COMP251: Greedy algorithms

Jérôme Waldispühl & Giulia Alberini
School of Computer Science
McGill University

Based on (Cormen *et al.*, 2002)

Based on slides from D. Plaisted (UNC) & (goodrich & Tamassia, 2009)

Overview

- Algorithm design technique to solve optimization problems.
- Problems exhibit optimal substructure.
- Idea (the greedy choice):
  - When we have a choice to make, make the one that looks best right now.
  - Make a locally optimal choice in hope of getting a globally optimal solution.

Outline

- Definition of the activity selection problem
- Greedy choice & optimal sub-structure
- Greedy algorithm for the activity selection problem
- Text compression & Huffman encoding

Greedy Strategy

The choice that seems best **at the moment** is the one we go with.

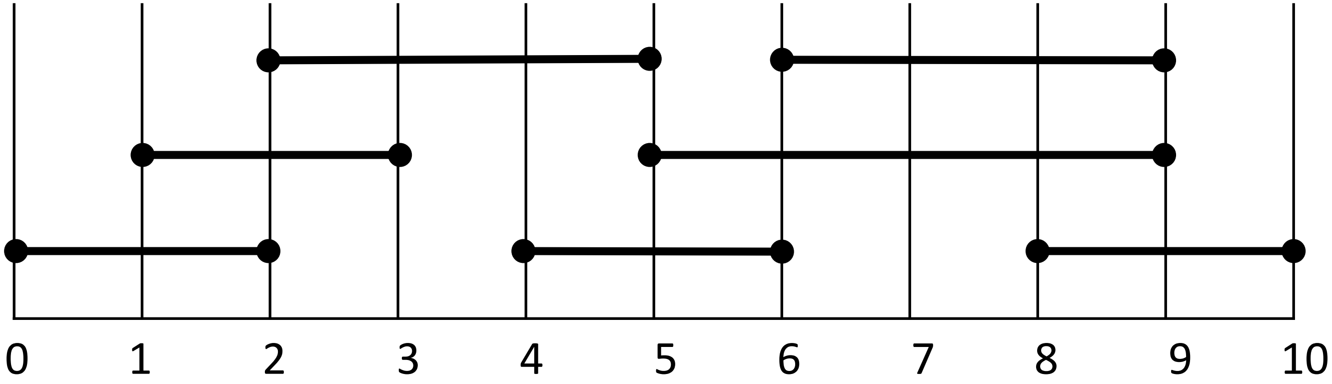
- Prove that when there is a choice to make, one of the optimal choices is the greedy choice. Therefore, it is always safe to make the greedy choice.
- Show that all but one of the sub-problems resulting from the greedy choice are empty.

Activity-selection Problem

- <u>Input:</u> Set *S* of *n* activities, a_1, a_2, …, a_n.
  - – s_i = start time of activity *i*.
  - – f_i = finish time of activity *i*.
- <u>Output:</u> Subset A of maximum **number** of compatible activities.
  - – 2 activities are compatible, if their intervals do not overlap.

Example:

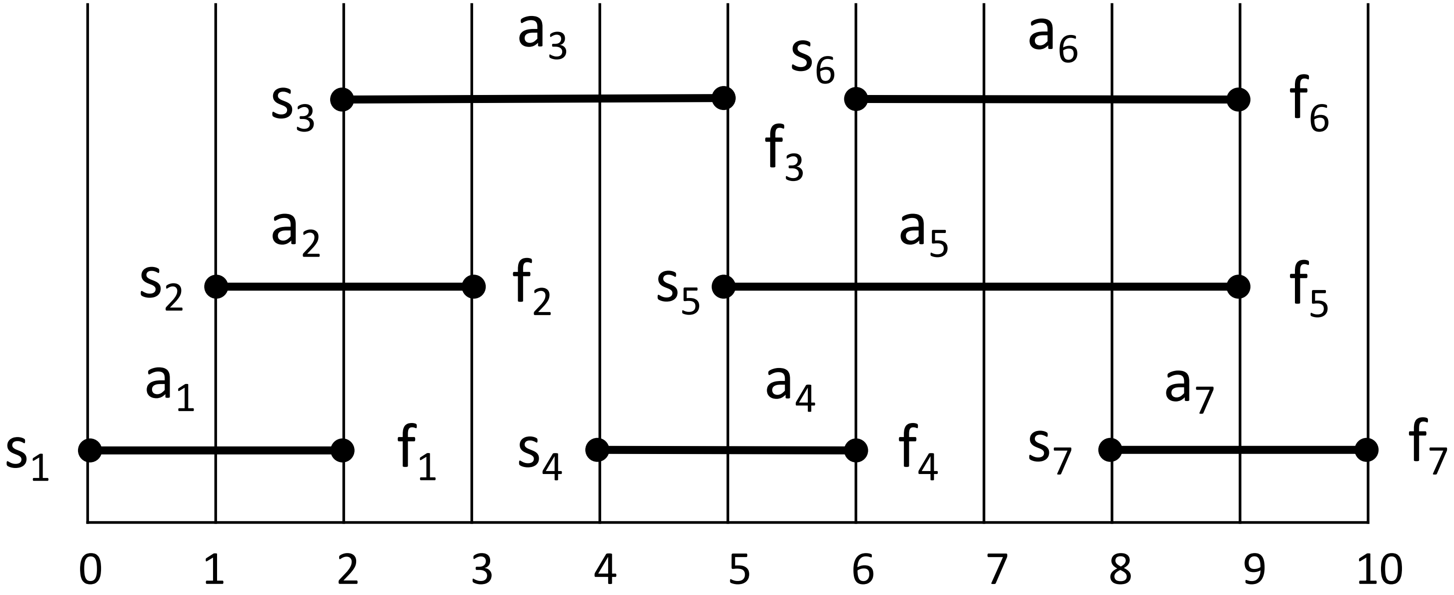
Activities in each line are compatible.



Activity-selection Problem

i	1	2	3	4	5	6	7
s_i	0	1	2	4	5	6	8
f_i	2	3	5	6	9	9	10

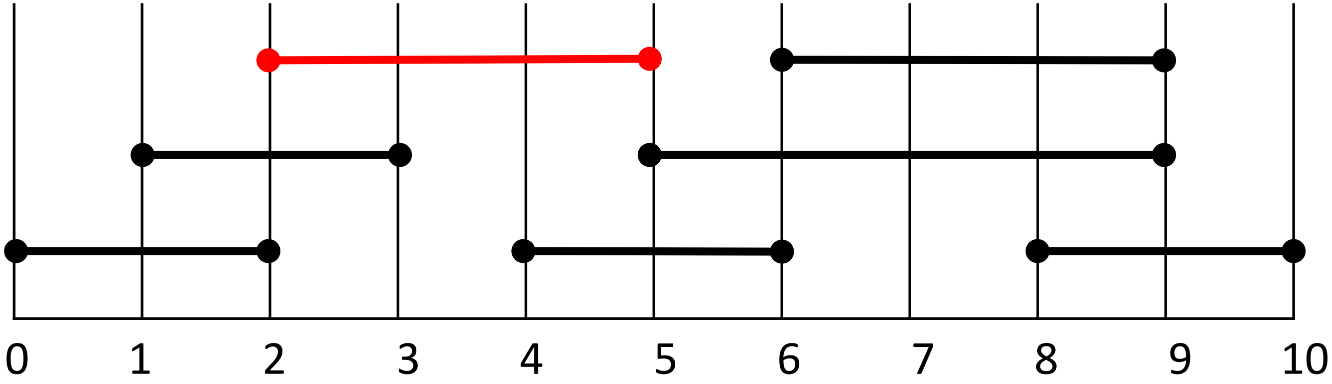
Activities sorted by finishing time.



Optimal compatible set: { a_1 , a_3 , a_5 }

Optimal Substructure

- Assume activities are sorted by finishing times.
- Suppose an optimal solution includes activity a_k. This solution is obtained from:
  - An optimal selection of a_1, …, a_{k-1} activities compatible with one another, and that finish **before** a_k starts.
  - An optimal solution of a_{k+1}, …, a_n activities compatible with one another, and that start **after** a_k finishes.



Optimal Substructure

- Let S_{ij} = subset of activities in S that start after a_i finishes and finish before a_j starts.

$$S_{ij} = \left\{ a_k \in S : \forall i, j \quad f_i \leq s_k < f_k \leq s_j \right\}$$

- A_{ij} = optimal solution to S_{ij}
- $A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj}$

Recursive Solution

- Subproblem: Selecting the maximum number of mutually compatible activities from S_{ij}.
- Let $c[i, j]$ = size of maximum-size subset of mutually compatible activities in S_{ij}.

Recursive solution:

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

Note: We do not know (yet) which k to use for the optimal solution.

Analysis of complexity

	Naïve approach	
# subproblems in optimal solution	2	
# choices to consider	j-i-1	

$A_{ij} = A_{ik} \text{ U } \{ a_k \} \text{ U } A_{kj}$

In other words, we have a linear number of decompositions to process (i.e., the choice of a_k) and each of these choice makes two recursivecalls (exponential growth).

Greedy choice

Theorem:

Let $S_{ij} \neq \emptyset$, and let a_m be the activity in S_{ij} with the earliest finish time $f_m = \min\{ f_k : a_k \in S_{ij}\}$. Then:

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.
2. $S_{im} = \emptyset$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem.

Greedy choice

Proof:

(1) a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.

- Let A_{ij} be a maximum-size subset of mutually compatible activities in S_{ij} (i.e. an optimal solution of S_{ij}).
- Order activities in A_{ij} in monotonically increasing order of finish time, and let a_k be the first activity in A_{ij}.
- If $a_k = a_m \Rightarrow$ done.
- Otherwise, let $A'_{ij} = A_{ij} - \{ a_k \} \cup \{ a_m \}$
- A'_{ij} is valid because a_m finishes before a_k
- Since $|A_{ij}|=|A'_{ij}|$ and A_{ij} maximal $\Rightarrow A'_{ij}$ maximal too.

Greedy choice

Proof:

(2) $S_{im} = \emptyset$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem.

If there is $a_k \in S_{im}$ *then* $f_i \leq s_k < f_k \leq s_m$ $\boldsymbol{< f_m}$ $\Rightarrow f_k < f_m$ which contradicts the hypothesis that a_m has the earliest finishing time.

Greedy choice

	Before theorem	After theorem
# subproblems in optimal solution	2	1
# choices to consider	j-i-1	1

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

$A_{ij} = \{ a_m \} \cup A_{mj}$

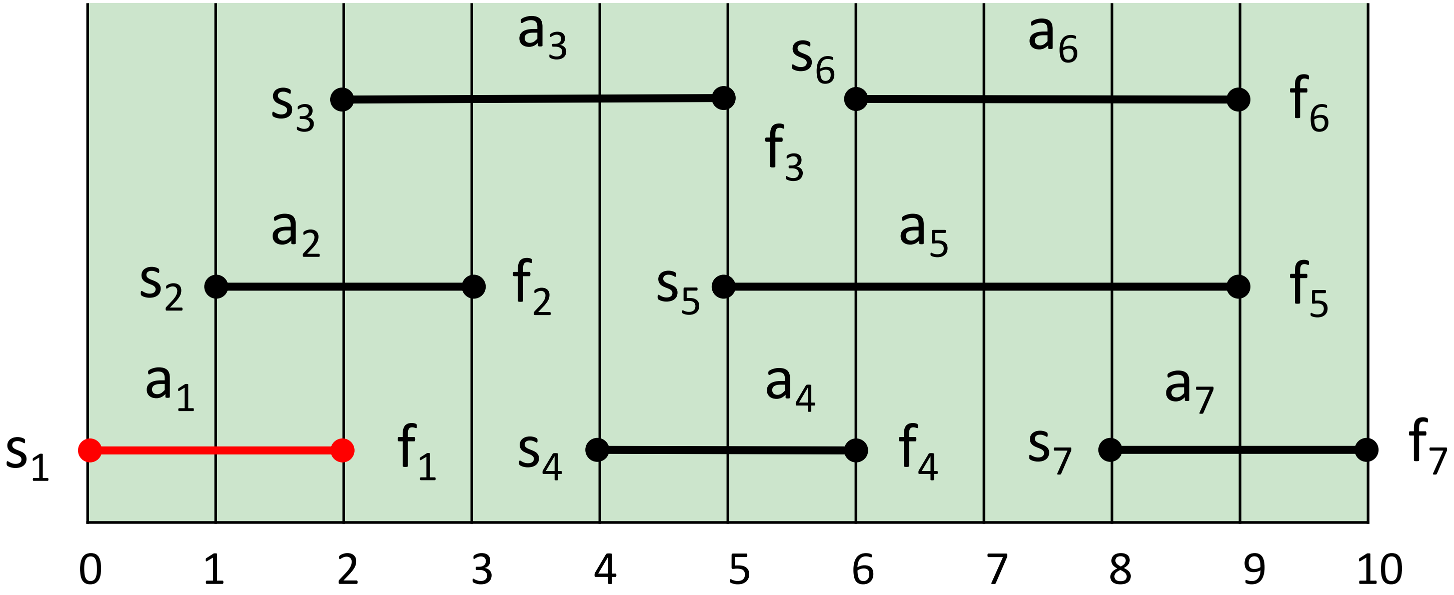
We can now solve the problem S_{ij} top-down:

- Choose $a_m \in S_{ij}$ with the earliest finish time (greedy choice).
- Solve S_{mj}.

Activity-selection Problem

i	1	2	3	4	5	6	7
s_i	0	1	2	4	5	6	8
f_i	2	3	5	6	9	9	10

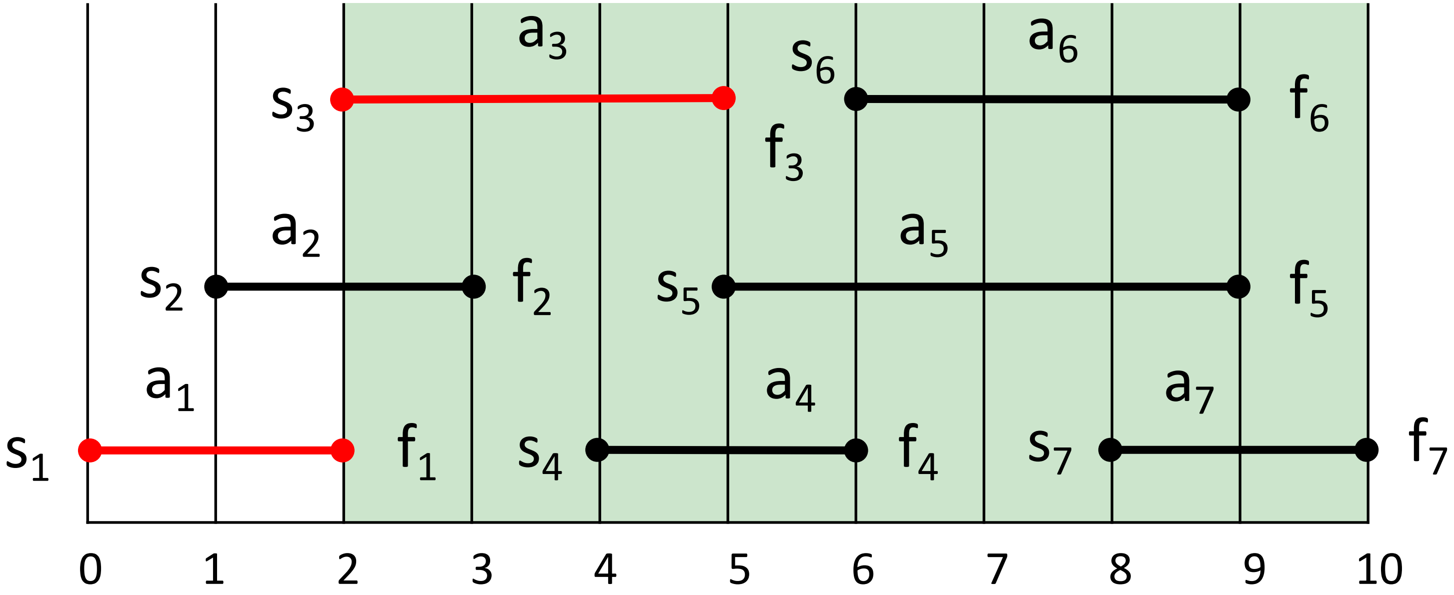
Activities sorted by finishing time.



Activity-selection Problem

i	1	2	3	4	5	6	7
s_i	0	1	2	4	5	6	8
f_i	2	3	5	6	9	9	10

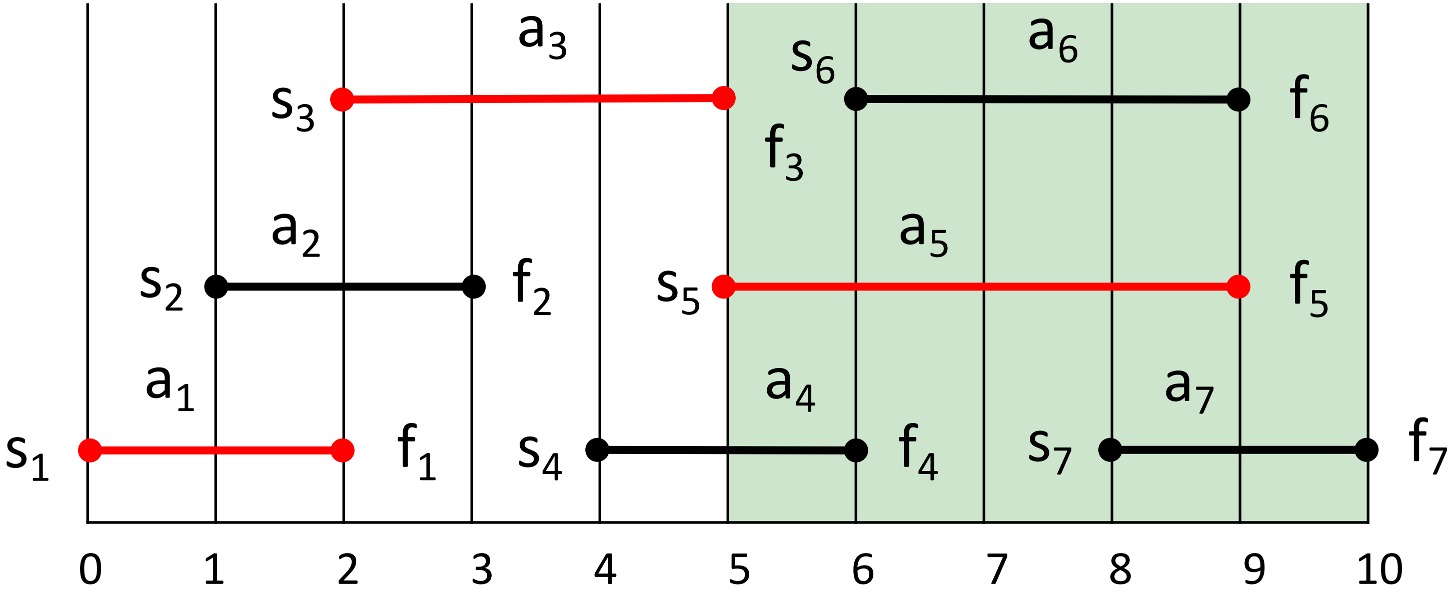
Activities sorted by finishing time.



Activity-selection Problem

i	1	2	3	4	5	6	7
s_i	0	1	2	4	5	6	8
f_i	2	3	5	6	9	9	10

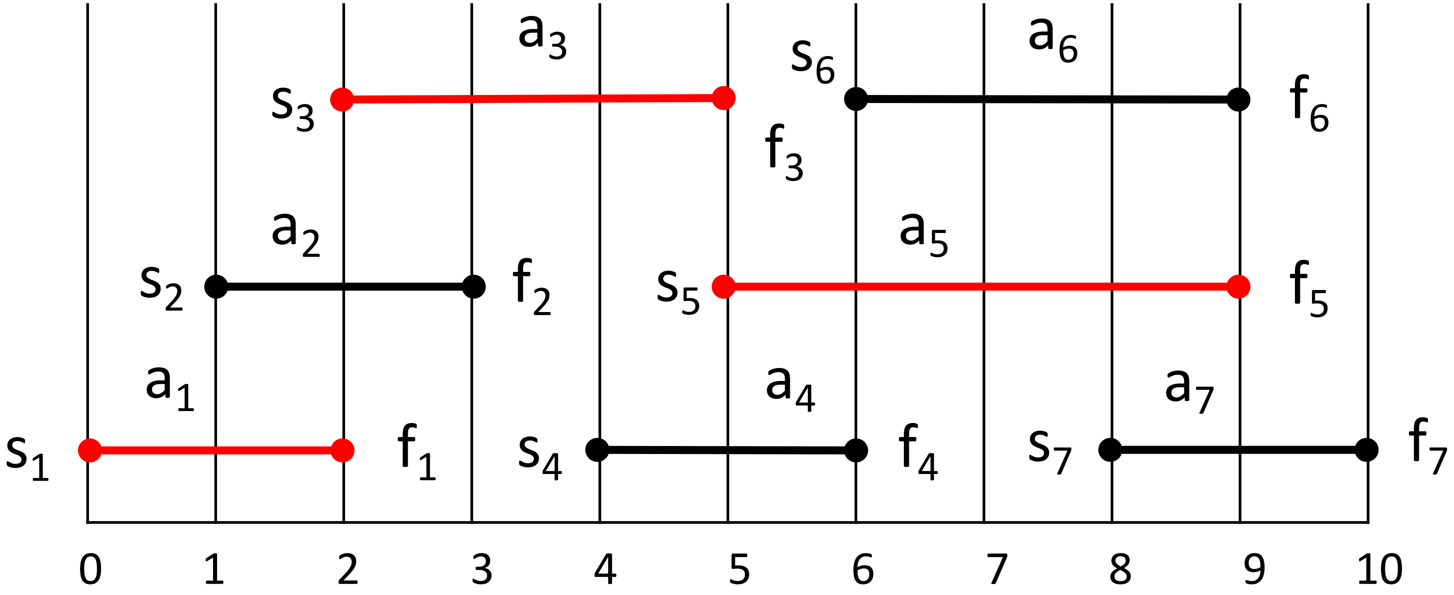
Activities sorted by finishing time.



Activity-selection Problem

i	1	2	3	4	5	6	7
s_i	0	1	2	4	5	6	8
f_i	2	3	5	6	9	9	10

Activities sorted by finishing time.



Recursive Algorithm

```
Recursive-Activity-Selector ($s, f, i, n$)
m ← i+1
while $m \le n$ and $s_m < f_i$        // Find first activity in $S_{i,n+1}$
    do $m \leftarrow m+1$
if $m \le n$
    then return $\{a_m\} \cup$
            Recursive-Activity-Selector($s, f, m, n$)
    else return $\emptyset$
```

Initial Call: Recursive-Activity-Selector (s, f, 0, n+1)

Complexity: $\Theta(n)$

Note 1: We assume activities are already ordered by finishing time.

Note 2: Straightforward to convert the algorithm to an iterative one.

Typical Steps

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- Prove that there is always an optimal solution that makes the greedy choice (greedy choice is safe).
- Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.
- Make the greedy choice and **solve top-down**.
- You may have to preprocess input to put it into greedy order (e.g. sorting activities by finish time).

Elements of Greedy Algorithms

No general way to tell if a greedy algorithm is optimal, but two key ingredients are:

- Greedy-choice Property.
  - We can build a globally optimal solution by making a locally optimal (greedy) choice.
- Optimal Substructure.

Text Compression

- Given a string X, efficiently encode X into a smaller string Y (Saves memory and/or bandwidth)

  A → 0; B → 10; C → 110; D → 1110
  DDCB → 1110 1110 110 10 (13 bits)

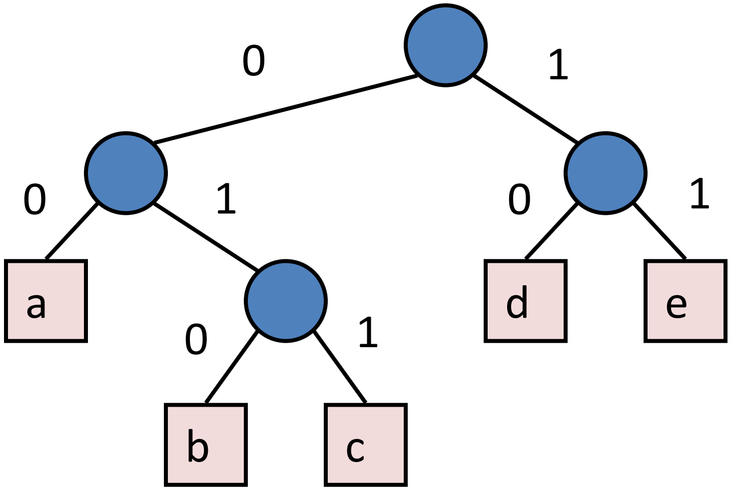
  A → 1110; B → 110; C → 10; D → 0
  DDCB → 0 0 10 110 (7 bits)

- A good approach: **Huffman encoding**
  - Compute frequency f(c) for each character c.
  - Encode high-frequency characters with short code words
  - No code word is a prefix for another code
  - Use an optimal encoding tree to determine the code words

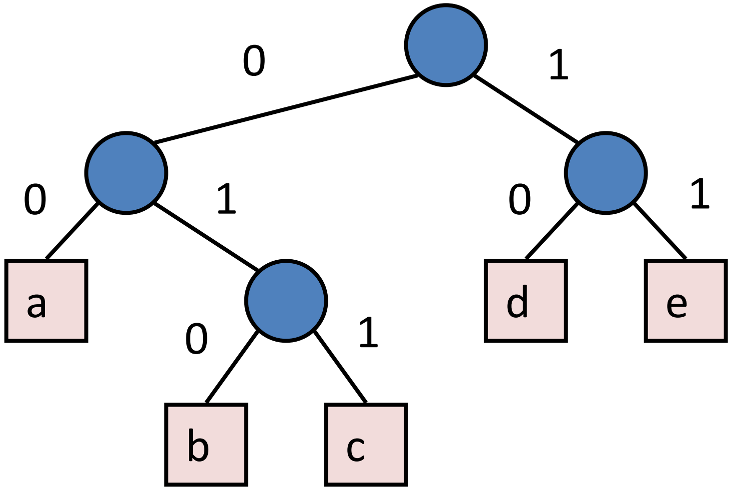
Encoding Tree Example

- A **code** is a mapping of each character of an alphabet to a binary code-word
- A **prefix code** is a binary code such that no code-word is the prefix of another code-word
- An **encoding tree** represents a prefix code
  - Each external node (leaf) stores a character
  - The code word of a character is given by the path from the root to the external node storing the character (0 for a left child and 1 for a right child)

00	010	011	10	11
a	b	c	d	e



Encoding Example



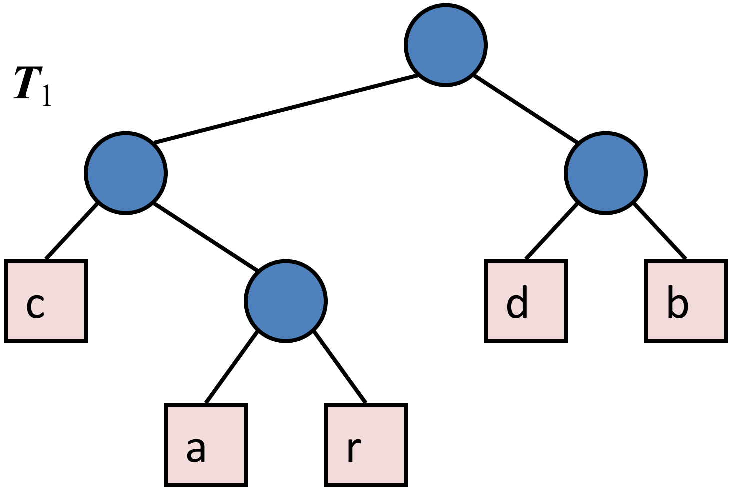
Initial string: X = acda

Encoded string: Y = 00 011 10 00

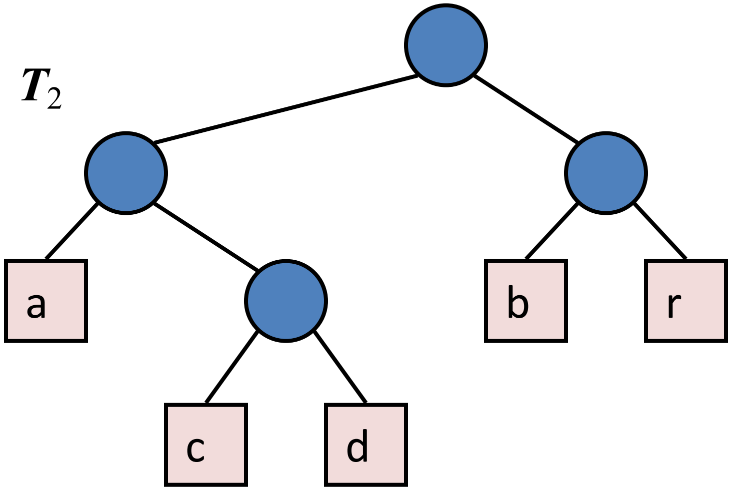
Encoding Tree Optimization

- Given a text string X, we want to find a prefix code for the characters of X that yields a small encoding for X
  - Rare characters should have long code-words
  - Frequent characters should have short code-words
- Example
  - X = abracadabra
  - T_1 encodes X into 29 bits
  - T_2 encodes X into 24 bits

T_1



T_2

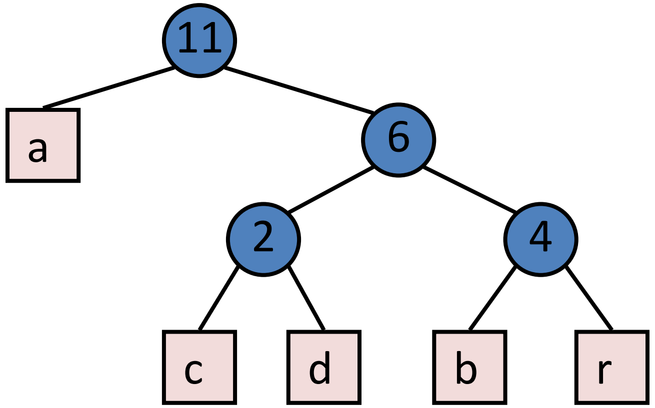


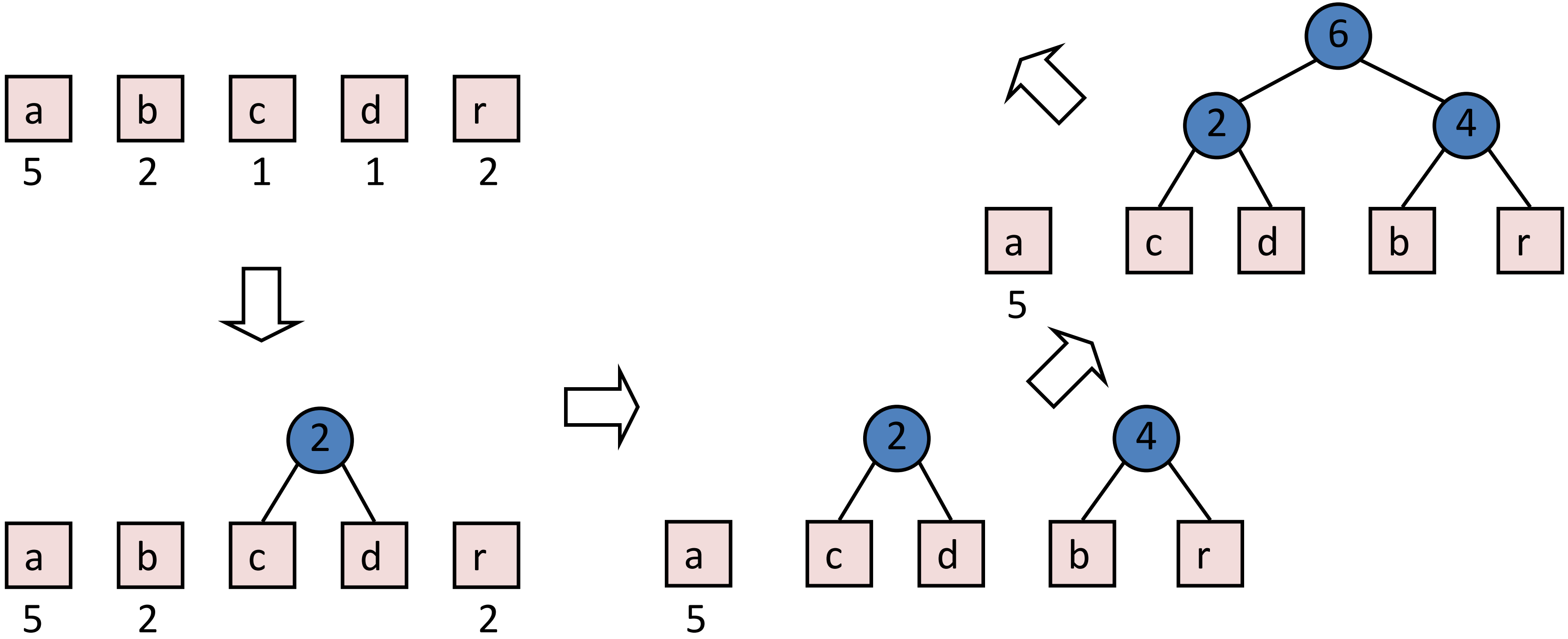
Example

X = abracadabra

Frequencies

a	b	c	d	r
5	2	1	1	2

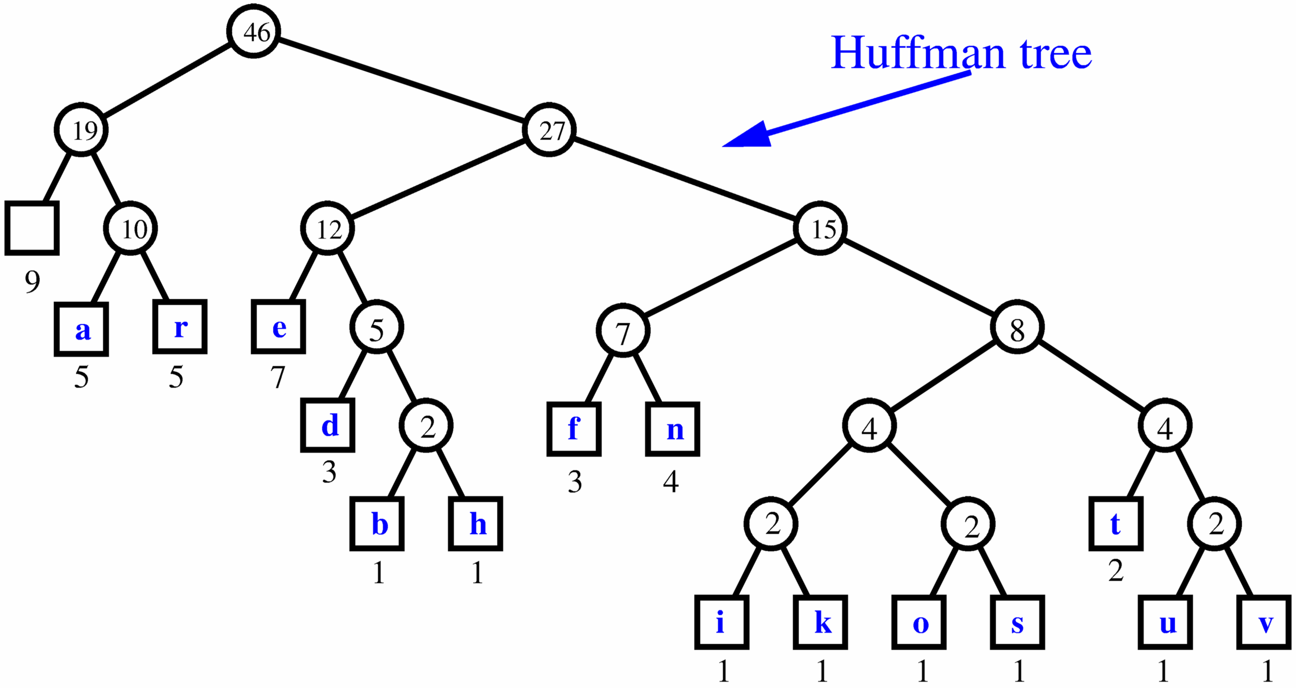




Extended Huffman Tree Example

String: **a fast runner need never be afraid of the dark**

Character		**a**	**b**	**d**	**e**	**f**	**h**	**i**	**k**	**n**	**o**	**r**	**s**	**t**	**u**	**v**
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



Huffman’s Algorithm

- Given a string X, Huffman’s algorithm construct a prefix code the minimizes the size of the encoding of X
- It runs in time $O(n + d \log d)$, where n is the size of X and d is the number of distinct characters of X
- A heap-based priority queue is used as an auxiliary structure

```
Algorithm HuffmanEncoding($X$)
    Input string $X$ of size $n$
    Output optimal encoding trie for $X$
    $C \leftarrow$ distinctCharacters($X$)
    computeFrequencies($C$, $X$)
    $Q \leftarrow$ new empty heap
    for all $c \in C$
        $T \leftarrow$ new single-node tree storing $c$
        $Q$.insert(getFrequency($c$), $T$)
    while $Q$.size() $> 1$
        $f_1 \leftarrow Q$.minKey()
        $T_1 \leftarrow Q$.removeMin()
        $f_2 \leftarrow Q$.minKey()
        $T_2 \leftarrow Q$.removeMin()
        $T \leftarrow$ join($T_1$, $T_2$)
        $Q$.insert($f_1 + f_2$, $T$)
    return $Q$.removeMin()
```