Introduction to Algorithms

Greedy Algorithms

Greedy algorithms

Suppose it is possible to build a solution through a sequence of partial solutions

- At each step, we focus on one particular partial solution and we attempt to extend that solution
- Ultimately, the partial solutions should lead to a feasible solution which is also optimal

Making change

Consider this commonplace example:

- Making the exact change with the minimum number of coins
- Consider the Euro denominations of 1, 2, 5, 10, 20, 50 cents
- Stating with an empty set of coins, add the largest coin possible into the set which does not go over the required amount

Making change

To make change for €0.72:

- Start with €0.50

Total €0.50

Making change

To make change for €0.72:

- Start with €0.50
- Add a €0.20

Total €0.70

Making change

To make change for €0.74:

- Start with €0.50
- Add a €0.20
- Skip the €0.10 and the €0. 05 but add a €0.02

Total €0.72

Making change

Notice that each digit can be worked with separately

- The maximum number of coins for any digit is three
- Thus, to make change for anything less than €1 requires at most six coins
- The solution is optimal

Making change

Does this strategy always work?

- What if our coin denominations grow quadraticly?

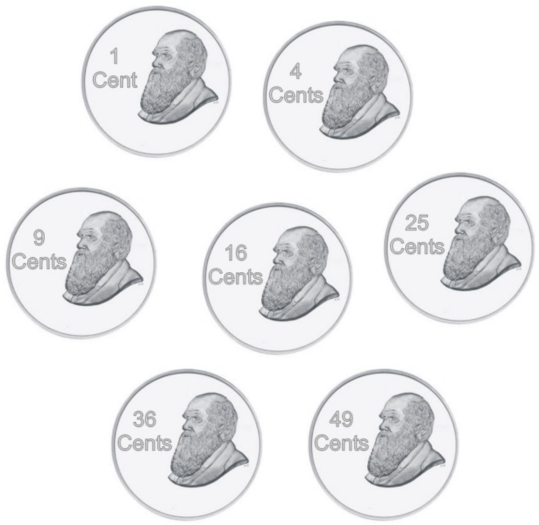
Consider 1, 4, 9, 16, 25, 36, and 49 dumbledores

Making change

Using our algorithm, to make change for 72 dumbledores, we require six coins:

$$72 = 49 + 16 + 4 + 1 + 1 + 1$$

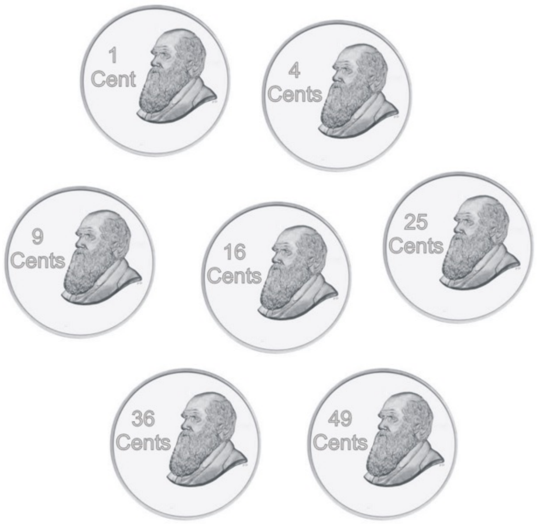
49 Cents
16 Cents
1 Cent
4 Cents
1 Cent
1 Cent



Making change

The optimal solution, however, is two 36 dumbledore coins

36 Cents
36 Cents



Greedy Algorithms

- A *greedy algorithm* always makes the choice that looks best at the moment
  - The hope: a locally optimal choice will lead to a globally optimal solution
  - For some problems, it works
  - My everyday examples:
    - Walking to the Corner
    - Playing a bridge hand
- Dynamic programming can be overkill; greedy algorithms tend to be easier to code

Definition

A greedy algorithm is an algorithm which has:

- A set of partial solutions from which a solution is built
- An *objective function* which assigns a value to any partial solution

Then given a partial solution, we

- Consider possible extensions of the partial solution
- Discard any extensions which are not feasible
- Choose that extension which minimizes the object function

This continues until some criteria has been reached

Data Compression

- Suppose we have 1000000000 (1G) character data file that we wish to include in an email.
- Suppose file only contains 26 letters {a,…,z}.
- Suppose each letter α in {a,…,z} occurs with frequency f_α.
- Suppose we encode each letter by a binary code
- If we use a fixed length code, we need 5 bits for each character
- The resulting message length is $5\left(f_a + f_b + \cdots f_z\right)$

- **Can we do better?**

Huffman Codes

- Most character code systems (ASCII, unicode) use fixed length encoding
- If frequency data is available and there is a wide variety of frequencies, variable length encoding can save 20% to 90% space
- Which characters should we assign shorter codes; which characters will have longer codes?

Data Compression: A Smaller Example

- Suppose the file only has 6 letters {a,b,c,d,e,f} with frequencies

a	b	c	d	e	f	
.45	.13	.12	.16	.09	.05	
000	001	010	011	100	101	***Fixed length***
0	101	100	111	1101	1100	***Variable length***

- Fixed length 3G=3000000000 bits
- Variable length

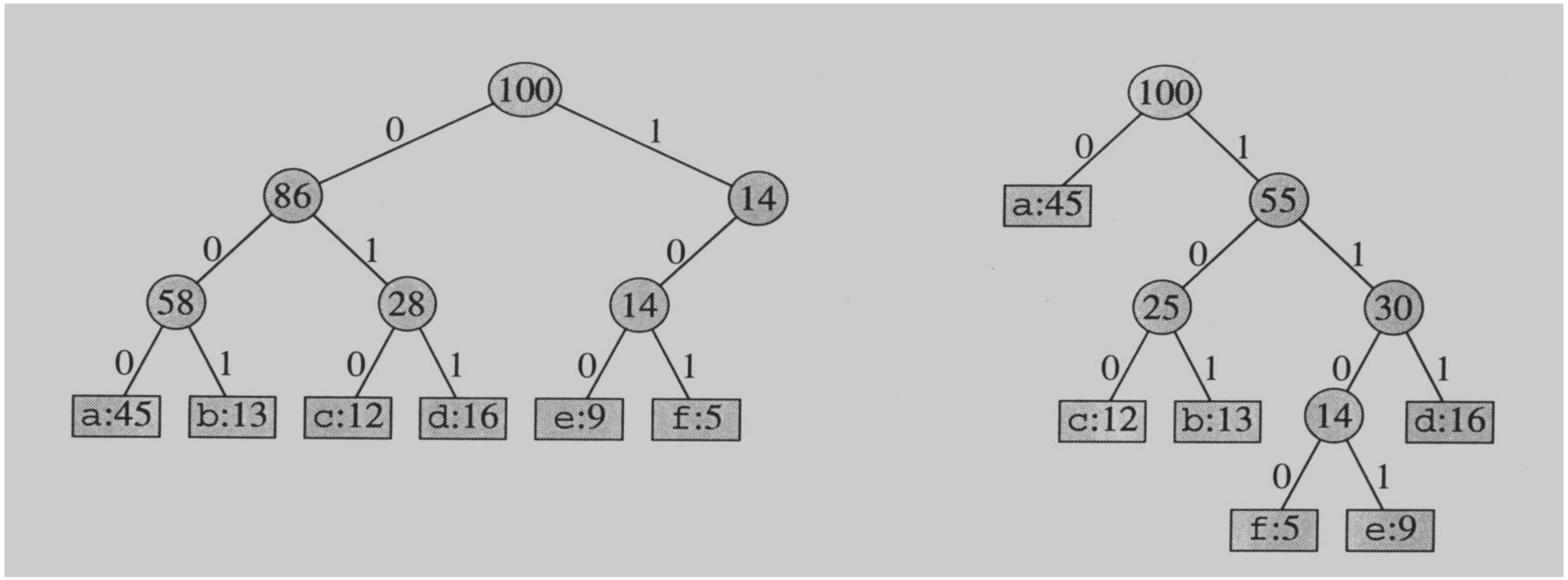
$$(.45 \bullet 1 + .13 \bullet 3 + .12 \bullet 3 + .16 \bullet 3 + .09 \bullet 4 + .05 \bullet 4) = 2.24G$$

How to decode?

- At first it is not obvious how decoding will happen, but this is possible if we use prefix codes

Prefix Codes

- No encoding of a character can be the prefix of the longer encoding of another character, for example, we could not encode *t* as 01 and *x* as 01101 since 01 is a prefix of 01101
- By using a binary tree representation we will generate prefix codes provided all letters are leaves



Prefix codes

- A message can be decoded uniquely.
- Following the tree until it reaches to a leaf, and then repeat!
- Draw a few more tree and produce the codes!!!

Some Properties

- Prefix codes allow easy decoding
  - Given a: 0, b: 101, c: 100, d: 111, e: 1101, f: 1100
  - Decode 001011101 going left to right, 0|01011101, a|0|1011101, a|a|101|1101, a|a|b|1101, a|a|b|e
- An optimal code must be a full binary tree (a tree where every internal node has two children)
- For *C* leaves there are *C-1* internal nodes
- The number of bits to encode a file is

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

where $f(c)$ is the freq of c, $d_T(c)$ is the tree depth of c, which corresponds to the code length of c

Optimal Prefix Coding Problem

- Input: Given a set of n letters ($c_1,\ldots, c_n$) with frequencies ($f_1,\ldots,f_n$).

- Construct a full binary tree T to define a prefix code that minimizes the average code length

$$\text{Average}(T) = \sum_{i=1}^{n} f_i \bullet \text{length}_T(c_i)$$

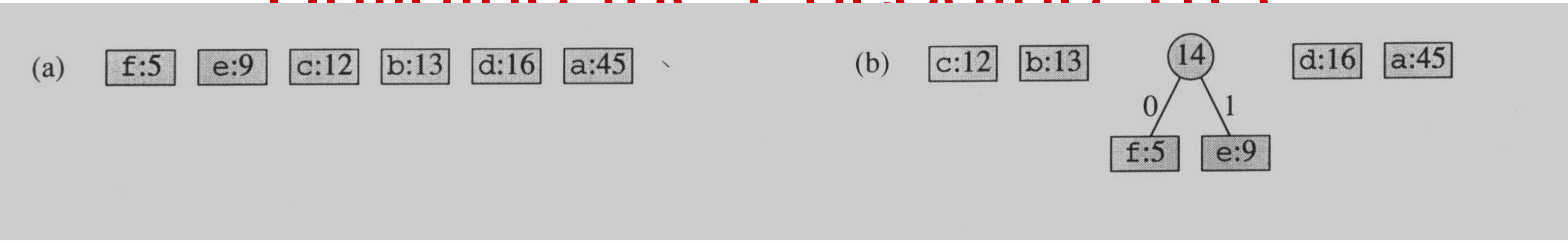
Greedy algorithms

- Many optimization problems can be solved using a greedy approach
  - The basic principle is that local optimal decisions may may be used to build an optimal solution
  - But the greedy approach may not always lead to an optimal solution overall for all problems
  - The key is knowing which problems will work with this approach and which will not
- We will study
  - **The problem of generating Huffman codes**

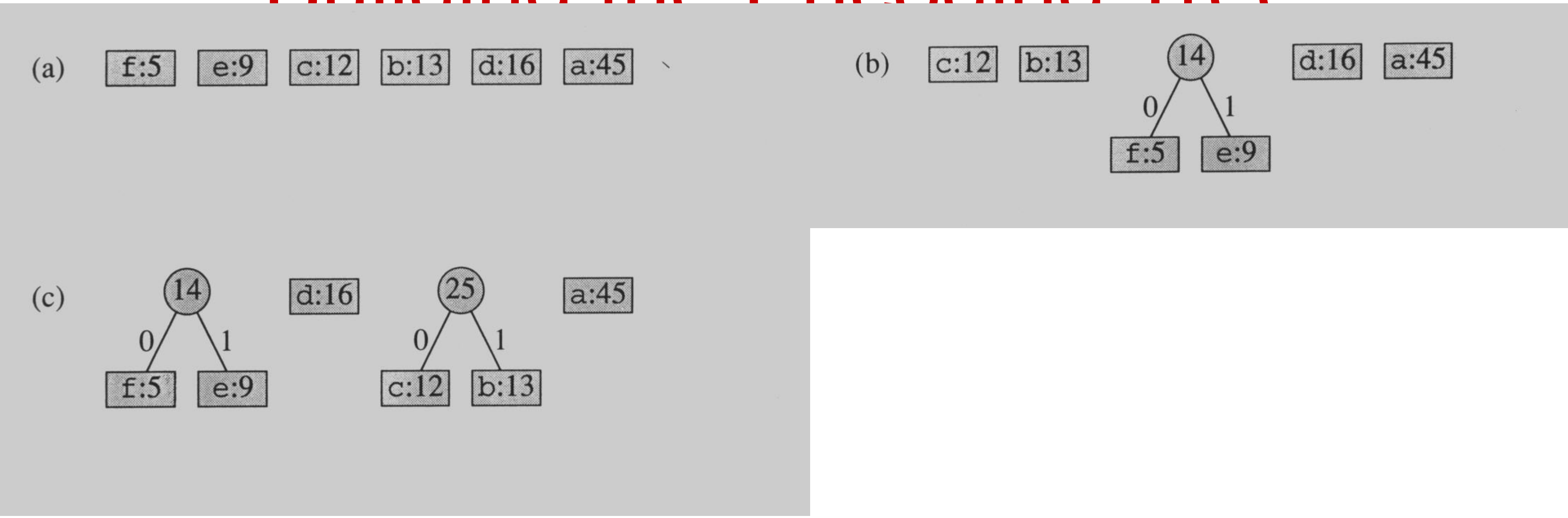
David Huffman’s idea

- Build the tree (code) bottom-up in a greedy fashion

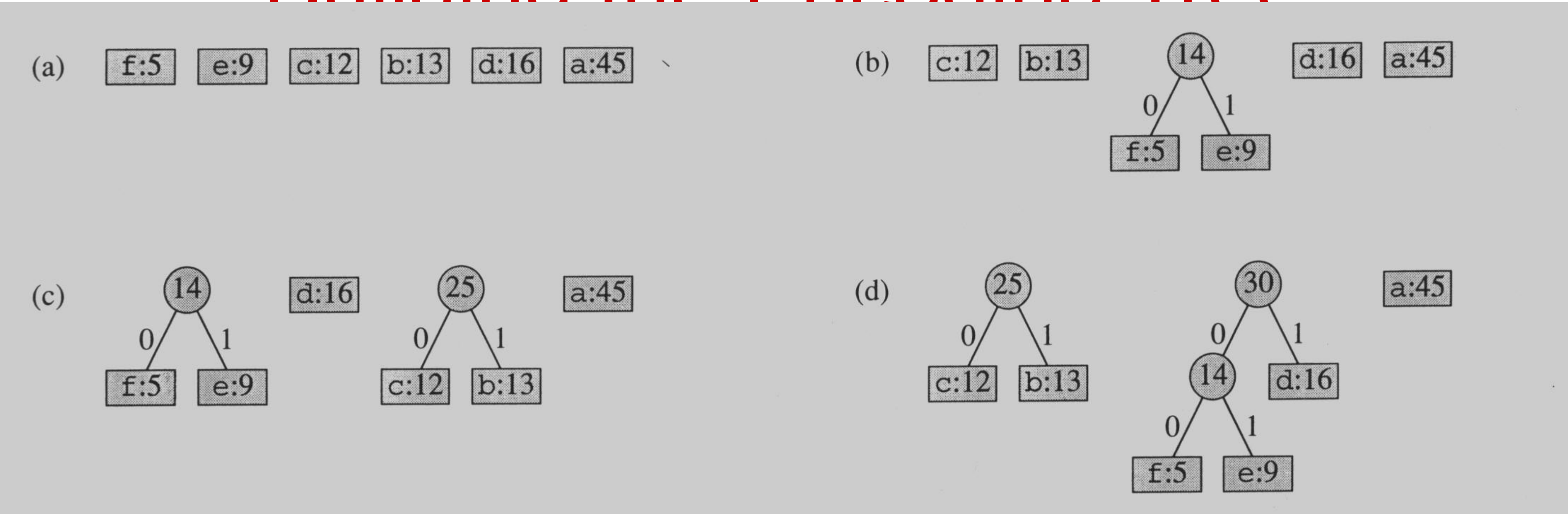
Building the Encoding Tree



Building the Encoding Tree

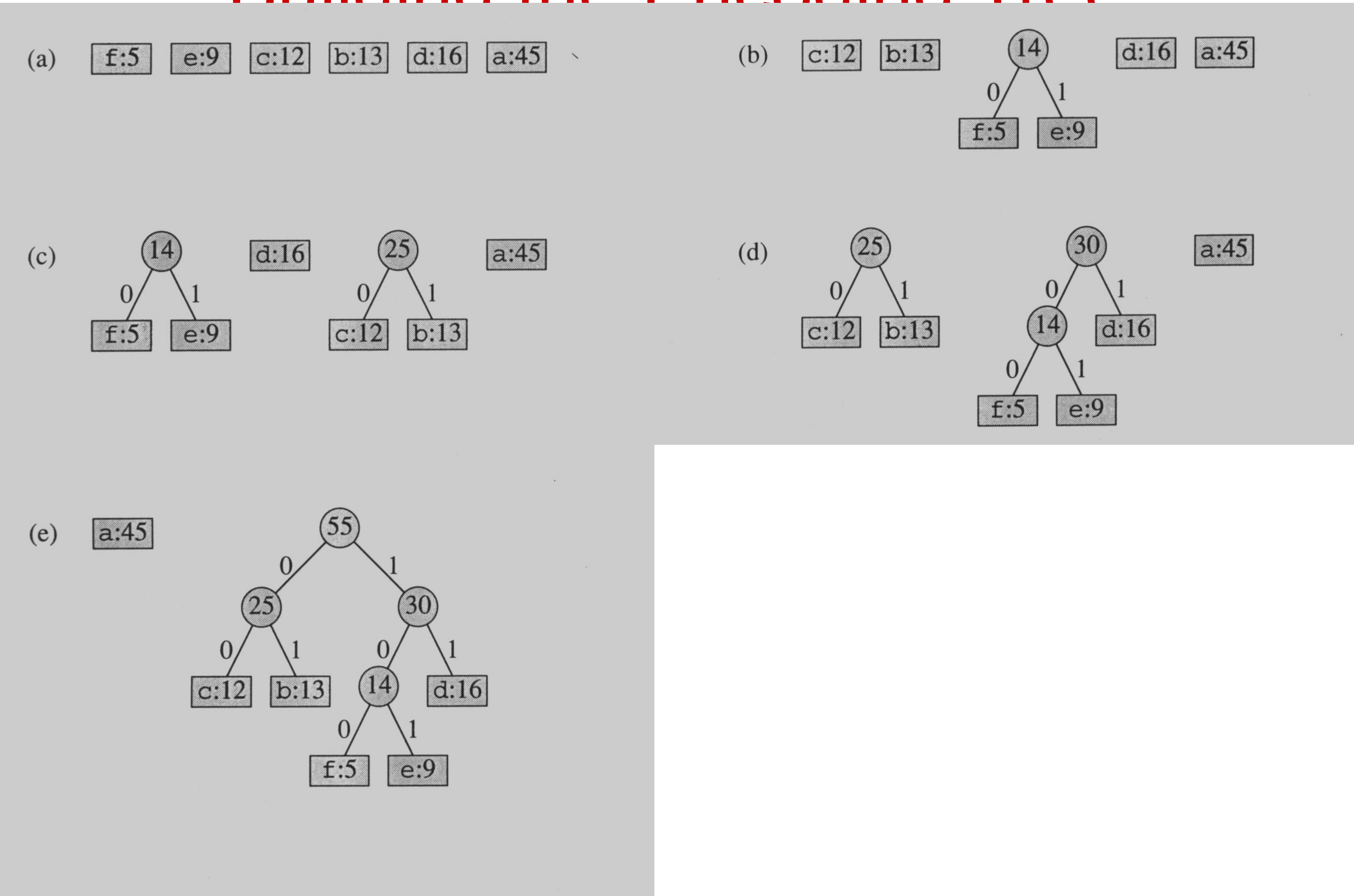


Building the Encoding Tree

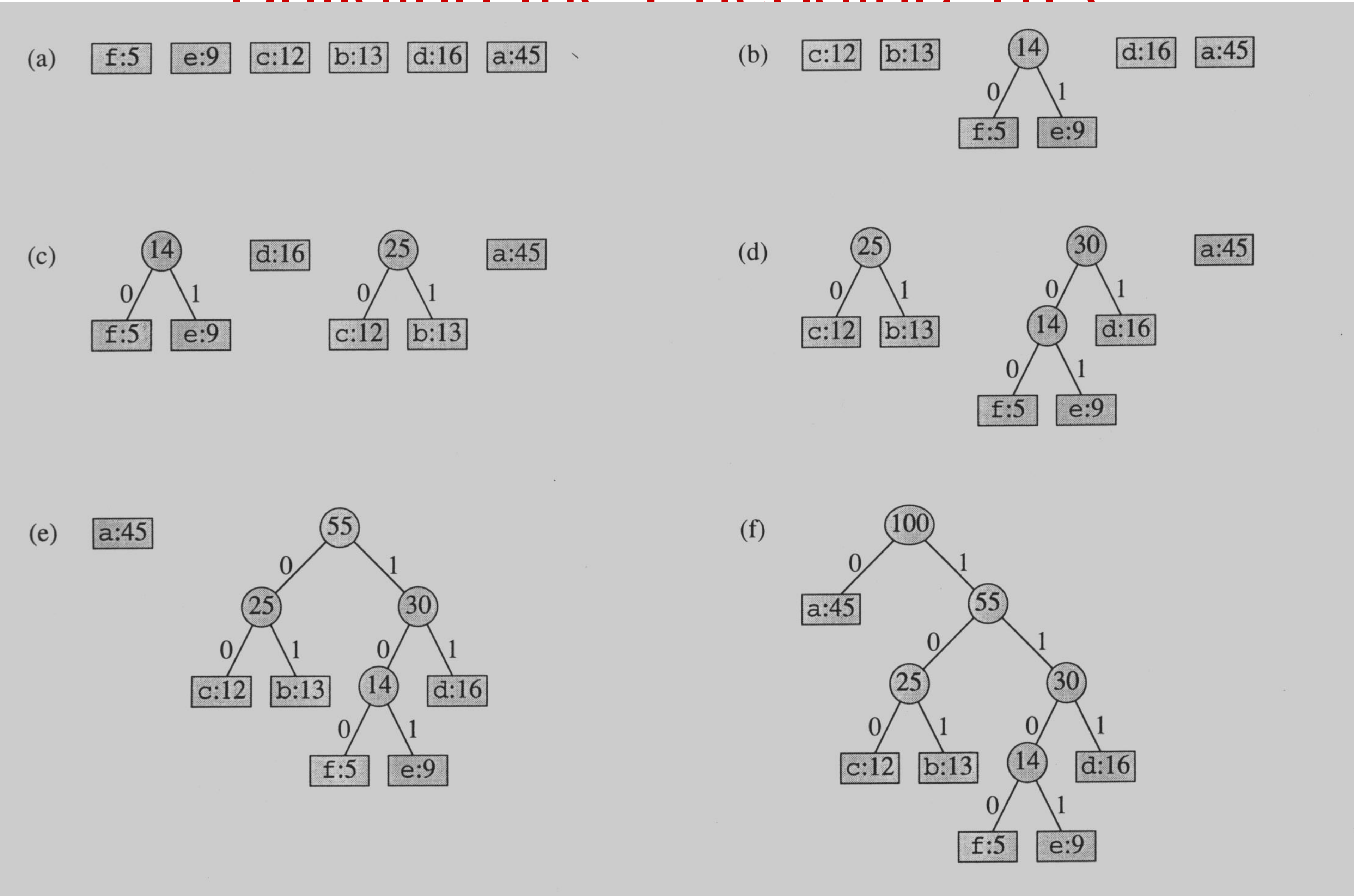


(a) f:5 e:9 c:12 b:13 d:16 a:45
(b) c:12 b:13 14 0 1 f:5 e:9 d:16 a:45
(c) 14 0 1 f:5 e:9 d:16 25 0 1 c:12 b:13 a:45
(d) 25 0 1 c:12 b:13 30 0 1 14 d:16 0 1 f:5 e:9 a:45

Building the Encoding Tree


(a) f:5 e:9 c:12 b:13 d:16 a:45
(b) c:12 b:13 14 0 1 f:5 e:9 d:16 a:45
(c) 14 0 1 f:5 e:9 d:16 25 0 1 c:12 b:13 a:45
(d) 25 0 1 c:12 b:13 30 0 1 14 d:16 0 1 f:5 e:9 a:45
(e) a:45 55 0 1 25 30 0 1 c:12 b:13 0 1 14 d:16 0 1 f:5 e:9

Building the Encoding Tree



(a) f:5 e:9 c:12 b:13 d:16 a:45
(b) c:12 b:13 14 0 f:5 1 e:9 d:16 a:45
(c) 14 0 f:5 1 e:9 d:16 25 0 c:12 1 b:13 a:45
(d) 25 0 c:12 1 b:13 30 0 14 0 f:5 1 e:9 1 d:16 a:45
(e) a:45 55 0 25 0 c:12 1 b:13 1 30 0 14 0 f:5 1 e:9 1 d:16
(f) 100 0 a:45 1 55 0 25 0 c:12 1 b:13 1 30 0 14 0 f:5 1 e:9 1 d:16

The Algorithm

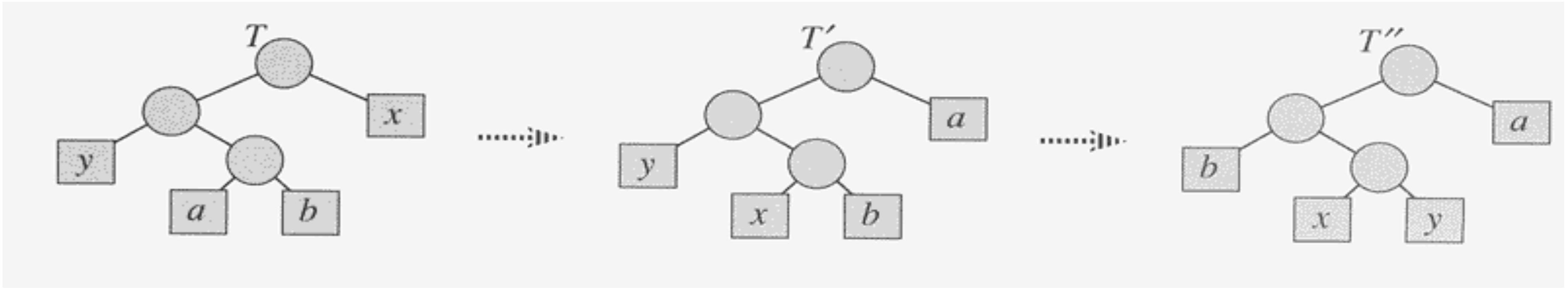
```
HUFFMAN($C$)
$n \leftarrow |C|$
$Q \leftarrow C$
for $i \leftarrow 1$ to $n - 1$
    do allocate a new node $z$
        $left[z] \leftarrow x \leftarrow$ EXTRACT-MIN($Q$)
        $right[z] \leftarrow y \leftarrow$ EXTRACT-MIN($Q$)
        $f[z] \leftarrow f[x] + f[y]$
        INSERT($Q$, $z$)
return EXTRACT-MIN($Q$)            ▷ Return the root of the tree.
```

- An appropriate data structure is a binary min-heap
- Rebuilding the heap is *lg n* and *n-1* extractions are made, so the complexity is O(*n lg n*)
- The encoding is NOT unique, other encoding may work just as well, but none will work better

Correctness of Huffman's Algorithm

Lemma 16.2

Let C be an alphabet in which each character $c \in C$ has frequency $f[c]$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.



Since each swap does not increase the cost, the resulting tree T'' is also an optimal tree

Lemma 16.2

- Without loss of generality, assume $f[a] \leq f[b]$ and $f[x] \leq f[y]$
- The cost difference between T and T' is

$$
\begin{aligned}
& B(T) - B(T') = \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
& = f[x]d_T(x) + f[a]d_T(a) - f[x]d_{T'}(x) - f[a]d_{T'}(a) \\
& = f[x]d_T(x) + f[a]d_T(a) - f[x]d_T(a) - f[a]d_T(x) \\
& = (f[a] - f[x])(d_T(a) - d_T(x)) \\
& \geq 0
\end{aligned}
$$

B(T'') ≤ B(T), but T is optimal,

B(T) ≤ B(T'') ➔ B(T'') = B(T)

Therefore T'' is an optimal tree in which x and y appear as sibling leaves of maximum depth

Correctness of Huffman’s Algorithm

Lemma 16.3

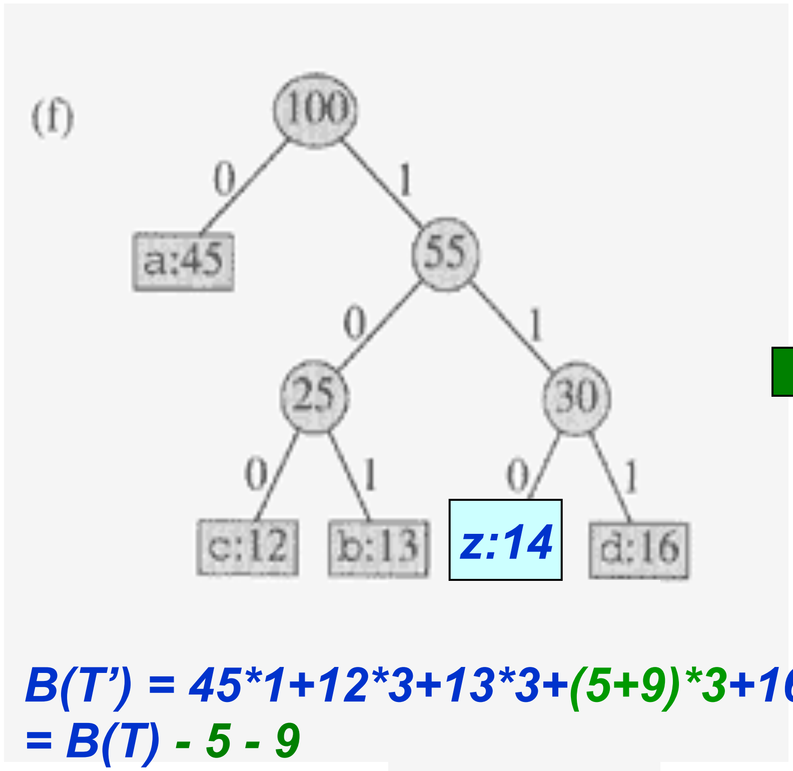
Let C be a given alphabet with frequency $f[c]$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with characters x, y removed and (new) character z added. so that $C' = C - \{x, y\} \cup \{z\}$; define f for C' as for C, except that $f[z] = f[x] + f[y]$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

- ***Observation:*** $B(T) = B(T') + f[x] + f[y] \Rightarrow B(T') = B(T) - f[x] - f[y]$
  - For each $c \in C - \{x, y\} \Rightarrow d_T(c) = d_{T'}(c) \Rightarrow f[c]d_T(c) = f[c]d_{T'}(c)$
  - $d_T(x) = d_T(y) = d_{T'}(z) + 1$
  - $f[x]d_T(x) + f[y]d_T(y) = (f[x] + f[y])(d_{T'}(z) + 1) = f[z]d_{T'}(z) + (f[x] + f[y])$

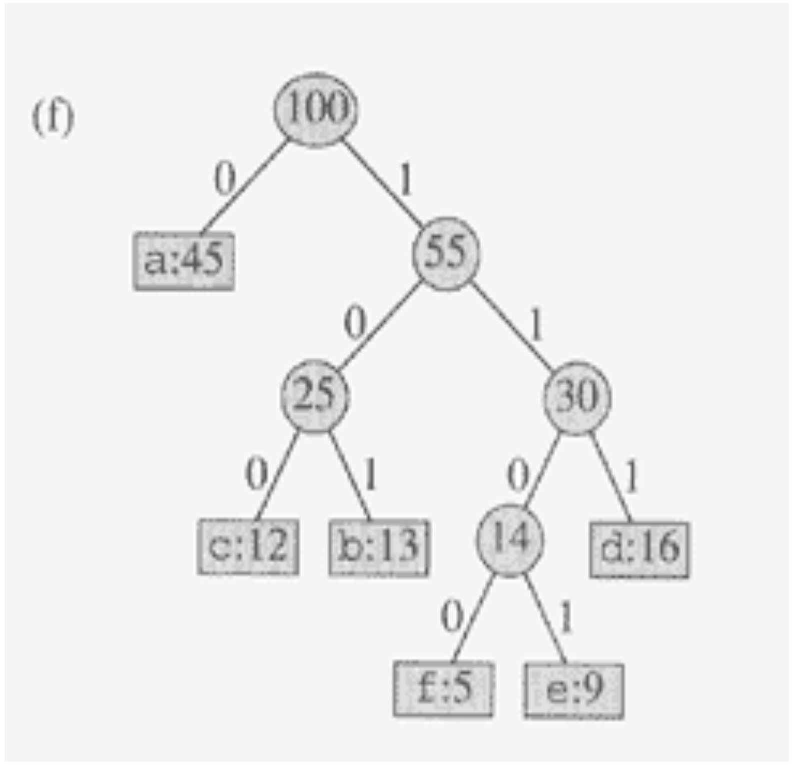
Theorem 16.4

Procedure HUFFMAN produces an optimal prefix code.

$B(T') = B(T)-f[x]-f[y]$



$B(T') = 45*1+12*3+13*3+(5+9)*3+16*3$
$= B(T) - 5 - 9$



$B(T) = 45*1+12*3+13*3+5*4+9*4+16*3$

Proof of Lemma 16.3

- Prove by contradiction.
- Suppose that T does not represent an optimal prefix code for C. Then there exists a tree *T''* such that $B(T'') < B(T)$.
- Without loss of generality, by Lemma 16.2, *T''* has *x* and *y* as siblings. Let *T'''* be the tree *T''* with the common parent x and *y* replaced by a leaf with frequency $f[z] = f[x] + f[y]$. Then
- $B(T''') = B(T'') - f[x] - f[y] < B(T) - f[x] - f[y] = B(T')$
  - *T'''* is better than *T'* ➔ contradiction to the assumption that *T'* is an optimal prefix code for C'

Greedy algorithms

- A *greedy algorithm* always makes the choice that looks best at the moment
  - My everyday examples:
    - Playing cards
    - Invest on stocks
    - Choose a university
  - The hope: a locally optimal choice will lead to a globally optimal solution
  - For some problems, it works
- greedy algorithms tend to be easier to code

An Activity Selection Problem (Conference Scheduling Problem)

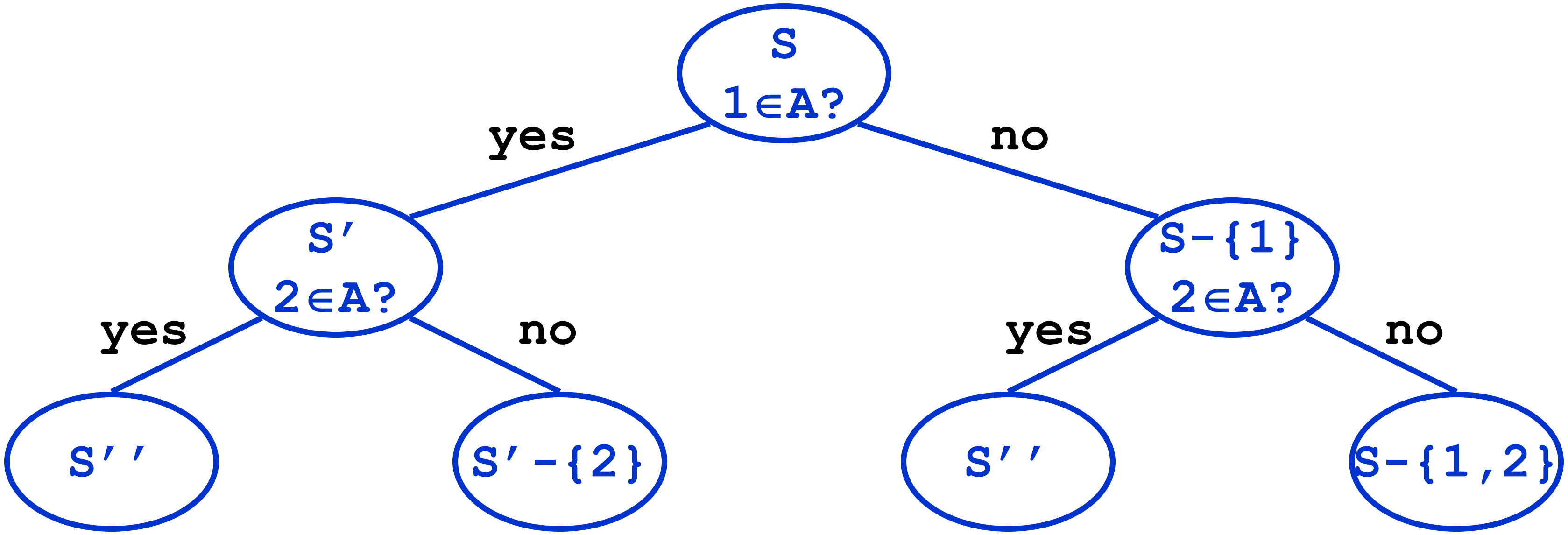
- **Input: A set of activities S = {$a_1, \ldots, a_n$}**
- Each activity has start time and a finish time
  - $a_i = (s_i, f_i)$
- Two activities are compatible if and only if their interval does not overlap
- **Output: a maximum-size subset of mutually compatible activities**

Activity Selection: Optimal Substructure

- Let k be the minimum activity in A (i.e., the one with the earliest finish time). Then A - $\{k\}$ is an optimal solution to $S' = \{i \in S: s_i \geq f_k\}$
  - In words: once activity #1 is selected, the problem reduces to finding an optimal solution for activity-selection over activities in S compatible with #1
  - Proof: if we could find optimal solution B' to S' with $|B| > |A - \{k\}|$,
    - Then $B \cup \{k\}$ is compatible

Activity Selection: Repeated Subproblems

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



Greedy Choice Property

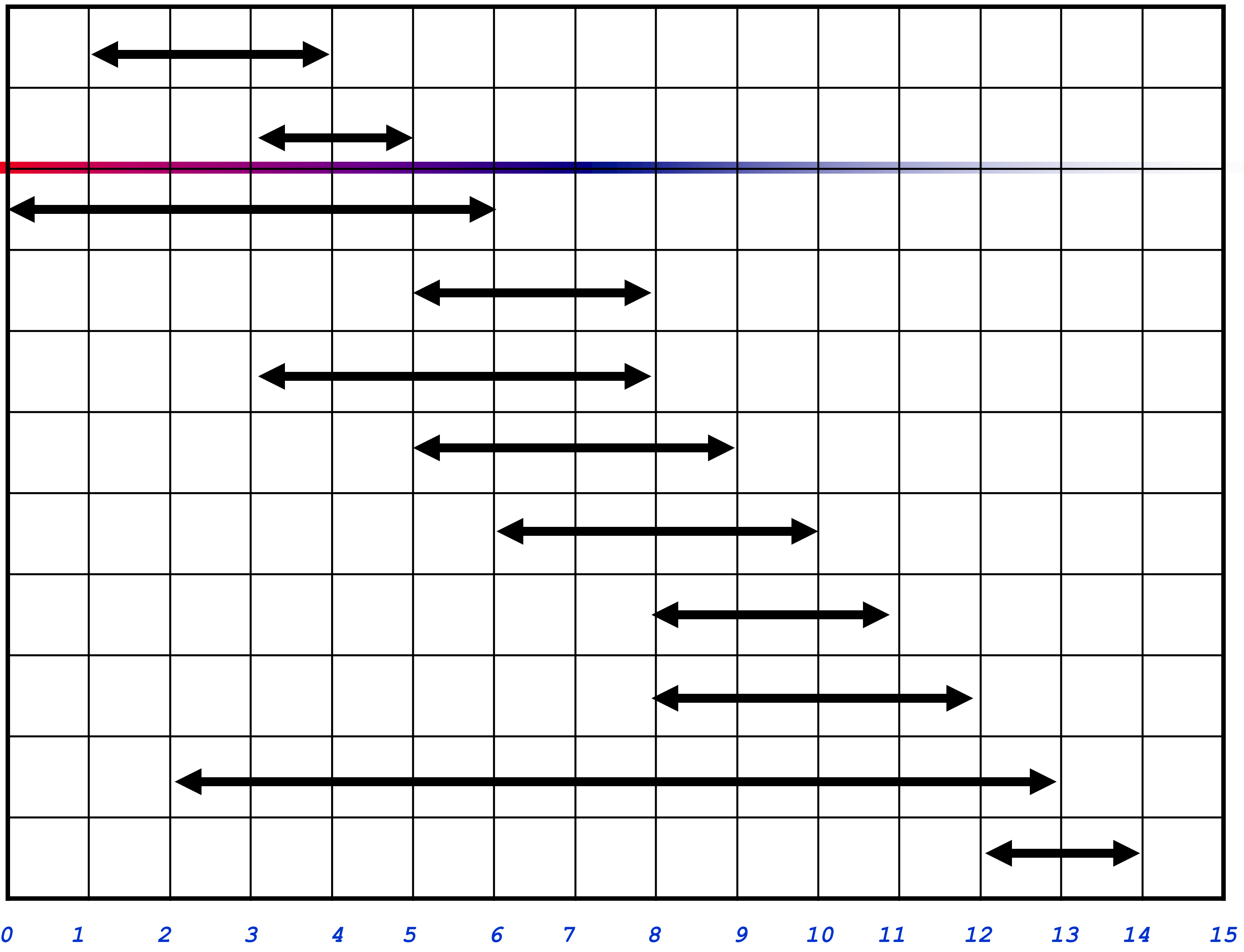
- Dynamic programming? Memoize? Yes, but…
- Activity selection problem also exhibits the *greedy choice* property:
  - Locally optimal choice $\Rightarrow$ globally optimal sol'n
  - Them 16.1: if S is an activity selection problem sorted by finish time, then $\exists$ optimal solution $A \subseteq S$ 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.

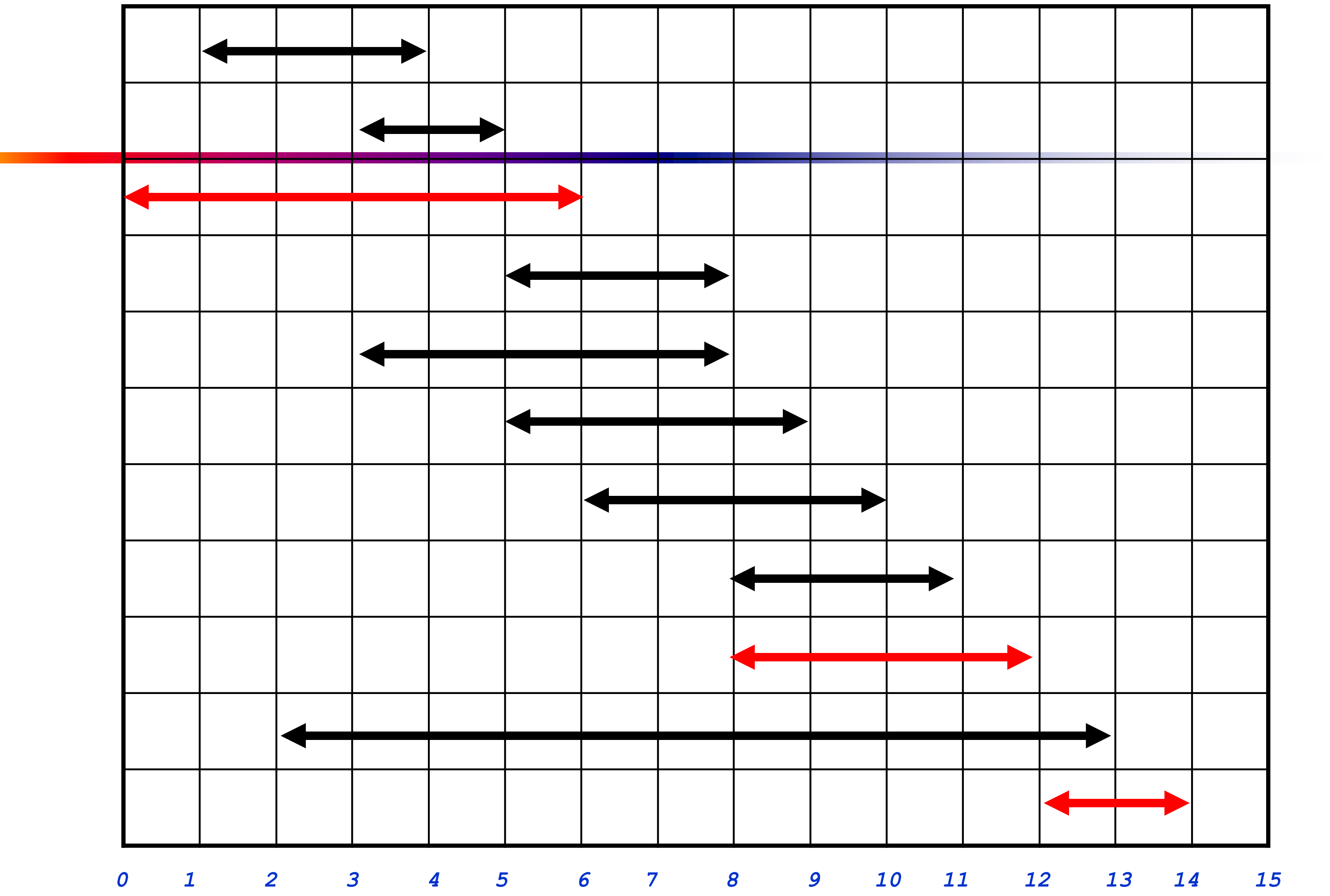
The Activity Selection Problem

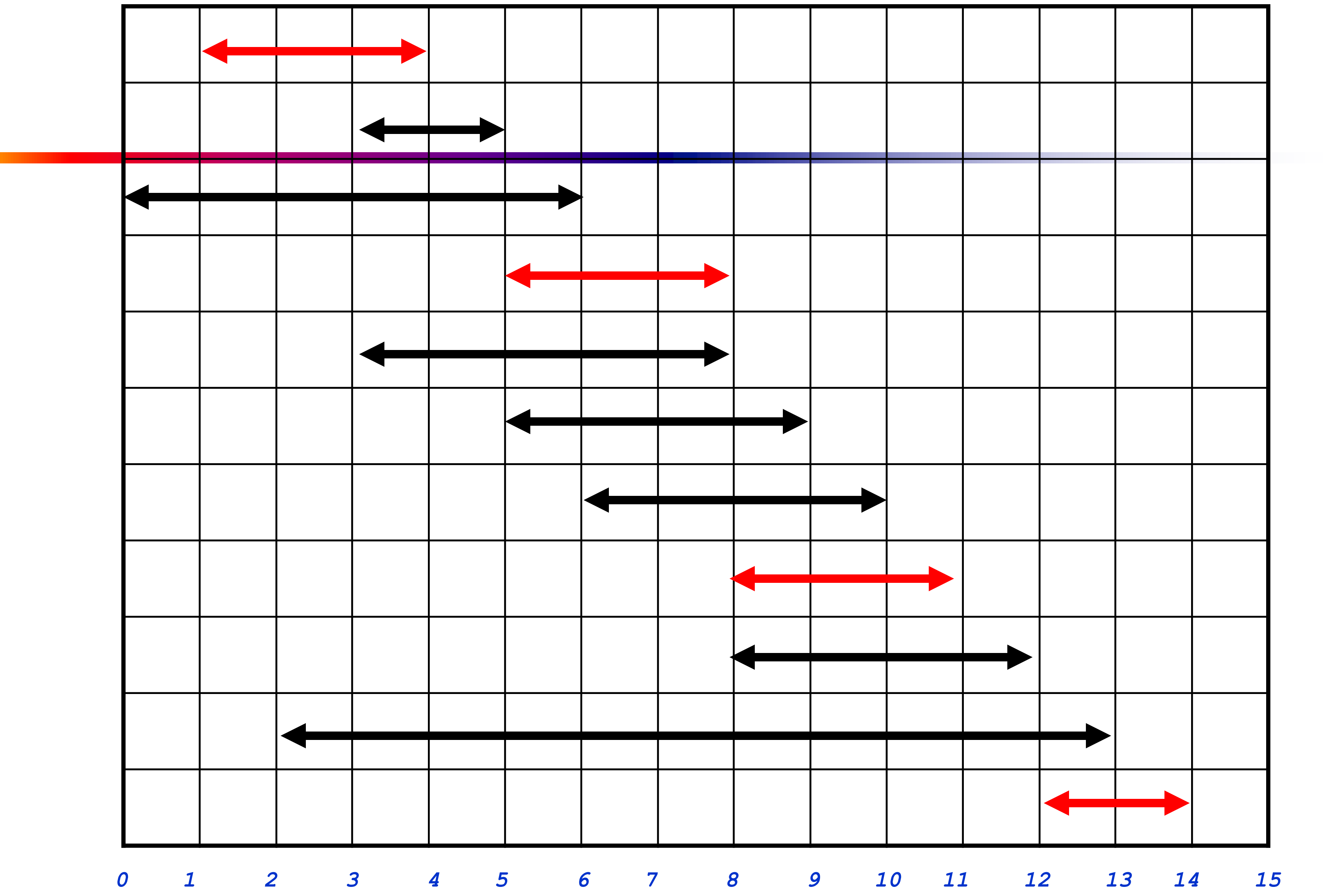
- Here are a set of start and finish times

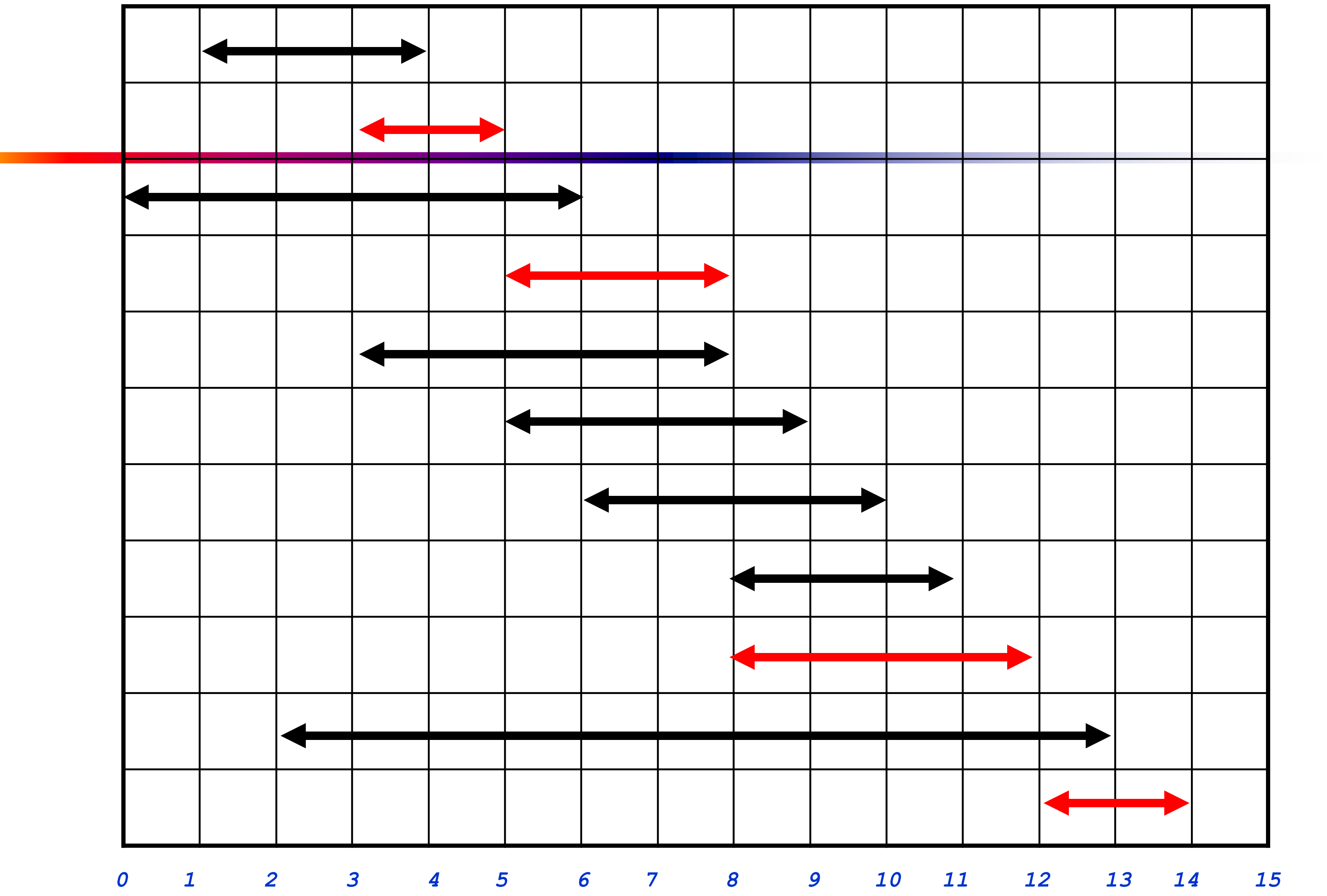
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- ***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 $\{a_2, a_4, a_9, a_{11}\}$***


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

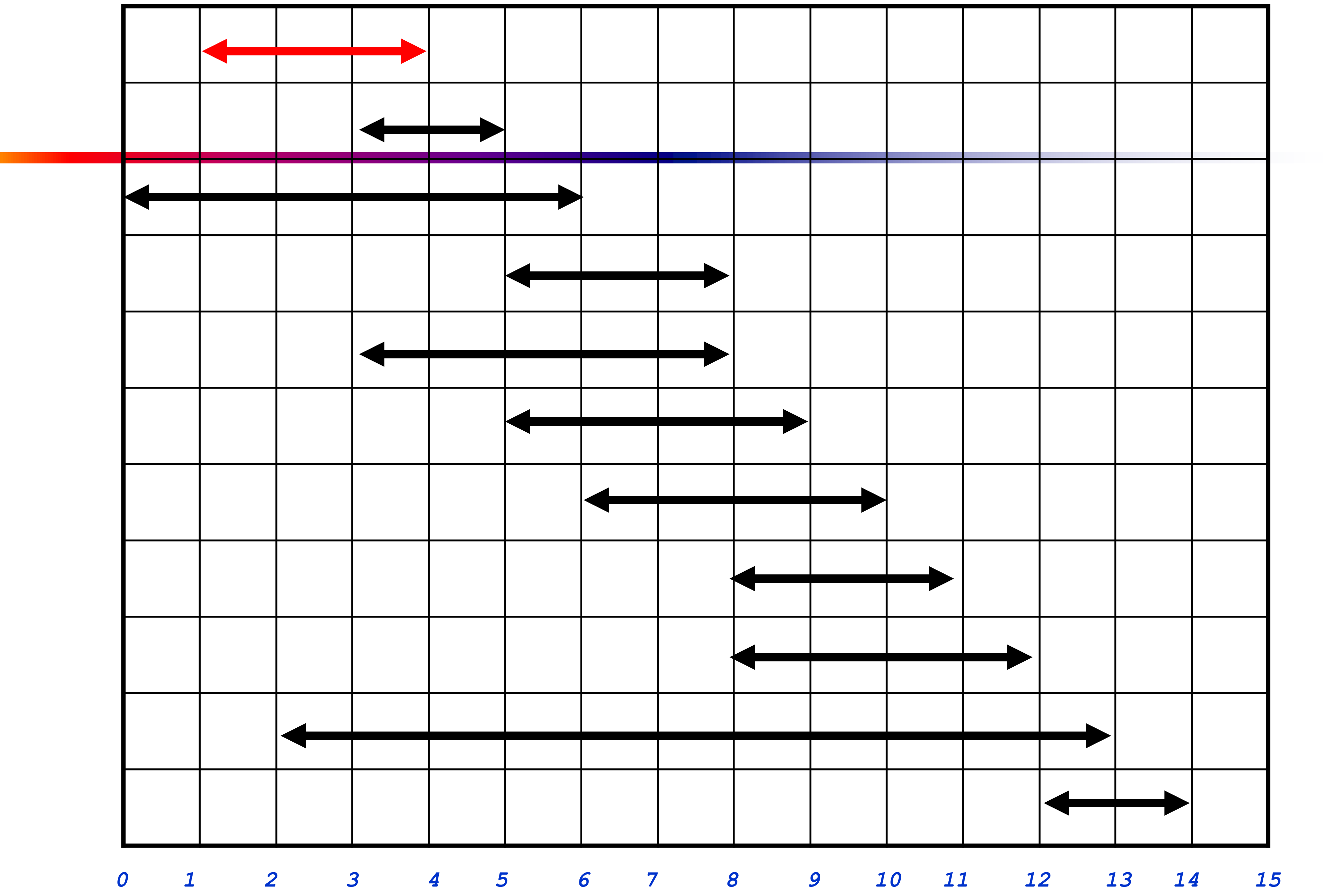

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

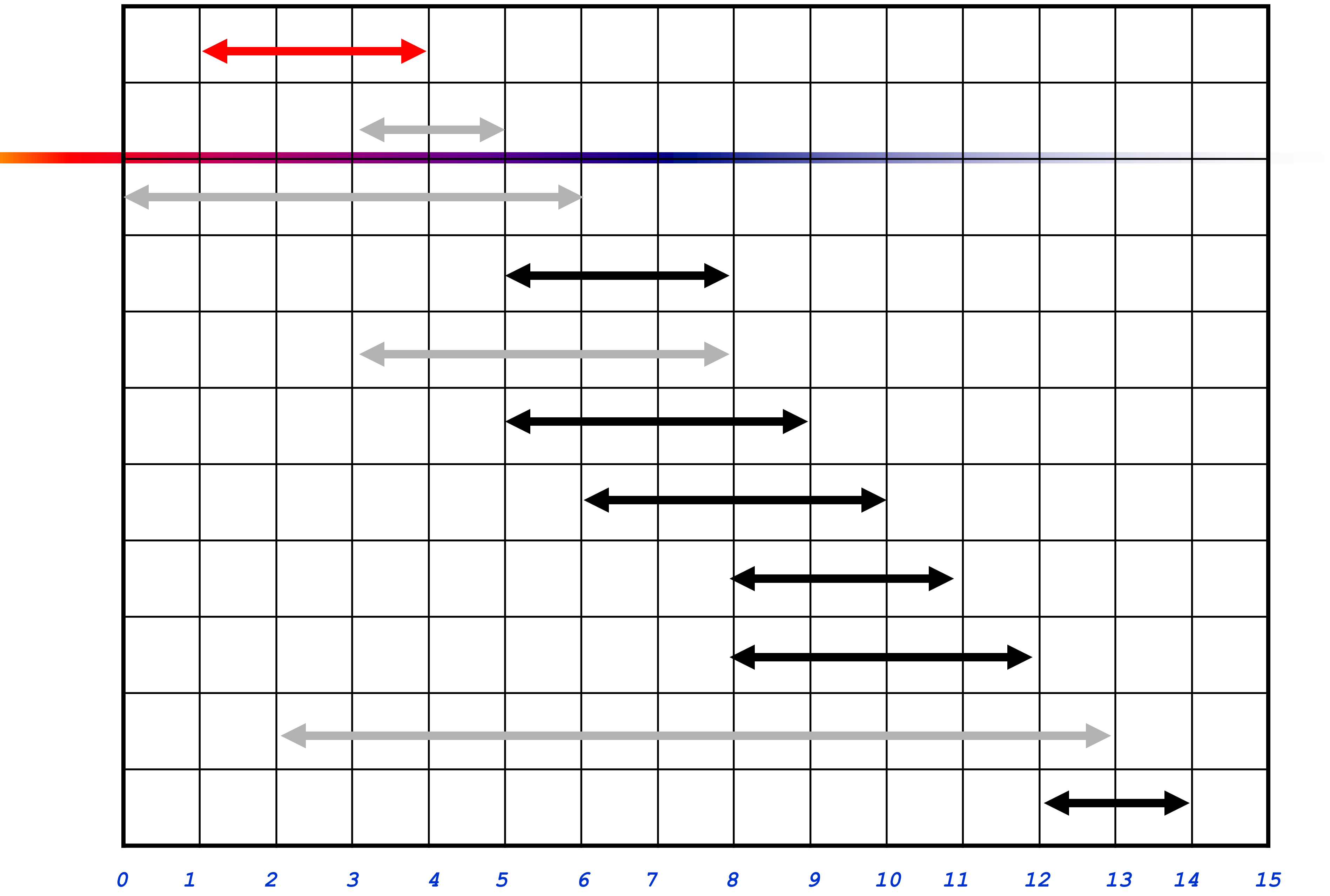

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

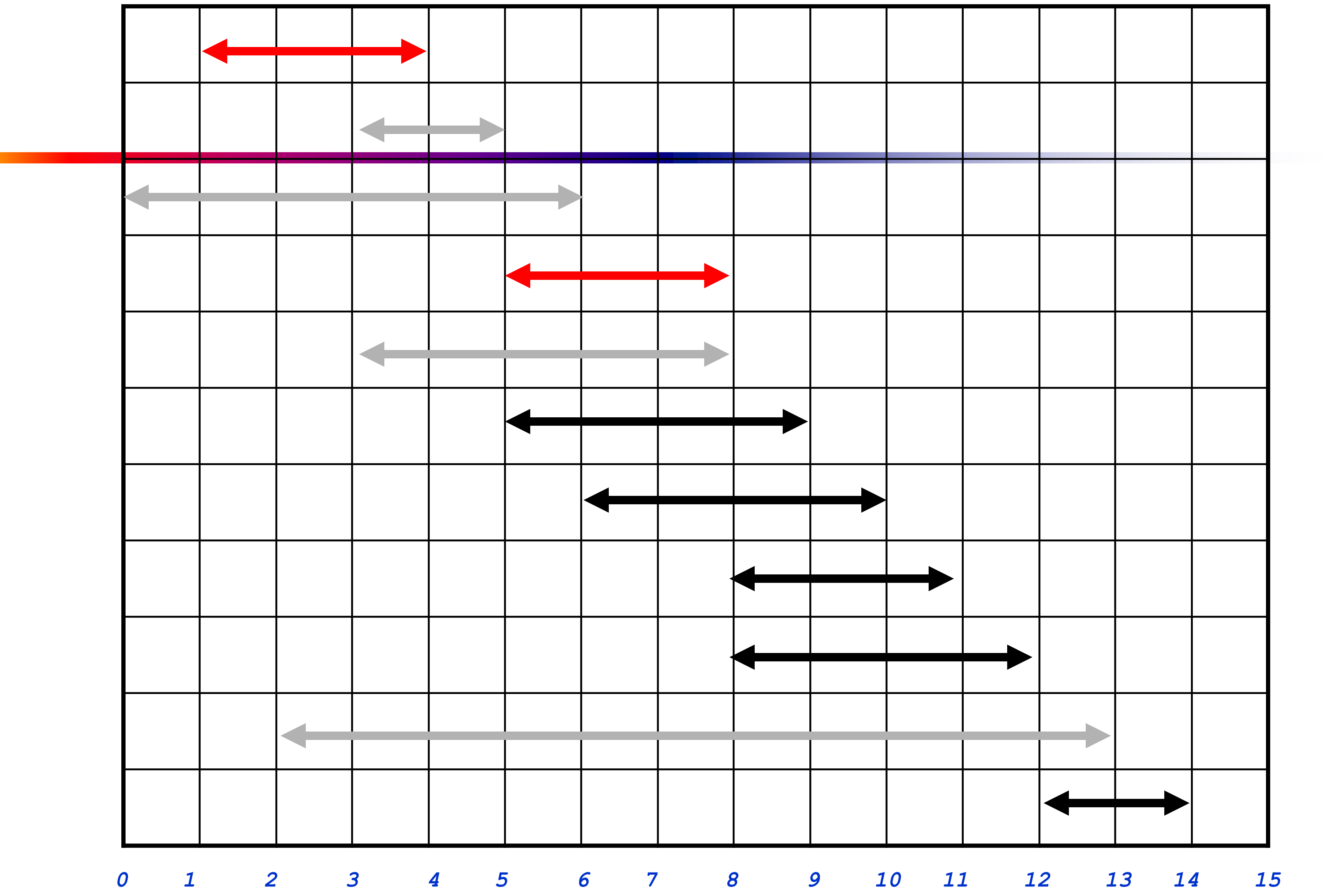

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

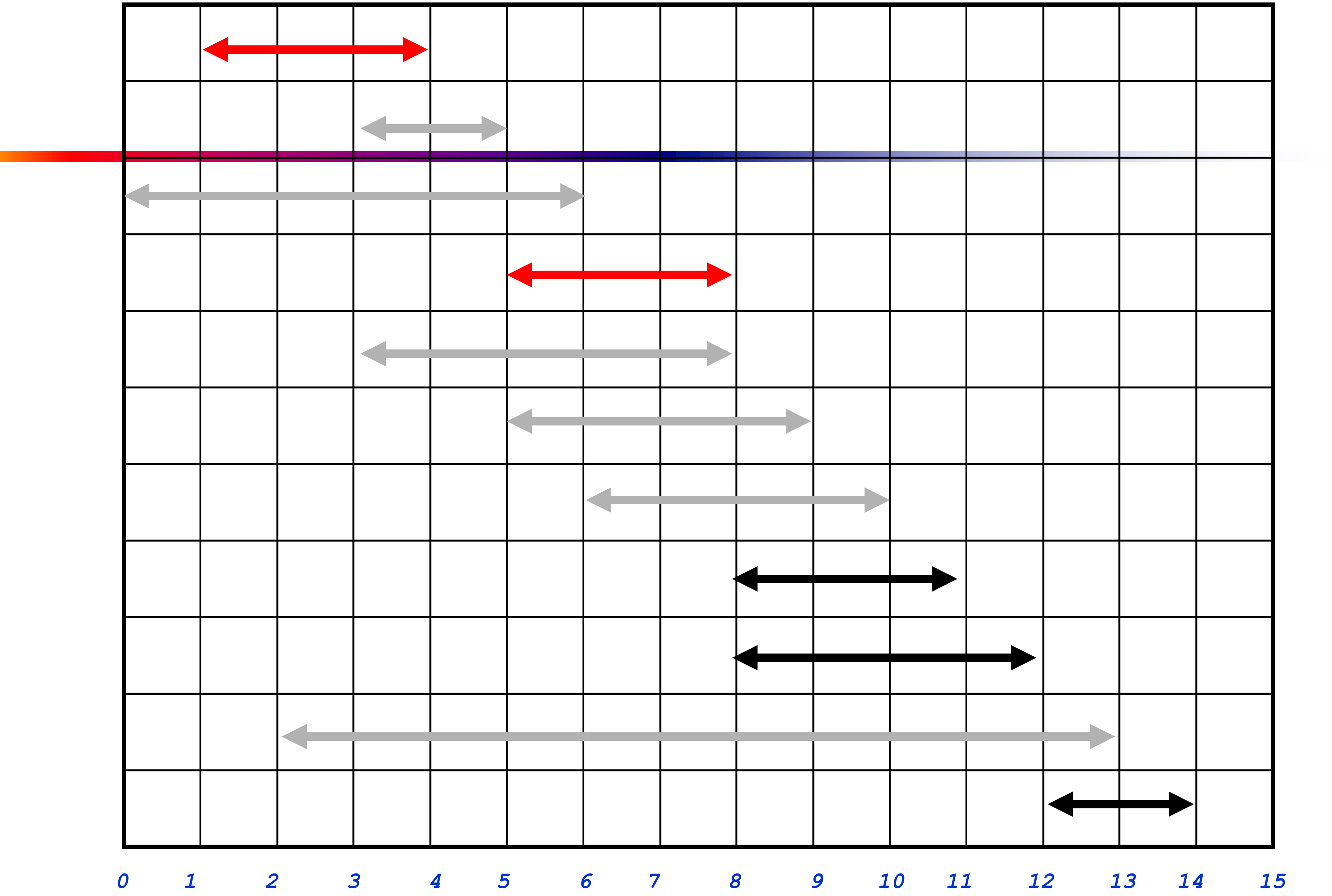
Activity Selection: A Greedy Algorithm

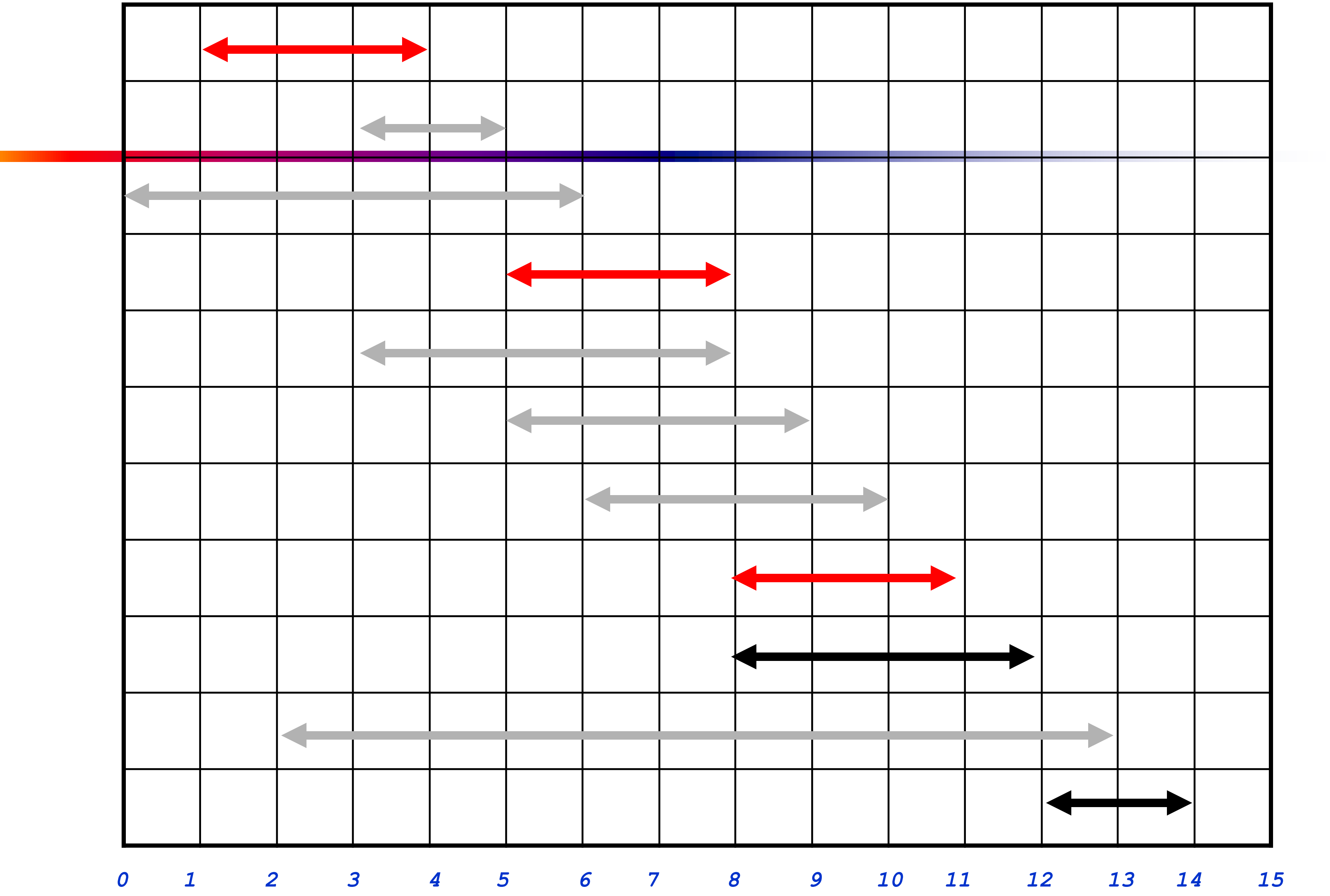
- So actual algorithm is simple:
  - Sort the activities by finish time
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes
  - Repeat until no more activities
- Intuition is even more simple:
  - Always pick the shortest ride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

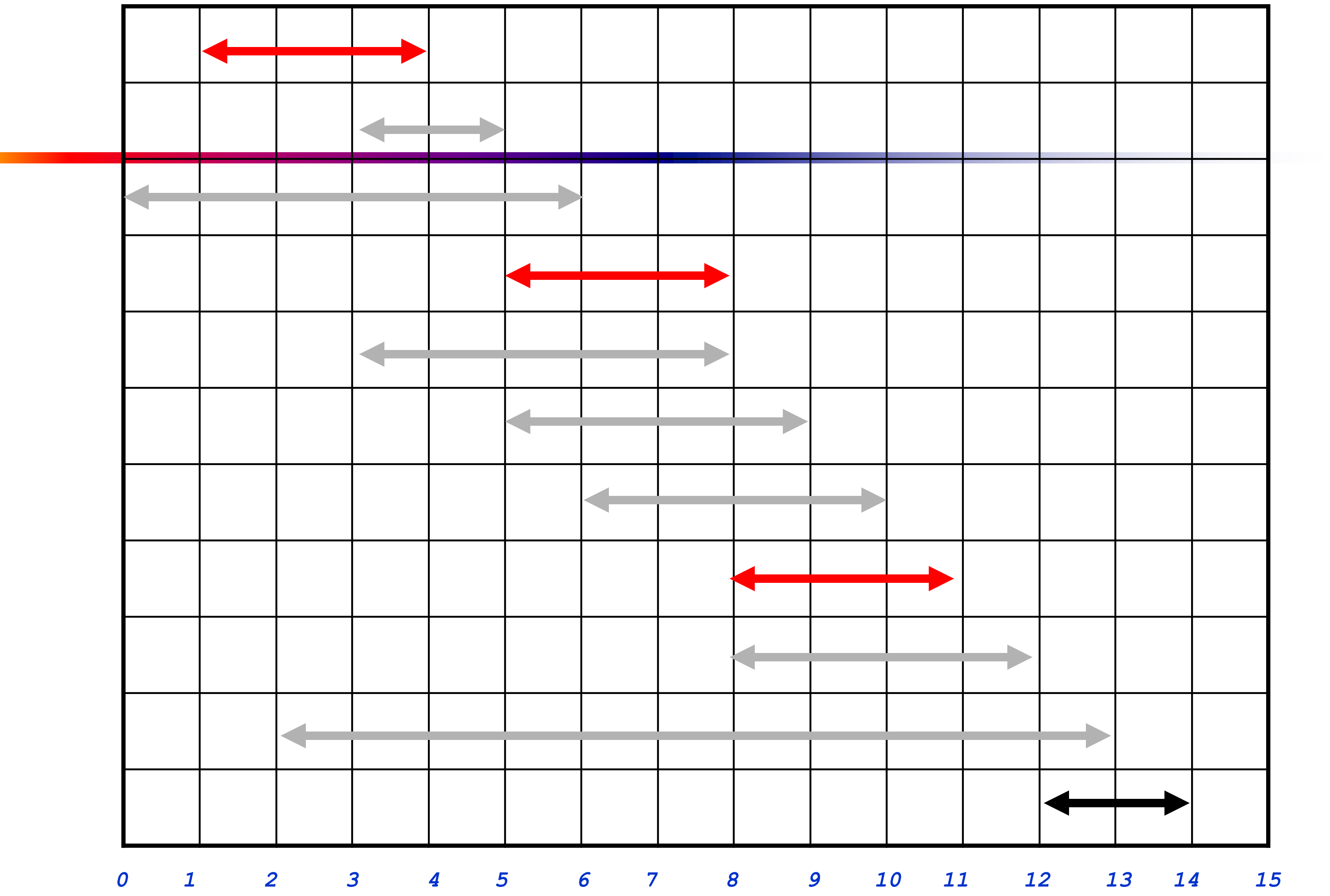

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

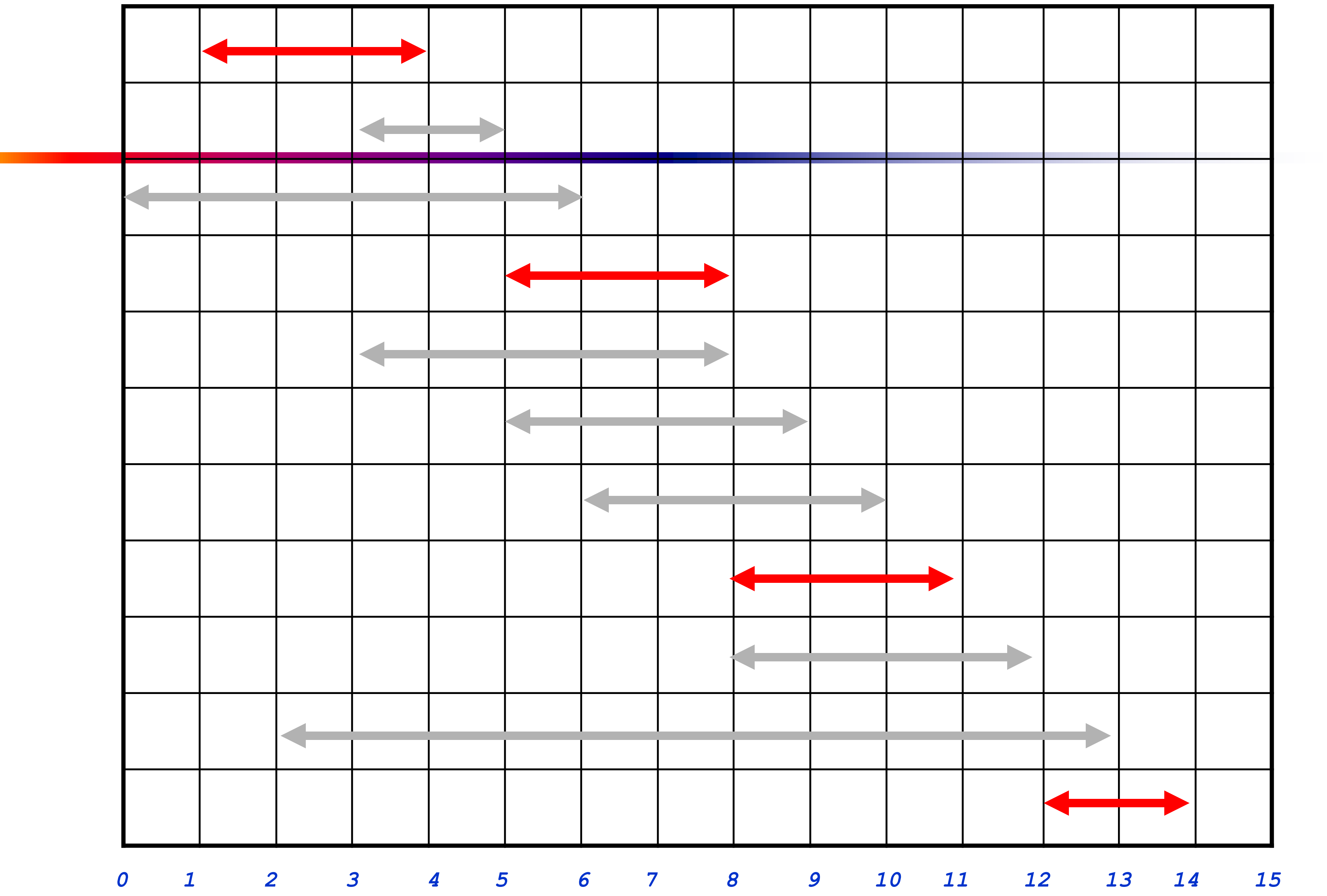

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


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

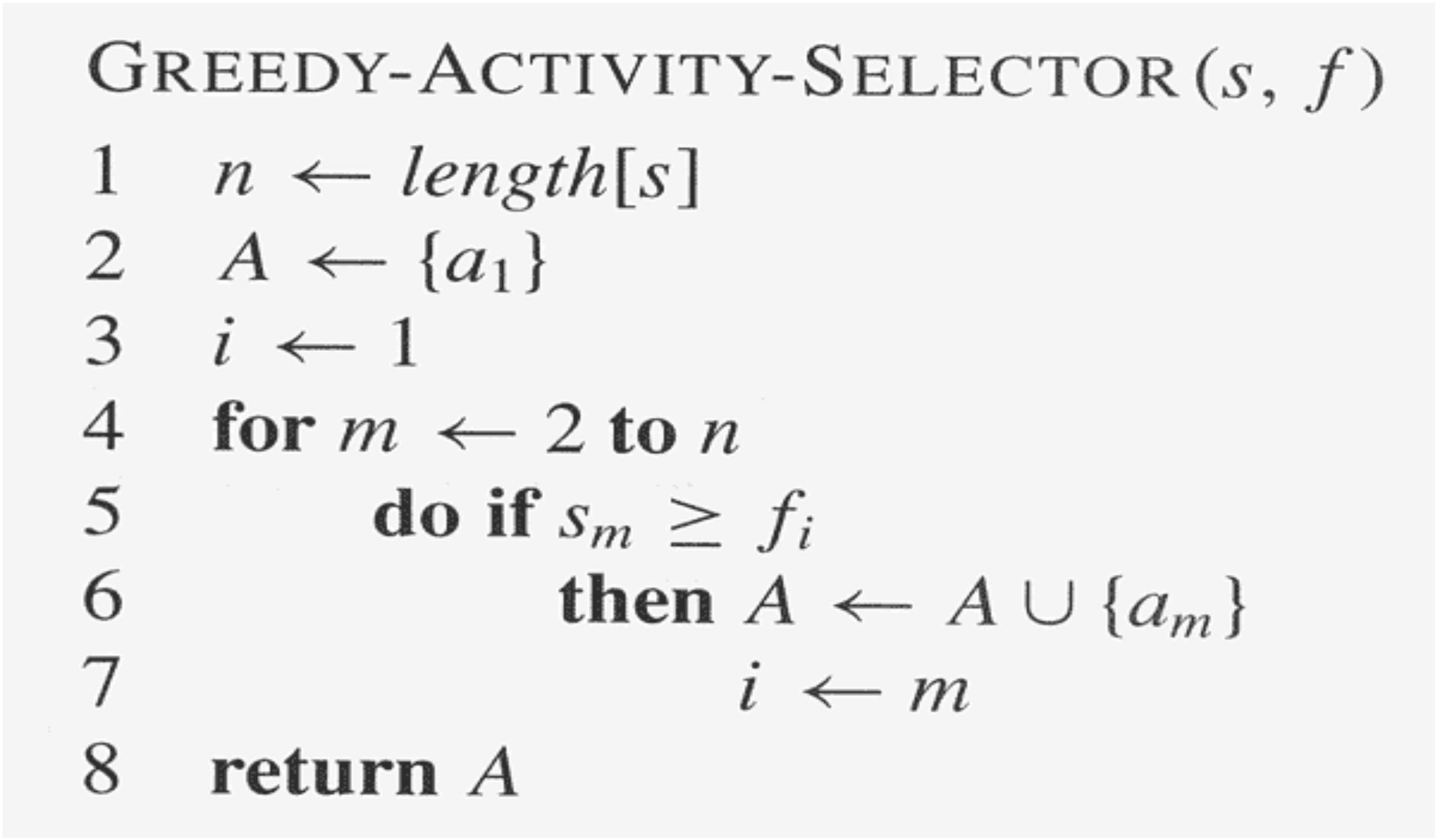

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


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 it is Greedy?

- 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

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 $A \subseteq S$ is 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

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
  - Optimal Substructure
  - If A is optimal to S, then $A' = A - \{1\}$ is optimal to $S'=\{i \in S: s_i \geq f_1\}$
  - Why?
    - 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

Elements of Greedy Strategy

- A 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

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

Optimal Substructures

- A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to sub-problems
  - If an optimal solution A to S begins with activity 1, then $A' = A - \{1\}$ is optimal to $S' = \{i \in S: s_i \geq f_1\}$

Greedy Vs. Dynamic Programming: 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?

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

The Knapsack Problem: 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

- 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*

The Knapsack Problem: Greedy Vs. Dynamic

- The fractional problem can be solved greedily
- The 0-1 problem cannot be solved with a greedy approach
  - It can, however, be solved with dynamic programming