Greedy Algorithm

Greedy algorithms

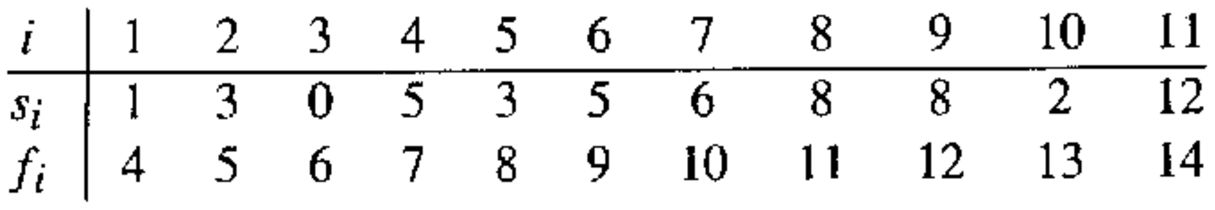
- A *greedy algorithm* always makes the choice that looks best at the moment
  - My everyday examples:
    - Driving in Los Angeles, NY, or Boston for that matter
    - Playing cards
    - Invest on stocks
    - Choose a university
  - The hope: a locally optimal choice will lead to a globally optimal 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**

The Activity Selection Problem

- Here are a set of start and finish times

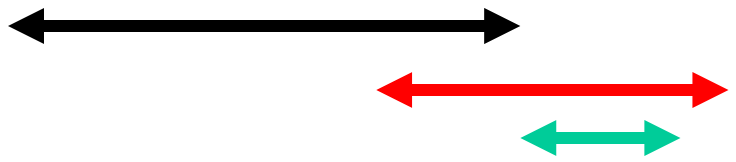


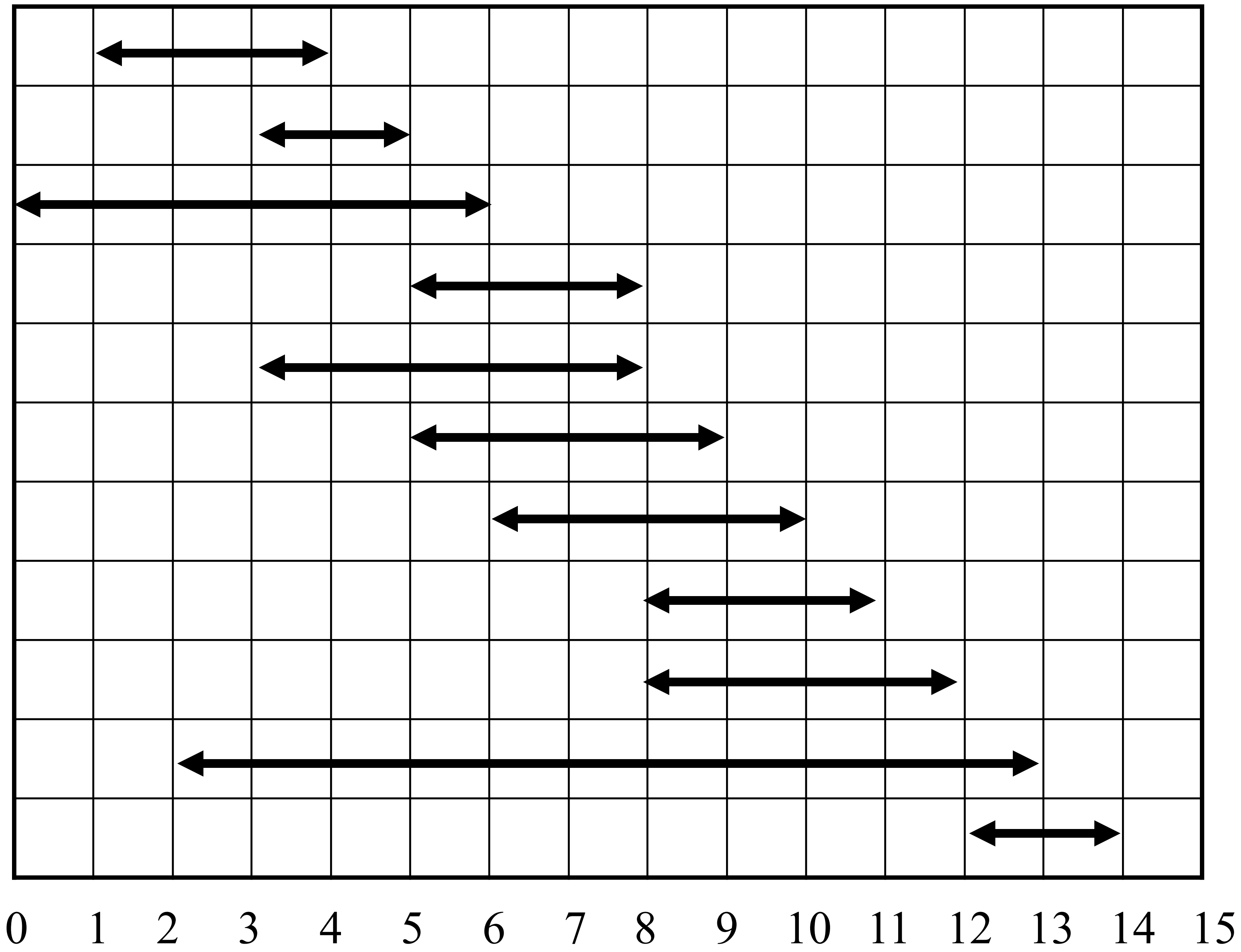
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

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

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




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

Greedy algorithms

- S_{ij} the set of activities that start after activity a_i finishes and that finish before activity a_j starts.
- Mutually exclusive maximum set is A_{ij} which includes some activity a_k.
- $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$
- $C[i, j] = c[i, k] + c[k, j] + 1$

Optimal substructures

- Define the following subset of activities which are activities that can start after a_i finishes and finish before a_j starts

$$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$$

- Sort the activities according to finish time

$$f_0 \leq f_1 \leq f_2 \leq \cdots \leq f_n < f_{n+1}$$

- We now define the the maximal set of activities from i to j as

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$$

- Let c[i,j] be the maximal number of activities

$$c[i, j] = c[i, k] + c[k, j] + 1$$

- Our recurrence relation for finding c[i, j] becomes

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

- We can solve this using dynamic programming, but a simpler approach exists

Can we make a greedy choice

- Choosing the activity which starts early
- Choosing the activity which finishes early
- Choosing the activity which is of smallest duration
  - **Intuition suggests that we should choose an activity that leaves the resource available for as many other activities as possible**

Can we make a greedy choice

- Now, of the activities we end up choosing, one of them must be the first one to finish
- In other words, since the activities are sorted in monotonically increasing order by finish time, the greedy choice is activity a_1.

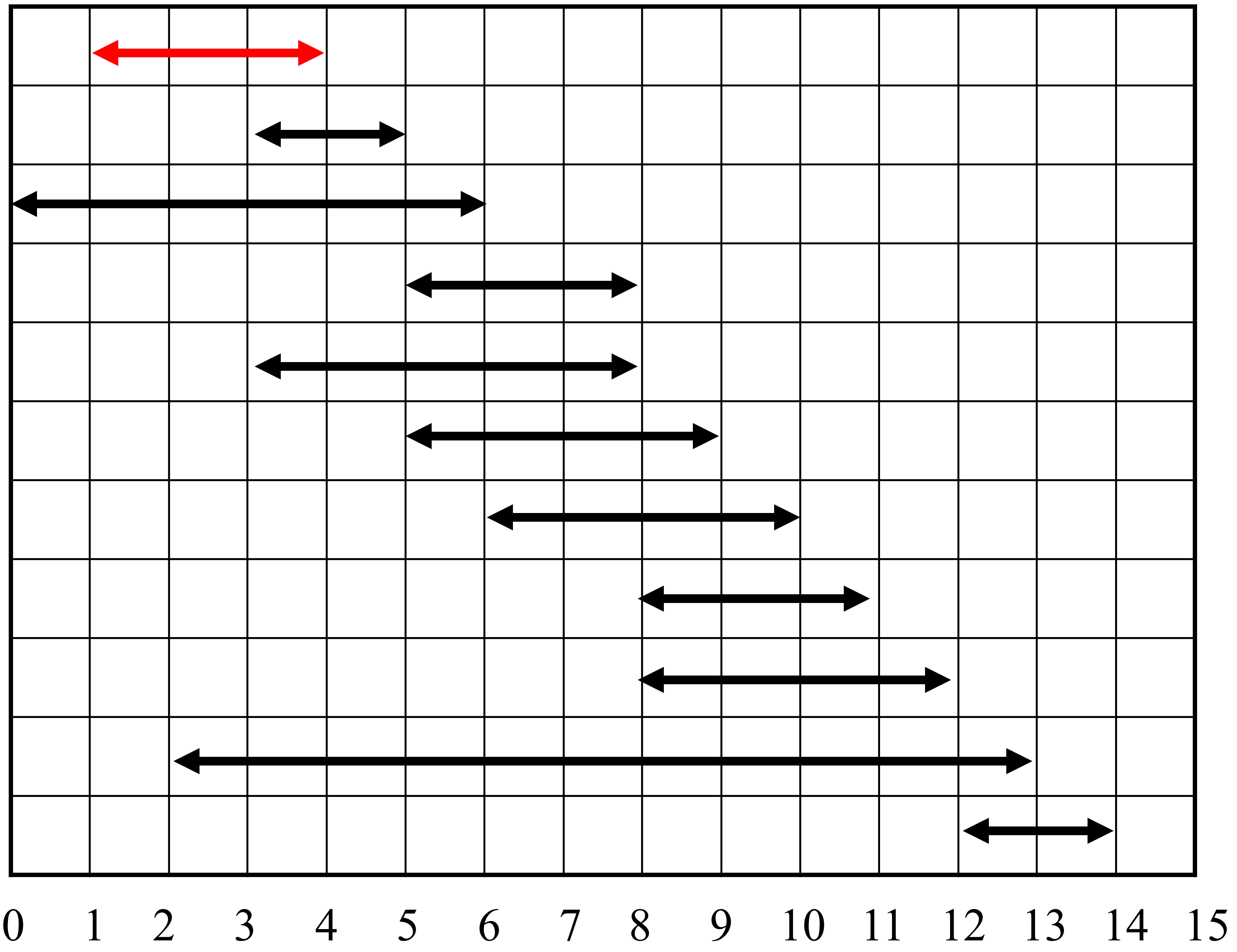
Let $S_k = \{a_i \in S : s_i \geq f_k\}$

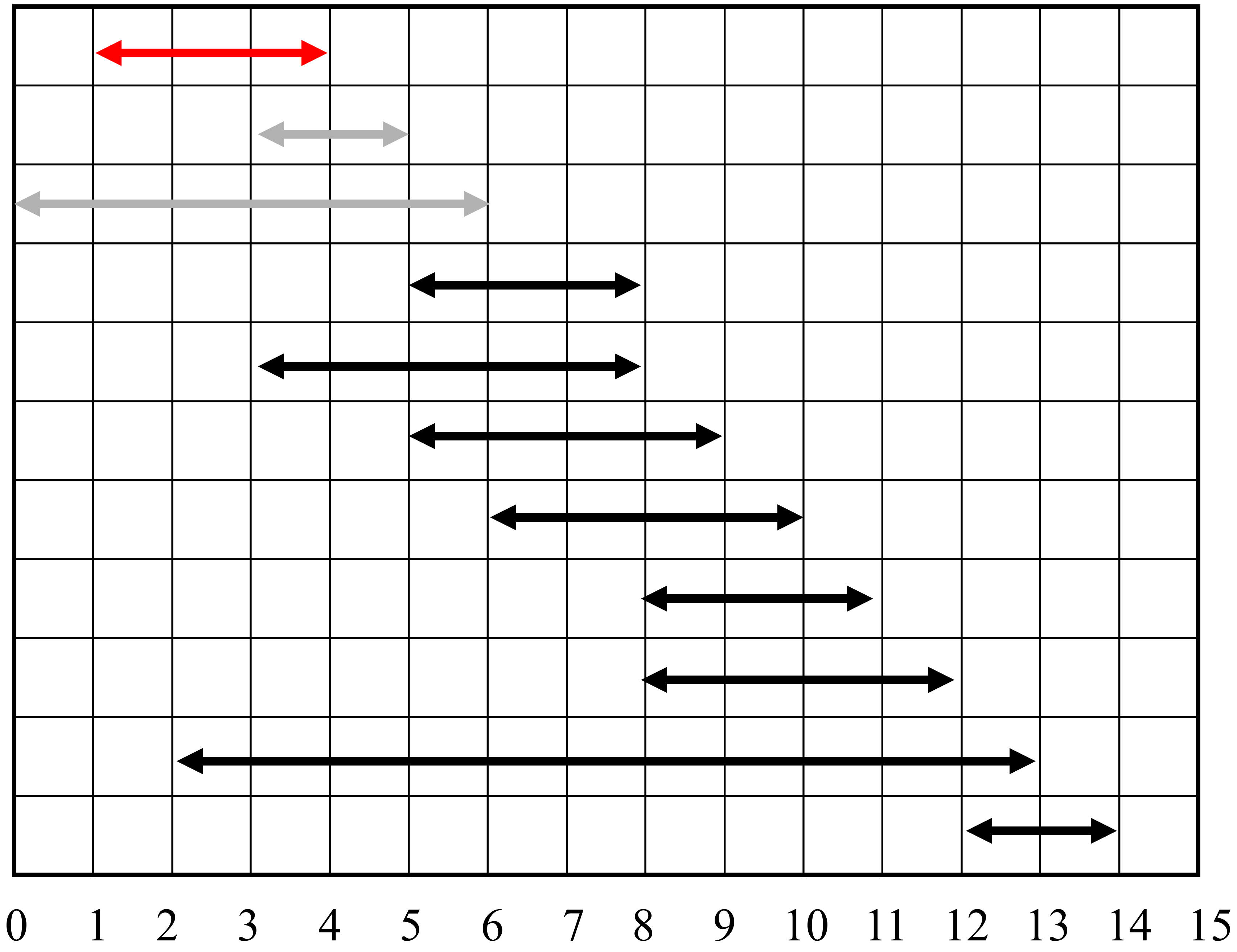
Early Finish Greedy

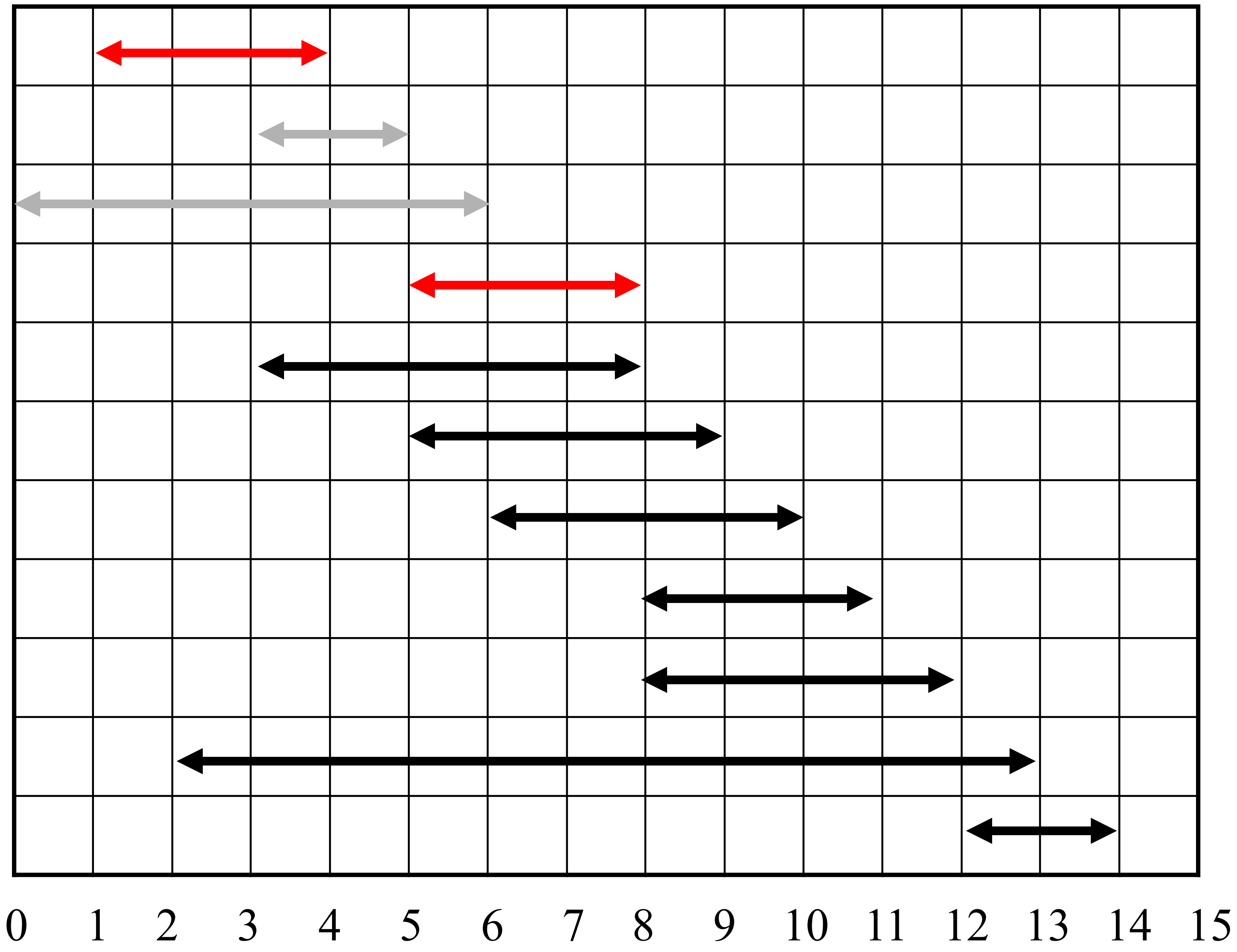
- Select the activity with the earliest finish
- Eliminate the activities that could not be scheduled
- Repeat!

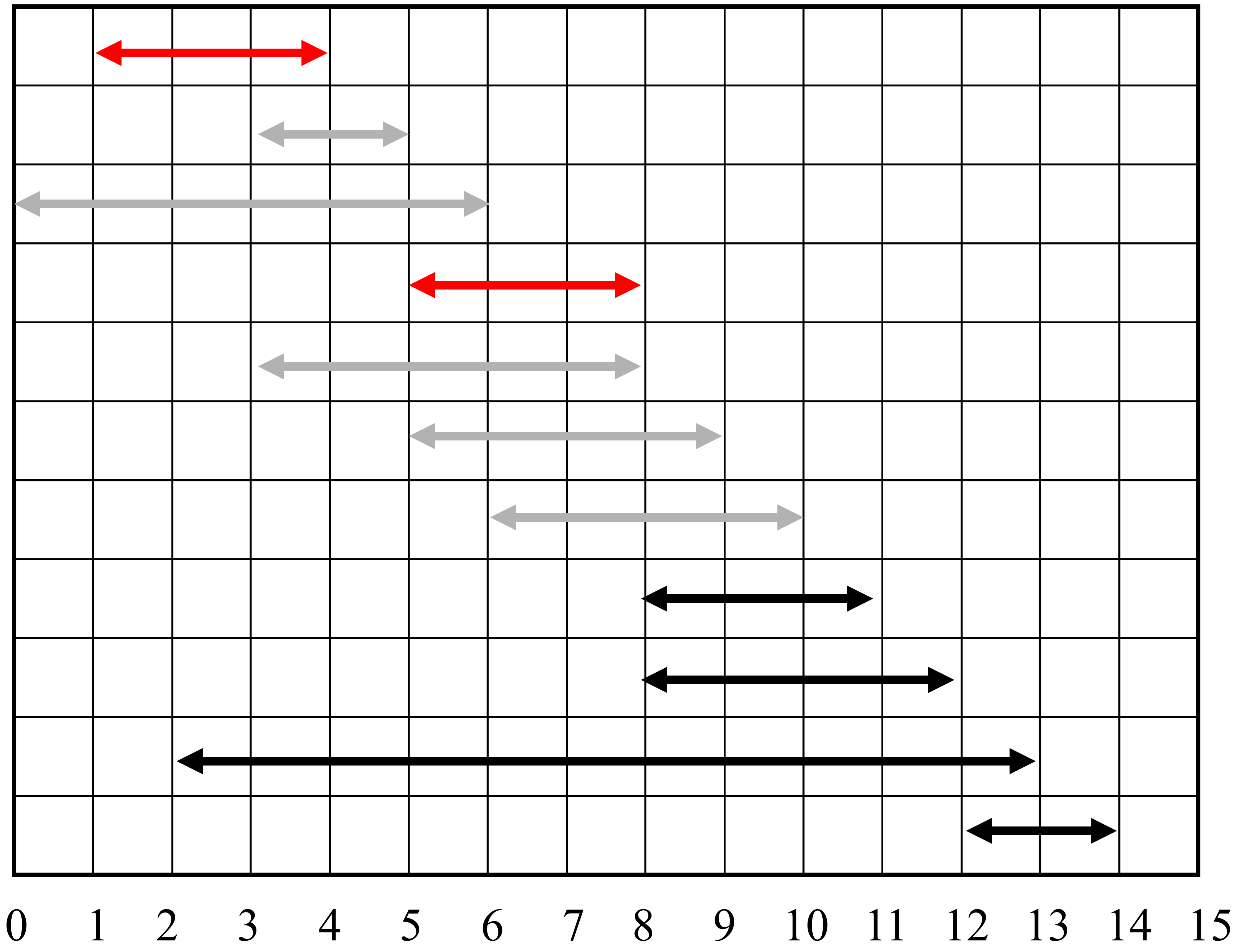
```
RECURSIVE-ACTIVITY-SELECTOR$(s, f, k, n)$
m = k + 1
while $m \leq n$ and $s[m] < f[k]$        // find the first activity in $S_k$ to finish
    $m = m + 1$
if $m \leq n$
    return $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR$(s, f, m, n)$
else return $\emptyset$
```

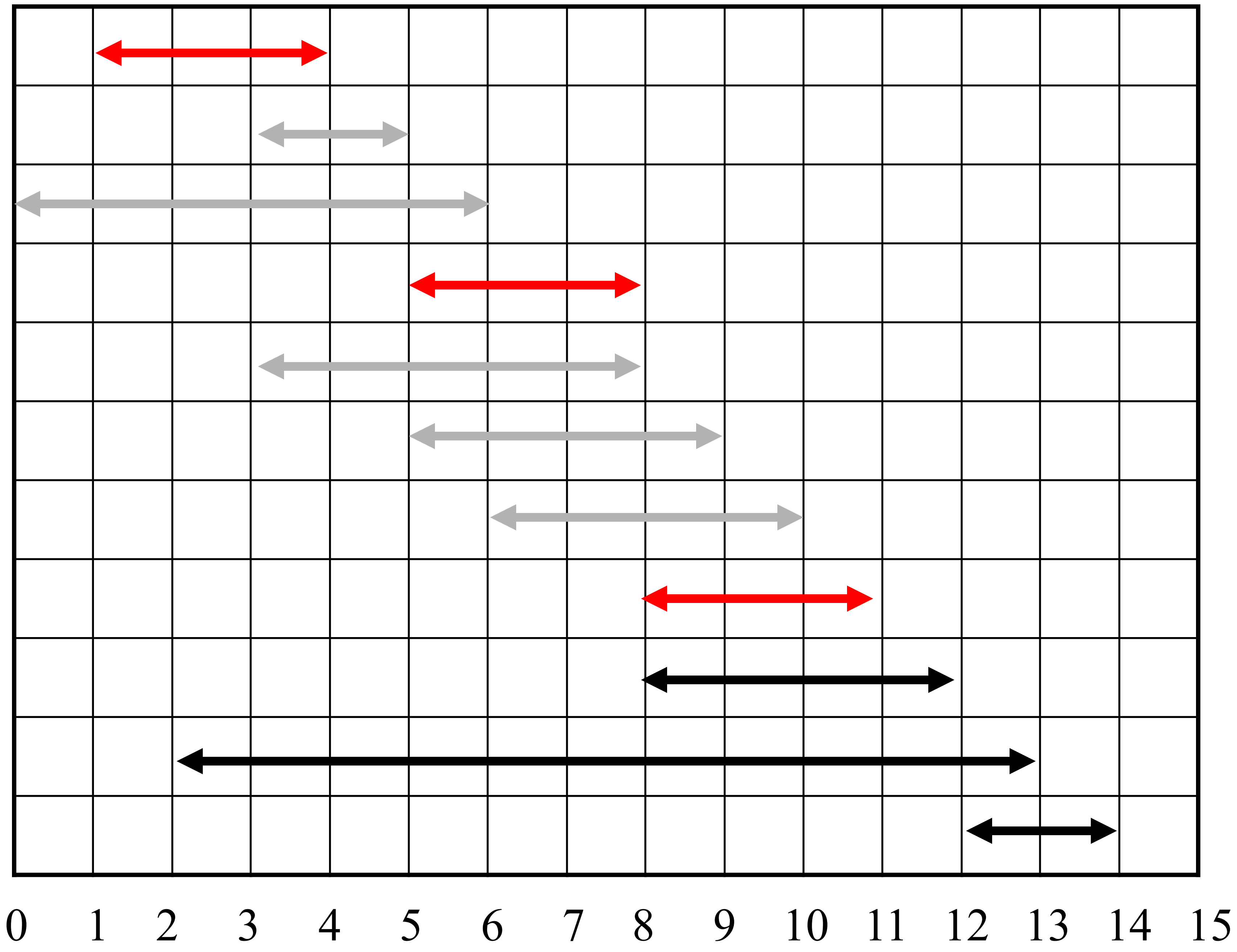
The initial call, which solves the entire problem, is RECURSIVE-ACTIVITY-SELECTOR(s,f,0,n).

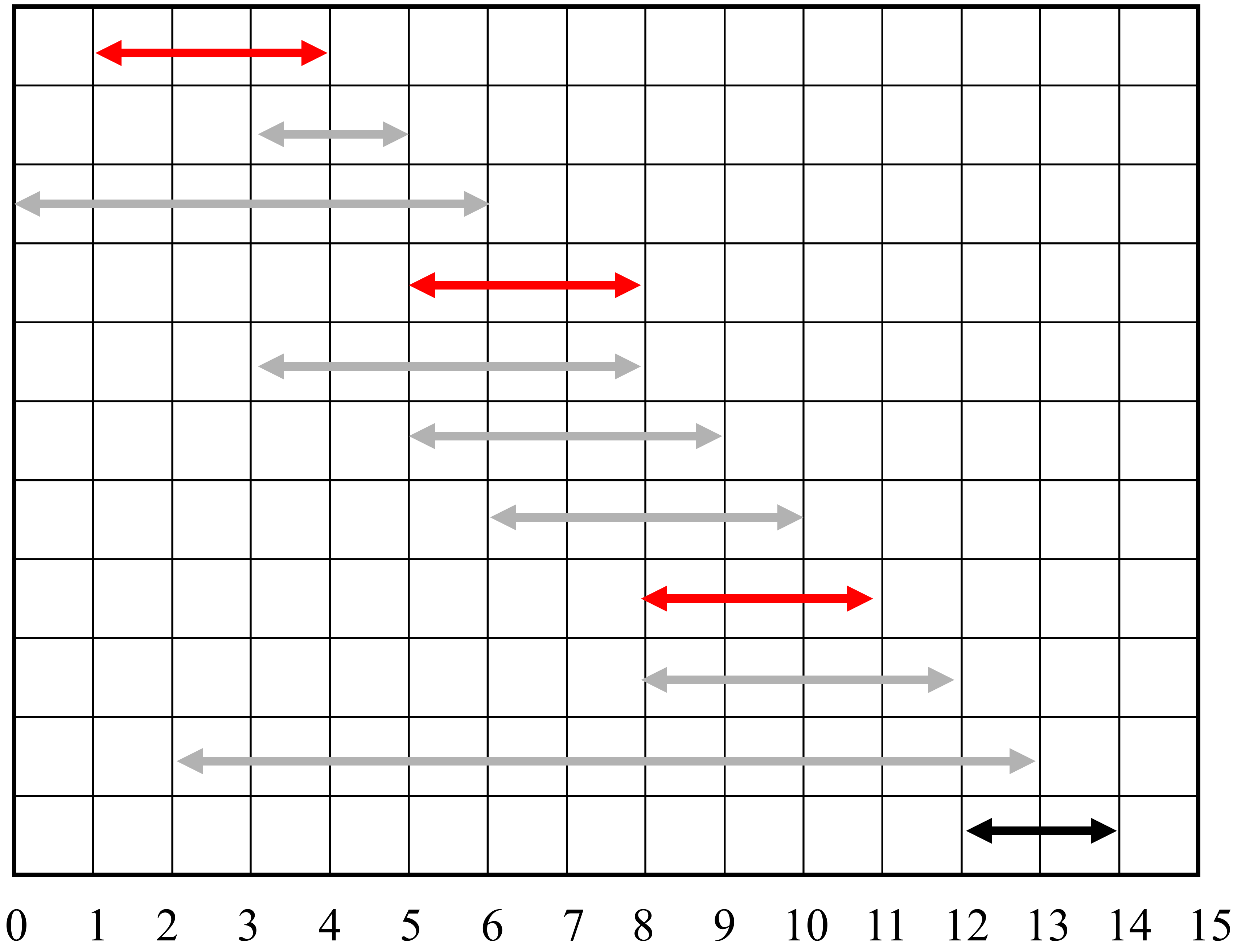

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

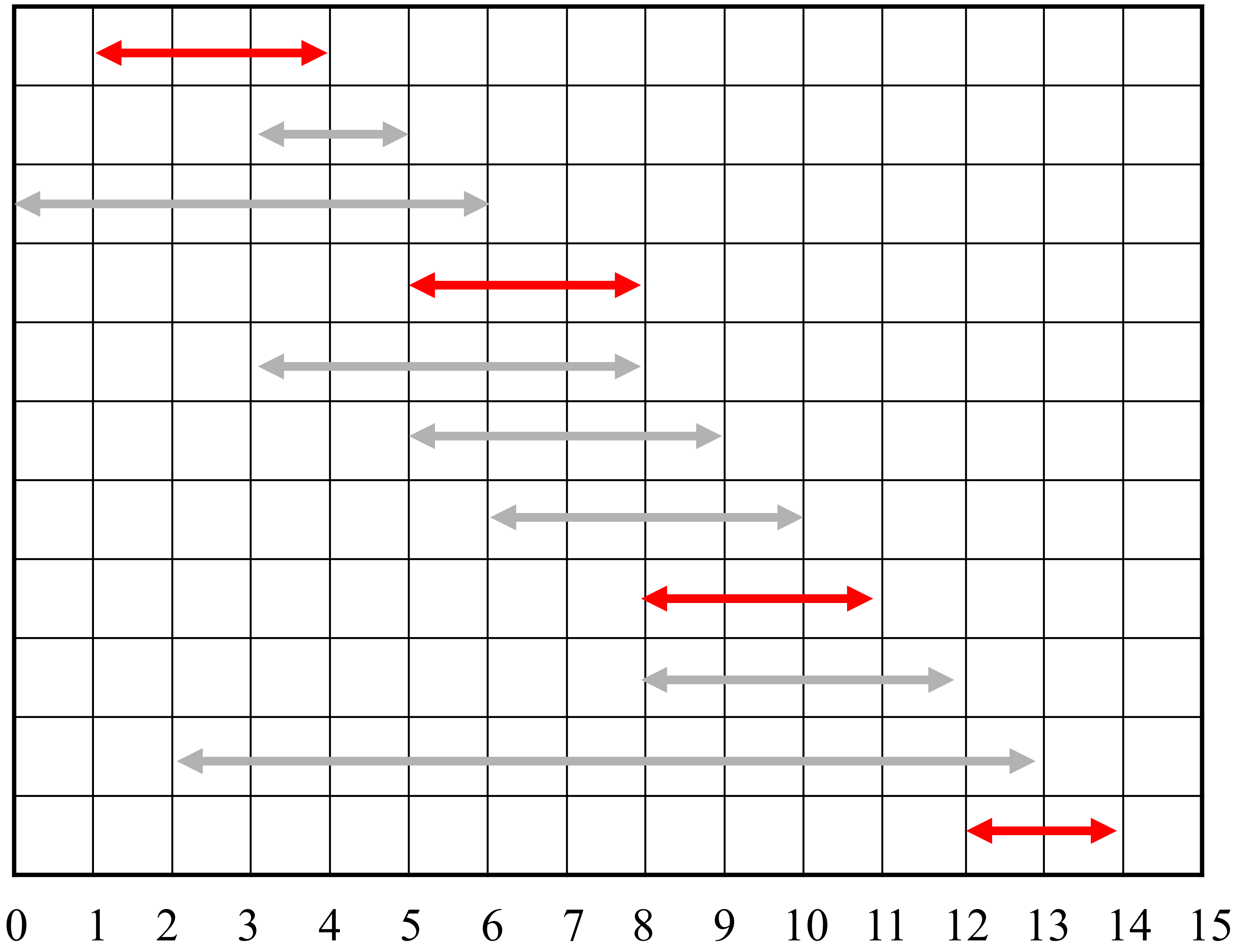

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


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

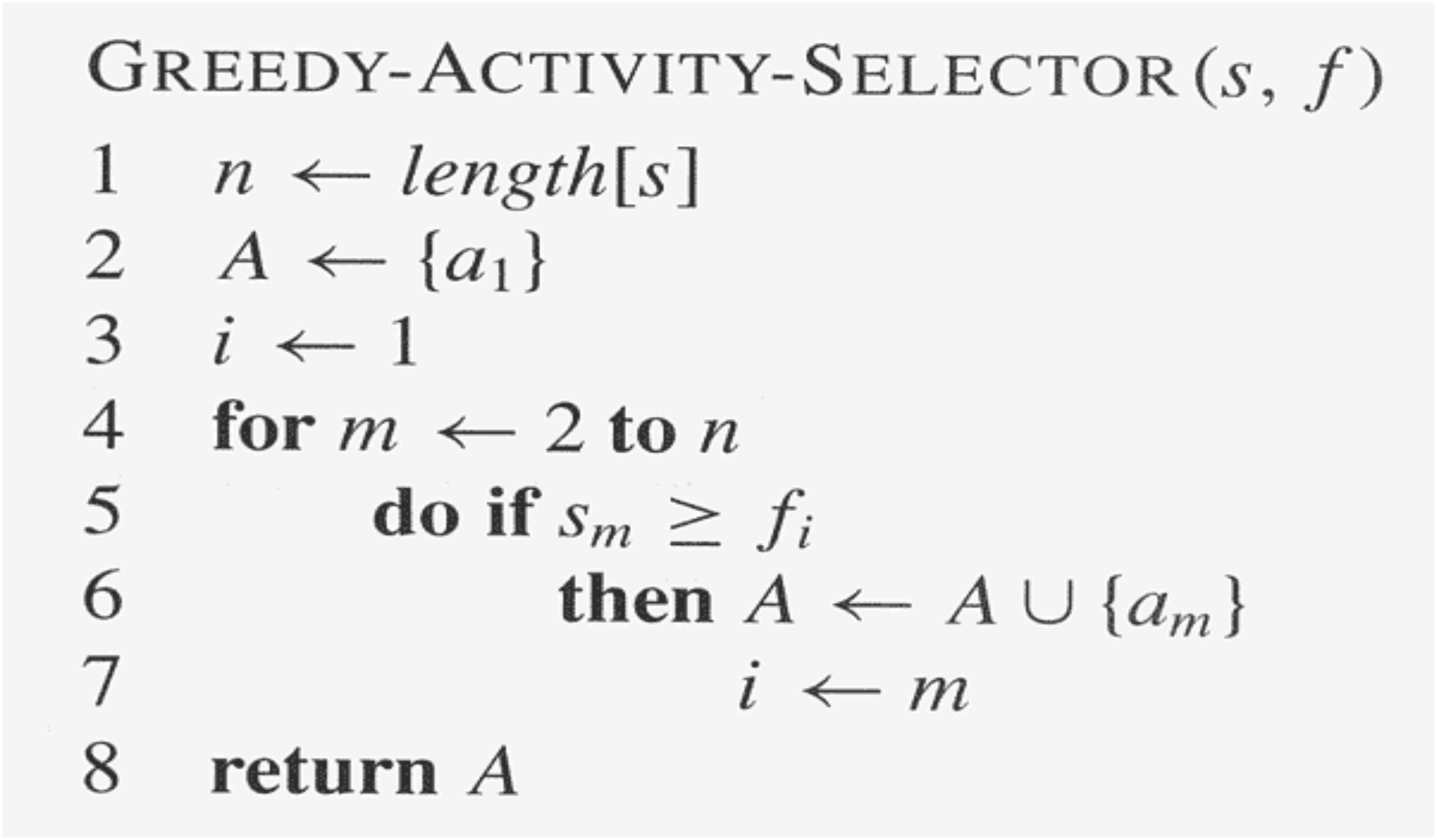

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


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 $A \subseteq S$ 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

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

- 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

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 $S' = \{i \in S : s_i \geq f_1\}$

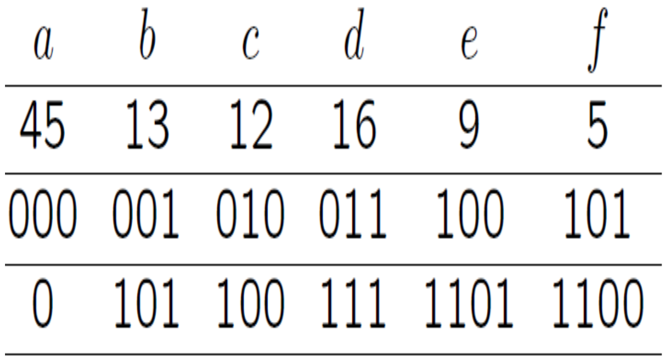
Huffman Coding

- Suppose that we have a 1,00,000 character data file that we wish to store . The file contains only 6 characters, appearing with the following frequencies:

a	b	c	d	e	f
45	13	12	16	9	5

- We would like to find a binary code that encodes the file using as few bits as possible.

- We can encode using two schemes
  - *fixed-length code*
  - *variable-length code*
- The fixed length-code requires 3,00,000 bits to store the file. The variable-length code uses only
- (45.1+13.3+12.3+16.3+9.4+5.4) .1000 = 2,24,000 bits,



a	b	c	d	e	f
45	13	12	16	9	5
000	001	010	011	100	101
0	101	100	111	1101	1100

a	b	c	d	e	f
45	13	12	16	9	5
000	001	010	011	100	101
0	101	100	111	1101	1100

- a *code* will be a set of codewords
- {000; 001; 010; 011; 100; 101}
- or {0; 101; 100; 111; 1101; 1100}

Encoding

- Given a code (corresponding to some alphabet Γ) and a message it is easy to *encode* the message. Just replace the characters by the corresponding codewords.
- Example: Γ = {a; b; c; d}
- If the code is C1{a= 00; b = 01; c = 10; d = 11}:
- then bad is encoded into 010011
- If the code is C2 {a = 0; b = 110; c = 10; d = 111}
- then bad is encoded into 1100111

Decoding

- Given an encoded message, *decoding* is the process of turning it back into the original message
- C1{a= 00; b = 01; c = 10; d = 11}
- C2 {a = 0; b = 110; c = 10; d = 111}
- C3 {a = 1; b = 110; c = 10; d = 111}
- C1: 010011 is uniquely decodable to bad
- C2: 1100111 is uniquely decodable to bad
- C3: 1101111 is not uniquely decodable to bad, It can be decoded to (acad, acda, acaaaa, bad, bda)

Prefix-Codes

- **Prefix Code:** A code is called a prefix (free) code if no codeword is a prefix of another one.
- **Example:** {a = 0; b = 110; c = 10; d = 111} is a prefix code.

- It is always uniquely decodable.

Optimum Source Coding Problem

- **The problem:** Given an alphabet A $\{a_1; : : : ; a_n\}$
- with frequency distribution $f(a_i)$ find a binary prefix code C for A that minimizes the number of bits
- $B(C) = \sum_{i=1}^{n} f(a_i) L(C(a_i))$

Building a Tree

Scan the original text

Eerie eyes seen near lake.

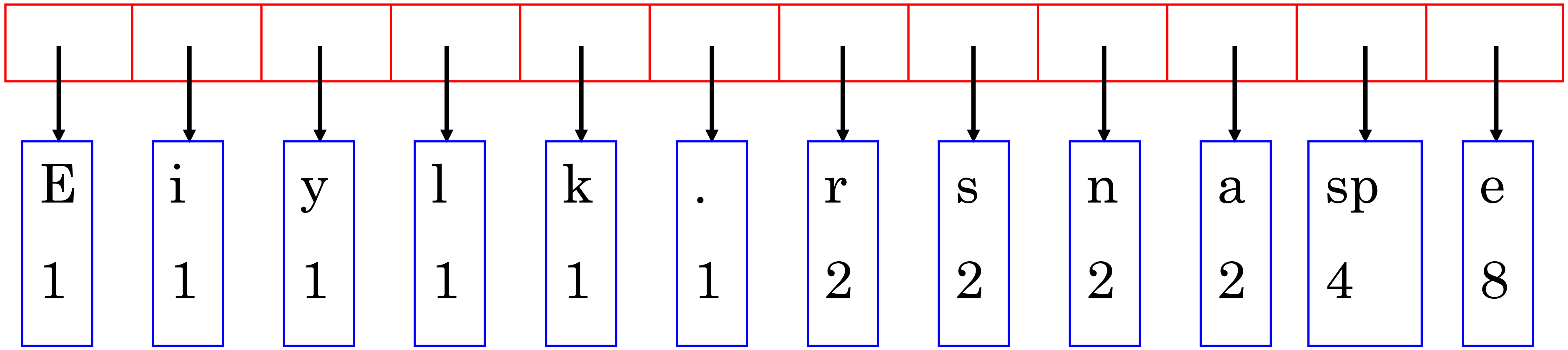
- What is the frequency of each character in the text?

Char	Freq	Char	Freq
E	1	s	2
e	8	n	2
r	2	a	2
i	1	l	1
space	4	k	1
y	1	.	1

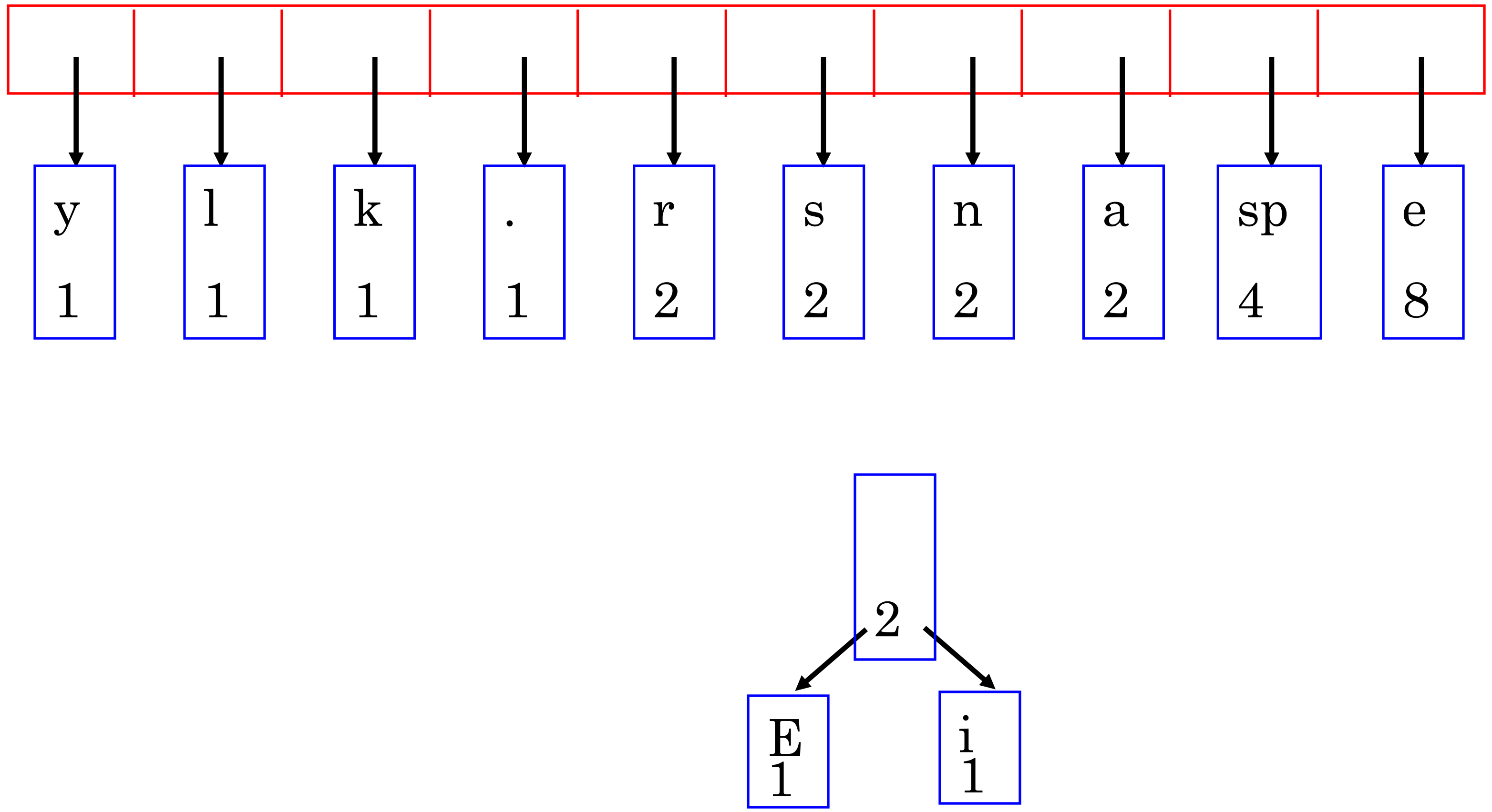
Building Tree

- Pick two letters x; y from alphabet A with the smallest frequencies and create a subtree that has these two characters as leaves. (greedy idea)
- Label the root of this subtree as z.
- Set frequency f(z) = f(x)+f(y).
- Remove x; y and add z creating new alphabet
- A’ = A U {z} – {x, y}
- Note that |A’| = |A| - 1.

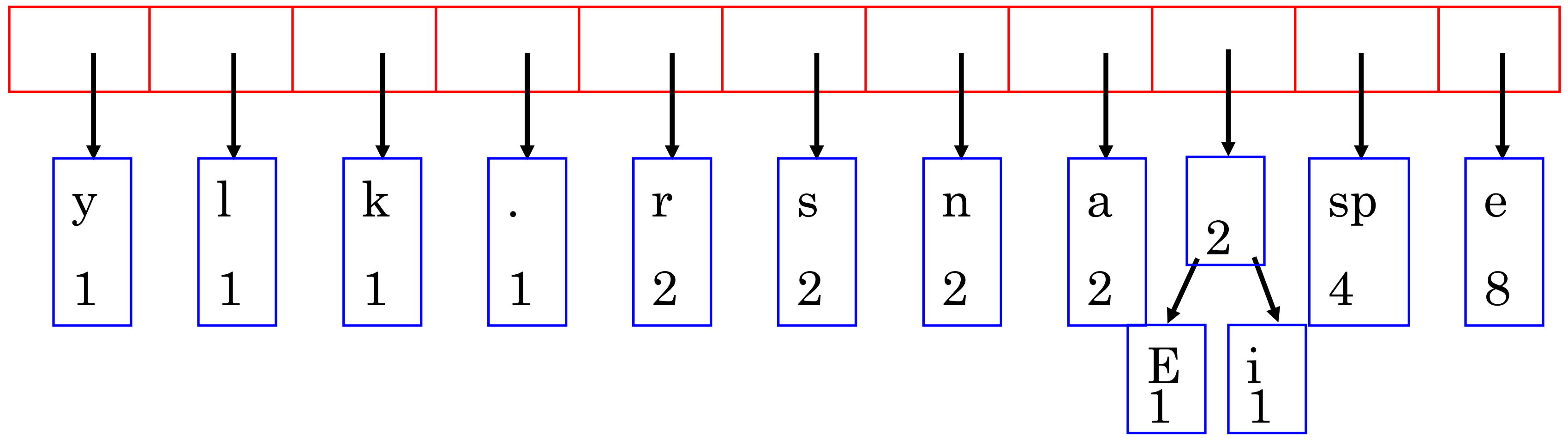
Building a Tree



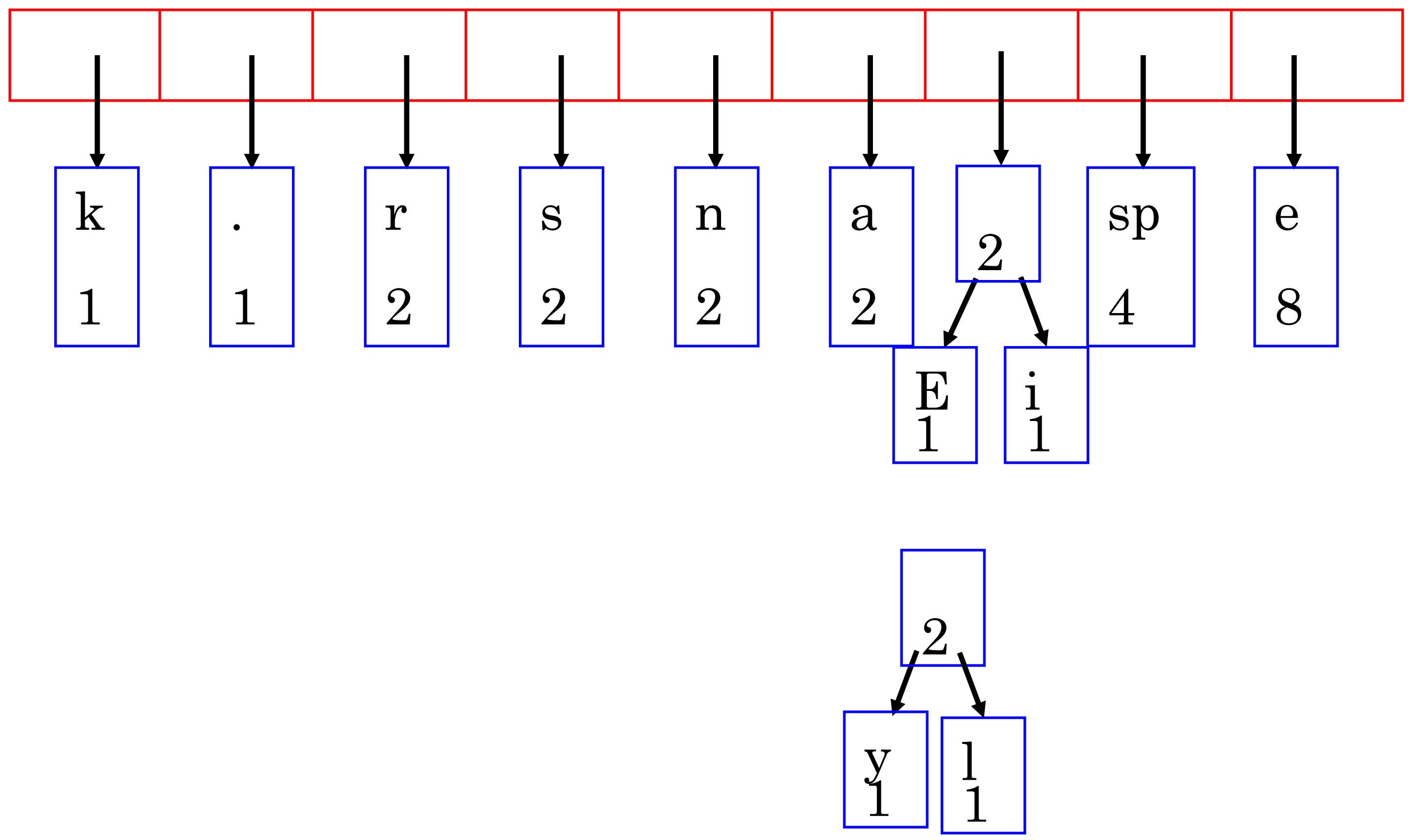
Building a Tree



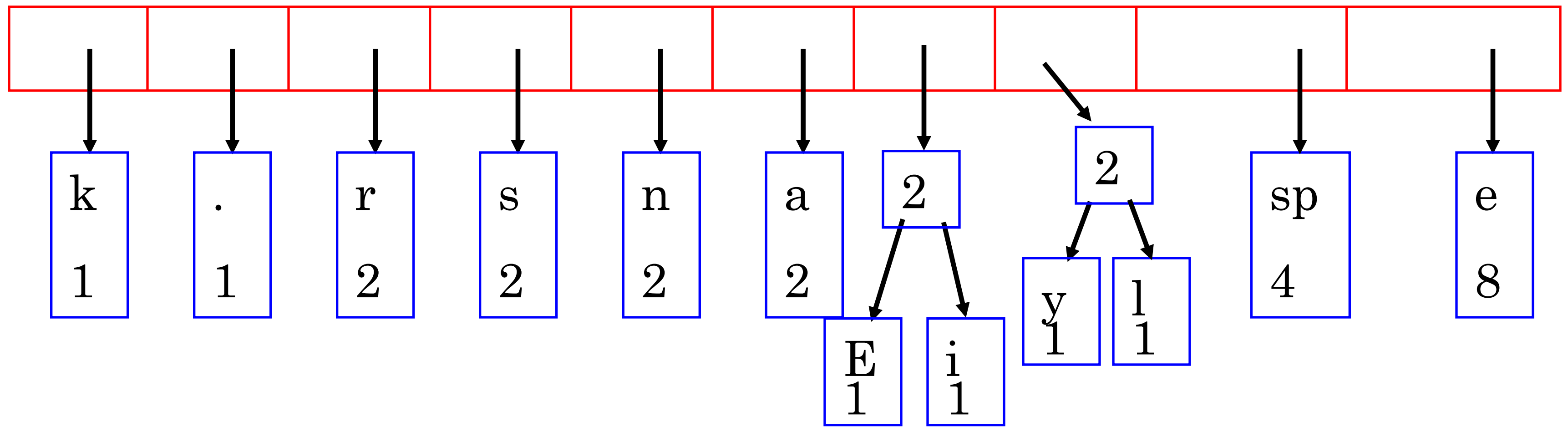
Building a Tree



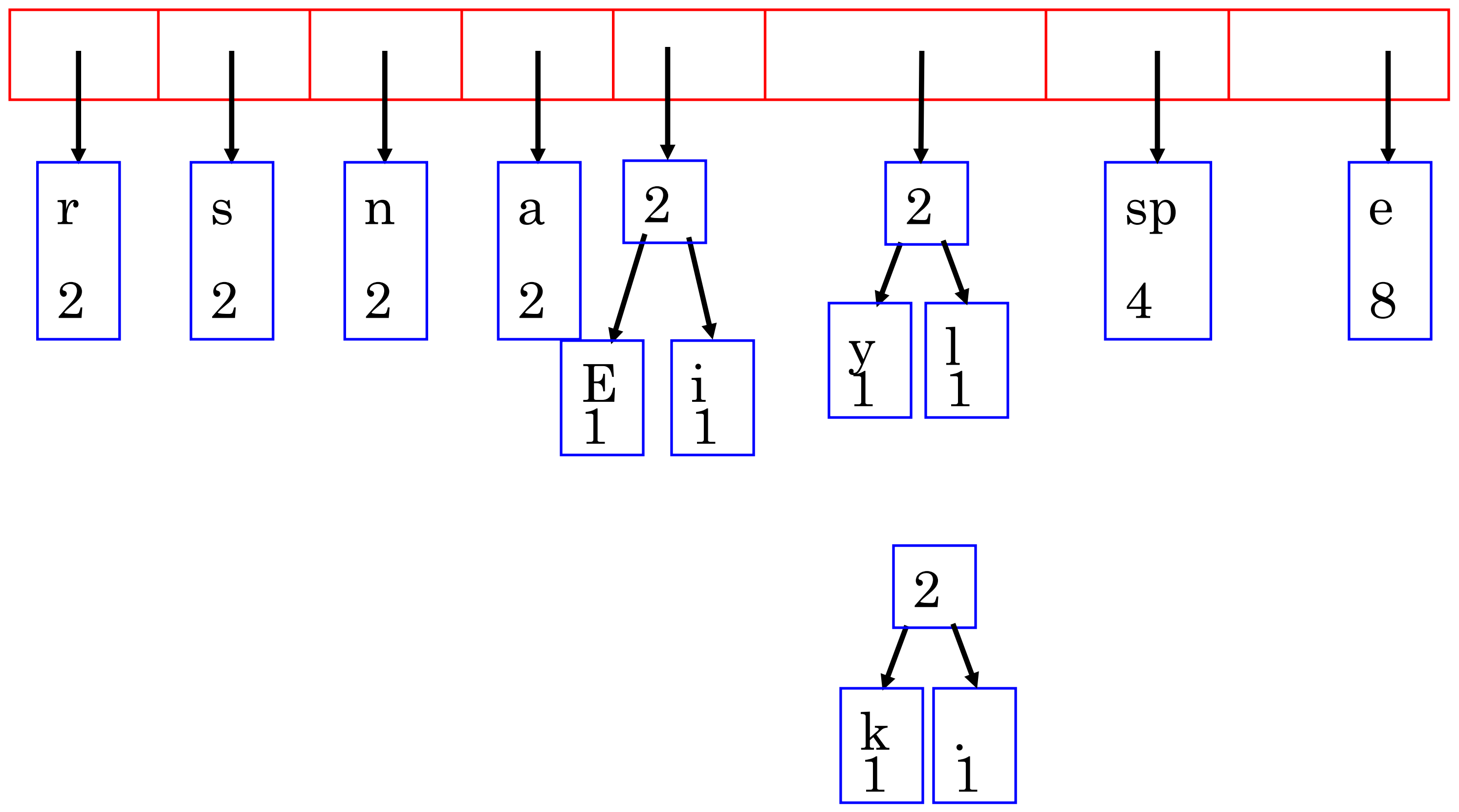
Building a Tree



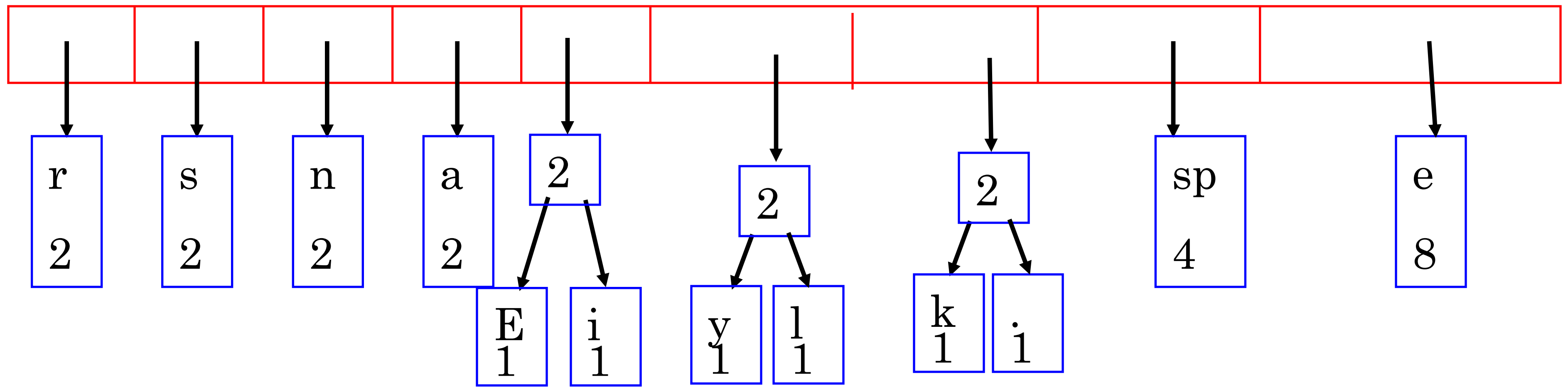
Building a Tree



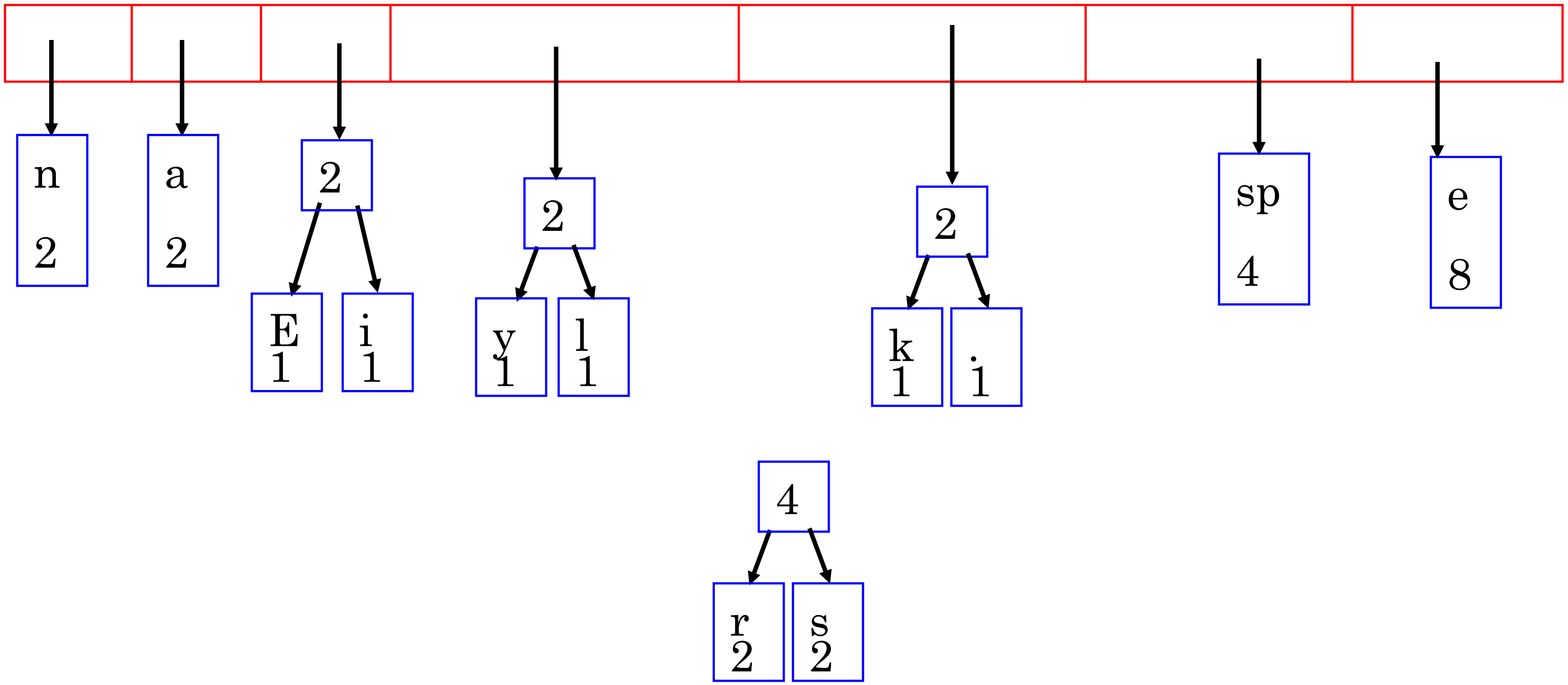
Building a Tree



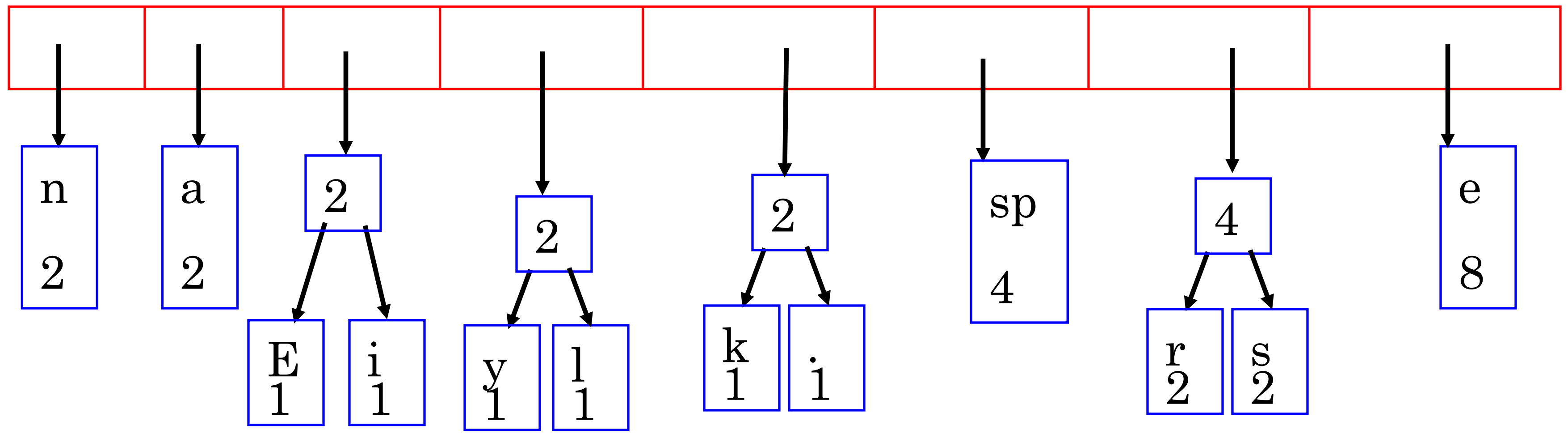
Building a Tree



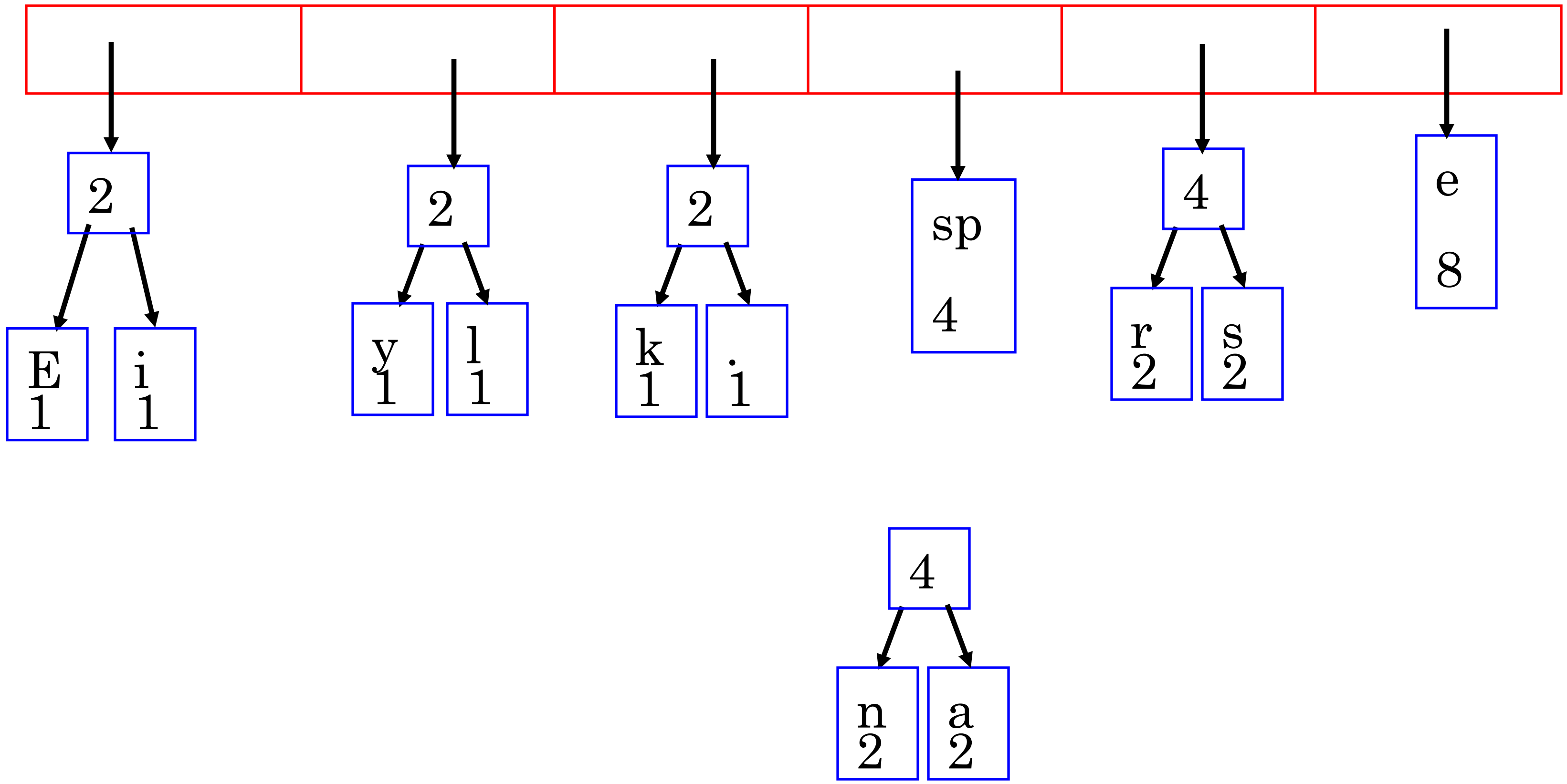
Building a Tree



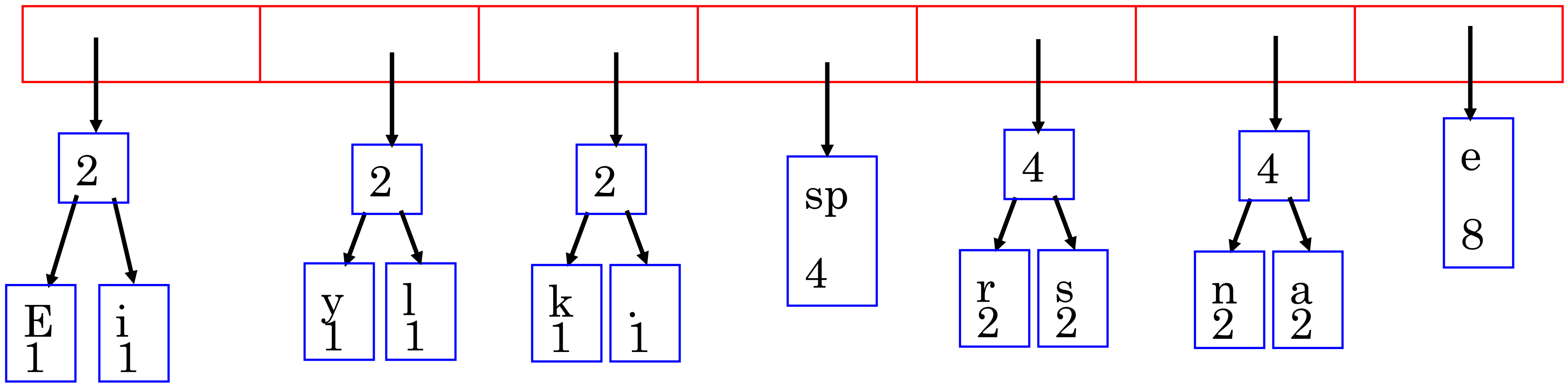
Building a Tree



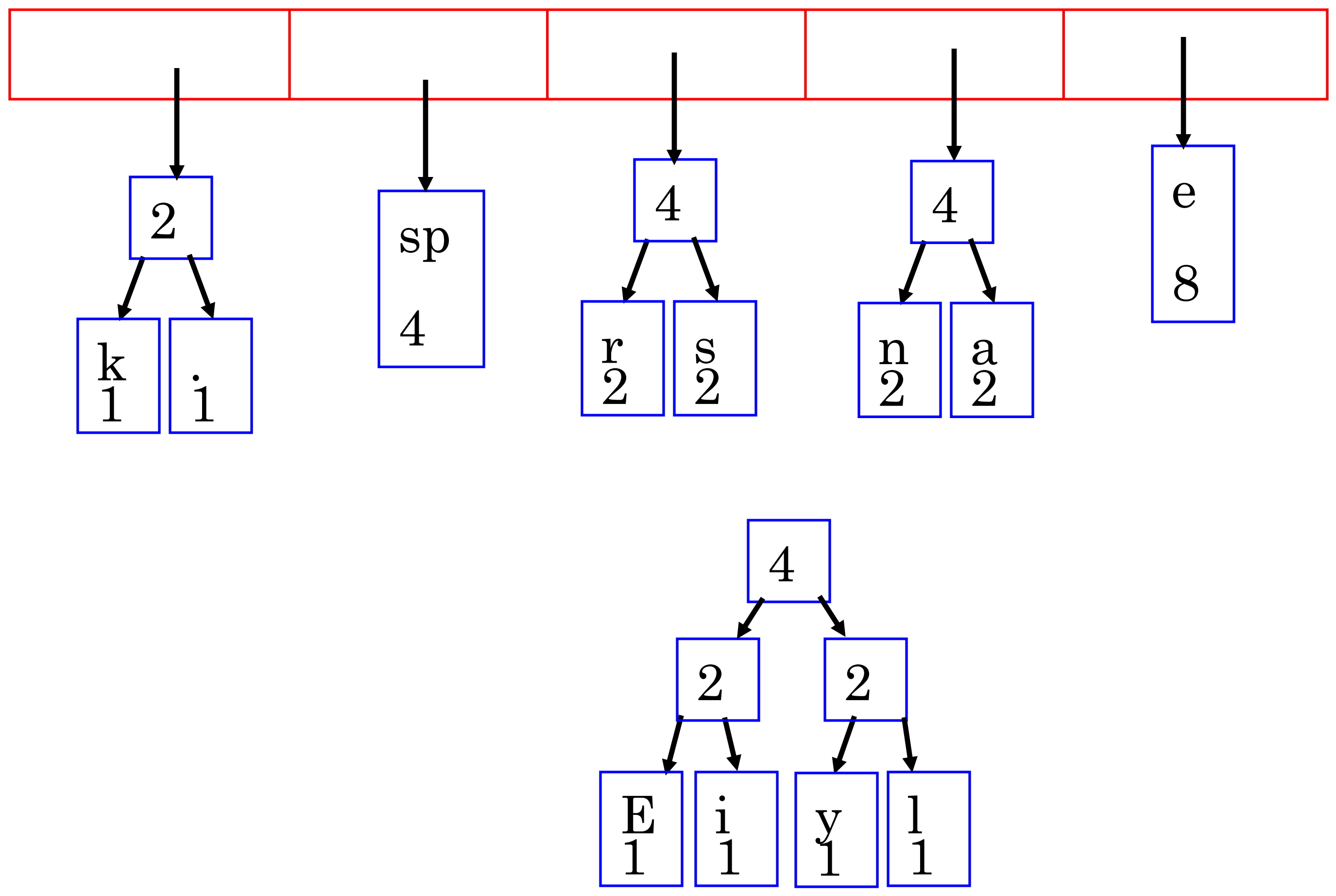
Building a Tree



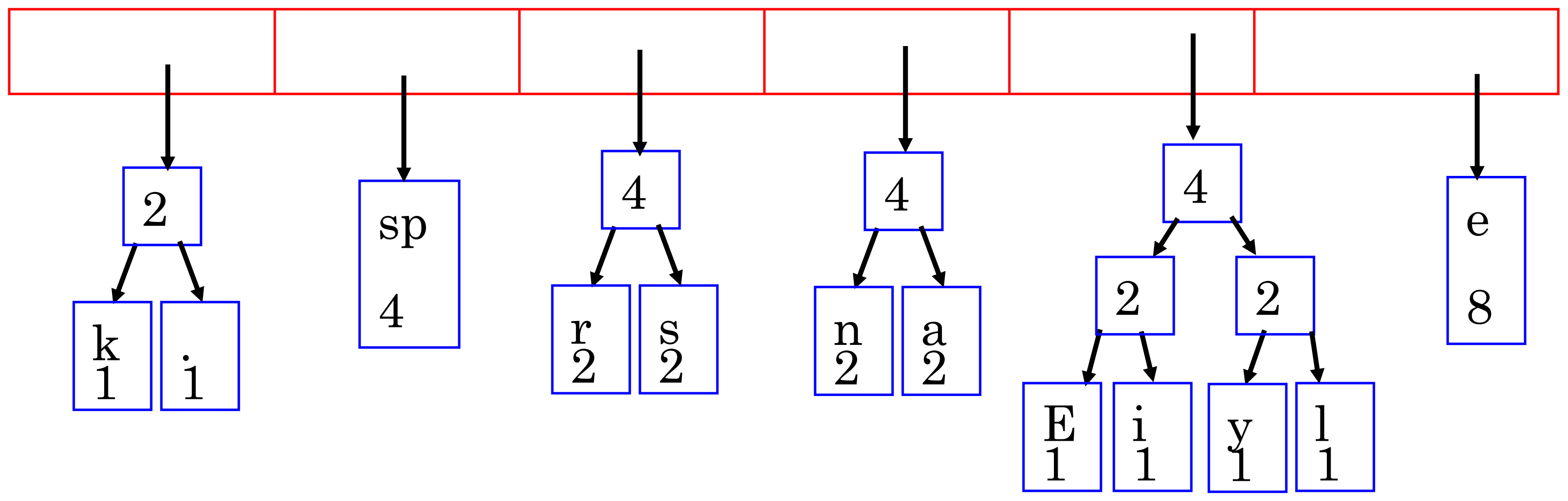
Building a Tree



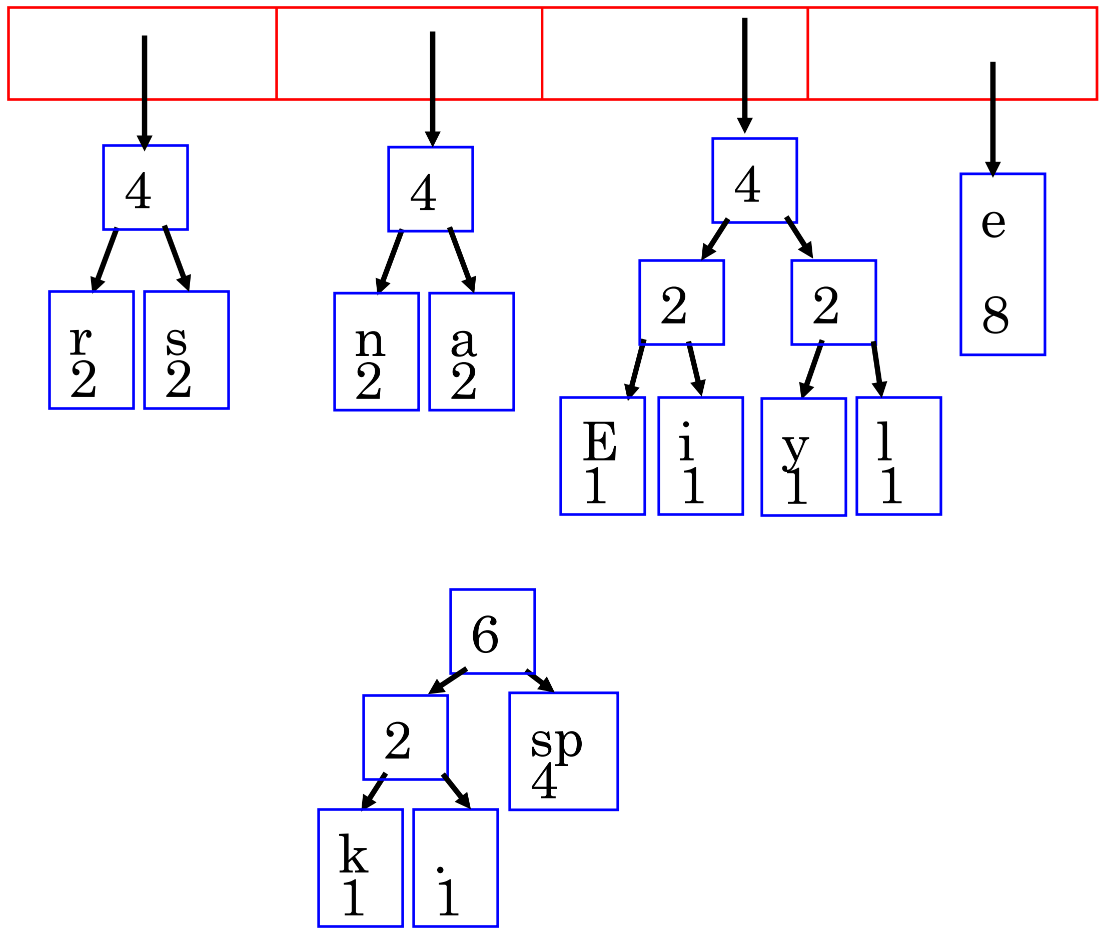
Building a Tree



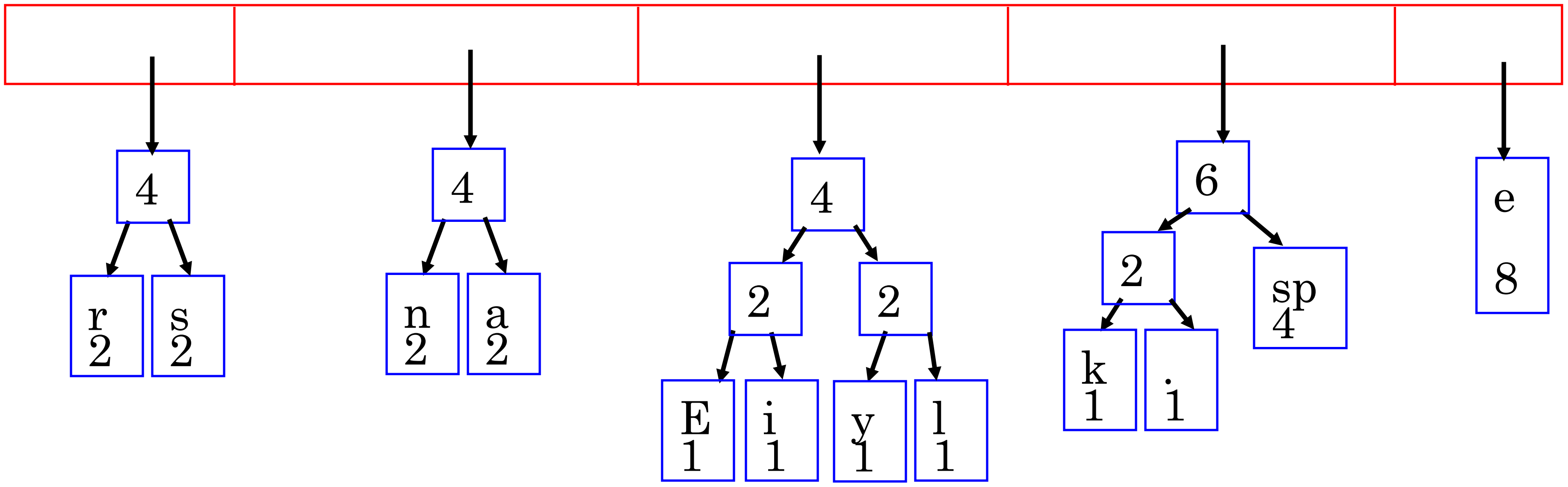
Building a Tree



Building a Tree

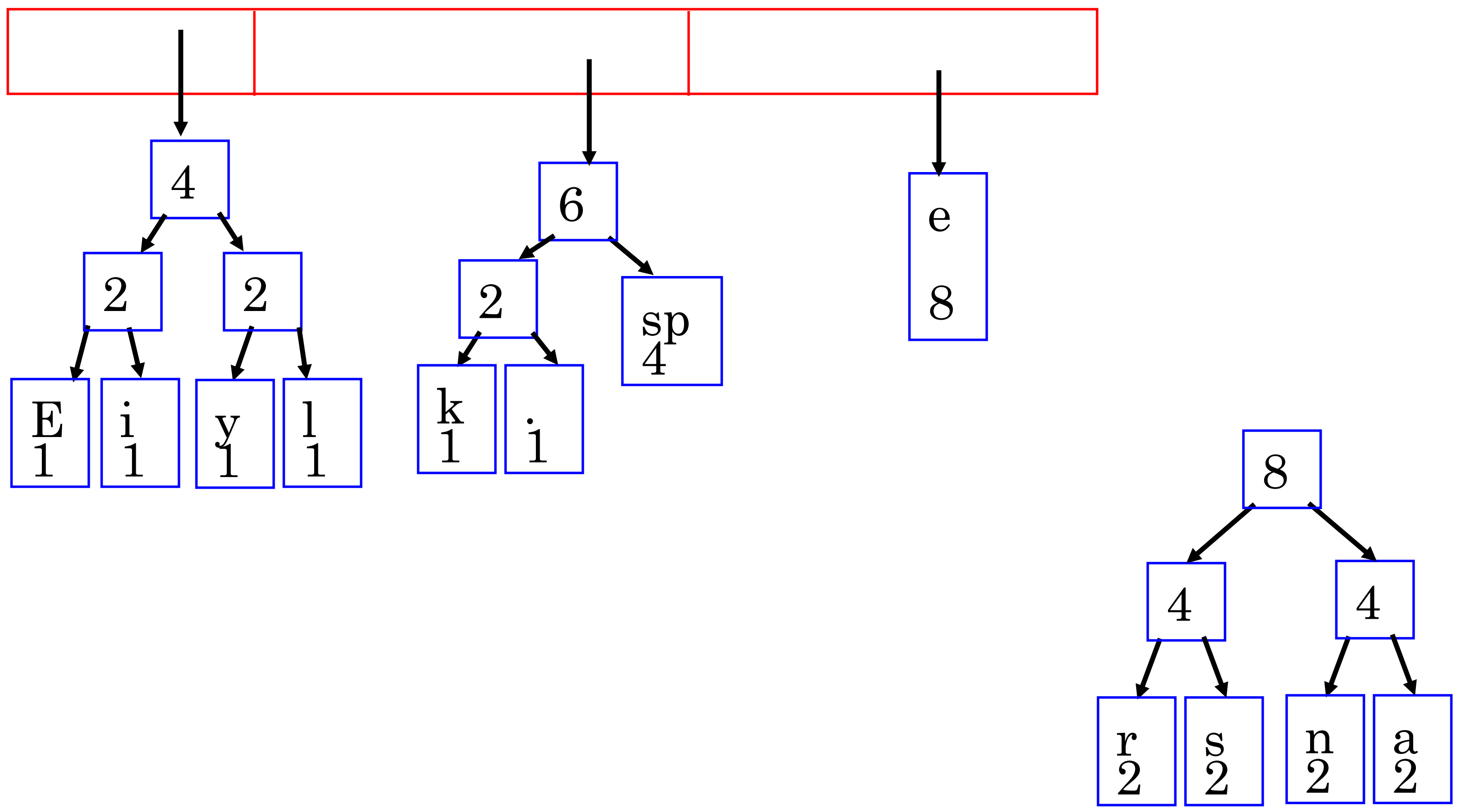


Building a Tree

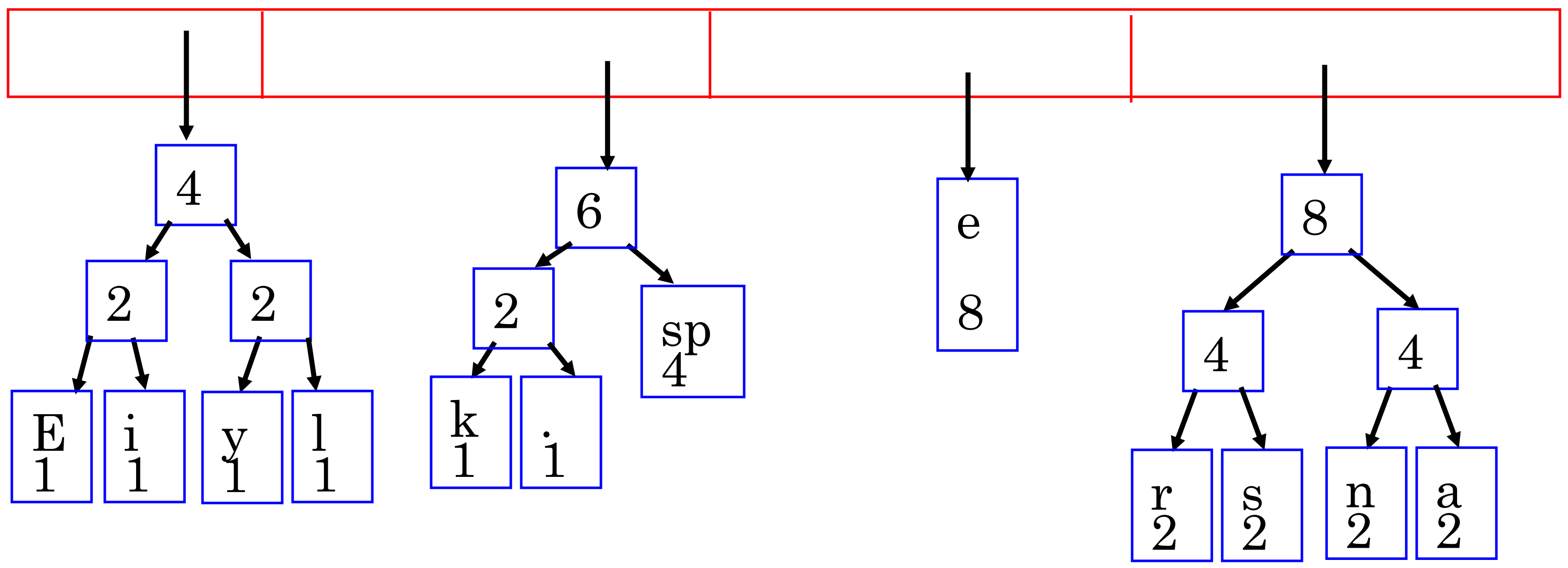


What is happening to the characters with a low number of occurrences?

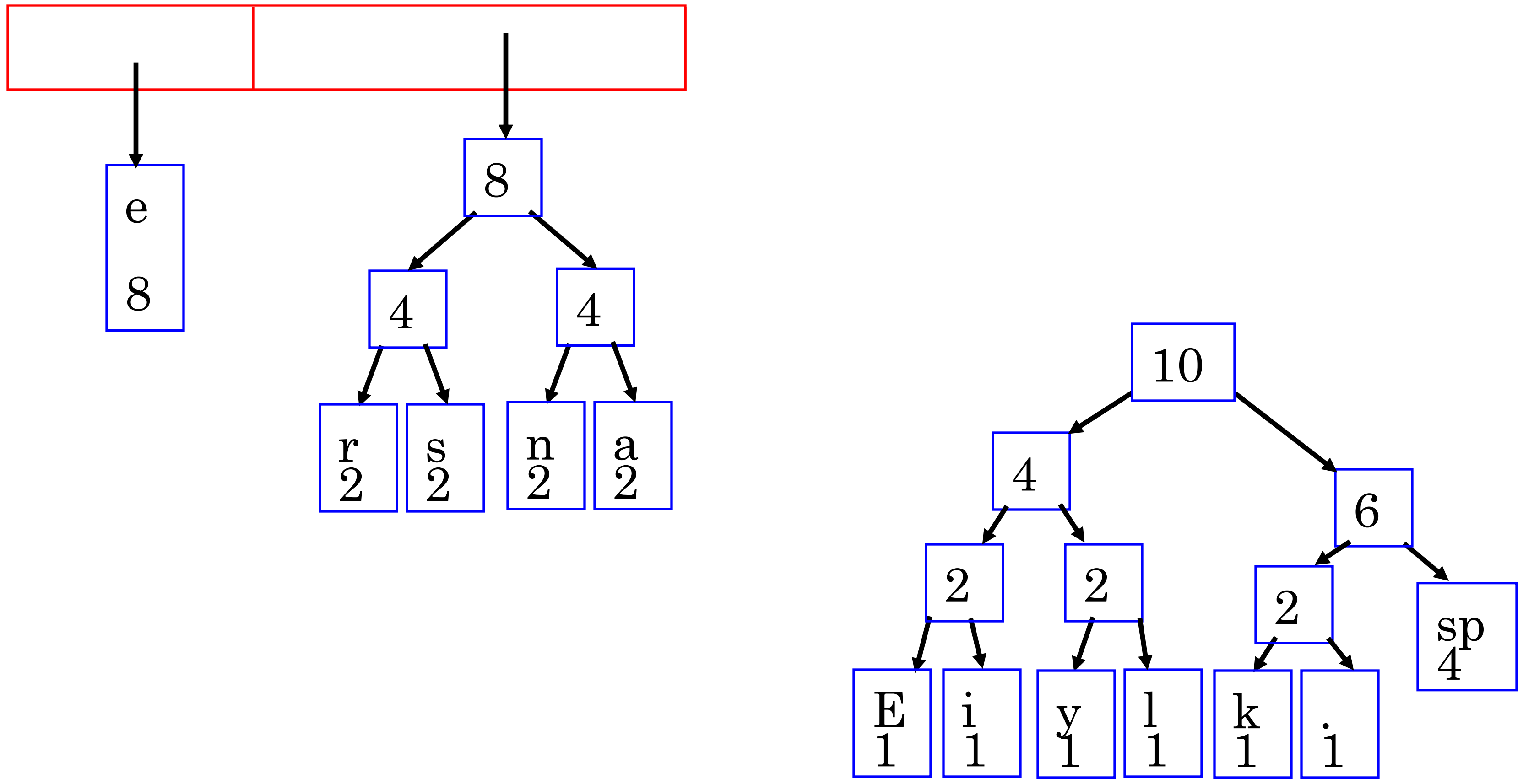
Building a Tree



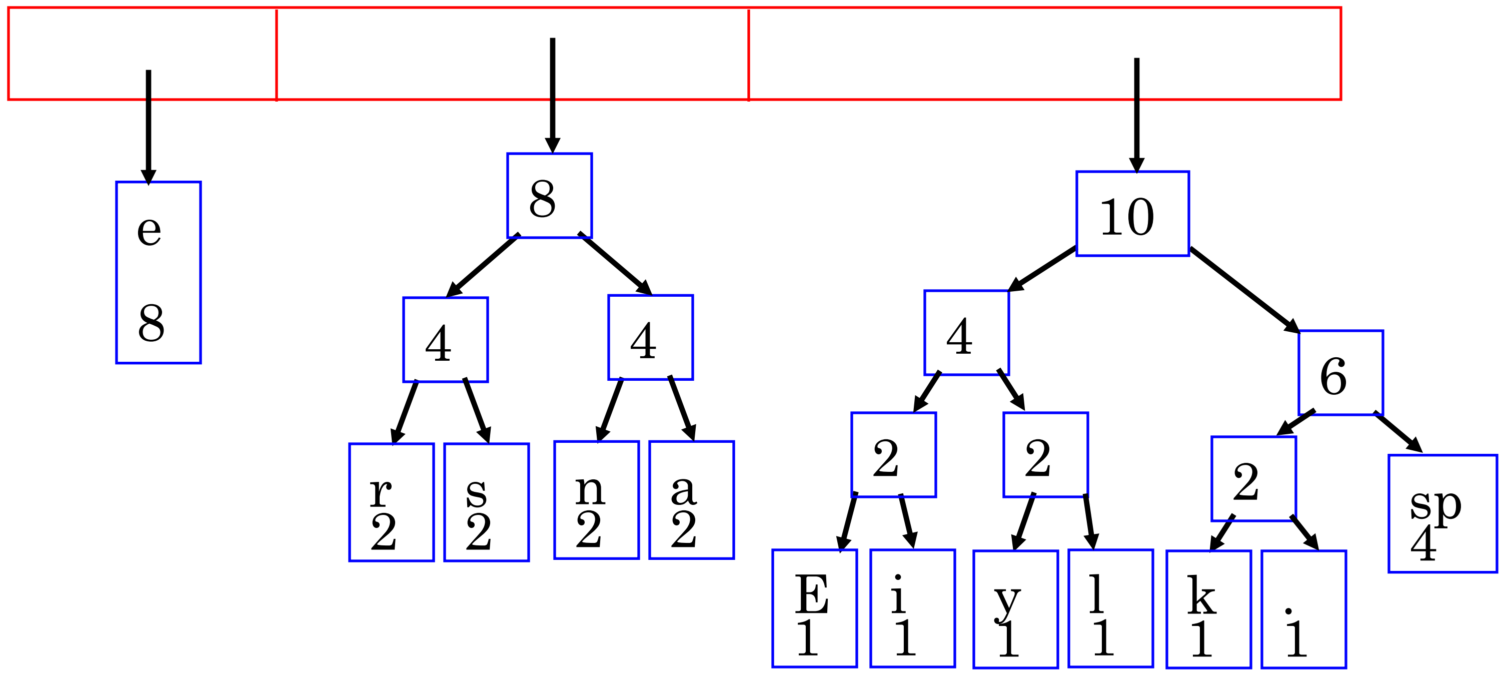
Building a Tree



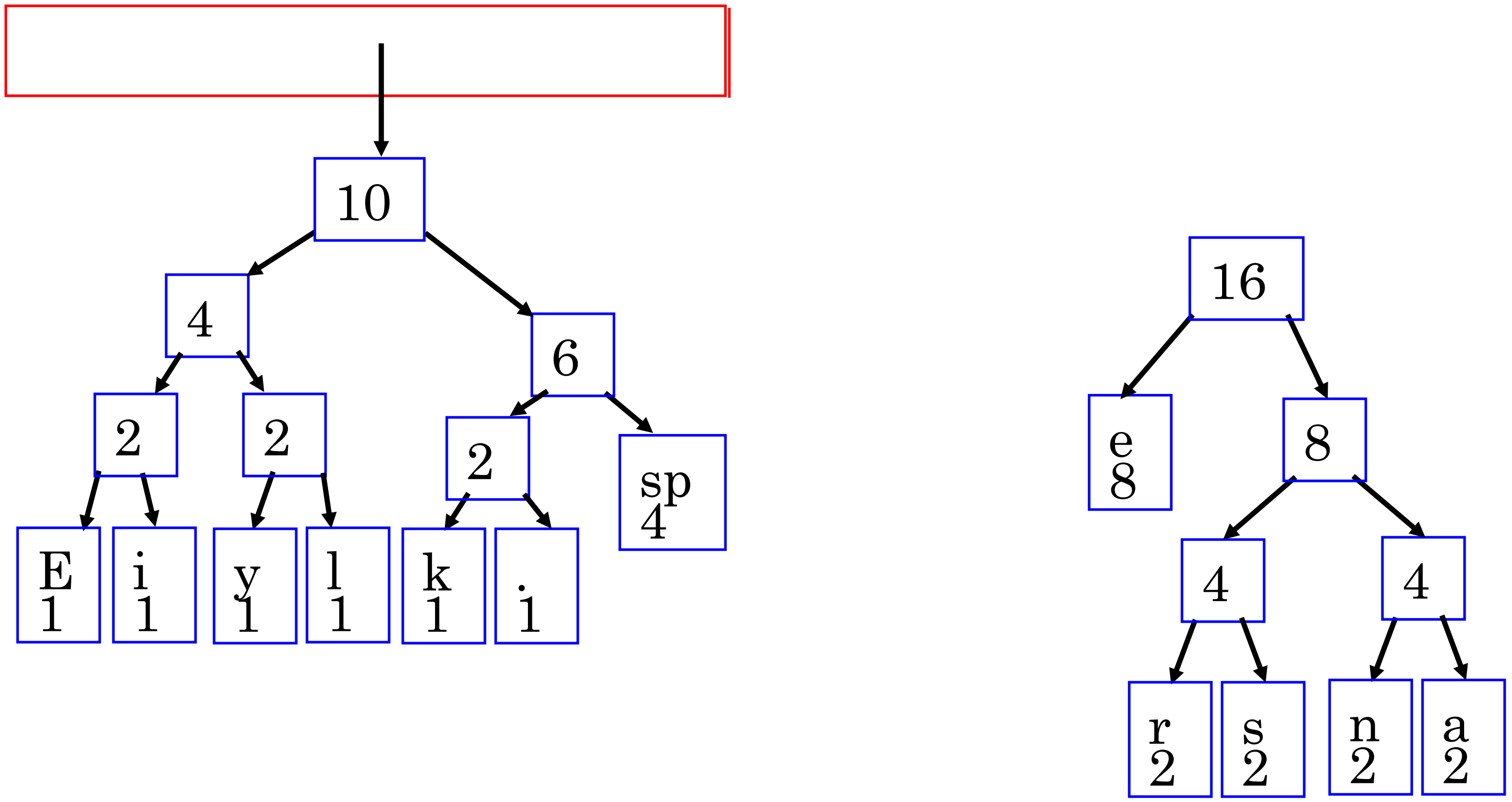
Building a Tree



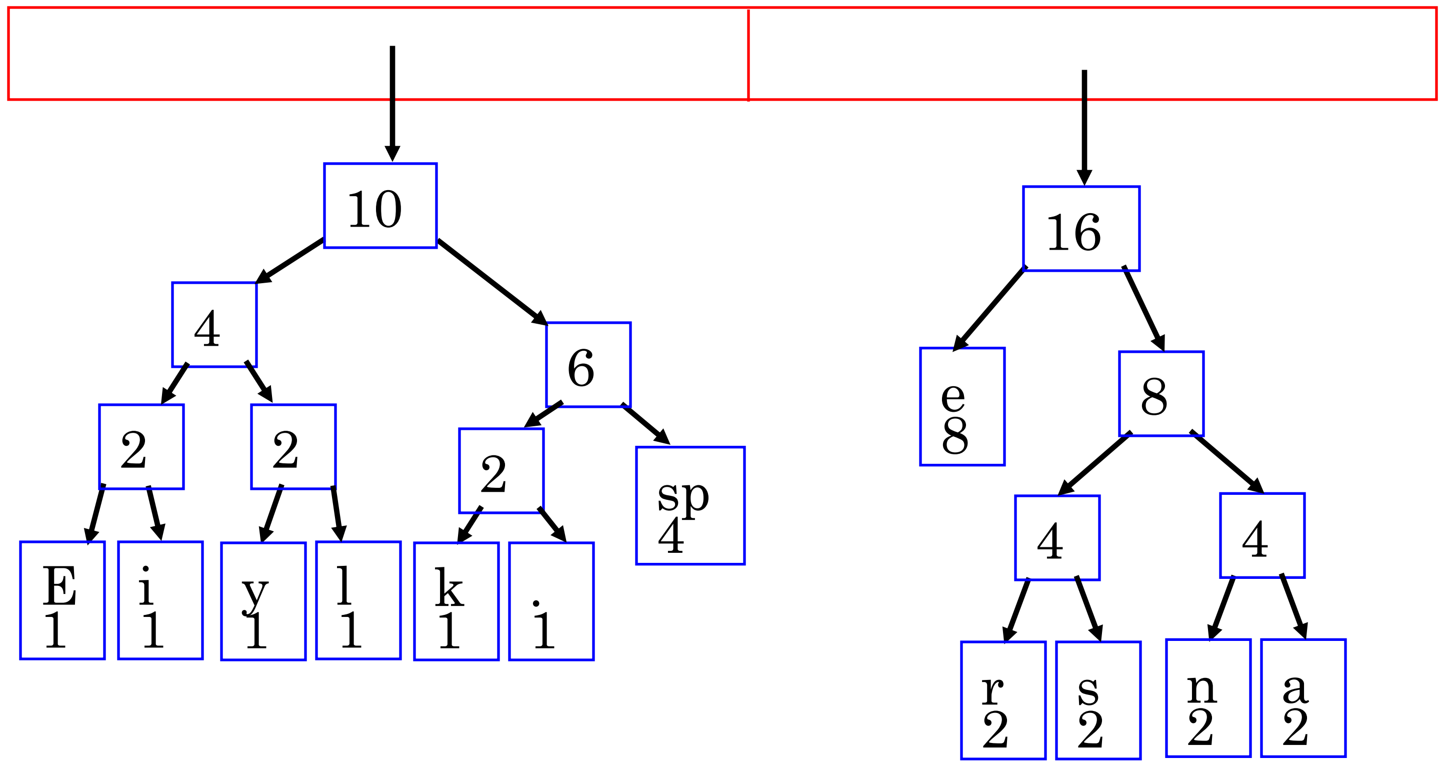
Building a Tree



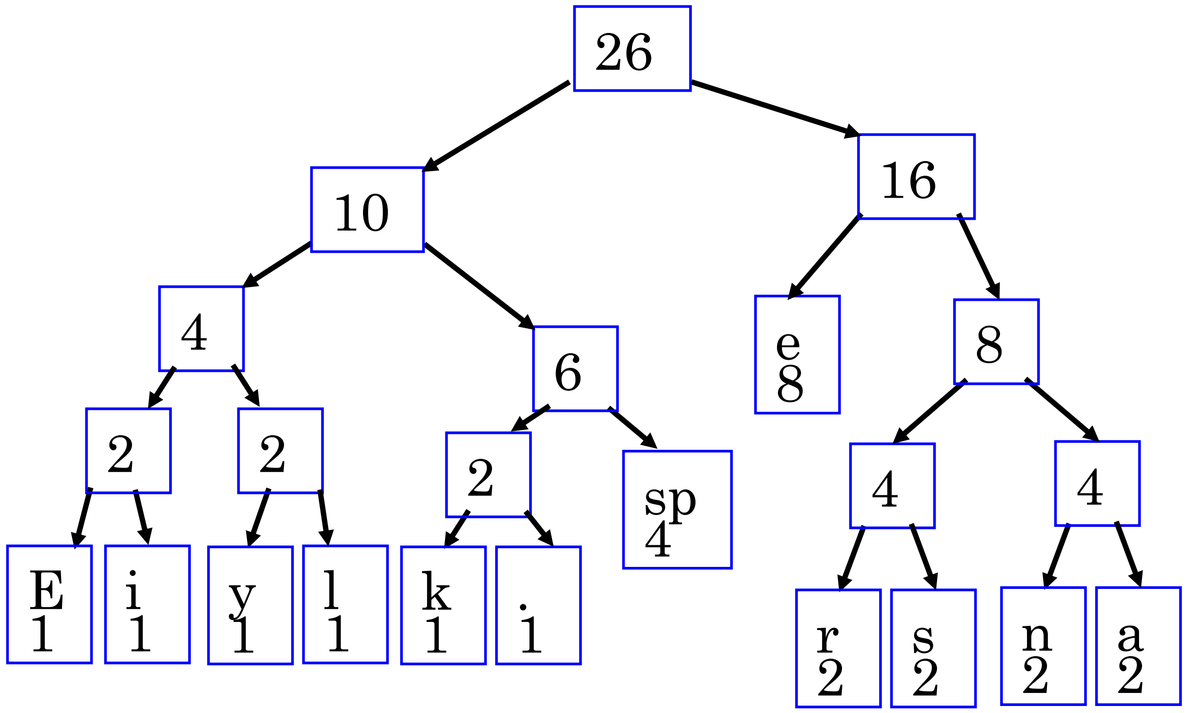
Building a Tree



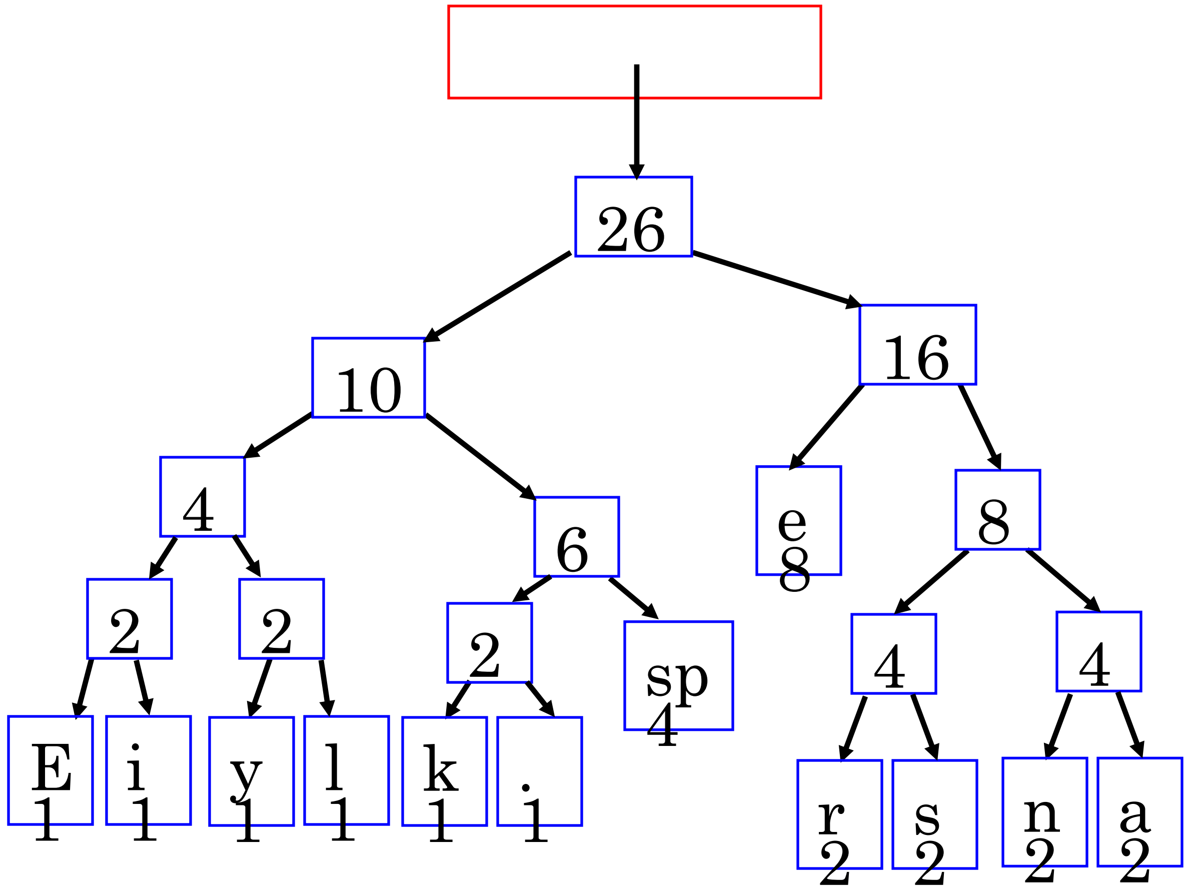
Building a Tree



Building a Tree



Building a Tree



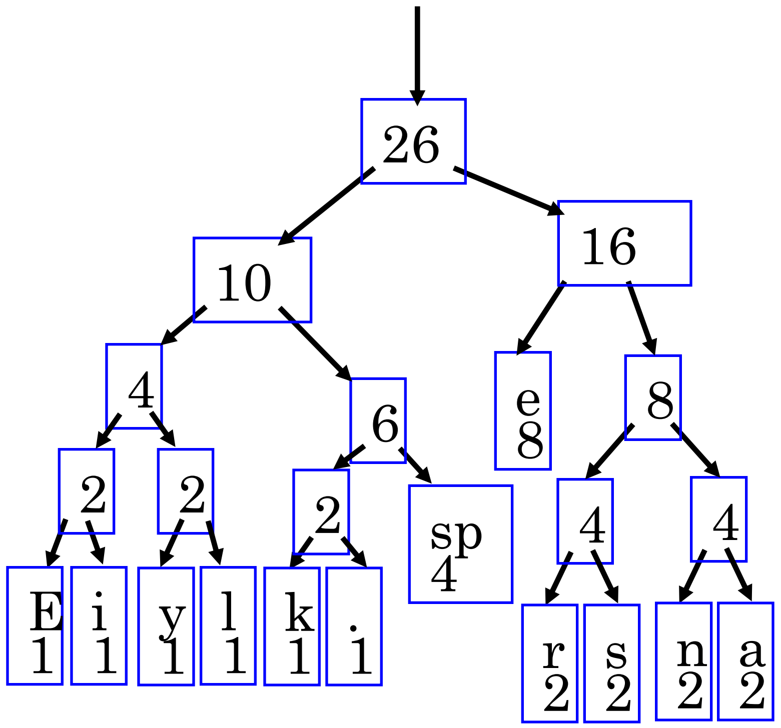
After enqueueing this node there is only one node left in priority queue.

Building a Tree

Dequeue the single node left in the queue.

This tree contains the new code words for each character.

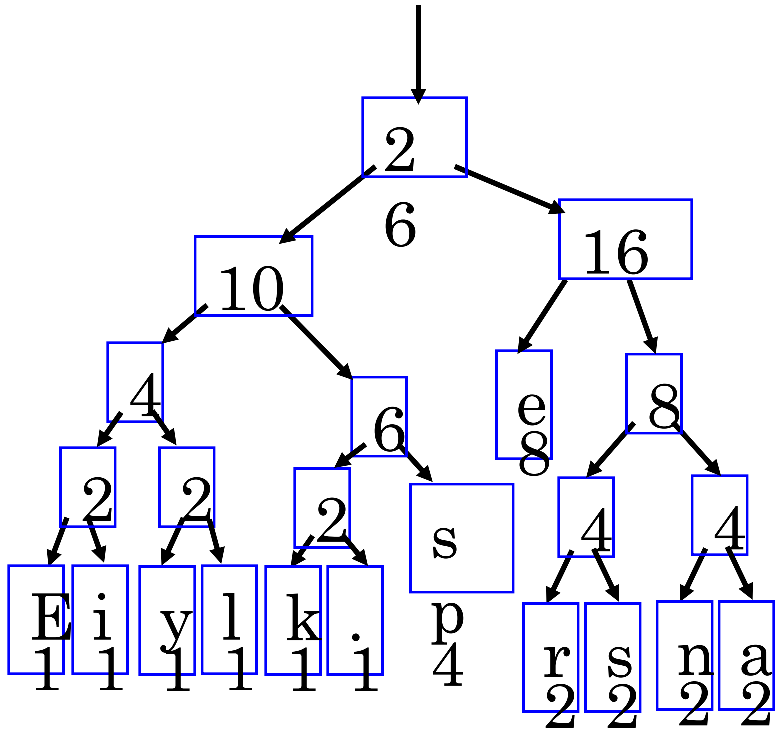
Frequency of root node should equal number of characters in text.



Eerie eyes seen near lake.   26 characters

Encoding the File
Traverse Tree for Codes

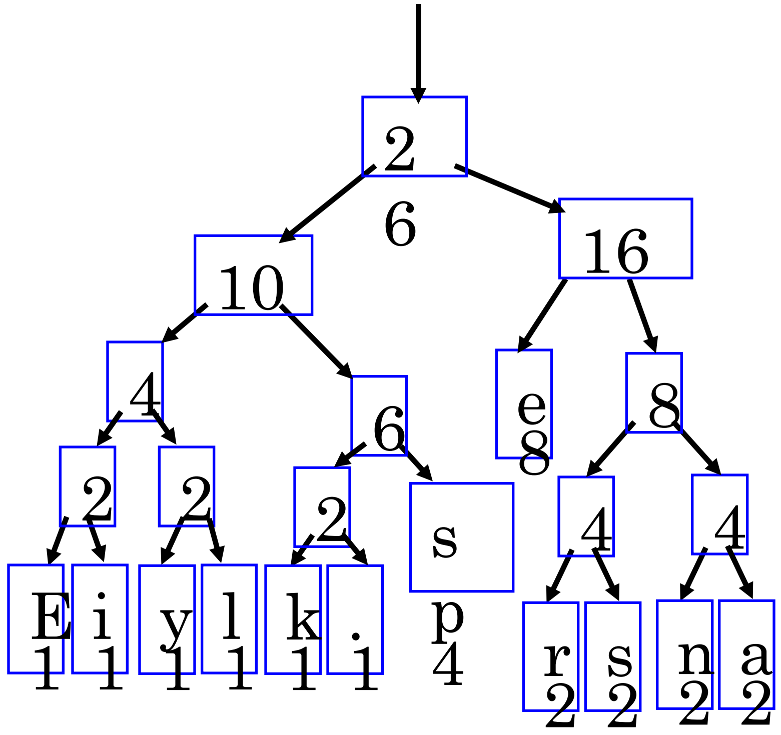
- Perform a traversal of the tree to obtain new code words
- Going left is a 0 going right is a 1
- code word is only completed when a leaf node is reached



Encoding the File

Traverse Tree for Codes

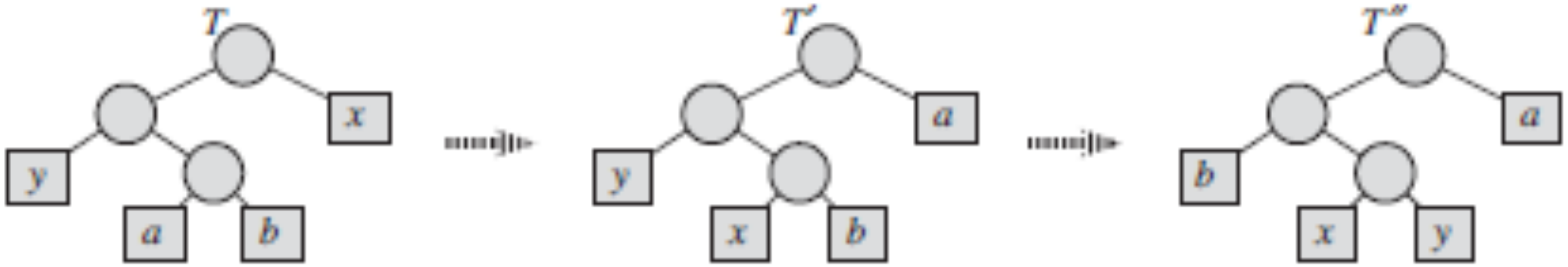
Char	Code
E	0000
i	0001
y	0010
l	0011
k	0100
.	0101
space	011
e	10
r	1100
s	1101
n	1110
a	1111

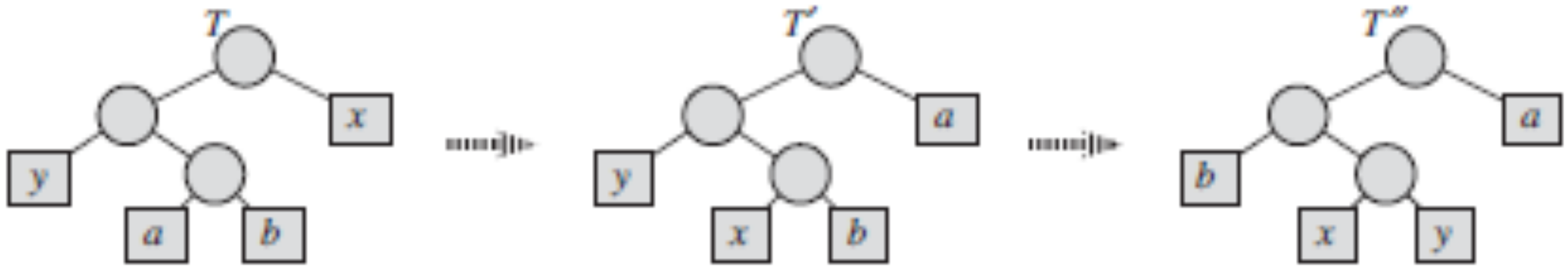


Huffman Coding Algorithm

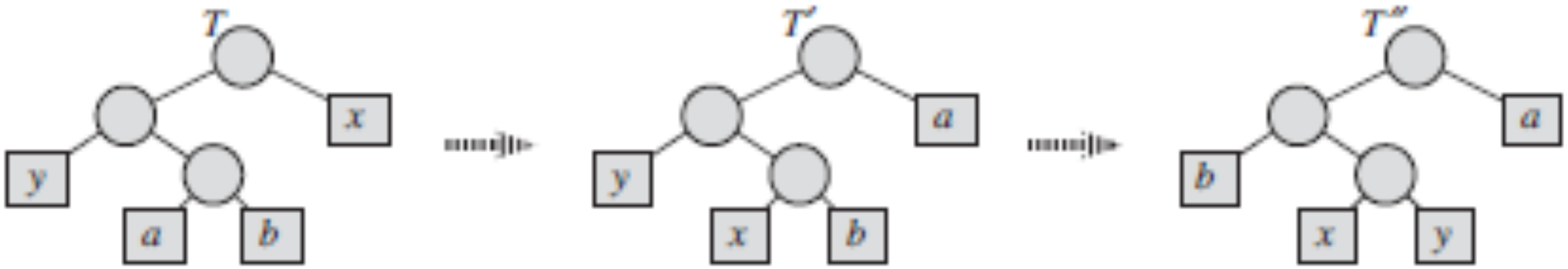
```
Huffman($A$)
{   $n = |A|$;
    $Q = A$;                        the future leaves
    for $i = 1$ to $n-1$            Why $n-1$?
    {   $z =$ new node;
        $left[z] =$Extract-Min($Q$);
        $right[z] =$Extract-Min($Q$);
        $f[z] = f[left[z]] + f[right[z]]$;
        Insert($Q, z$);
    }
    return Extract-Min($Q$)  root of the tree
}
```

- Let C be an alphabet in which each character c ∈ C has frequency c:*freq*. 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.





- Let a and b be two characters that are sibling leaves of maximum depth in T . Without loss of generality, we assume that a:*freq* ≤ b:*freq* and x:*freq* ≤ y:*freq*. Since x:*freq* and y:*freq* are the two lowest leaf frequencies, in order, and a:*freq* and b:*freq* are two arbitrary frequencies, in order, we have x:*freq* ≤ a:*freq* and y:*freq* ≤ b:*freq*.



$$
\begin{aligned}
B(T) - B(T') \\
&= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
&= (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\geq 0 ,
\end{aligned}
$$

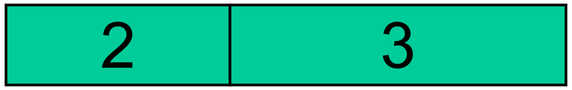
Scheduling tasks

- Each task has a length t_i and a deadline d_i
- All tasks are available at the start
- One task may be worked on at a time
- All tasks must be completed

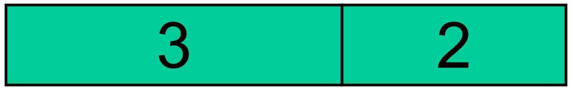
- Goal: minimize maximum lateness
  - Lateness = $f_i - d_i$ if $f_i >= d_i$

Example

Time	Deadline
2	2
3	4



Lateness 1



Lateness 3

Determine the minimum lateness

Time	Deadline
2	6
3	4
4	5
5	12

Greedy Algorithm

- Earliest deadline first
- Order jobs by deadline

- This algorithm is optimal

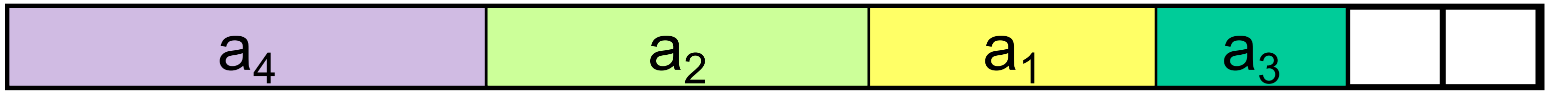
Analysis

- Suppose the jobs are ordered by deadlines, $d_1 <= d_2 <= . . . <= d_n$
- A schedule has an *inversion* if job j is scheduled before i where j > i

- The schedule A computed by the greedy algorithm has no inversions.
- Let O be the optimal schedule, we want to show that A has the same maximum lateness as O

Find the inversions

	Time	Deadline
a_1	3	4
a_2	4	5
a_3	2	6
a_4	5	12

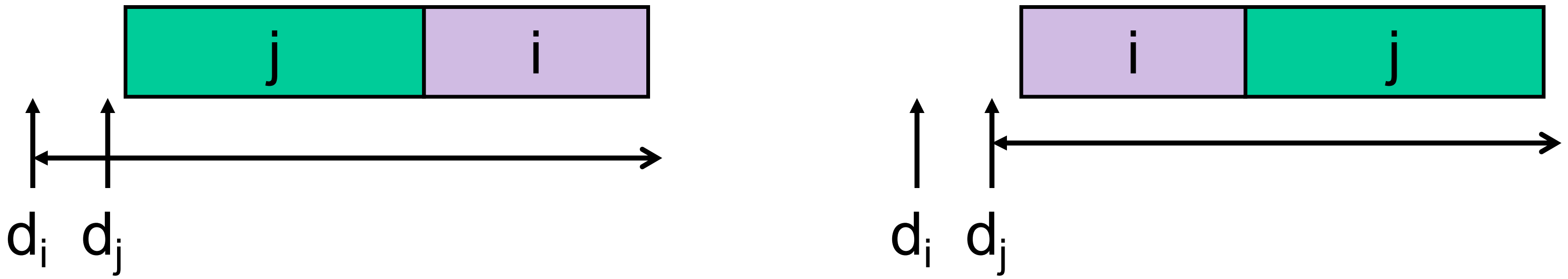


Lemma

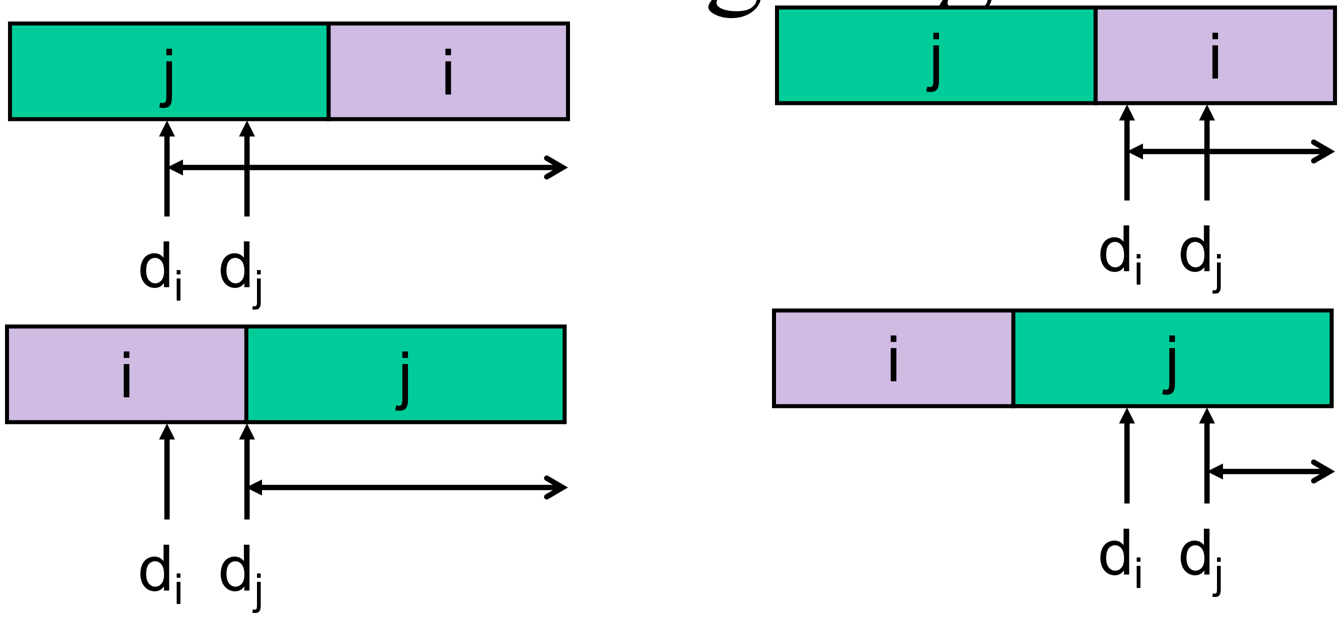
- If there is an inversion i, j, there is a pair of adjacent jobs i’, j’ which form an inversion

Interchange argument

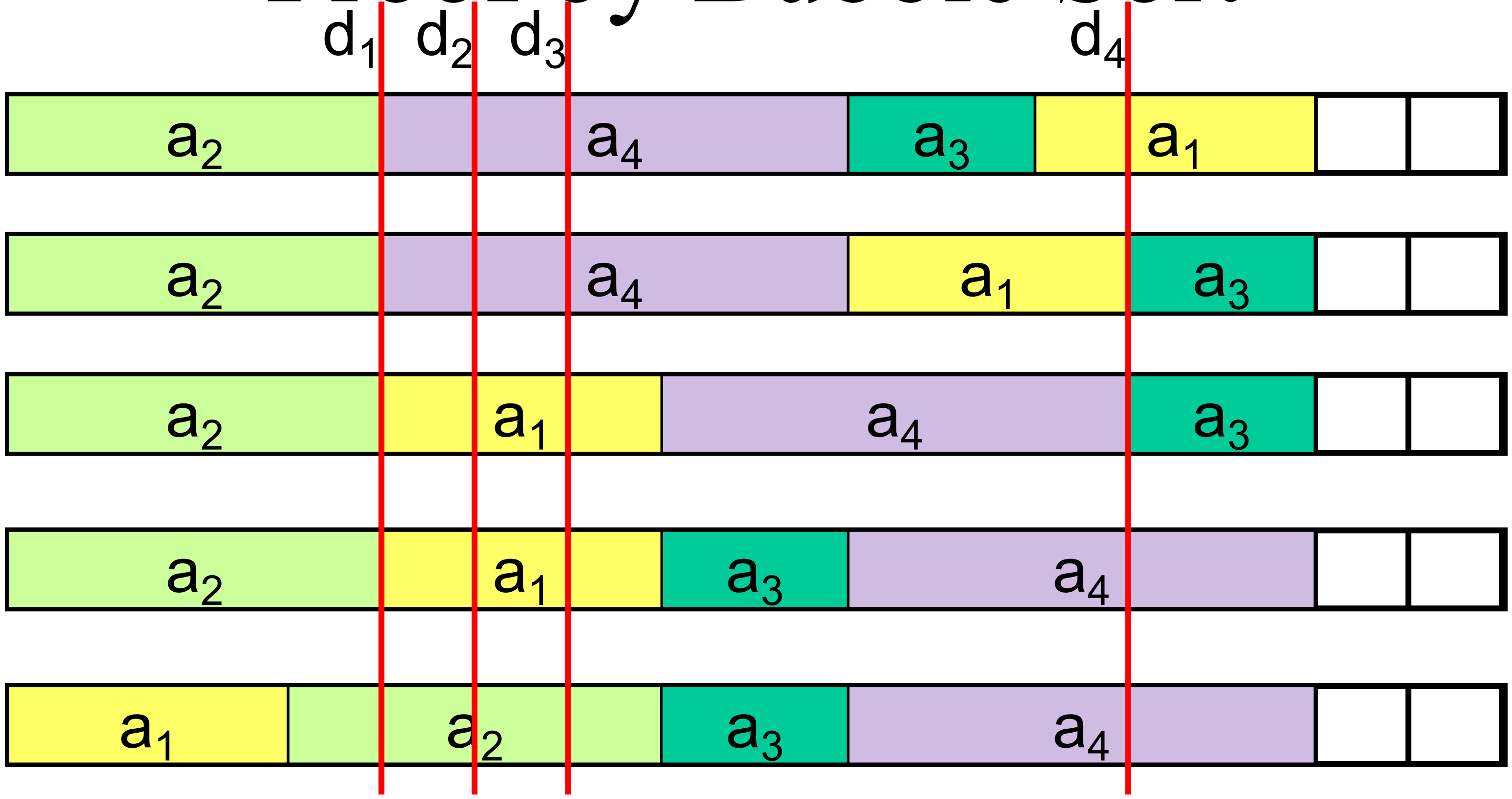
- Suppose there is a pair of jobs i and j, with d_i <= d_j, and j scheduled immediately before i. Interchanging i and j does not increase the maximum lateness.



Interchange argument



Proof by Bubble Sort



Determine maximum lateness

Real Proof

- There is an optimal schedule with no inversions and no idle time.
- Let O be an optimal schedule k inversions, we construct a new optimal schedule with k-1 inversions
- Repeat until we have an optimal schedule with 0 inversions
- This is the solution found by the earliest deadline first algorithm

Conclusion

- Earliest Deadline First algorithm constructs a schedule that minimizes the maximum lateness