- Suppose we have the Following code: 1010111

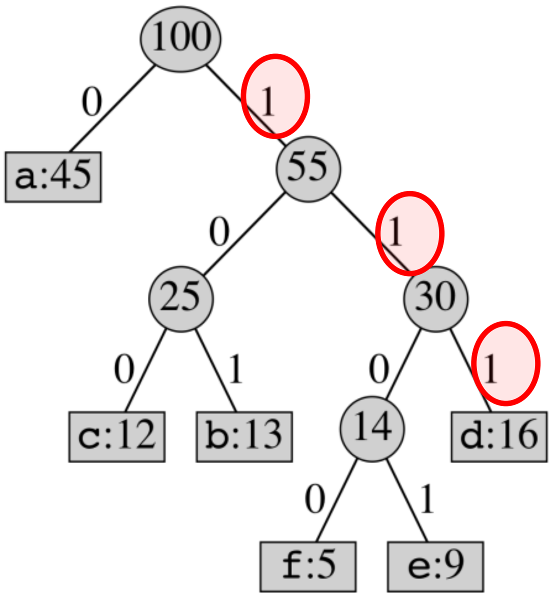
- What is the decode result? ba



Decoding

- Suppose we have the Following code: 1010111

- What is the decode result? bad



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

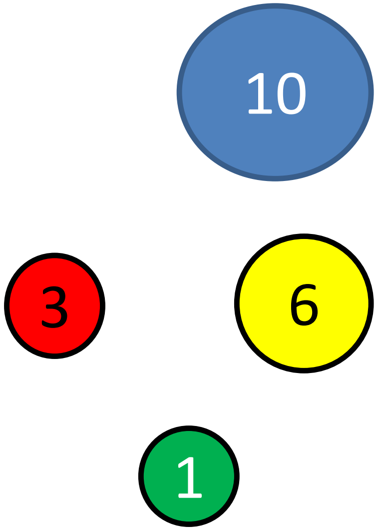
- For coin denominations of 25, 10, 5, 1
  - The greedy choice property is not violated

A failure of the Greedy Algorithm

- Suppose in a fictional monetary system, we have 1 , 3, 6 , and 10 cent coins
- The greedy algorithm results in a solution, but not in an optimal solution

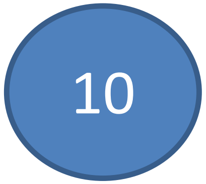
Coin Change Fail

- Step 1: A = 12

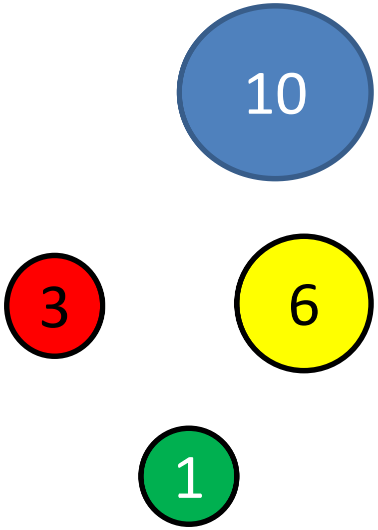


Coin Change Fail

- Step 1: A = 12

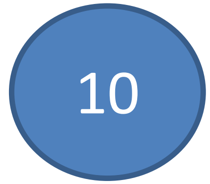


- Step2: (12-10) = 2

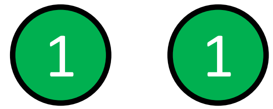


Coin Change Fail

- Step 1: A = 12

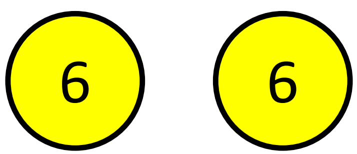


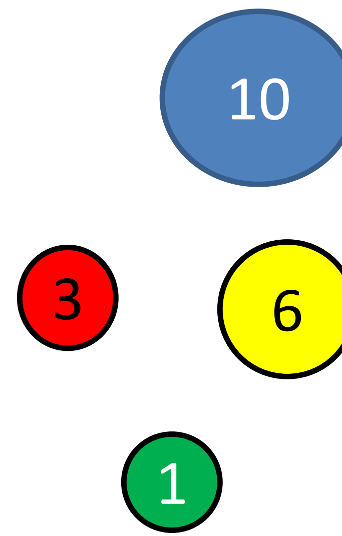
- Step2: (12-10) = 2



This is three coins

The optimal solution is two coins





Making Change DP

Given coins of denominations (value) $1 = v_1 < v_2 < \ldots < v_n$, we wish to make change for an amount A using as few coins as possible. Assume that v_i's and A are integers. Since $v_1 = 1$ there will always be a solution.

Formally, an algorithm for this problem should take as input:

- An array V where V[i] is the value of the coin of the i^{th} denomination.
- A value A which is the amount of change we are asked to make

The algorithm should return an array C where C[i] is the number of coins of value V[i] to return as change and m the minimum number of coins it took.

The objective is to minimize the number of coins returned or:

$$\mathrm{m} = \min \sum_{i=1}^{n} C[i]$$

Subject to : $\sum_{i=1}^{n} V[i] \cdot C[i] = A$

Let coins[k] be the number of coins used to make change for k cents. V[i] is the value (denomination) of the ith coin.

sol[k] is the index of last denomination V[sol[k]] used to obtain change for k cents

Formulas

coins[i] = inf   if i < 0

coins[0] = 0

coins[1]= 1

coins[j] = $\min\limits_{1 \le i \le n} \{1 + coins[j - v_i]\}$**, sol[j] = i**

Sample pseudocode

```
minCoins(A, V[], n)
        coins[0]=0; coins[1] = 1
        for j = 2 to A do  {
                min = inf
                for i = 1 to n do {
                        if ( j >= V[i] )
                           if ( (coins[j-V[i]] + 1) < min )
                           {
                                min = coins[j-V[i]] + 1
                                index = i
                           }
                }
                coins[j] = min
                sol[j] = index
        }
        return coins[A], sol[A]
```

To reconstruct the series of coins used to obtain change for A with minimum number of coins. Call MakeChange(sol, V, A).

```
MakeChange(sol[], V[], m)
        C[] = 0;
        while m > 0  {
                Print V[sol[m]]
                C[sol[m]] = C[sol[m]] + 1
                m = m – V[sol[m]]
        }
```

- V[i] is the value of the ith coin.
- Sol[j] is the index of the last coin to make change for an amount of j
- C[i] is the number of times the ith coin is used in the solution.

Theoretical running time

- The running time of MakeChange is O(A)
- The running time of minCoins is $\Theta(nA)$ since the outer loop is j = 2 ..A and inner loop is i = 1 ..n.
- **Overall $\Theta(nA)$**

Example

V = { 1, 3, 6, 10} V[1]=1, V[2]=3, V[3] = 6, V[4] = 10,   A = 12

Coin[a] minimum # of coins to make amount a

Coin[0] = 0   no coins

Coin[1] = 1   sol[1]=1 use a 1 cent

Coin[2] = 2   sol[2]=1

Coin[1] + 1 = 2 use a 1 cent coin

Coin[3] = 1   sol[3] = 2

Coin[2] + 1 = 3 use the 1 cent coin and have 2 cents left over

Coin[0] + 1 = 1   use the 3 cent coin v[2]

Coin[4] = 2   sol[4] = 1

Coin[3] + 1 = 2   use a 1 cent

Coin[1] + 1 = 2   use a 3 cent

V = { 1, 3, 6, 10} V[1]=1, V[2]=3, V[3] = 6, V[4] = 10,   A = 12

Coin[5] = 3   sol[5] = 1

Coin[4] + 1 = 3   use a 1 cent

Coin[2] + 1 = 3   use the 3 cent

Coin[6] = 1   sol[6] = 3

Coin[5] + 1 = 4 use a 1 cent (1 , 1 3)

Coin[3] + 1 = 2 use a 3 cent (3 cent)

Coin[0] + 1 = 1 use the 6 cents

……

Coin[12]   sol[12] = 3

Coin[11] + 1 = 3   last coin 1 cent

Coin[9] + 1 =   3   last coin 3 cent

Coin[6] + 1 =   2   last coin 6 cent

Coin[2] + 1 =   3   last coin 10 cent

To reconstruct the series of coins used to obtain change for A with minimum number of coins. Call MakeChange(sol, V, A).

```
MakeChange(sol[], V[], m)
        C[] = 0;
        while m > 0  {
                Print V[sol[m]]
                C[sol[m]] = C[sol[m]] + 1
                m = m – V[sol[m]]
        }
```

sol[12] = 3, V[3] = 6, C[3] = 1

m= 12-6 = 6

sol[6] = 3, V[3] = 6, C[3] = 1+1 = 2

m = 6-6 = 0;