January 19, 2022

Greedy Algorithms

Chapters 5 of Dasgupta *et al.*

Outline

- Activity selection
- Fractional knapsack
- Huffman encoding
- Later:
  - Dijkstra (single source shortest path)
  - Prim and Kruskal (minimum spanning tree)

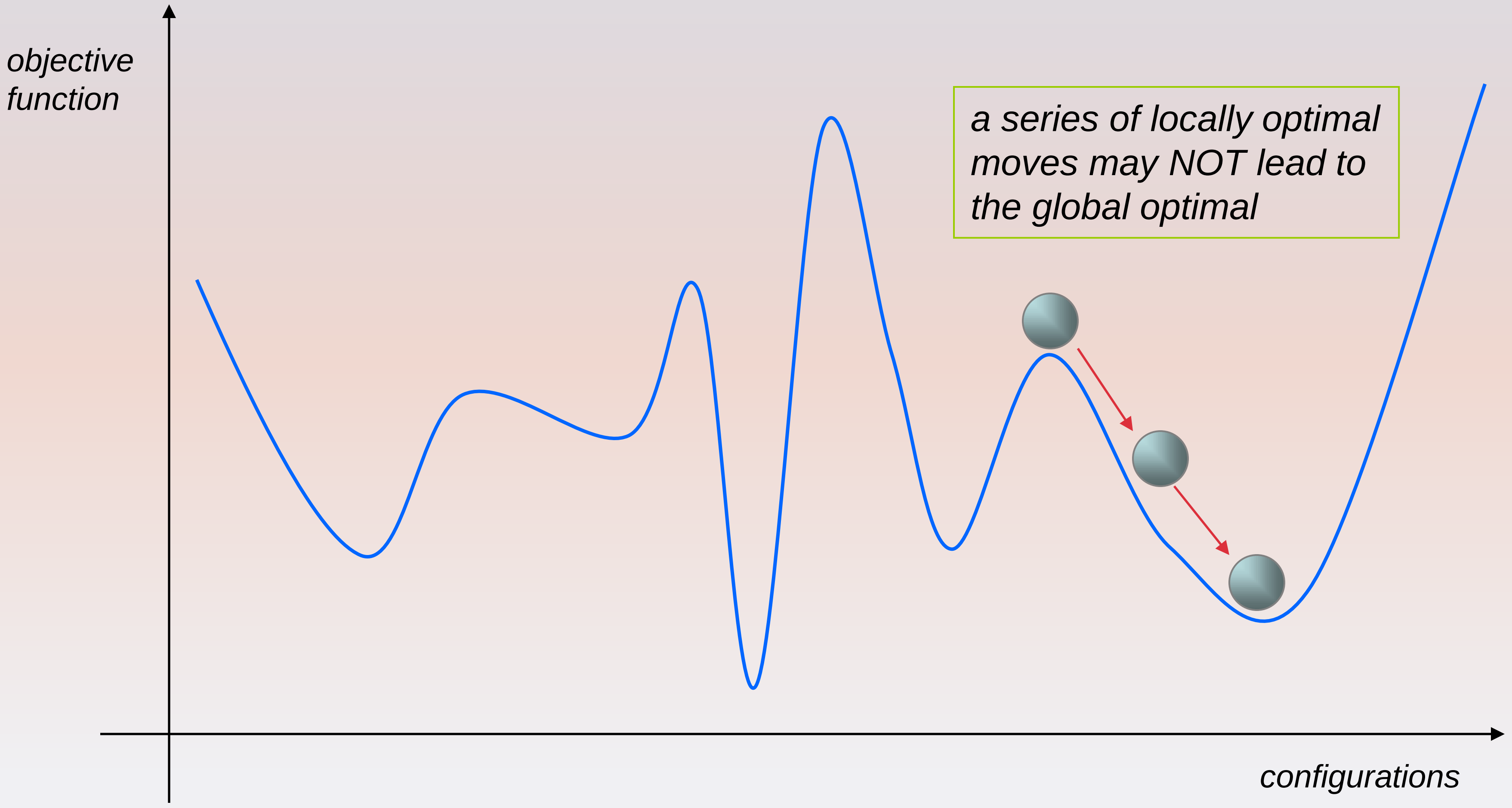
Optimization problems

- A class of problems in which we are asked to find a set (or a sequence) of “items” that satisfy some constraints and simultaneously optimize (i.e., maximize or minimize) some objective function

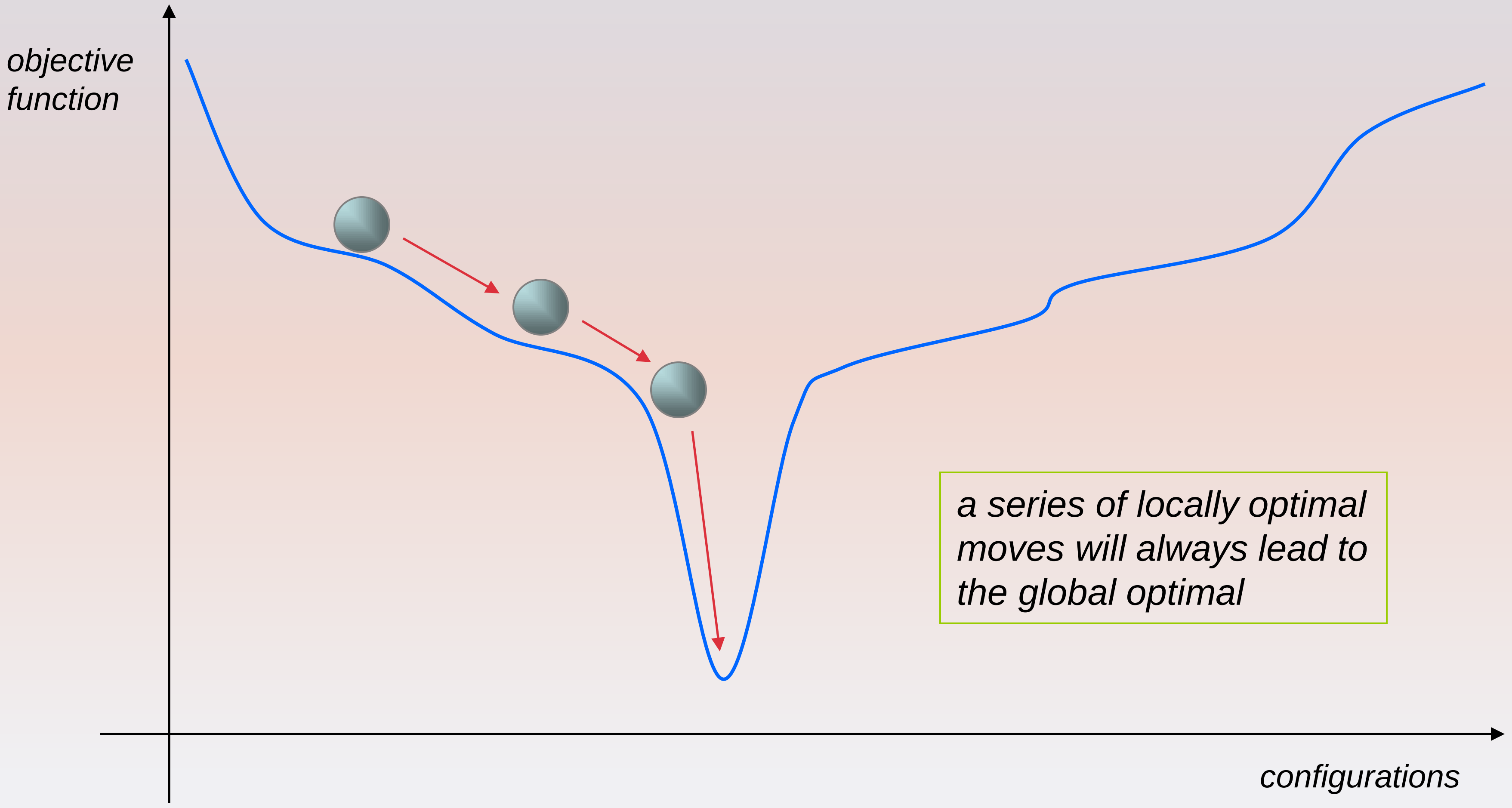
Greedy method

- Typically applied to *optimization problems*, that is, problems that involve searching through a set of *configurations* to find one that minimizes/maximizes an *objective function* defined on these configuration
- *Greedy strategy:* at each step of the optimization procedure, choose the configuration which seems the best between all of those possible

Searching for the global minimum


objective function
a series of locally optimal moves may NOT lead to the global optimal
configurations

Searching for the global minimum



a series of locally optimal moves will always lead to the global optimal

Greedy method

- There are problems for which the globally optimal solution can be found by making a series of locally optimal (greedy) choices
  - 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
- The greedy strategy does not always lead to the global optimal solution

Elements of greedy strategy

- Two ingredients that are exhibited by most problems that lend themselves to a greedy strategy
  - Greedy-choice property: a globally optimal solution can be reached by making a locally optimal choice
  - Optimal substructure: optimal solution to the problem consists of optimal solutions to sub-problems

Activity selection

(aka, “task scheduling”)

Activity Selection

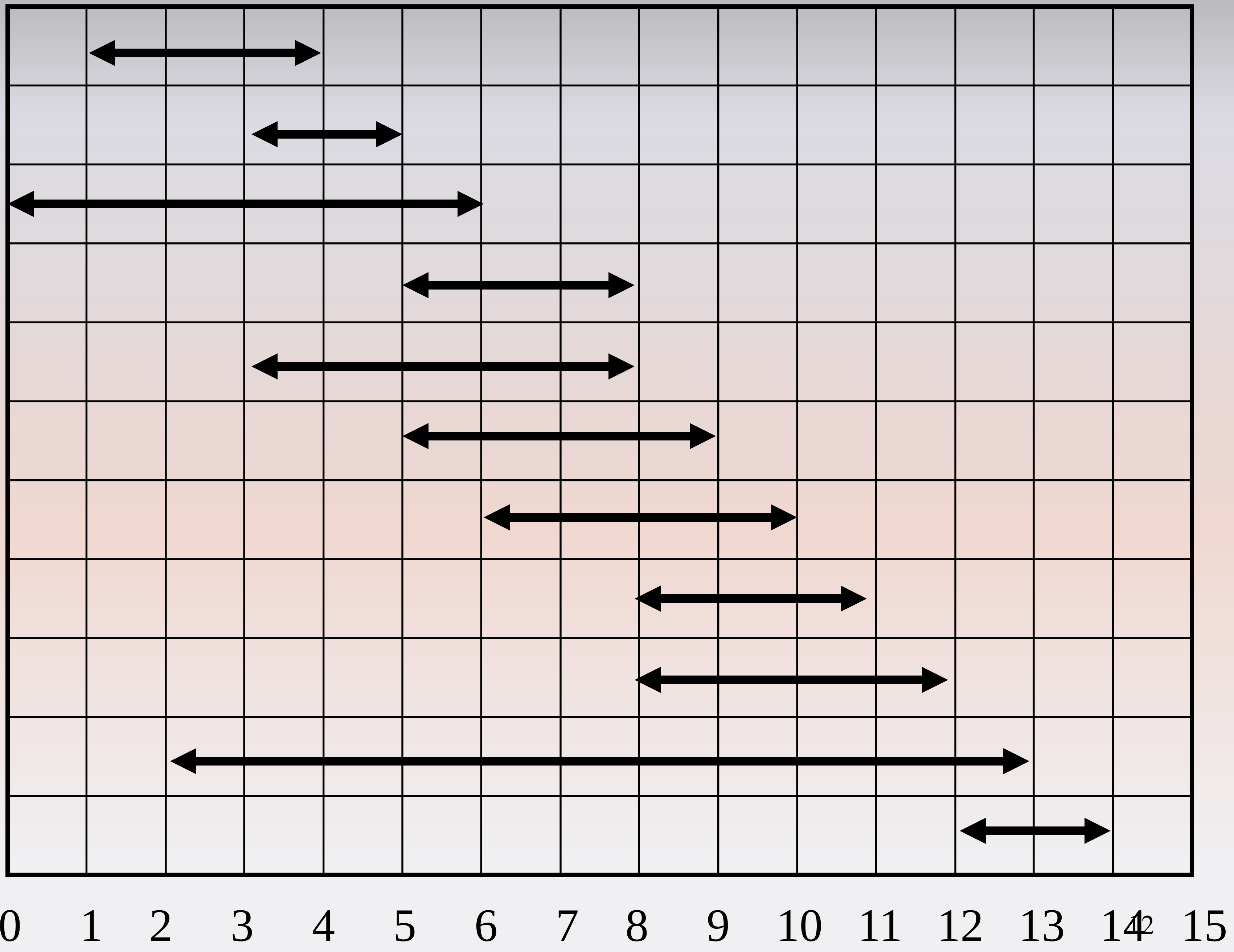
- Input: A set of activities $S = \{a_1, \ldots, a_n\}$
- Each activity has start time and a finish time $a_i = (s_i, f_i)$
- Two activities are *conflicting* if and only if their interval overlap
- Output: a maximum-size subset of non-conflicting activities

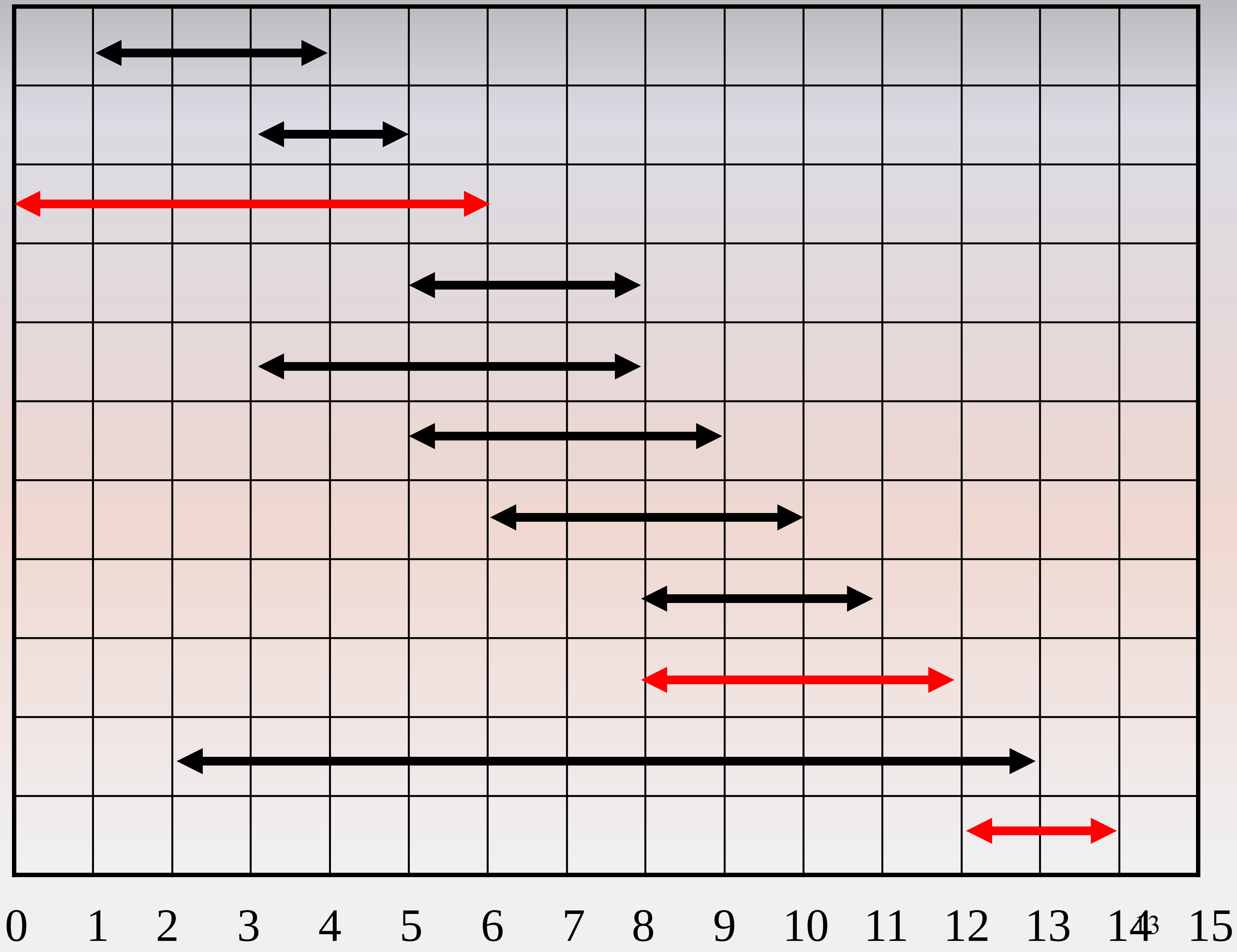
Activity Selection

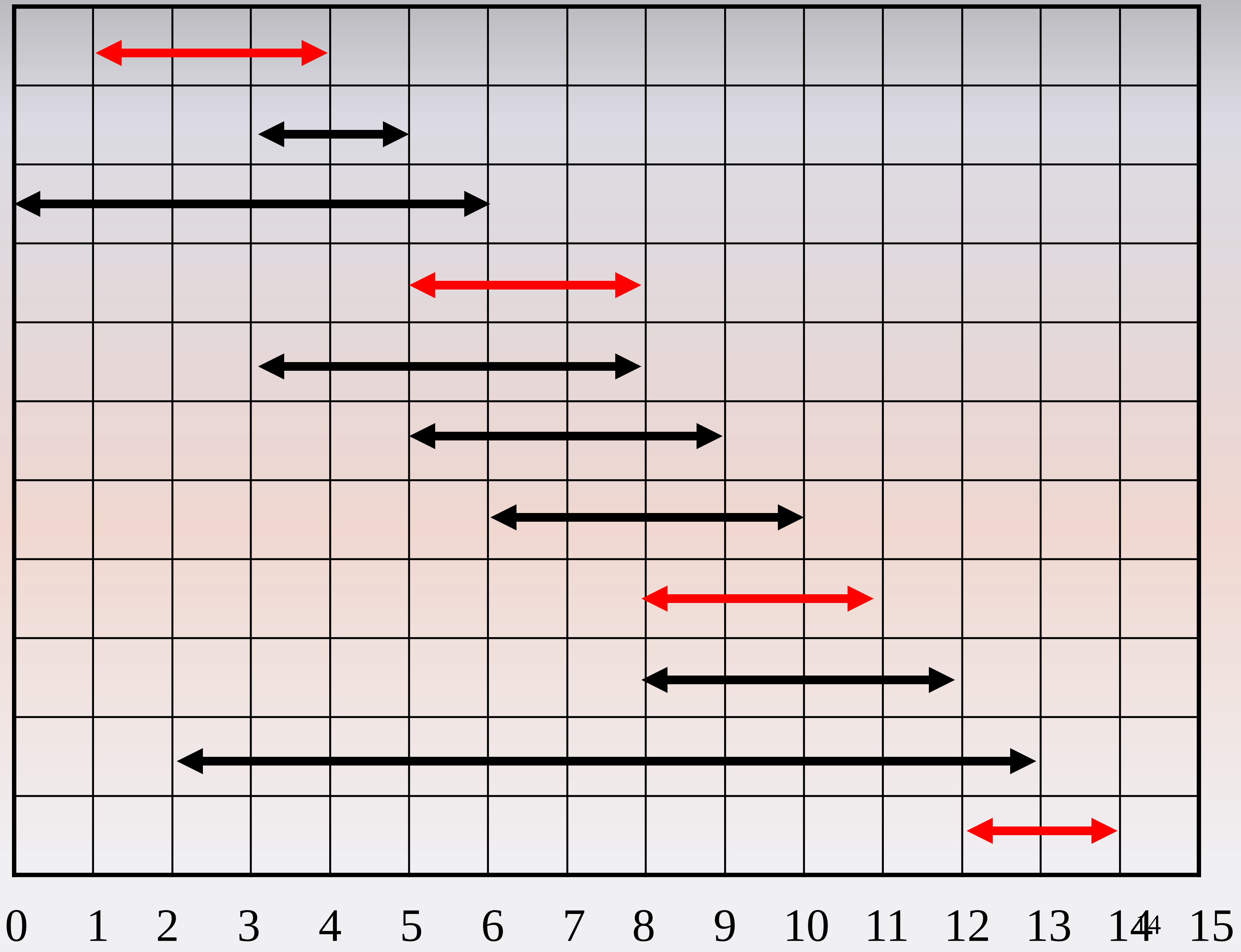
- Here are a set of start and finish times

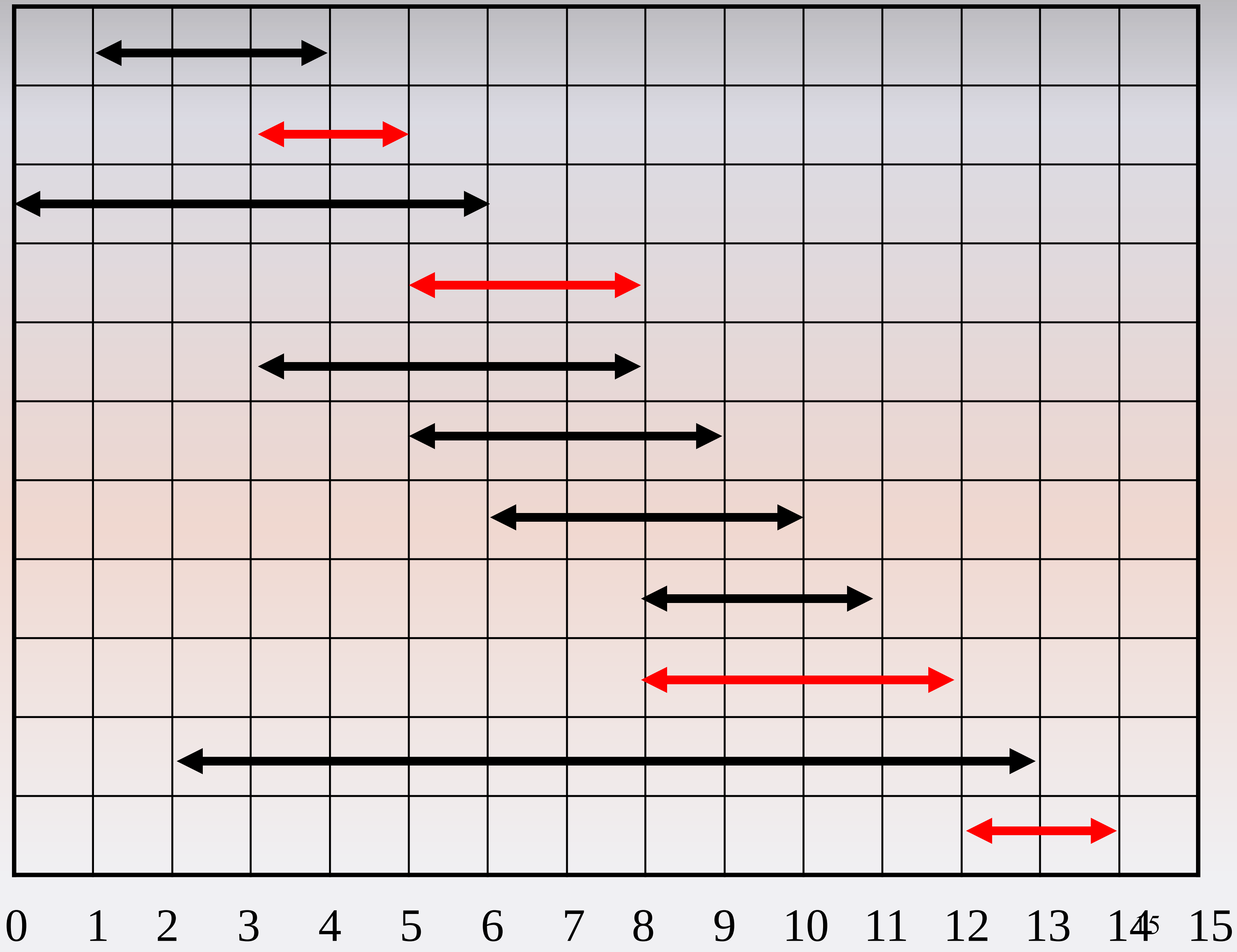
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- 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

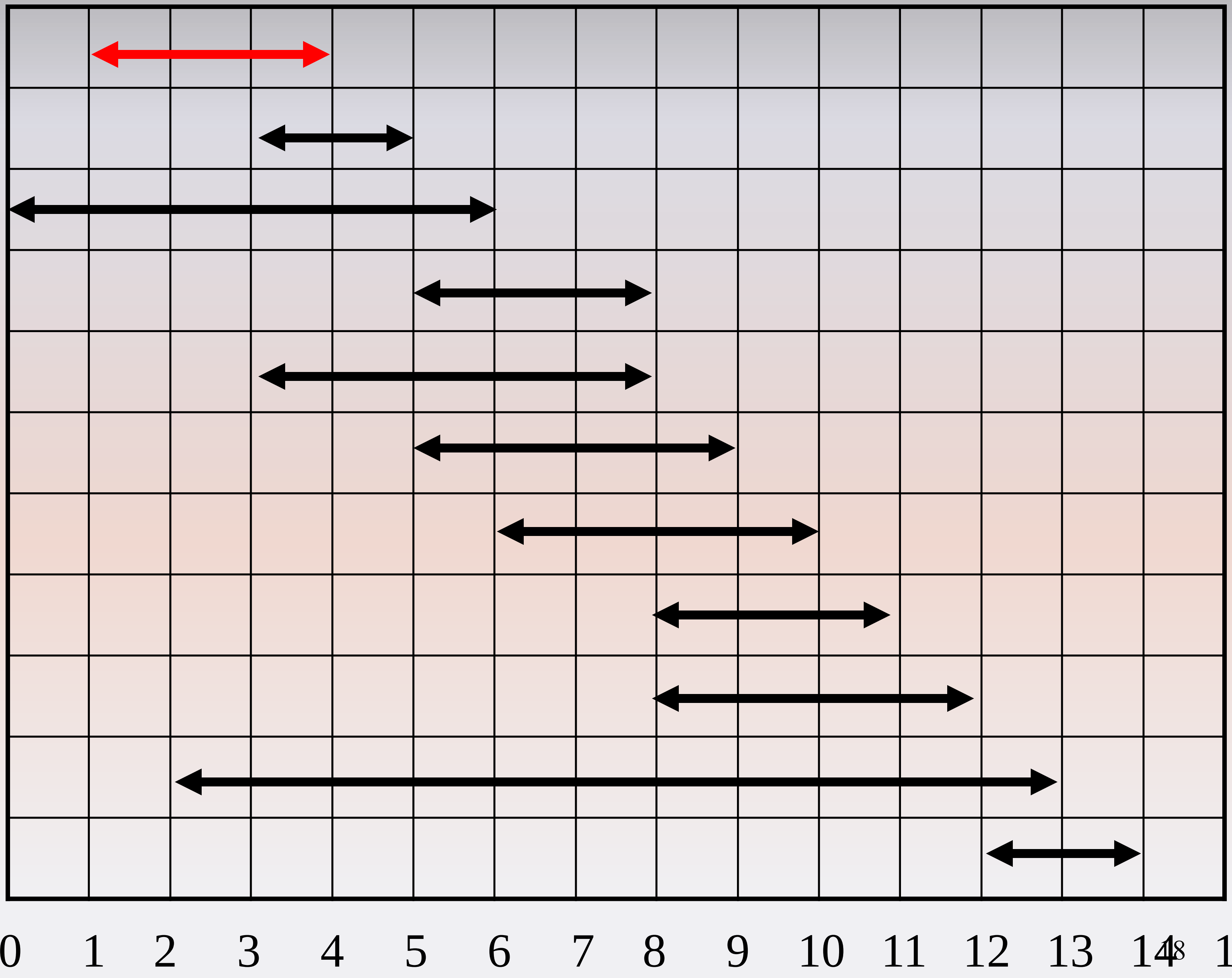

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

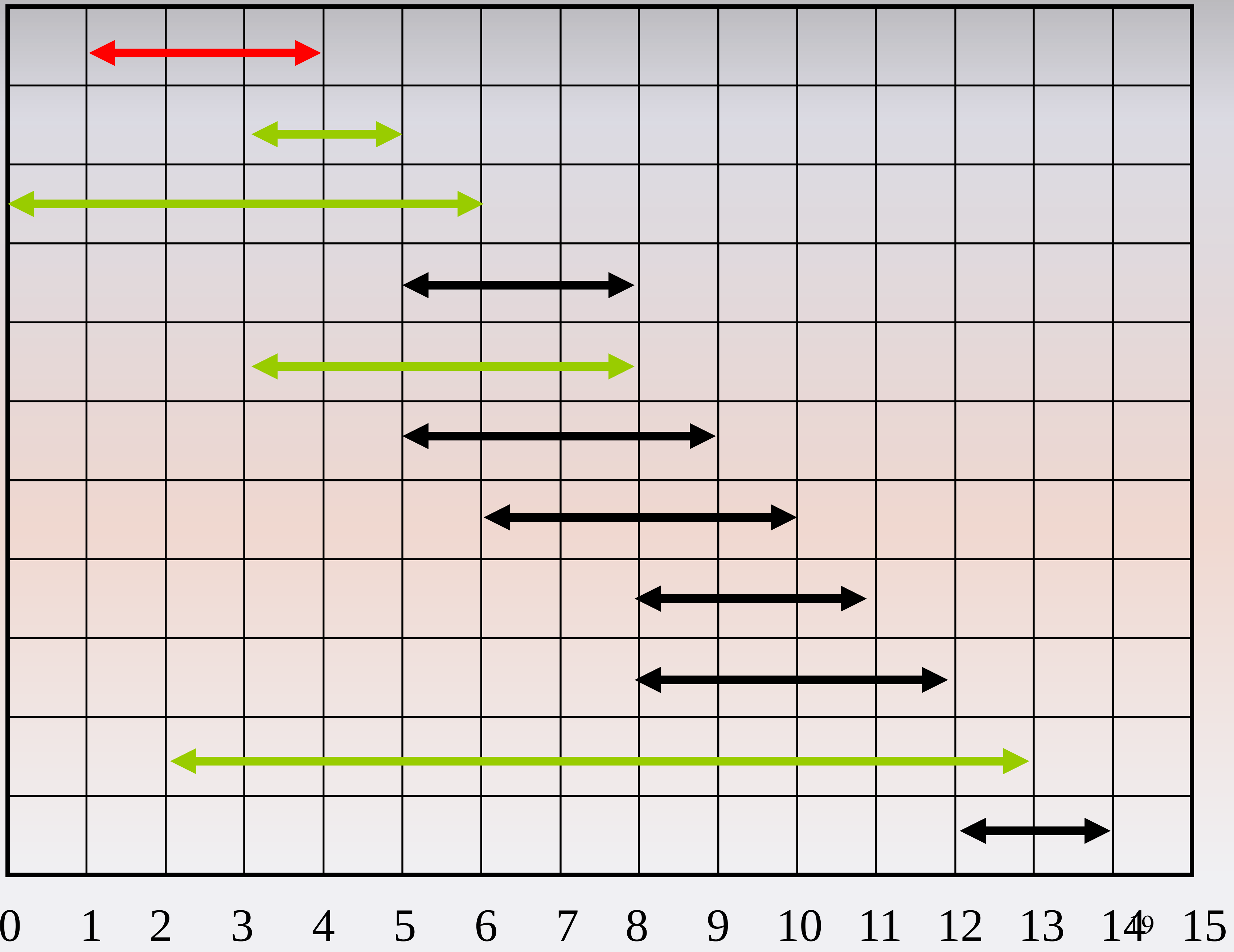
“Greedy” Strategies

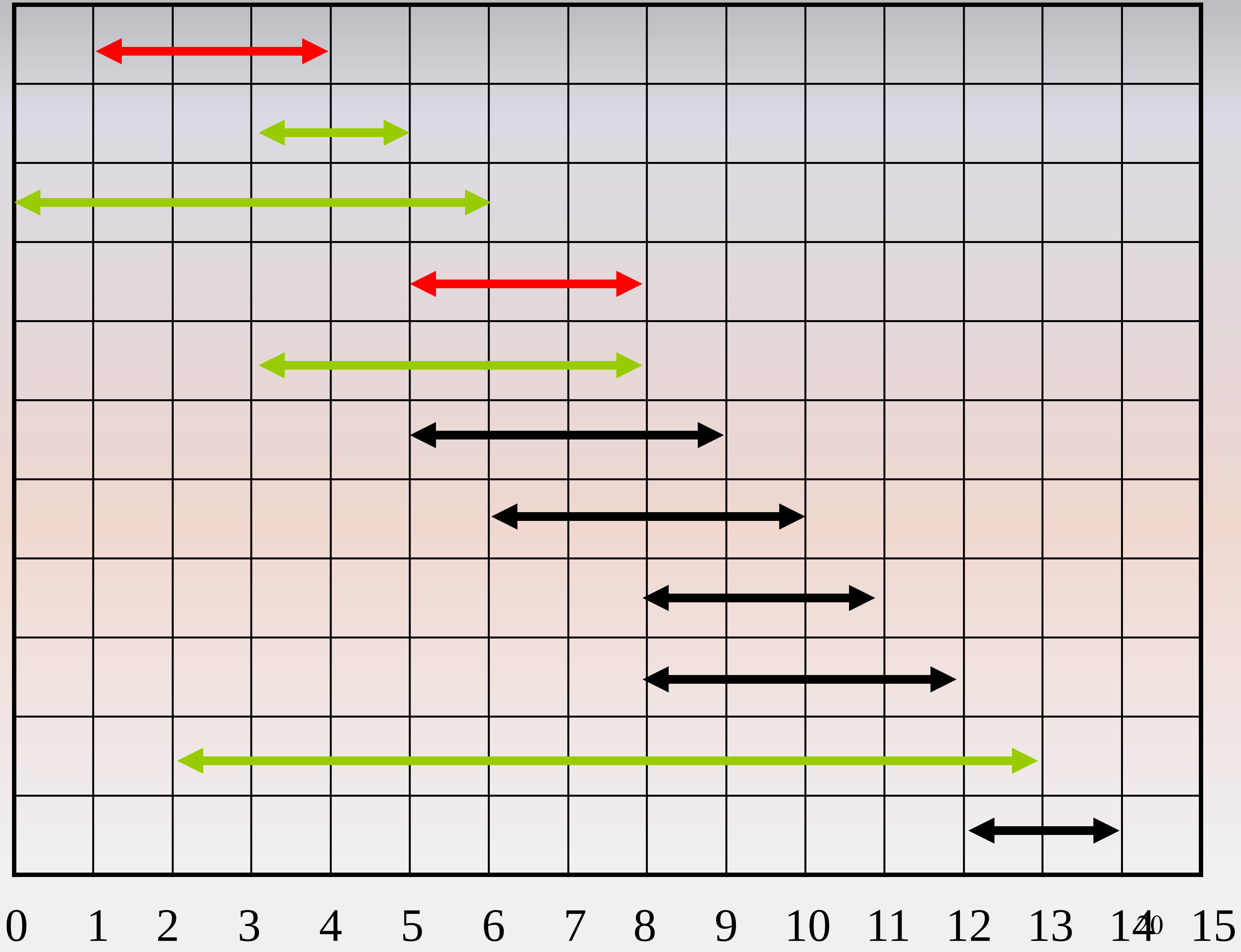
1. Longest first
2. Shortest first
3. Early start first
4. Early finish first
5. None of the above

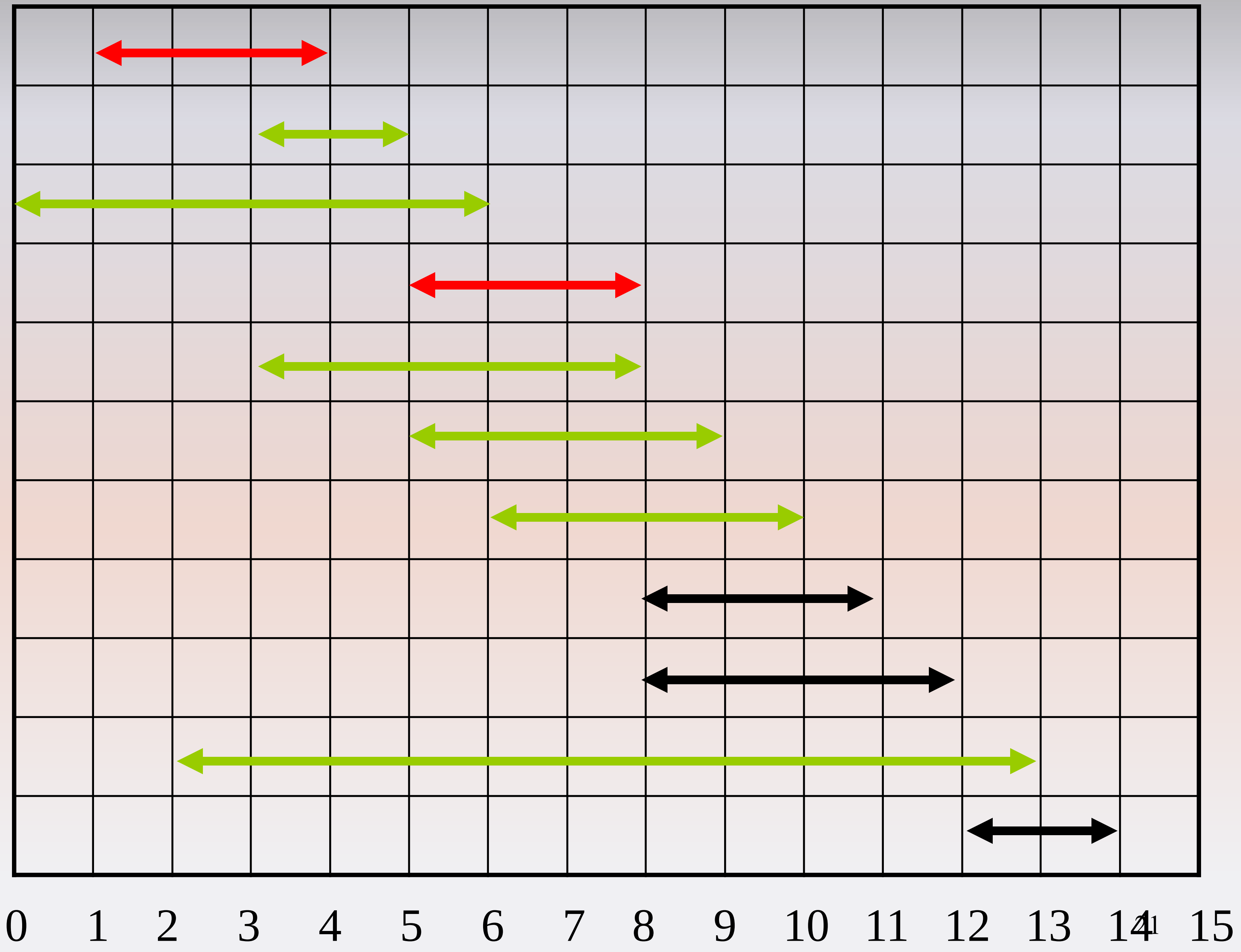
Early Finish Greedy strategy

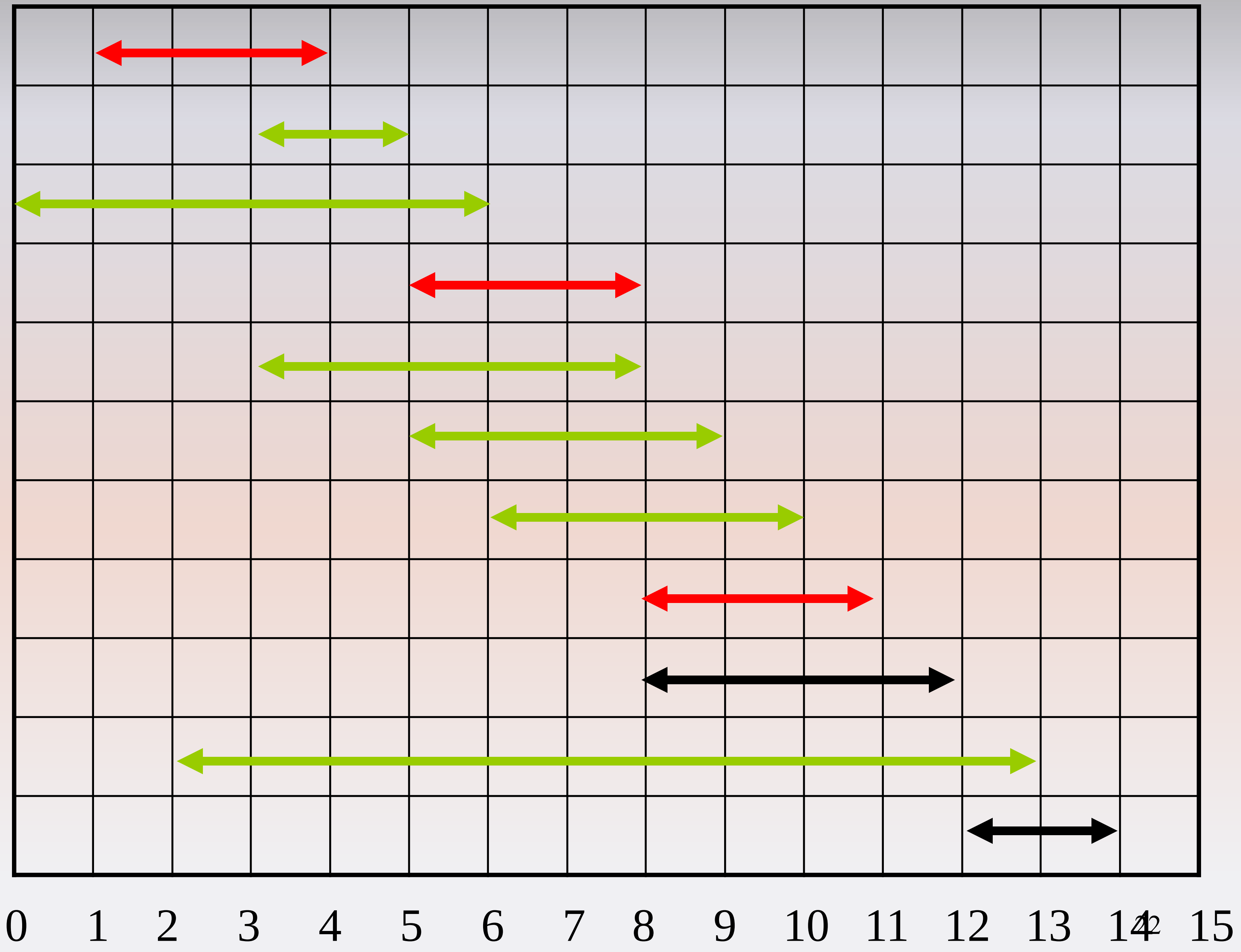
- Sort the activities by finish time
- Schedule the first activity
- Then, schedule the next activity (in sorted list) which starts after previous activity finishes (first non-conflicting task)
- Repeat until no more activities

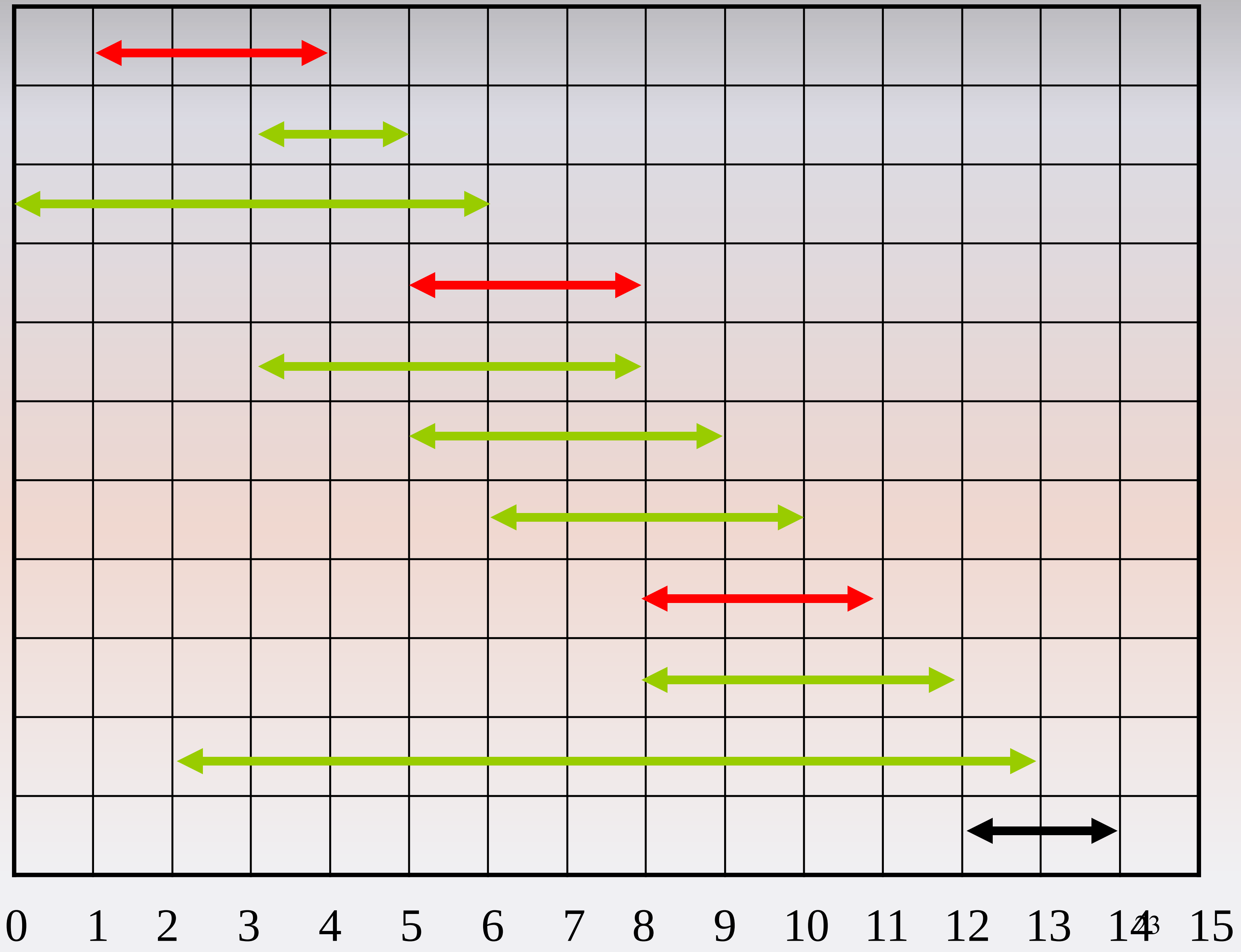

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

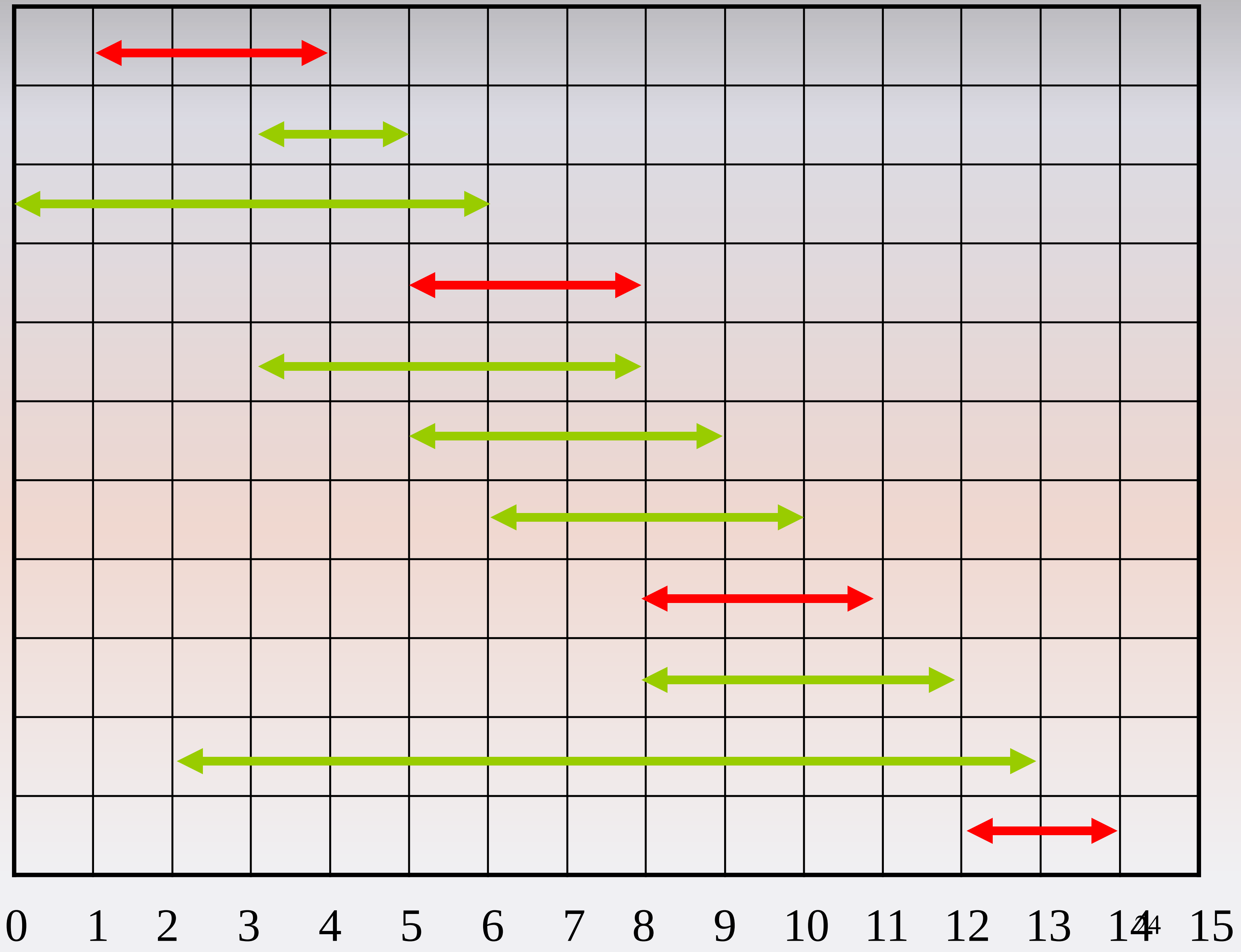

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


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

Activity selection in Python

```
def greedy_activity_selection(A):
    A.sort(key=itemgetter(1))
    result = [A[0]]
    i = 0
    for j in range(1,len(A)):
        if A[j][0] >= A[i][1]:
            result.append(A[j])
            i = j
    return result
```

Remark: sort **A** by finish time

Remark: first activity in the solution

Remark: start[j] >= finish[i]

Time complexity? *O(n log n)* to sort, the rest is linear.

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

Correctness (optimality)

- We will show that
  - the algorithm satisfies the greedy-choice property (*a globally optimal solution can be reached by making a locally optimal choice*
  - the problem has the optimal substructure property (*optimal solution to the problem consists of optimal solutions to sub-problems*)

Thus, the algorithm always finds the optimal solution

Greedy-Choice Property

- We want to show there is an optimal solution that begins with a greedy choice (i.e., with the first activity, which has the earliest finish time)

Greedy-Choice Property

- Suppose $A \subseteq S$ is an optimal solution
  - Order the activities in *A* by finish time
    Let *k* be the first activity in *A*
    - If $k = 1$, the schedule *A* begins with a greedy choice
    - If $k \neq 1$, show that there is another optimal solution *B* that begins with the greedy choice (activity 1)
  - Let $B = (A - \{k\}) \cup \{1\}$
    - Activities in *B* are non-conflicting because activities in *A* are non-conflicting, *k* is the first activity to finish and $f_1 \le f_k$
    - *B* has the same number of activities as *A* thus, *B* is optimal

Optimal Substructure

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

- Optimal Substructure: if *A* is optimal to *S* and *A* contains task *1*, 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*

Optimal Substructure

- 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

Fractional Knapsack

Fractional Knapsack

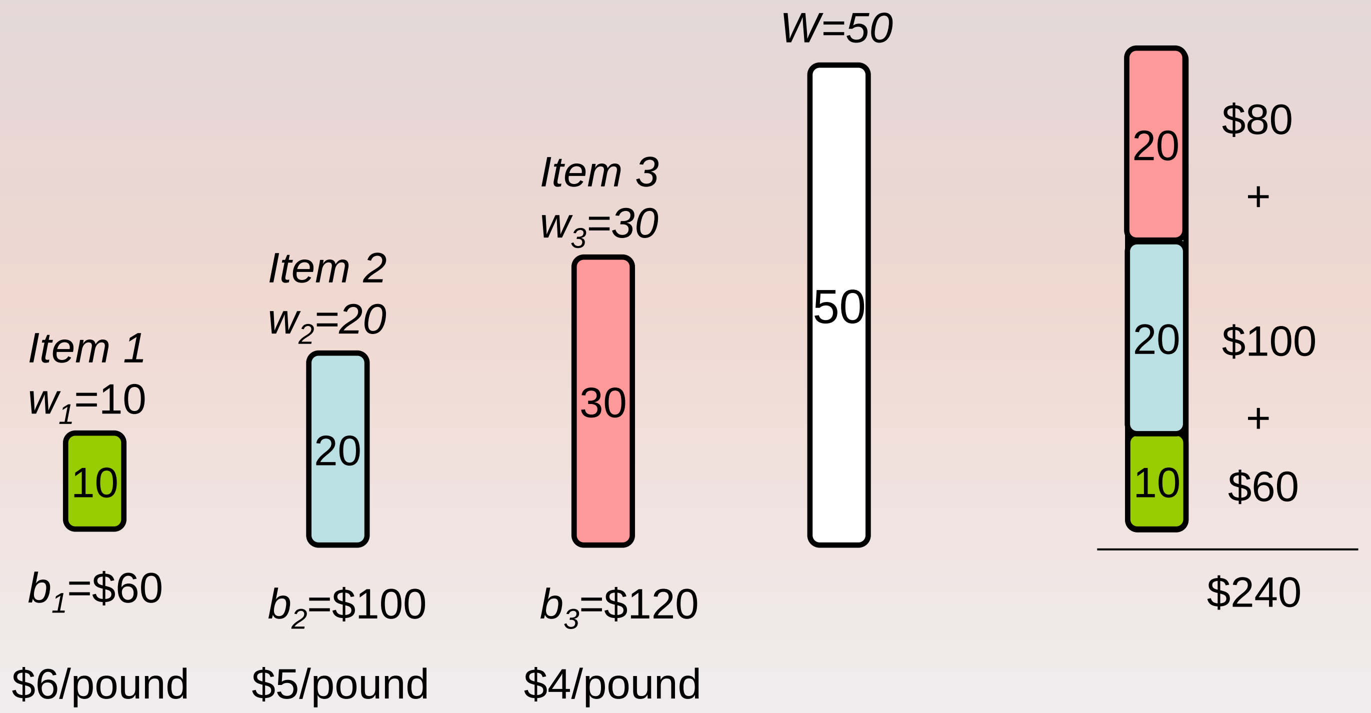
- Given a set *S* of *n* items, such that each item *i* has a positive benefit b_i and a positive weight w_i; the size of the knapsack *W*
- The problem is to find the amount x_i of each item *i* which maximizes the total benefit

  $$\sum_i b_i (x_i / w_i)$$

  under the condition that $0 \le x_i \le w_i$ and

  $$\sum_i x_i \le W$$

Fractional Knapsack - Example



Fractional Knapsack in Python

```
def fractional_knapsack(S, W):
    v = []
    for item in S:
        value = float(item[1]) / float(item[0])
        v.append((value,item[1],item[2]))
    v.sort(key=itemgetter(0))
    w, result = 0, []
    while w < W:
        high = v[-1]
        v.pop()
        a = min(high[1], W-w)
        w += a
        result.append((a,high[2]))
    return result
```

Remark: sort **v** by value = benefit/weight

Remark: select and remove the highest value (**high**)

Remark: **a** is how much item **high** we took

Fractional Knapsack

- Time complexity is *O(n log n)*
- <u>Fact</u>: Greedy strategy is optimal for the fractional knapsack problem
- <u>Proof</u>: We will show that the problem has the optimal substructure and the algorithm satisfies the greedy-choice property

Greedy Choice

Items (sorted by b_i/w_i)	1	2	3	… j …	n
“Optimal” solution:	x_1	x_2	x_3	x_j	x_n
Greedy solution:	x_1'	x_2'	x_3'	x_j'	x_n'

- We want to prove that taking <u>as much as possible</u> from item *1* is optimal
- Let us assume that is not the case, and in the optimal solution $x_1 < x_1'$
- Because of taking more of item *1* in the greedy solution, we have to decrease the quantity taken of some other item *j*
- Therefore, in the greedy solution x_j is decreased by $(x_1' - x_1)$
- In the greedy solution, we gain $(x_1' - x_1)b_1/w_1$, and we lose $(x_1' - x_1)b_j/w_j$

$$(x_1' - x_1)\, b_1/w_1 \geq (x_1' - x_1)\, b_j/w_j \ \ ?$$

$$\frac{b_1}{w_1} \geq \frac{b_j}{w_j}$$

True, since x_1 had the best benefit/weight ratio

Optimal substructure

Items:	1	2	3	…	*j*	…	*n*
Solution *U*:	x_1	x_2	x_3	…	x_j	…	x_n
Solution *U'*:	*0*	x_2'	x_3'	…	x_j'	…	x_n'

- *(S,W)* is the original problem, <u>assume</u> *U* is optimal for *(S,W)*
- *S'* is the sub-problem *{2,3,…,n}*
- *U* contains the greedy choice x_1
- <u>Prove</u> that *U'* is optimal for $(S', W\text{-}x_1)$ where $x_i' = x_i$ for all $i>1$
- By contradiction: if *U'* was not optimal, then *U"* exists such that
- But $\sum_{2 \le i \le n} x_i''(b_i / w_i) > \sum_{2 \le i \le n} x_i'(b_i / w_i)$

$$\sum_{1 \le i \le n} x_i(b_i / w_i) = \sum_{2 \le i \le n} x_i'(b_i / w_i) + x_1(b_1 / w_1) < \sum_{2 \le i \le n} x_i''(b_i / w_i) + x_1(b_1 / w_1)$$

which means that *U* was not optimal for *(S,W)* → contradiction

Huffman codes

Data Compression

- Text files are usually stored by representing each character with an 8-bit ASCII code
- The ASCII encoding is an example of fixed-length encoding, where each character is represented with the same number of bits
- In order to reduce the space required to store a text file, we can exploit the fact that some characters are more likely to occur than others

Data Compression

- Variable-length encoding uses binary codes of different lengths for different characters; thus, we can assign fewer bits to frequently used characters, and more bits to rarely used characters
- Huffman coding (section 5.2)

File Compression: Example

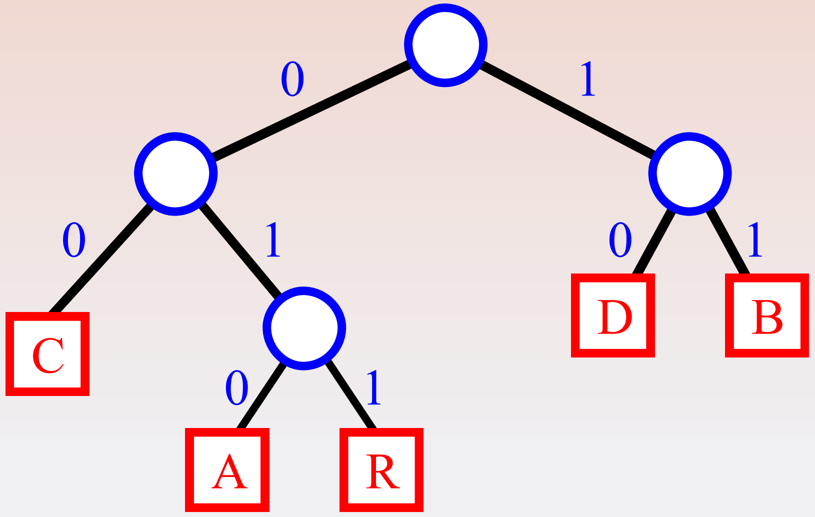
- An example
  - text: “java”
  - encoding: a = “0”, j = “11”, v = “10”
  - encoded text: 110100 (6 bits)
- How to decode in the case of ambiguity?
  - encoding: a = “0”, j = “11”, v = “00”
  - encoded text: 11<u>0000</u> (6 bits)
  - could be “<u>java</u>”, or “<u>jvv</u>”, or “<u>jaaaa</u>”, or …

Encoding

- To prevent ambiguities in decoding, we require that the encoding satisfies the prefix rule: no code is a prefix of another
- Example
  - a = “0”, j = “11”, v = “10” satisfies the prefix rule
  - a = “0”, j = “11”, v= “00” does not satisfy the prefix rule (the code of 'a' is a prefix of the codes of 'v')

Trie

- We use an encoding trie to satisfy this prefix rule
  - the characters are stored at the external nodes
  - a left child (edge) means 0
  - a right child (edge) means 1



A = 010

B = 11

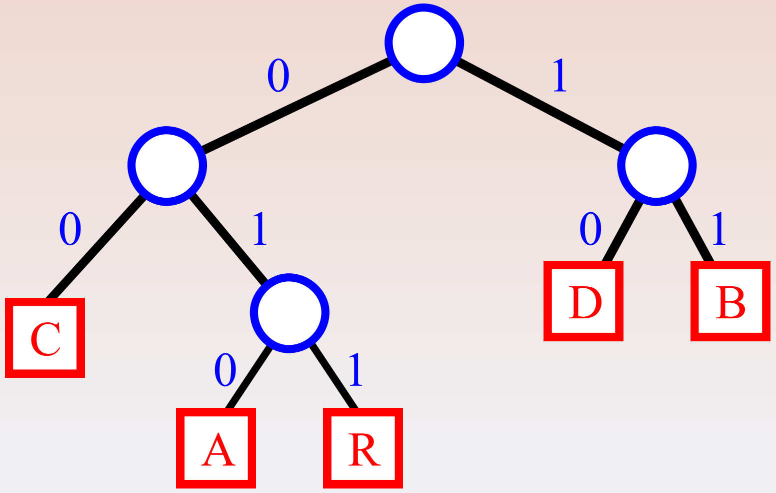
C = 00

D = 10

R = 011

Example of Decoding

- encoded text:
  01011011010000101001011011010
- text: ABRACADABRA (11 bytes=88 bits)



A = 010

B = 11

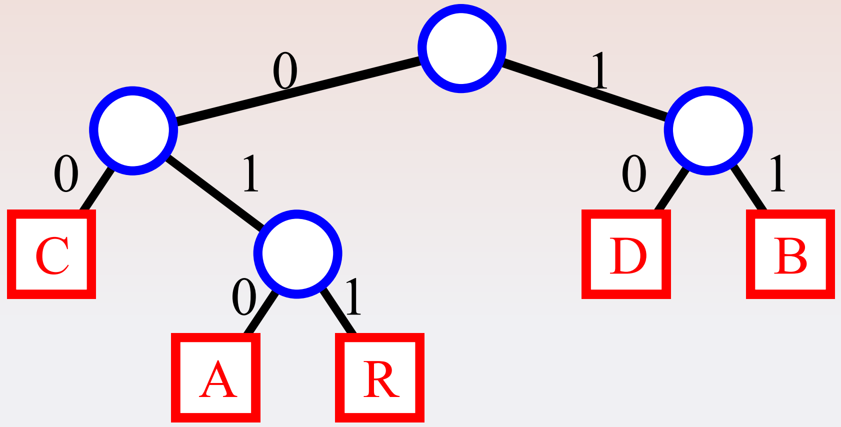
C = 00

D = 10

R = 011

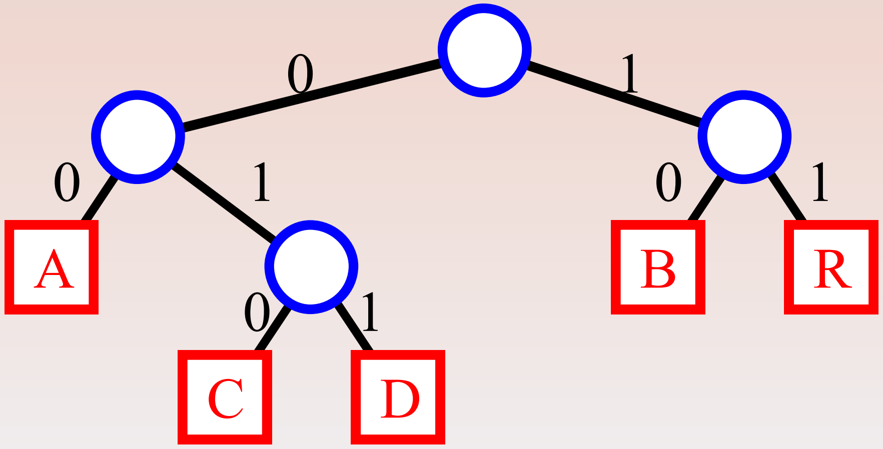
Data Compression

- Problem: We want the encoded text as short as possible
- Example: ABRACADABRA
  01011011010000101001011011010 **29 bits**



Data Compression

- Example2: ABRACADABRA
  001011000100001100101100 **24 bits**



Optimization problem

- Given a character c in the alphabet Σ
  - let $f(c)$ be the frequency of c in the file
  - let $d_T(c)$ be the depth of c in the tree = the length of the codeword
- We want to minimize the number of bits required to encode the file, that is

$$\min_{\substack{\text{binary trees } T \\ \text{with } |\Sigma| \text{ leaves}}} \sum_{c \in \Sigma} f(c) d_T(c)$$

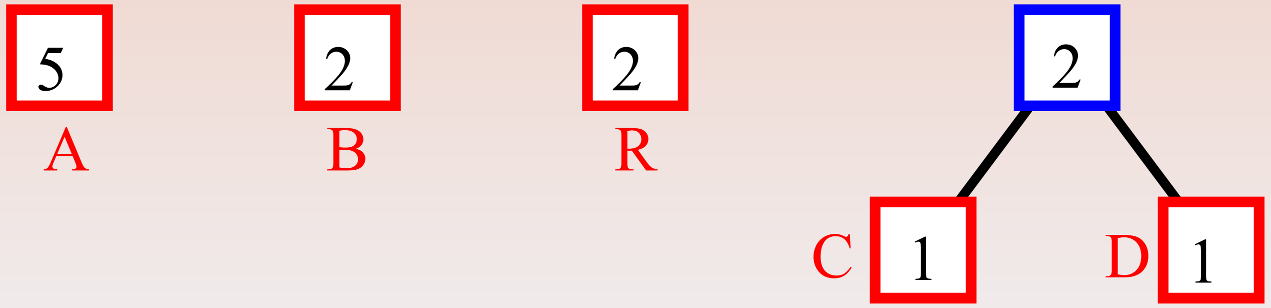
Huffman Encoding: Example

Step *0*

ABRACADABRA

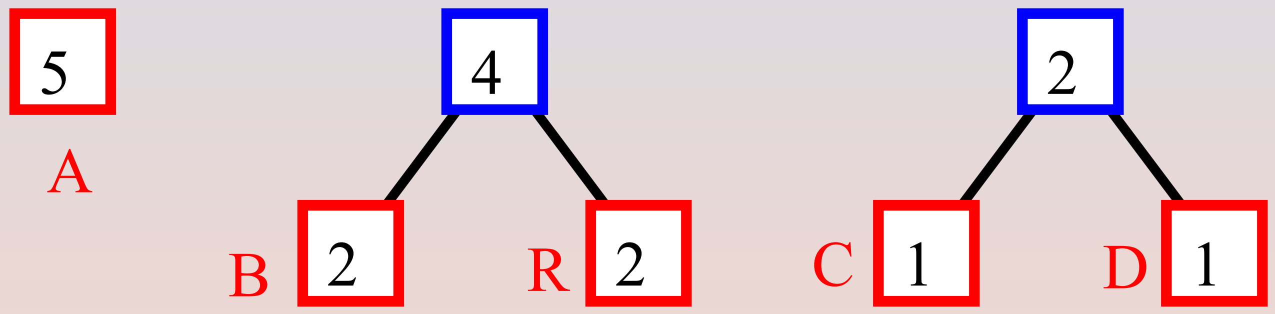
character	A	B	R	C	D
frequency	5	2	2	1	1

Step *1*

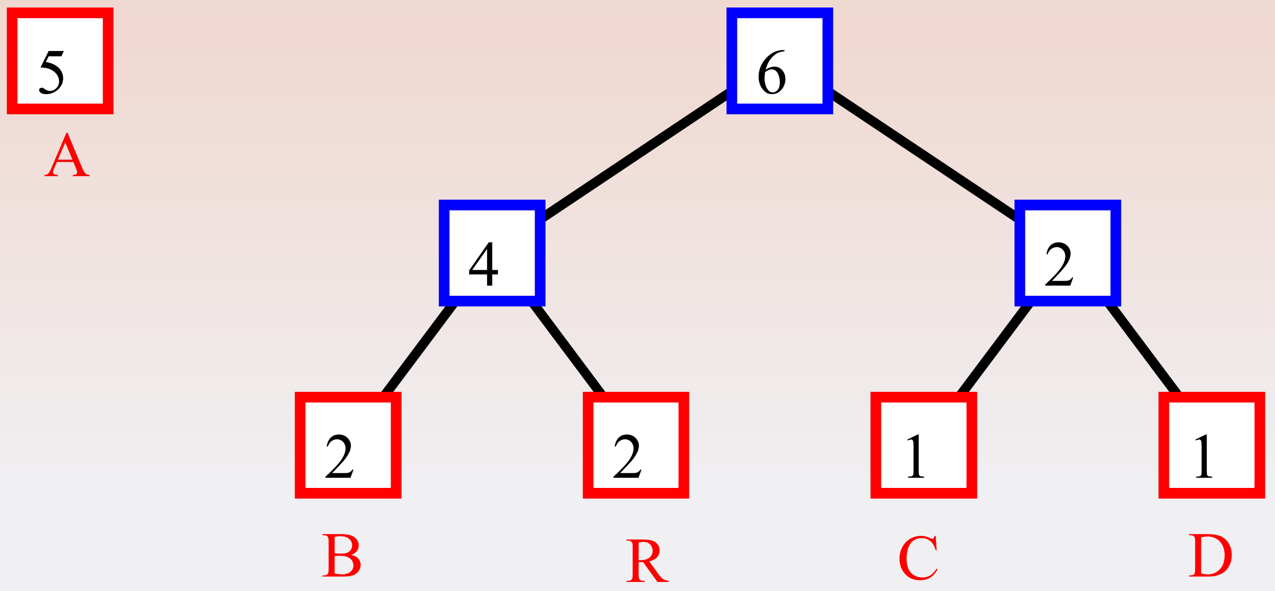


Huffman Encoding: Example

Step *2*

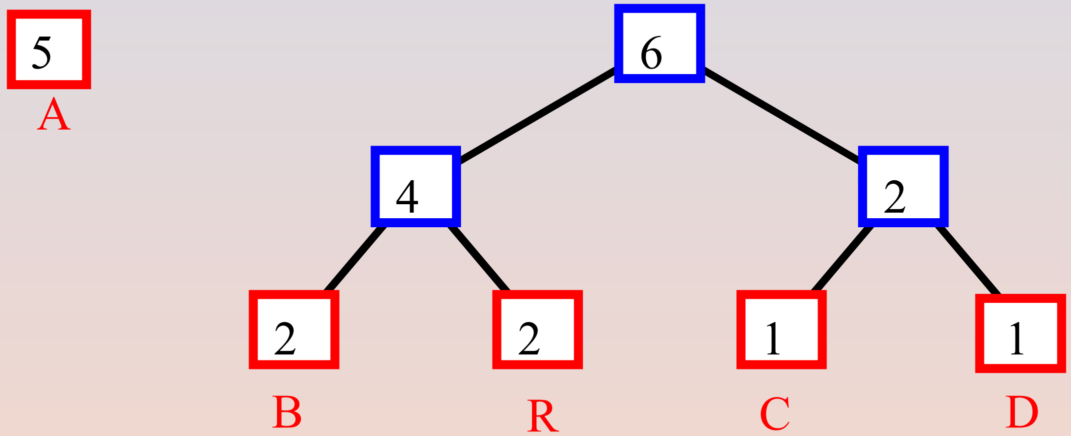


Step *3*

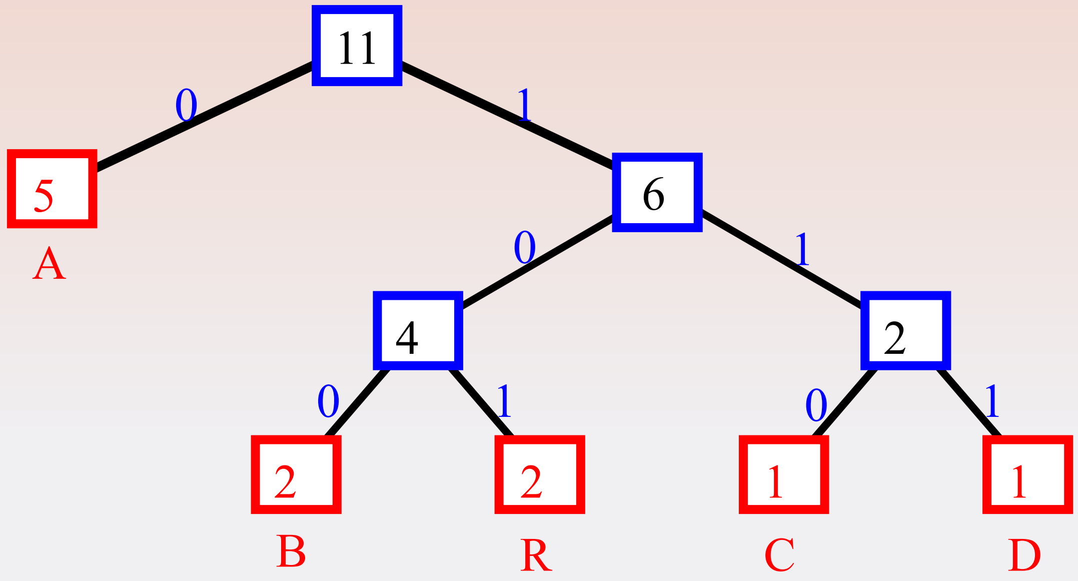


Huffman Encoding: Example

Step *4*



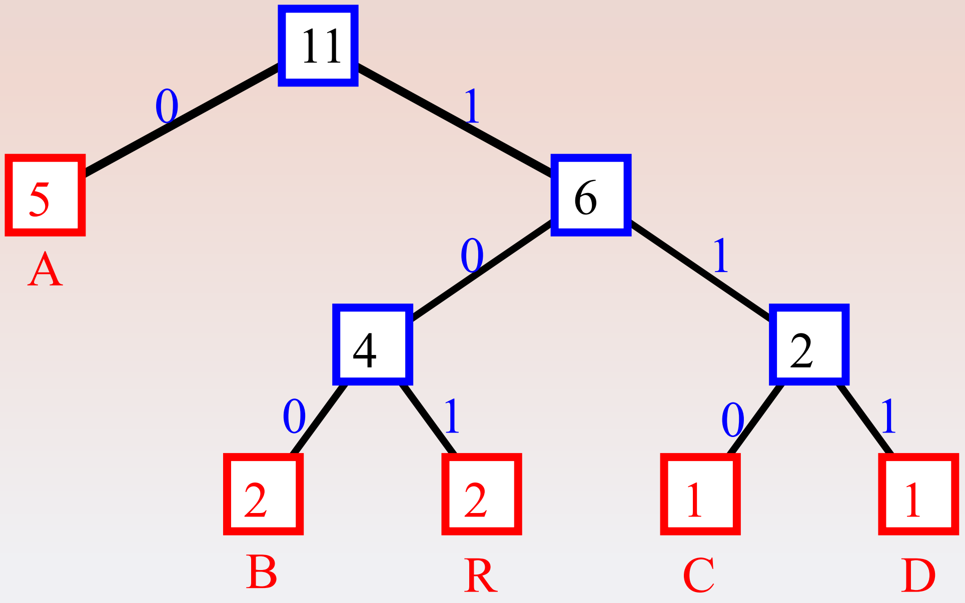
Step *5*



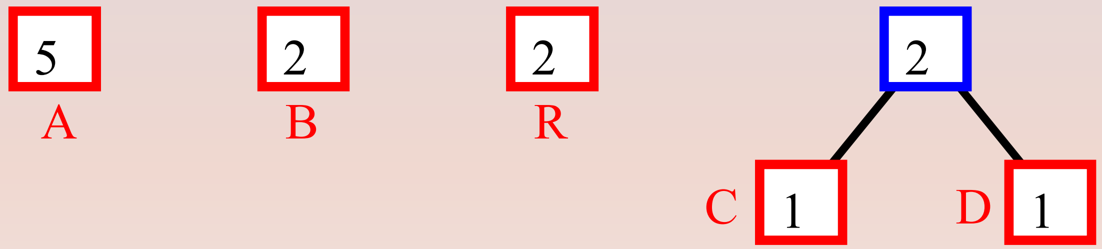
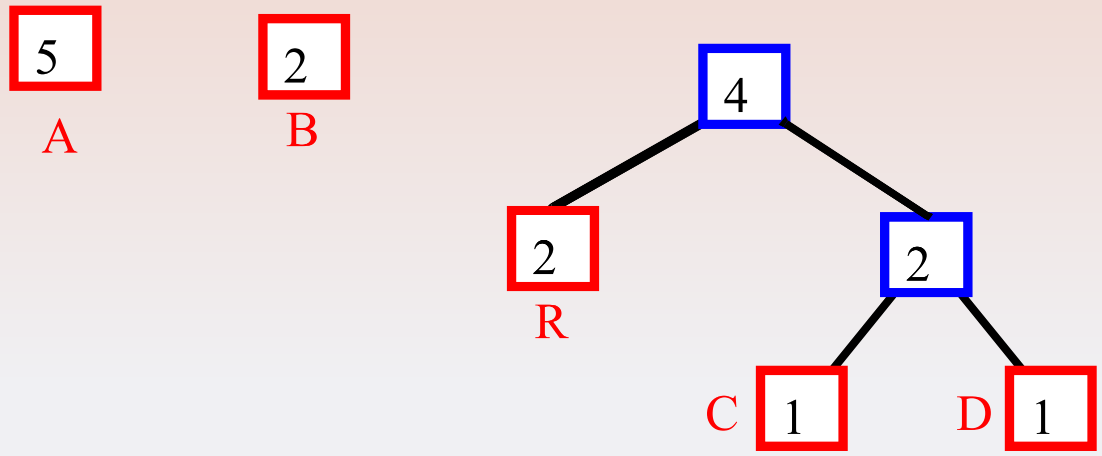
Final Huffman Trie

A B R A C A D A B R A

0 100 101 0 110 0 111 0 100 101 0 (23 bits)

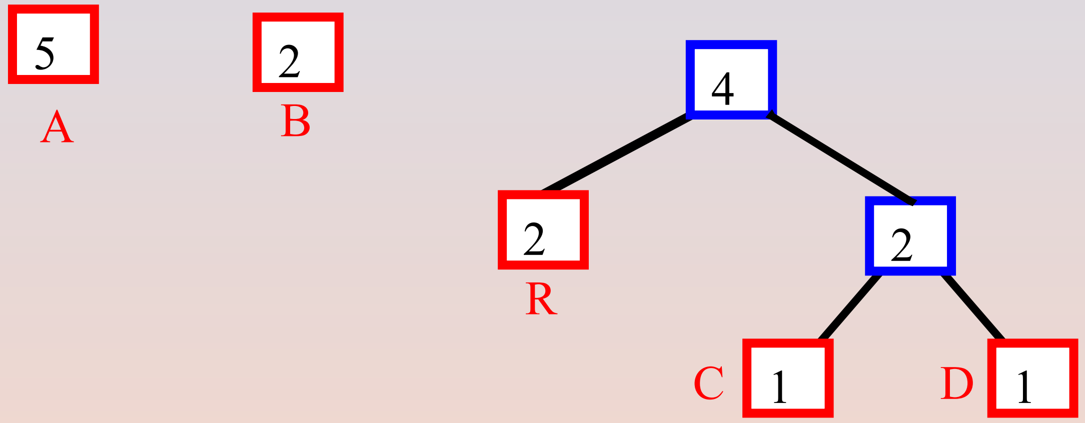


Another Example

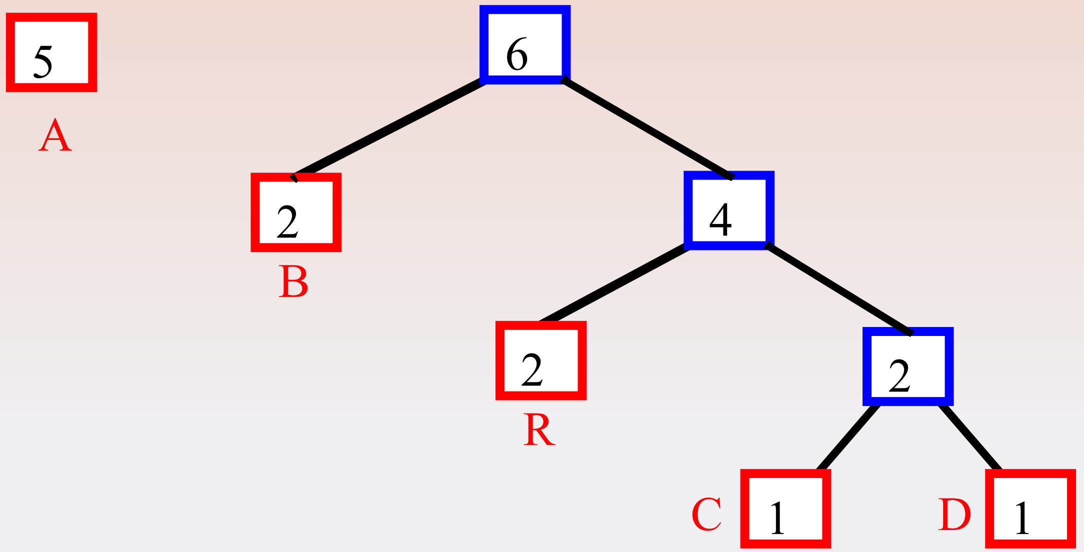
Step *0*	ABRACADABRA character: A B R C D frequency: 5 2 2 1 1
Step *1*	
Step *2*	

Another Example

Step 3

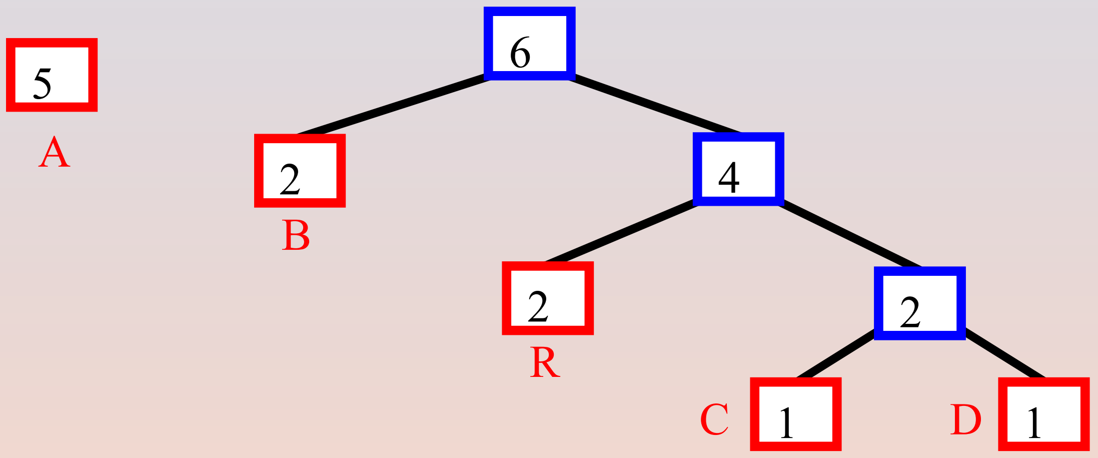


Step 4

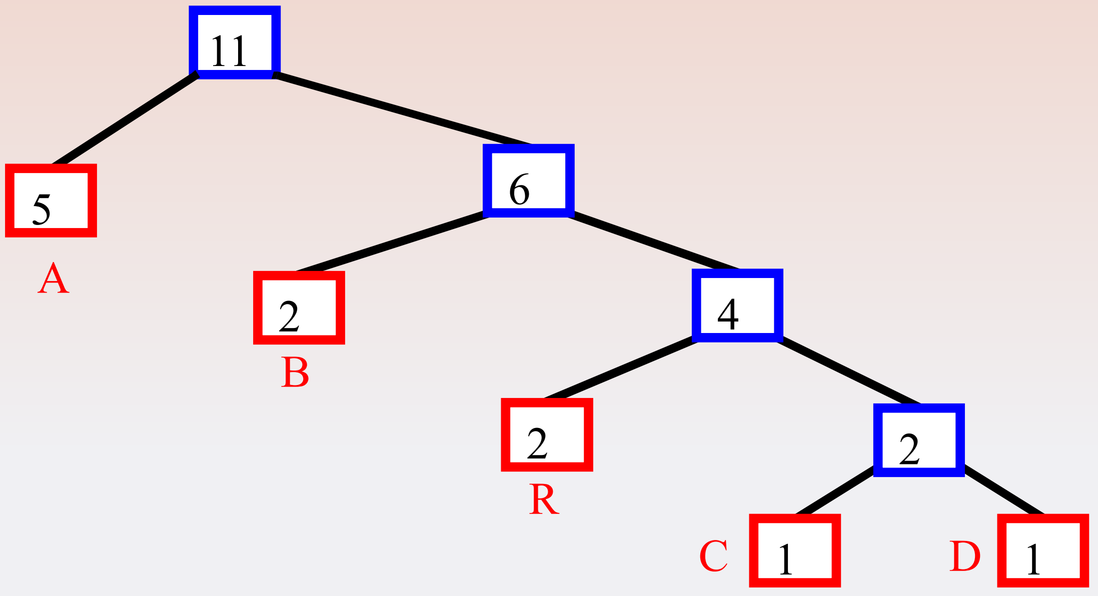


Another Example

Step 5



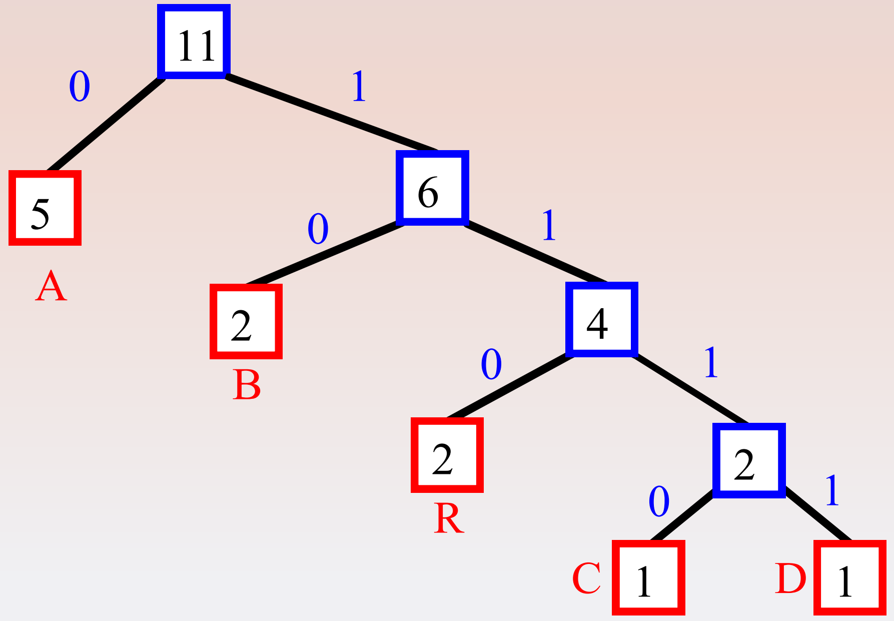
Step 6



Final Trie

A B R A C A D A B R A

0 10 110 0 1110 0 1111 0 10 110 0 (23 bits)



Another Example

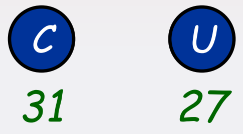
Huffman Code Construction

- Character count in text.

Char	Freq
E	125
T	93
A	80
O	76
I	73
N	71
S	65
R	61
H	55
L	41
D	40
C	31
U	27

Huffman Code Construction

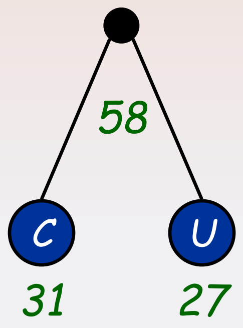
Char	Freq
E	125
T	93
A	80
O	76
I	73
N	71
S	65
R	61
H	55
L	41
D	40
C	31
U	27



Huffman Code Construction

Char	Freq
E	125
T	93
A	80
O	76
I	73
N	71
S	65
R	61
	58
H	55
L	41
D	40

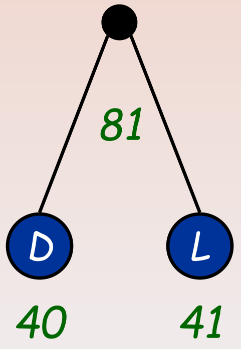
C	31
U	27

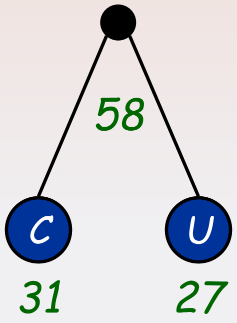


Huffman Code Construction

Char	Freq
E	125
T	93
	81
A	80
O	76
I	73
N	71
S	65
R	61
	58
H	55

L	41
D	40

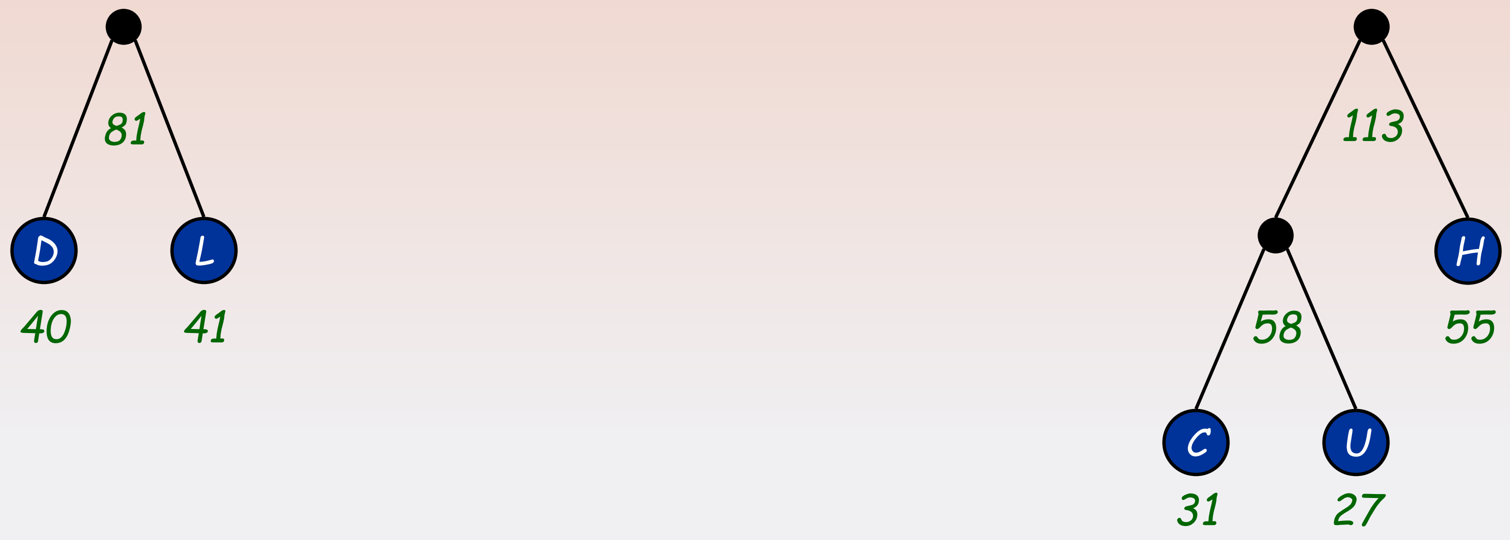




Huffman Code Construction

Char	Freq
E	125
	113
T	93
	81
A	80
O	76
I	73
N	71
S	65
R	61

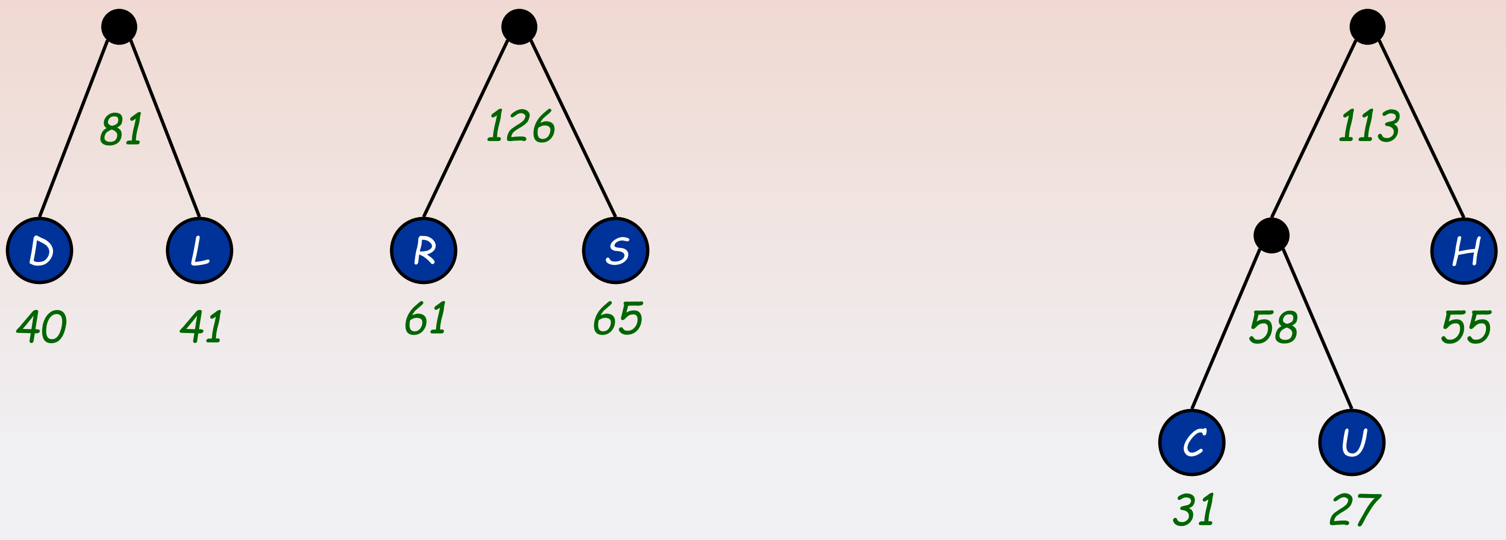
	58
H	55



Huffman Code Construction

Char	Freq
	126
E	125
	113
T	93
	81
A	80
O	76
I	73
N	71

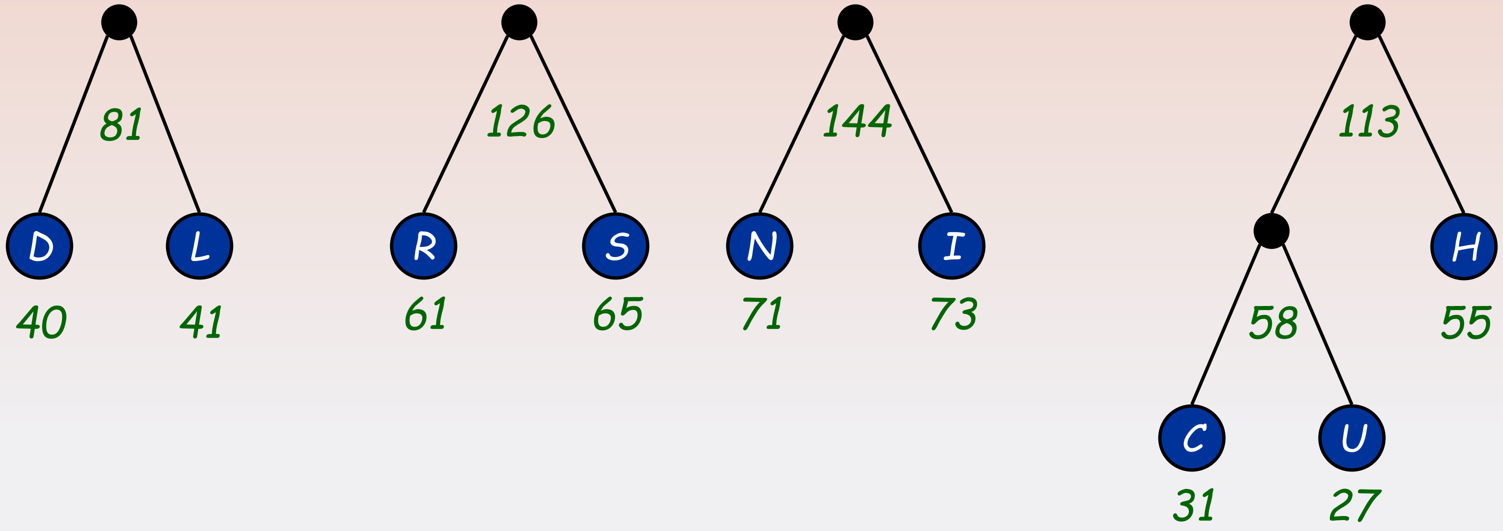
S	65
R	61



Huffman Code Construction

Char	Freq
	144
	126
E	125
	113
T	93
	81
A	80
O	76

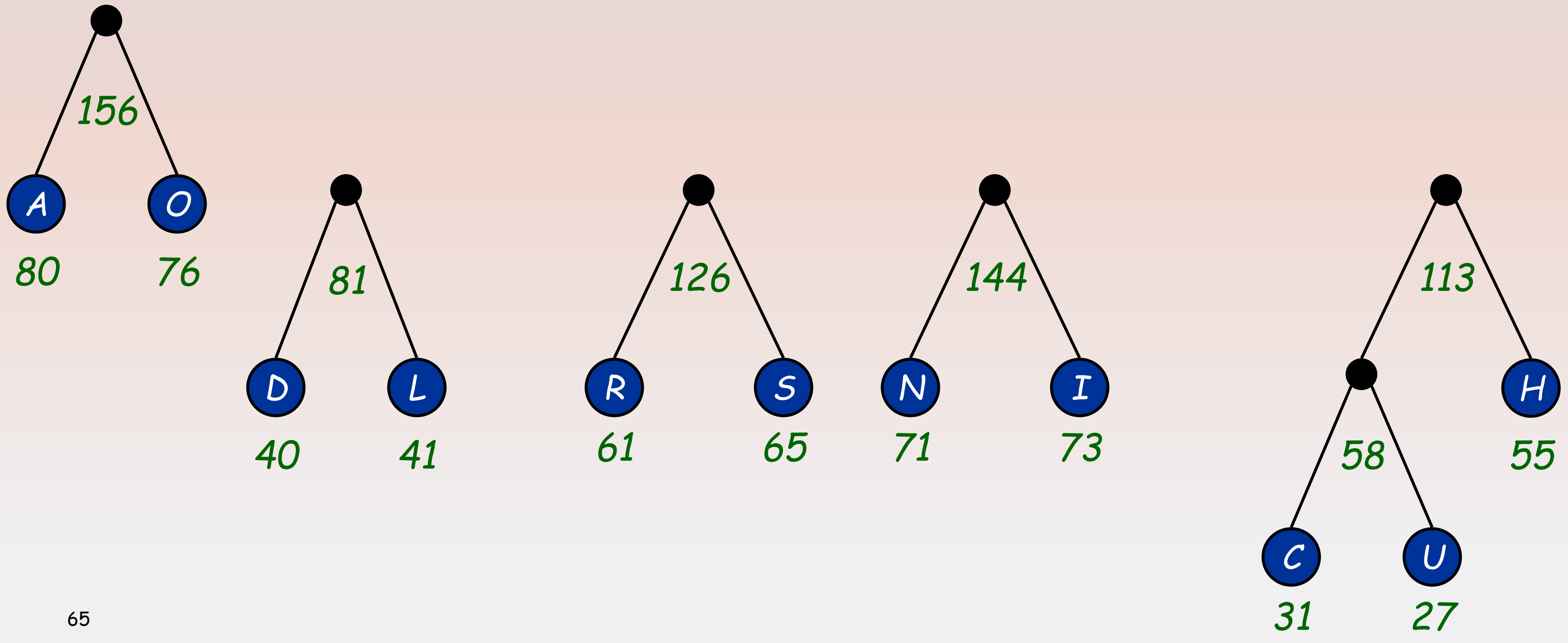
I	73
N	71



Huffman Code Construction

Char	Freq
	156
	144
	126
E	125
	113
T	93
	81

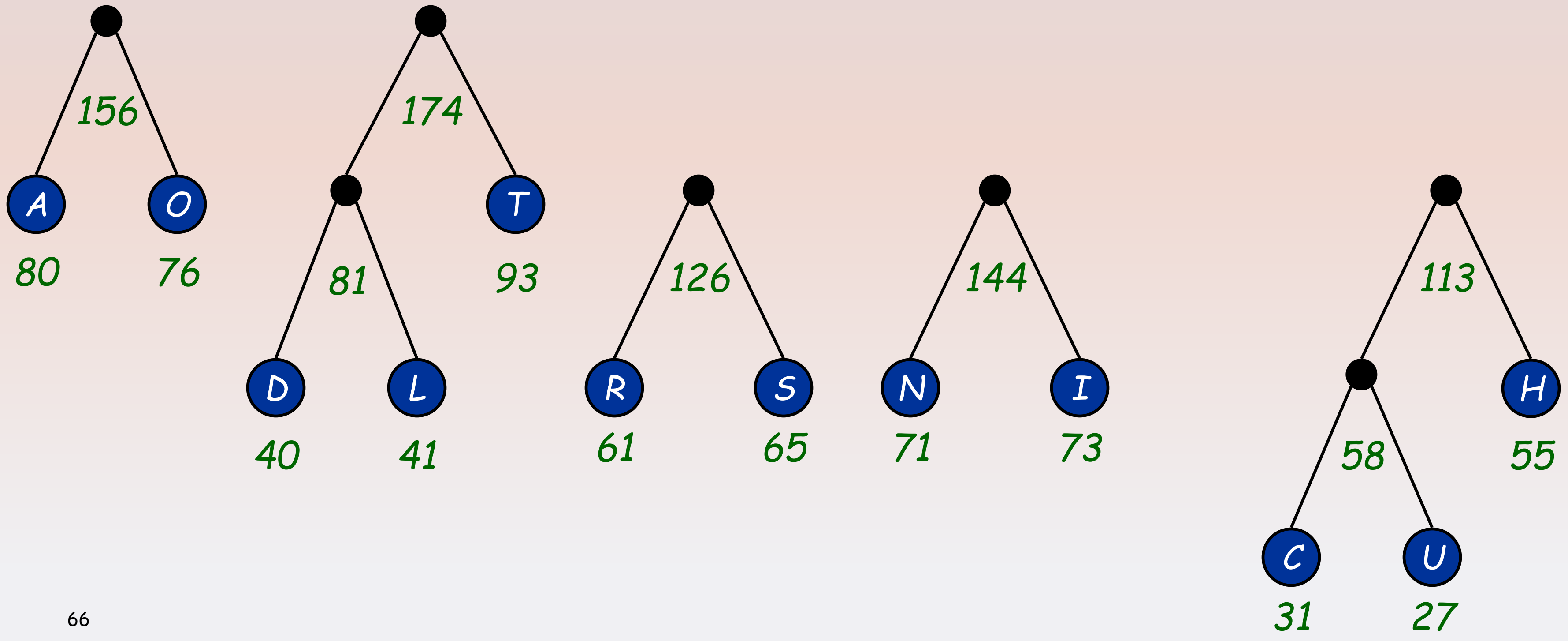
A	80
O	76



Huffman Code Construction

Char	Freq
	174
	156
	144
	126
E	125
	113

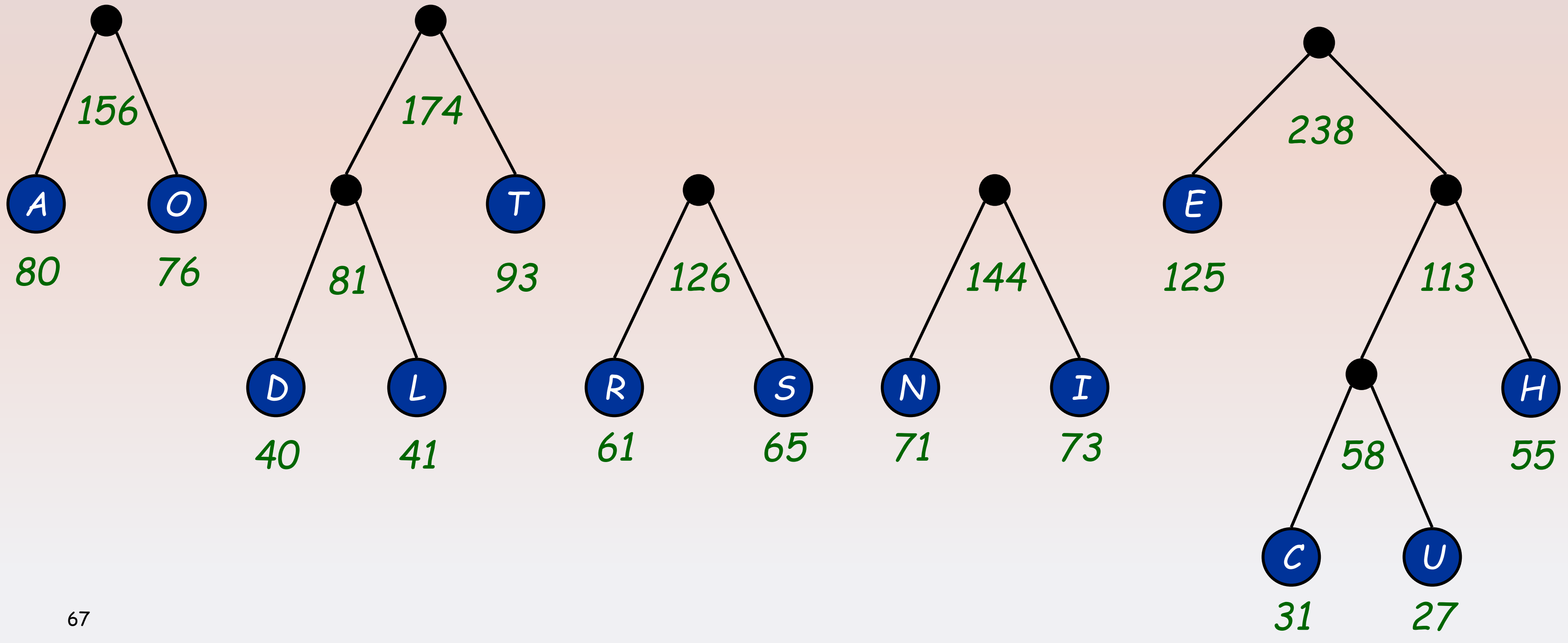
T	93
	81



Huffman Code Construction

Char	Freq
	238
	174
	156
	144
	126

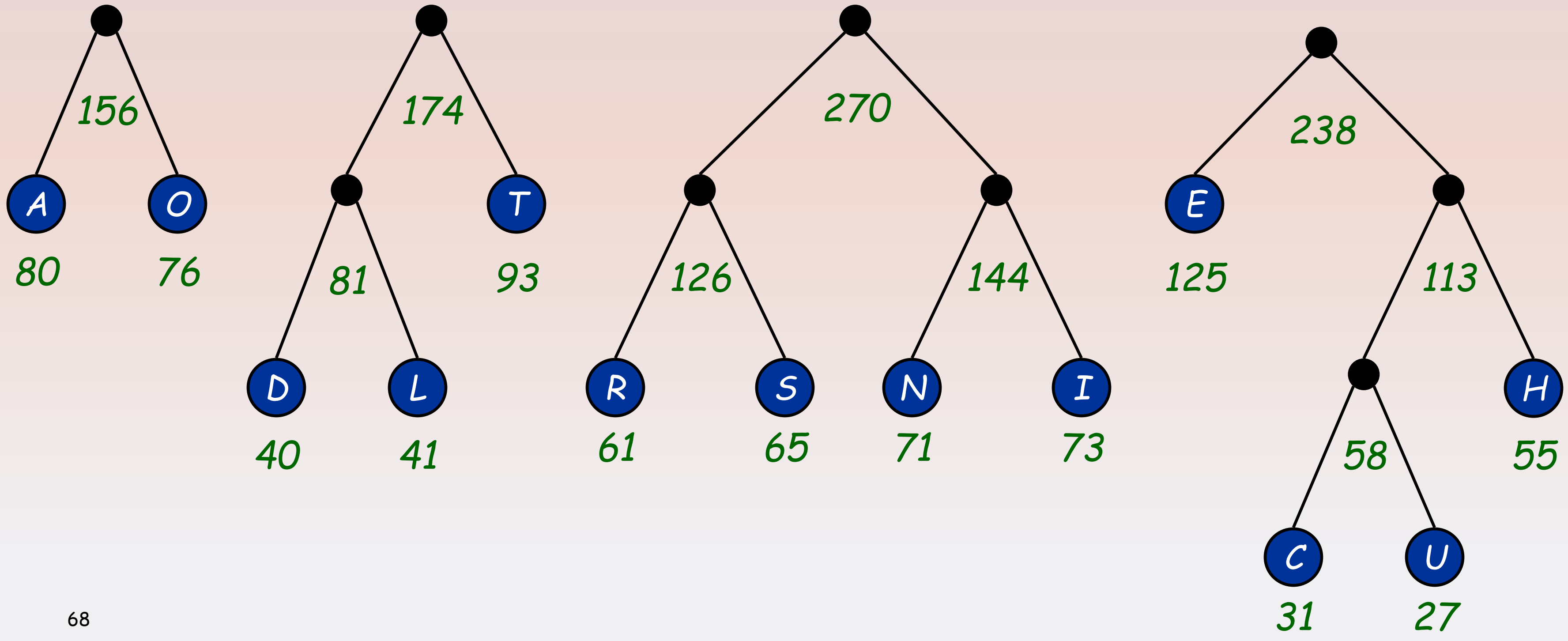
E	125
	113



Huffman Code Construction

Char	Freq
	270
	238
	174
	156

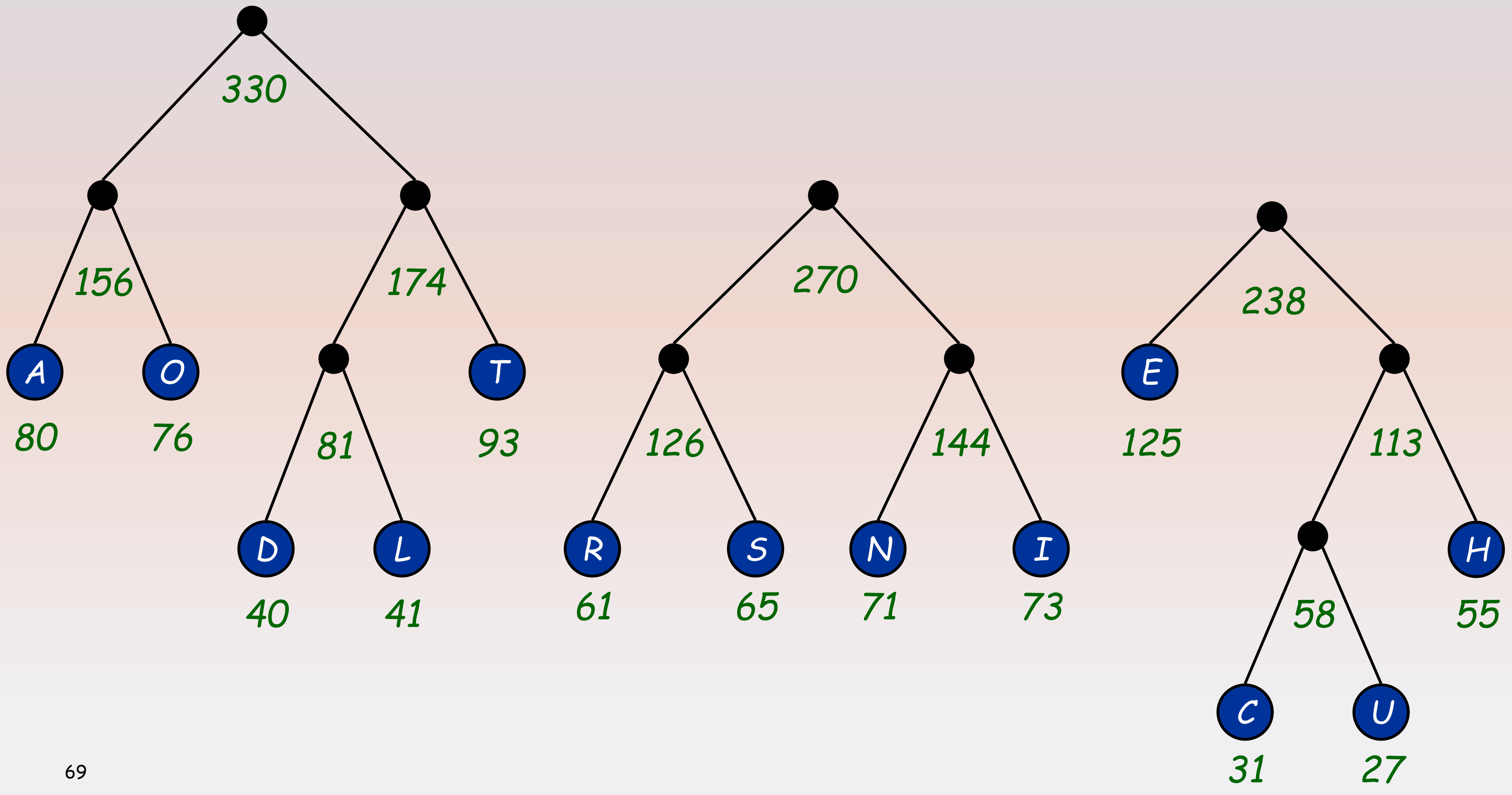
	144
	126



Huffman Code Construction

Char	Freq
	330
	270
	238

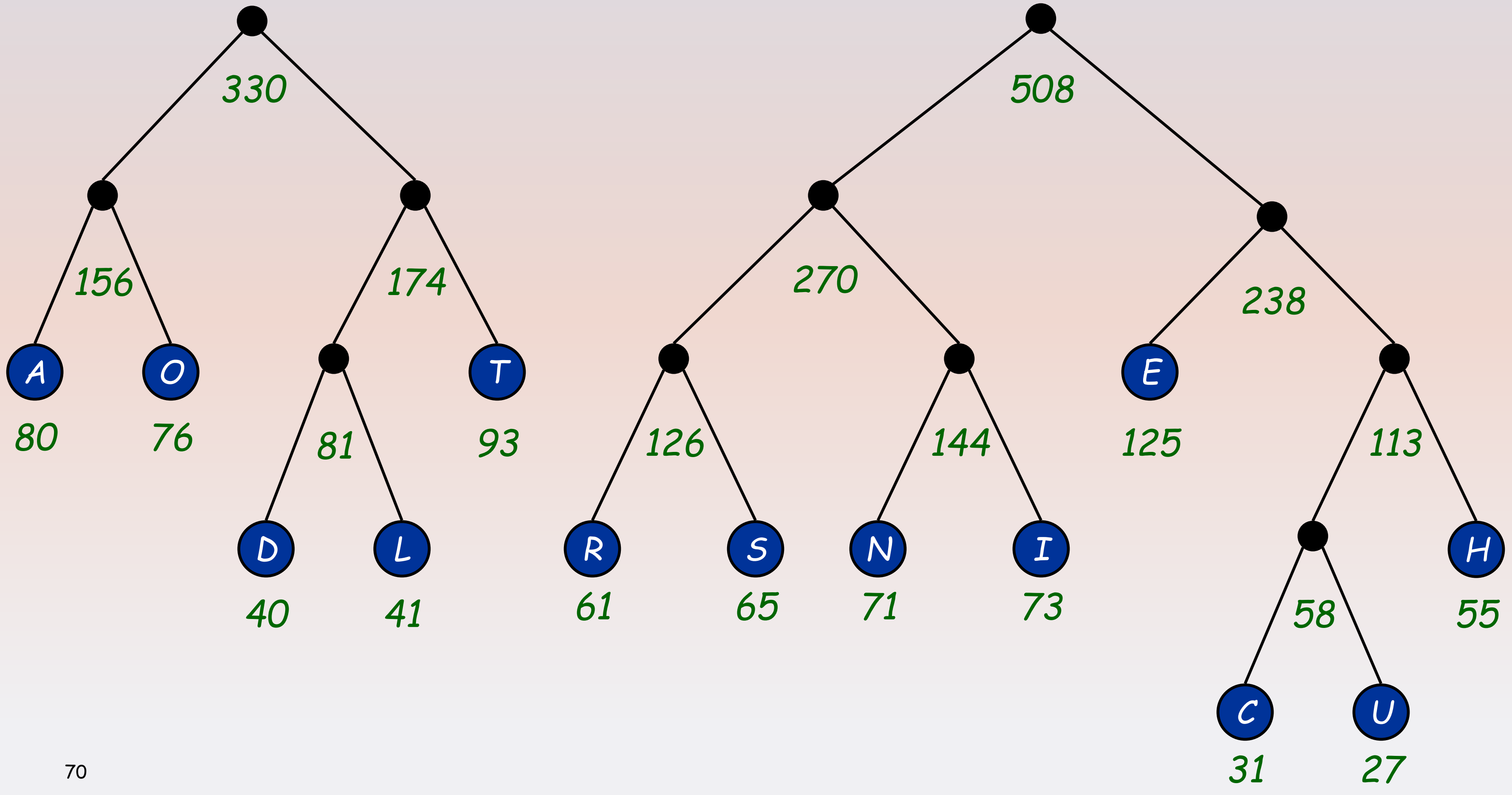
	174
	156



Char	Freq
	508
	330

Huffman Code Construction

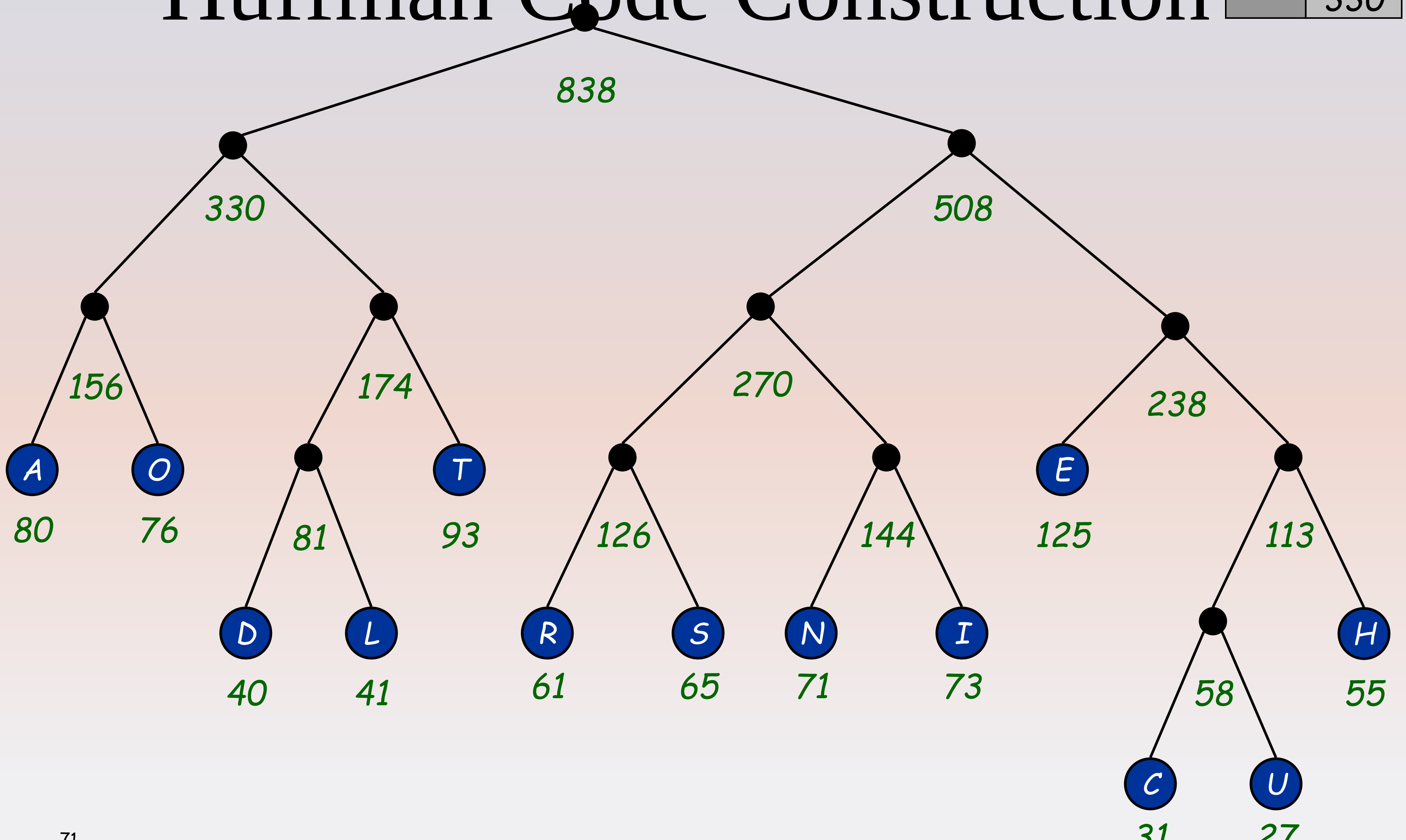
	270
	238



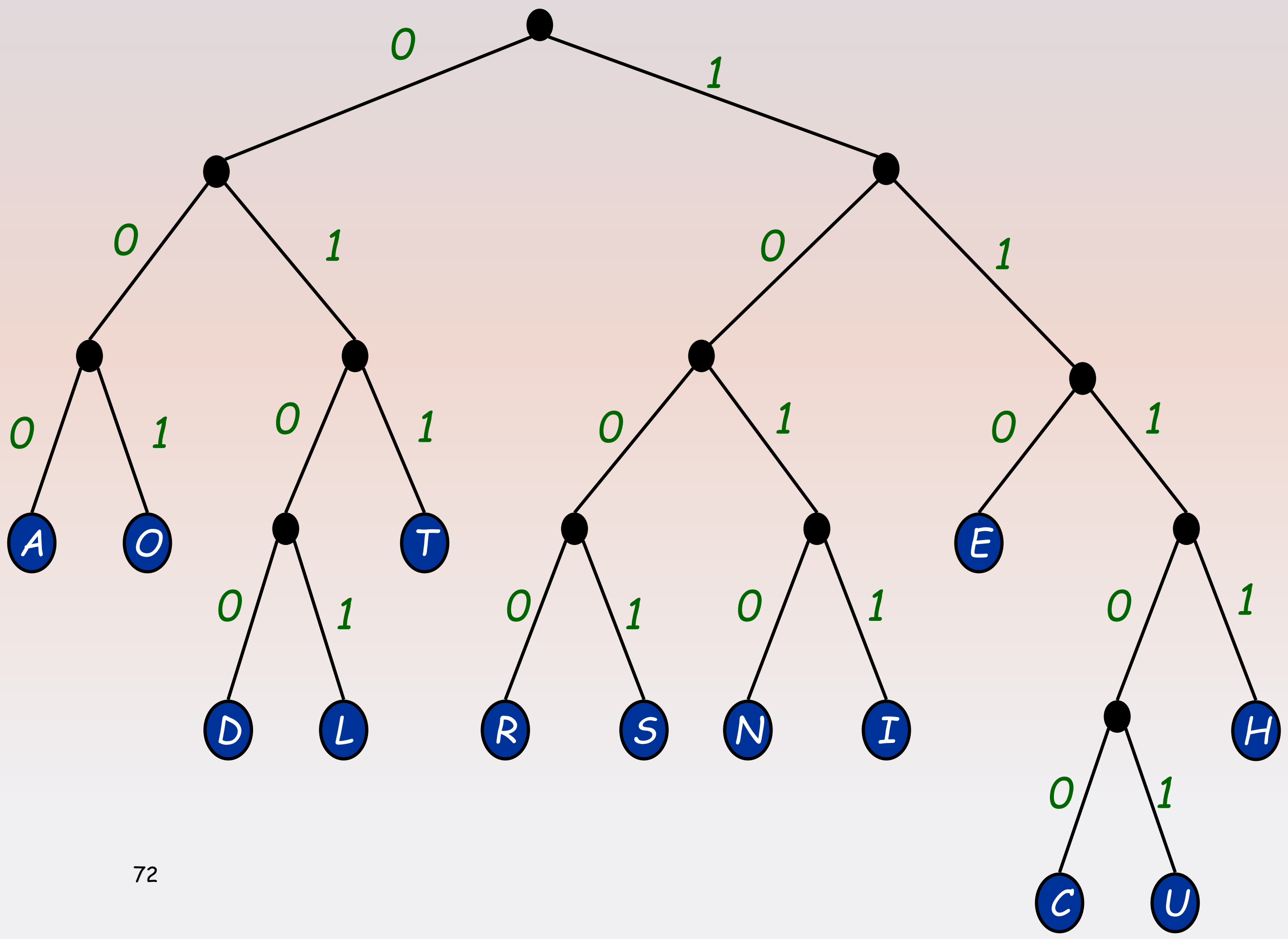
Huffman Code Construction

Char	Freq
	838

	508
	330



Huffman Code Construction



Char	Freq	Fixed	Huff
E	125	0000	110
T	93	0001	011
A	80	0010	000
O	76	0011	001
I	73	0100	1011
N	71	0101	1010
S	65	0110	1001
R	61	0111	1000
H	55	1000	1111
L	41	1001	0101
D	40	1010	0100
C	31	1011	11100
U	27	1100	11101
Total	838	4.00	3.62

Priority queue

- Use a priority queue for storing the nodes
- Priority queue is a queue ordered by priority (heap)
- For our application, priority = frequency
- If there are *k* elements in the queue:
  - Extracting the lowest priority is *O(log k)*
  - Inserting takes *O(log k)*

Huffman algorithm in Python

```
def makeHuffTree(t):
    heapq.heapify(t)
    while len(t) > 1:
        L, R = heapq.heappop(t), heapq.heappop(t)
        parent = (L[0] + R[0], L, R)
        heapq.heappush(t, parent)
    return t[0]

def printHuffTree(t, prefix = ''):
    if len(t) == 2:
        print t[1], prefix
    else:
        printHuffTree(t[1], prefix + '0')
        printHuffTree(t[2], prefix + '1')
```

Remark: transforms list **`t`** in a heap in linear time

Remark: returns the tree represented a nested tuple

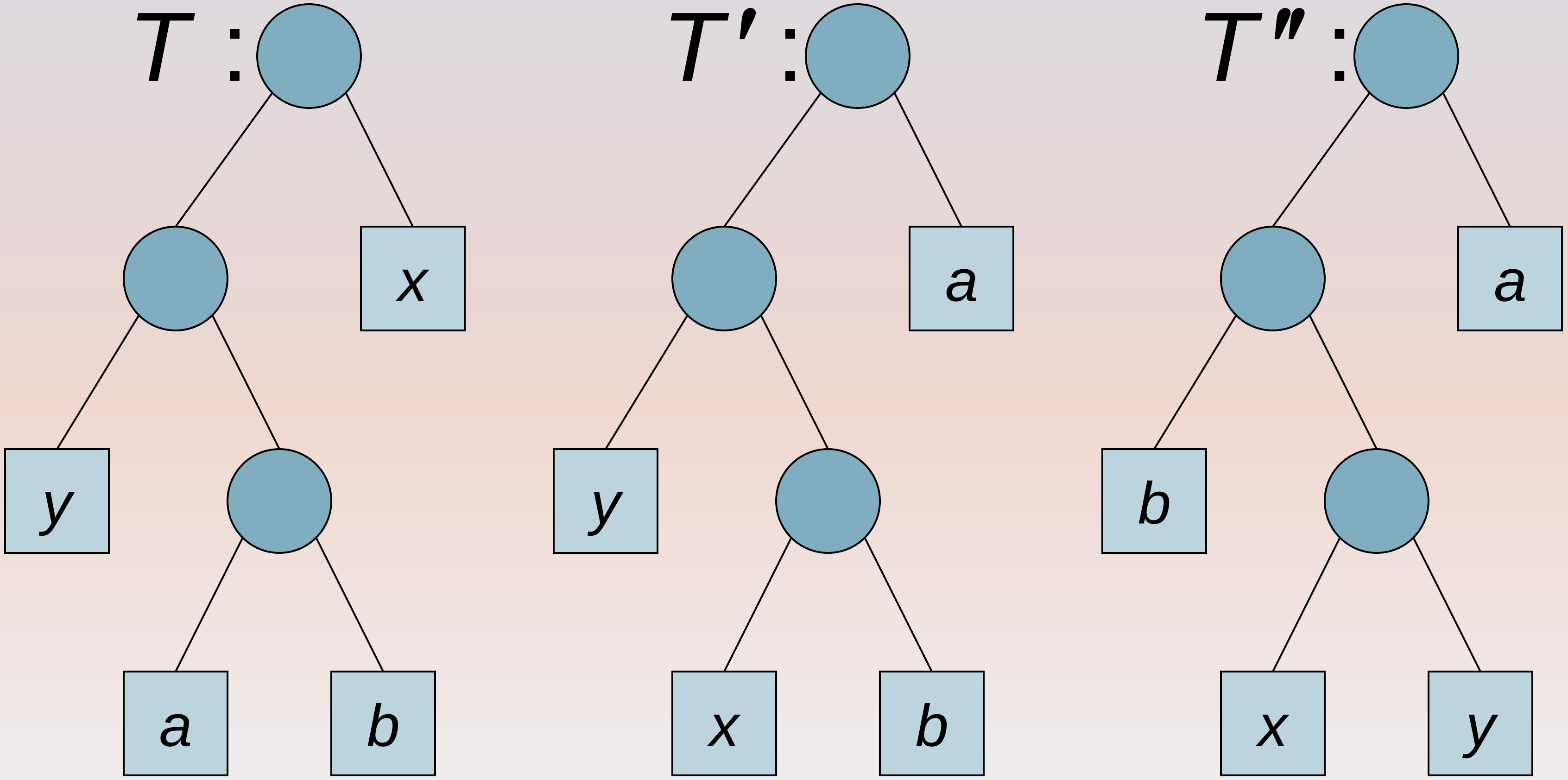
Huffman Algorithm

- Running time for a text of length *n* with *k* distinct characters: $O(n + k \log k)$
- If we assume *k* to be a constant (i.e., not a function of *n*) then the algorithm runs in
  $O(n)$ time
- <u>Fact</u>: Using a Huffman encoding trie, the encoded text has minimal length

Greedy-choice property

- Claim: Consider the two characters x and y with the lowest frequencies. Then there is an optimal tree in which x and y are siblings at the deepest level of the tree.

- Proof
  - *Let T* be an arbitrary ***optimal*** prefix code tree
  - Let a and b be two siblings at the deepest level of T.
  - We will show that we can convert T to another prefix tree where x and y are siblings at the deepest level without increasing the cost.



$\boxed{B(T') \le B(T)}$ $\boxed{B(T'') \le B(T')}$

- Assume $f(x) \le f(y)$ and $f(a) \le f(b)$
- We know that $f(x) \le f(a)$ and $f(y) \le f(b)$

Switch a and x

Switch b and y

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

Non-negative because x has (at least) the lowest frequency

Non-negative because a is at the max depth

Since $B(T) - B(T') \geq 0$, T' is at least as good as *T*.

But *T* is optimal, so *T′* must be optimal too.

Thus, moving x to the bottom (similarly, y to the bottom) yields an optimal solution

- The previous claim asserts that the greedy-choice of Huffman’s algorithm is the proper one to make.

Optimal substructure property

- Claim:

  Huffman's algorithm produces an optimal prefix code tree.

- Proof (by induction on n=|C|)

  Base case: n=1

  the tree consists of a single leaf—optimal

  Inductive assumption:

  Assume for strictly less than n characters, Huffman's algorithm produces an optimal tree

  Prove:

  Huffman's algorithm produces an optimal tree for exactly n characters

- (According to the previous claim) in the optimal tree, the lowest frequency characters x and y are siblings at the deepest level.

- Remove x and y replacing them with z, where f(z)= f(x)+ f(y),
  - Thus, n-1 characters remain in the alphabet.

- Let T' be any tree representing any prefix code for this (n-1) character alphabet. Then, we can obtain a prefix-code tree *T for the original set of n characters* by replacing the leaf node for *z* with an internal node having *x* and *y* as children. The cost of T is

$$\begin{aligned} B(T) &= B(T') - f(z)\,d(z) + f(x)\,(d(z) + 1) + f(y)\,(d(z)+1) \\ &= B(T') - (f(x) + f(y))\,d(z) + (f(x) + f(y))\,(d(z)+1) \\ &= B(T') + f(x) + f(y) \end{aligned}$$

- According to the assumption, B(T') is optimal.

- Thus B(T) is optimal as well.

Greedy method: summary

- Task scheduling
- Fractional knapsack
- Huffman encoding (section 5.2)

- Other greedy algorithms that will be covered later: Prim (section 5.1.5), Kruskal (section 5.1.3) and Dijkstra (section 4.4)