

# Analysis of Algorithms

## GREEDY ALGORITHMS

## Greedy Algorithms

- ★ Greedy algorithms make decisions that **seem** to be the **best** at the time.
- ★ Usually, there is **preprocessing** before decisions are made.
- ★ Usually, there exist greedy **criteria**.
- ★ In **Real-Time** and **On-Line** problems:
  - The **present** cannot change the **past**.
  - The **present** cannot rely on the un-known **future**.

## How and When to use Greedy Algorithms?

- ★ Establish **trivial** solutions for a problem of a small size; usually  $n = 0$  or  $n = 1$ .
- ★ For a problem of size  $n$ , look for a greedy decision that **reduces** the size of the problem to some  $k < n$ .
- ★ Then, apply **recursion**.

# The Coin Changing Problem

## Input:

- ★ Integer coin denominations  $d_n > \dots > d_2 > d_1 = 1$ .
- ★ An integer amount to pay:  $A$ .

**Output:** Number of coins  $n_i$  for each denomination  $d_i$  to get the exact amount.

- ★  $A = n_n d_n + n_{n-1} d_{n-1} + n_2 d_2 + n_1 d_1$ .

**Goal:** Minimize total number of coins.

- ★  $\mathcal{N} = n_n + \dots + n_2 + n_1$ .

**Remark:** There is always a solution with  $\mathcal{N} = A$  since  $d_1 = 1$ .

## Examples

- ★ **USA:**  $d_6 = 100, d_5 = 50, d_4 = 25, d_3 = 10, d_2 = 5, d_1 = 1$ .
  - $\mathcal{A} = 73 = 2 \cdot 25 + 2 \cdot 10 + 3 \cdot 1$ .
  - $\mathcal{N} = 2 + 2 + 3 = 7$ .
- ★ **Old British:**  $d_3 = 240, d_2 = 20, d_1 = 1$ .
  - $\mathcal{A} = 307 = 1 \cdot 240 + 3 \cdot 20 + 7 \cdot 1$ .
  - $\mathcal{N} = 1 + 3 + 7 = 11$ .

## Greedy Solution

**Idea:** Use the largest possible denomination and update  $\mathcal{A}$ .

**Implementation:**

**Coin-Changing**( $d_n > \dots > d_2 > d_1 = 1$ )

**for**  $i = n$  **downto** 1

$n_i = \lfloor \mathcal{A}/d_i \rfloor$

$\mathcal{A} = \mathcal{A} \bmod n_i$

**Return**( $\mathcal{N} = n_n + \dots + n_2 + n_1$ )

**Correctness:**  $\mathcal{A} = n_n d_n + n_{n-1} d_{n-1} + n_2 d_2 + n_1 d_1$ .

**Complexity:**  $\Theta(n)$  division and mod integer operations.

## Optimality

- ★ Greedy is optimal for the USA system.
- ★ A coin system for which Greedy is not optimal:
  - $d_3 = 4, d_2 = 3, d_1 = 1$  and  $\mathcal{A} = 6$ :
  - Greedy:  $6 = 1 \cdot 4 + 2 \cdot 1 \Rightarrow \mathcal{N} = 3$ .
  - Optimal:  $6 = 2 \cdot 3 \Rightarrow \mathcal{N} = 2$ .
- ★ A coin system for which Greedy is very “bad”:
  - $d_3 = x + 1, d_2 = x, d_1 = 1$  and  $\mathcal{A} = 2x$ :
  - Greedy:  $2x = 1 \cdot (x + 1) + (x - 1) \cdot 1 \Rightarrow \mathcal{N} = x$ .
  - Optimal:  $2x = 2 \cdot x \Rightarrow \mathcal{N} = 2$ .

## Efficiency

**Optimal solution:** Check **all** possible combinations.

- ★ Not a **polynomial** time algorithm.

**Another optimal solution:** Polynomial in both  $n$  and  $\mathcal{A}$ .

- ★ Not a **strongly polynomial** time algorithm.

**Objective:**

- ★ Find a solution that is polynomial only in  $n$ .
- ★ **Probably** impossible!

# The Knapsack Problem

## Input:

- ★ A thief enters a store and finds  $n$  items  $I_1, \dots, I_n$ .
- ★ The value of item  $I_i$  is  $v(I_i)$  and its weight is  $w(I_i)$ .
  - Both are positive integers.
- ★ The thief can carry at most weight  $W$ .
- ★ The thief either takes all of item  $I_i$  or not.

**Goal:** Carry items with maximum total value.

- ★ Which are these items?
- ★ What is their total value?

## A General Greedy Scheme

- ★ Order the items according to some greedy criterion.
- ★ If weight of item  $I_i$  from the top of the list is " $\leq W$ ":
  - Take item  $I_i$ .
  - Continue recursively with  $I_1, \dots, I_{i-1}, I_{i+1}, \dots, I_n$  and new maximum weight  $W - w(I_i)$ .
- ★ Otherwise:
  - Ignore item  $I_i$ .
  - Continue recursively with  $I_1, \dots, I_{i-1}, I_{i+1}, \dots, I_n$  and maximum weight  $W$ .

## A General Greedy Scheme – Implementation

**Non-Recursive Knapsack**( $I_1, \dots, I_n, w(\cdot), v(\cdot), W$ )

**Let**  $J_1, \dots, J_n$  be the new order on the items.

$S = \emptyset$  (\* the set of items the thief takes \*)

$V = 0$  (\* the value of these items \*)

**for**  $i = 1$  **to**  $n$

**if**  $w(J_i) \leq W$  **then**

$S = S \cup \{J_i\}$

$V = V + v(J_i)$

$W = W - w(J_i)$

**Return**( $S, V$ )

## Greedy Criteria

### Greedy criterion I

- ★ Order the items by their **value**:
  - from the **most expensive** to the **cheapest**.

### Greedy criterion II

- ★ Order the items by their **weight**:
  - from the **lightest** to the **heaviest**.

### Greedy criterion III

- ★ Order the items by their **ratio** of value over weight:
  - from the **largest** ratio to the **smallest** ratio.

## The three criteria are not optimal

### Counter example for criteria I and III:

- ★ Three items  $I_1, I_2, I_3$  and  $W = 10$ .
  - Weights and values are:  $\langle 6, 10 \rangle, \langle 5, 6 \rangle, \langle 5, 6 \rangle$ .
- ★ **Optimal** takes items  $I_2$  and  $I_3$  for a profit of 12.
- ★ **Greedy I** or **Greedy III** take only item  $I_1$  for a profit of 10.

### Counter example for criterion II:

- ★ The value of  $I_1$  is 13. The rest is as before.
- ★ **Optimal** takes only item  $I_1$  for a profit of 13.
- ★ **Greedy II** that takes items  $I_2$  and  $I_3$  for a profit of 12.

## The Fractional Knapsack Problem

- ★ The thief can take **portions** of items.
- ★ If the thief takes a **fraction**  $0 \leq p_i \leq 1$  of item  $i$ :
  - Its value is  $p_i v_i$ .
  - Its weight is  $p_i w_i$ .

**Theorem:** Greedy that uses **Criterion III** is **optimal**.

## Proof

- ★ Assume that **Greedy** fails on the input  $I_1, \dots, I_n$  and the weight  $W$ .
- ★ Let the portions taken by **Optimal** be  $p_1, \dots, p_n$ .
  - $p_i = 1$ : **all** of item  $I_i$  is taken.
  - $p_i = 0$ : **none** of item  $I_i$  is taken.
  - $0 < p_i < 1$ : **some** but **not all** of item  $I_i$  is taken.
- ★ Since **Greedy** fails, there exist  $I_i$  and  $I_j$  such that:
  - $\frac{v(I_i)}{w(I_i)} > \frac{v(I_j)}{w(I_j)}$  and  $p_i < 1$  and  $p_j > 0$ .
- ★ But, it is **more** profitable to take **more** of item  $I_i$  and **less** of item  $I_j$ .
- ★ Because each **unit of weight** of item  $I_i$  has **more** value than each **unit of weight** of item  $I_j$ .
- ★ A **contradiction** to the optimality of **Optimal**.

## The 0 – 1 Knapsack Problem

**Optimal solution:** Check **all** possible sets of items.

- ★ Not a **polynomial** time algorithm.

**Another optimal solution:** Polynomial in both  $n$  and  $W$ .

- ★ Not a **strongly polynomial** time algorithm.

**Objective:**

- ★ Find a solution that is polynomial only in  $n$ .
- ★ **Probably** impossible!
- ★ However, **Greedy** produces “**good**” solutions.

# The Activity-Selection Problem

## Input:

- ★ Activities  $A_1, \dots, A_n$  that need the service of a common resource.
- ★ Activity  $A_i$  is associated with a time interval  $[s_i, f_i)$  for  $s_i < f_i$ .
  - $A_i$  needs the service from time  $s_i$  until just before time  $f_i$ .

**Mutual Exclusion:** The resource serves at most one activity at any time.

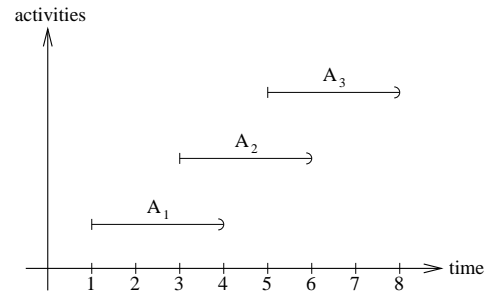
**Definition:**  $A_i$  and  $A_j$  are compatible if either  $f_i \leq s_j$  or  $f_j \leq s_i$ .

**Goal:** Find a maximum size set of compatible activities.

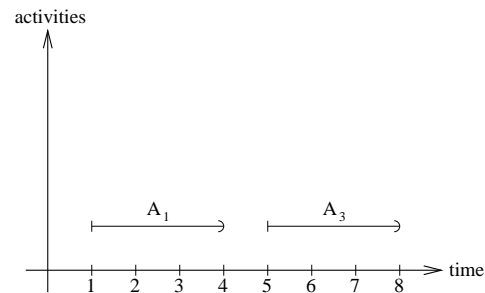
## Example

**Input:** 3 activities  $A_1 = [1, 4)$ ,  $A_2 = [3, 6)$ ,  $A_3 = [5, 8)$ .

**A graphical representation:**



**The best solution:**



## Static vs. Dynamic

**Static:** The **greedy** criterion is determined in advance and cannot be changed during the execution of the algorithm.

**Dynamic:** The **greedy** criterion may change during the execution of the algorithm based on some decisions.

**Remark:** A **static** criterion is also a **dynamic** criterion.

## A General Static Greedy Scheme

- ★ **Maintain** a set  $\mathcal{S}$  of the activities that have been selected so far.
- ★ Initially,  $\mathcal{S} = \emptyset$  and at the end,  $\mathcal{S}$  is an **optimal** solution.
- ★ **Order** the activities following some greedy criterion and **consider** the activities according to this order.
- ★ Let  $A$  be the current considered activity. **If**  $A$  is compatible with all the activities in  $\mathcal{S}$ :
  - **Then add**  $A$  to  $\mathcal{S}$ .
  - **Else ignore**  $A$ .
- ★ **Continue** until there are no activities to consider

## A General Dynamic Greedy Scheme

- ★ **Maintain** two sets of activities:
  - $S$  those that have been selected so far.
  - $\mathcal{R}$  those that can still be selected.
  - Initially,  $S = \emptyset$  and  $\mathcal{R} = \{A_1, \dots, A_n\}$ .
  - At the end,  $S$  is an **optimal** solution and  $R = \emptyset$ .
- ★ **Select** a “good” activity  $A$  from  $\mathcal{R}$ , following some greedy criterion.
- ★ **Add**  $A$  to  $S$ .
- ★ **Throw** from  $\mathcal{R}$  all the activities that are not compatible with activity  $A$ .
- ★ **Continue** until  $\mathcal{R}$  is empty.

## Greedy Criteria

### Three criteria:

- Prefer **short** activities.
- Prefer activities intersecting **few** other activities.
- Prefer activities that start **earlier**.
- Prefer activities that terminate **earlier**.

**Optimality:** Only the **fourth** criterion is **optimal**.

### Remarks:

- All four criteria are **static** in their nature.
- The second criterion can be implemented to be **dynamic**.

## An Optimal Greedy Solution

**Preprocessing**( $A_1, \dots, A_n$ )

**Sort** the activities according to their finish time

**Let** this order be  $A_1, \dots, A_n$  (\*  $i < j \Rightarrow f_i \leq f_j$  \*)

**Greedy-Activity-Selector**( $A_1, \dots, A_n$ )

$\mathcal{S} = \{A_1\}$  (\*  $A_1$  terminates the earliest \*)

$j = 1$  (\*  $A_j$  is the current selected activity \*)

**for**  $i = 2$  **to**  $n$  (\* scan all the activities \*)

**if**  $s_i \geq f_j$  (\* check compatibility \*)

**then** (\* select  $A_i$  that is compatible with  $\mathcal{S}$  \*)

$\mathcal{S} = \mathcal{S} \cup \{A_i\}$

$j = i$

**else** (\*  $A_i$  is not compatible \*)

**Return**( $\mathcal{S}$ )

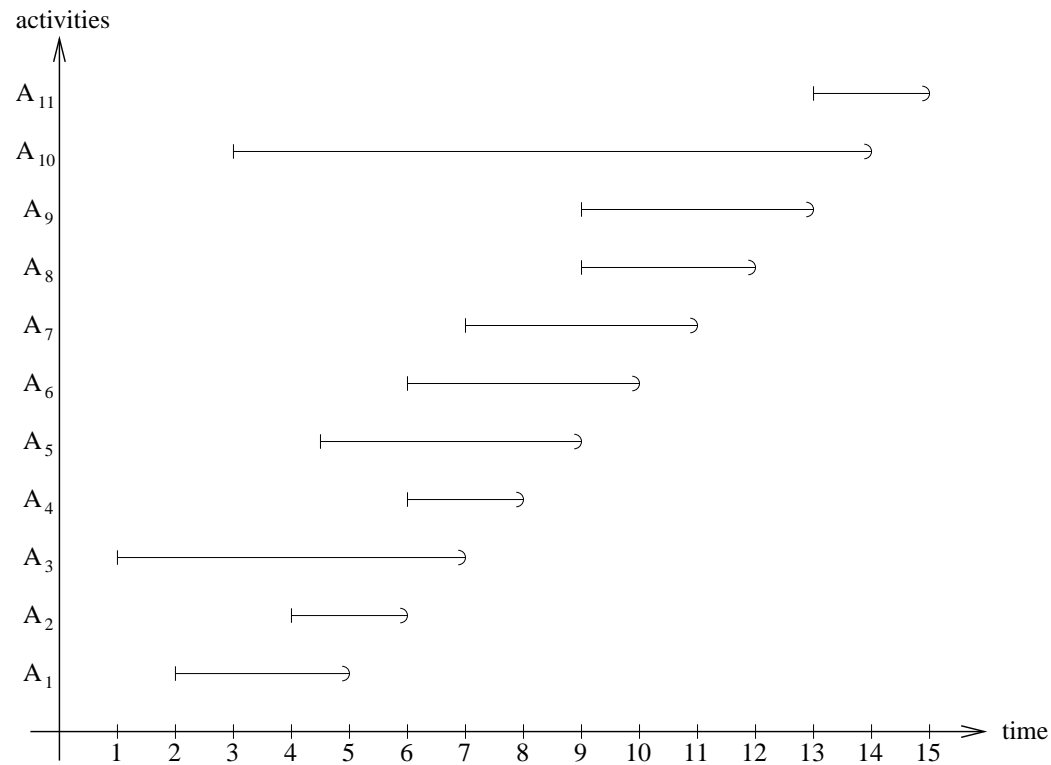
## Correctness and Complexity

**Correctness:** By definition.

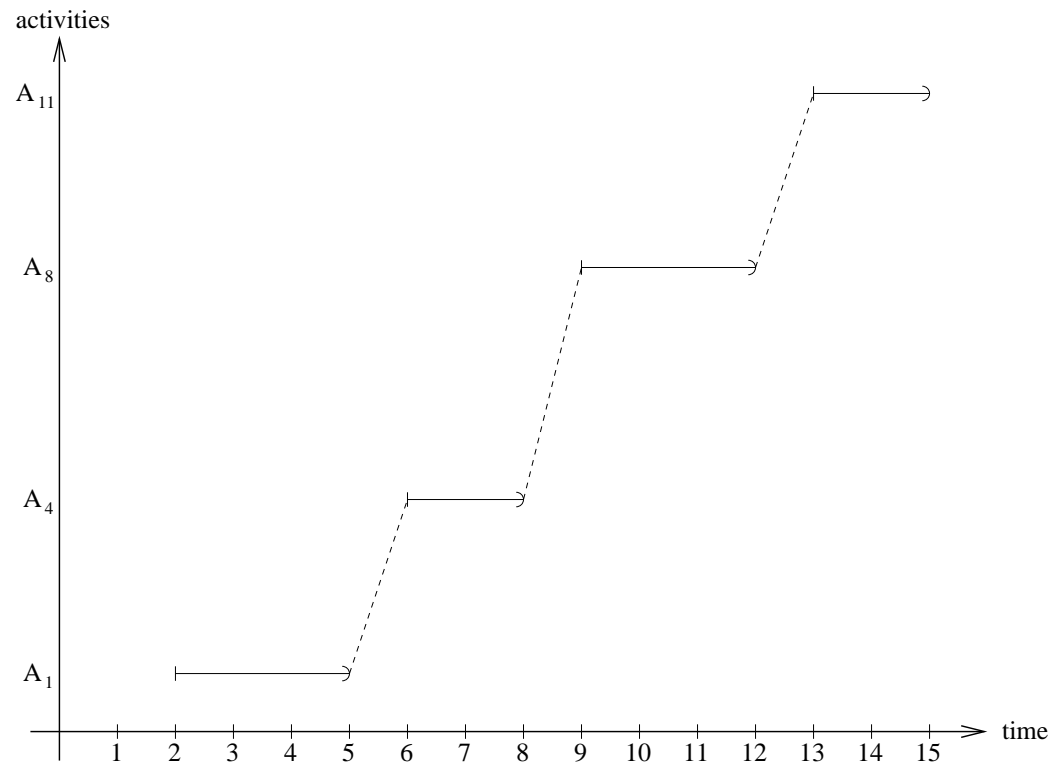
**Complexity:**

- ★ The sorting can be done in  $O(n \log n)$  time.
- ★ There are  $O(1)$  operations per each activity.
- ★ All together:  $O(n \log n) + n \cdot O(1) = O(n \log n)$  time.

# Example - Input



## Example - Output



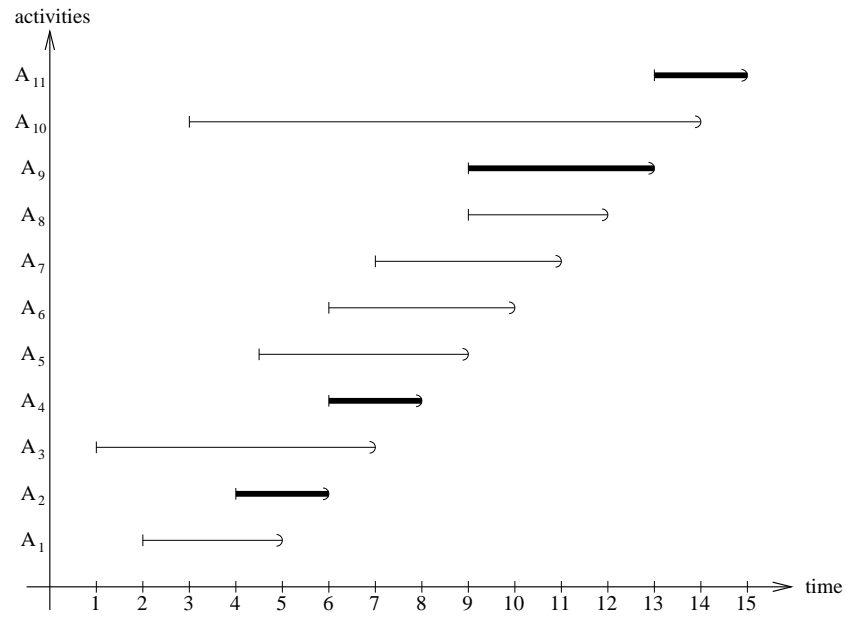
## Optimality

- ★ Let  $\mathcal{T}$  be an **optimal** set of activities.
- ★ **Transform**  $\mathcal{T}$  to  $\mathcal{S}$  **preserving** the size of  $\mathcal{T}$ .
- ★ Let  $A_1, \dots, A_n$  be ordered by their **finish** time.
- ★ Let  $A_i$  be the **first** activity that is in  $\mathcal{T}$  and not in  $\mathcal{S}$ .
- ★ All the activities in  $\mathcal{T}$  that finish **before**  $A_i$  are also in  $\mathcal{S}$ .

## Optimality

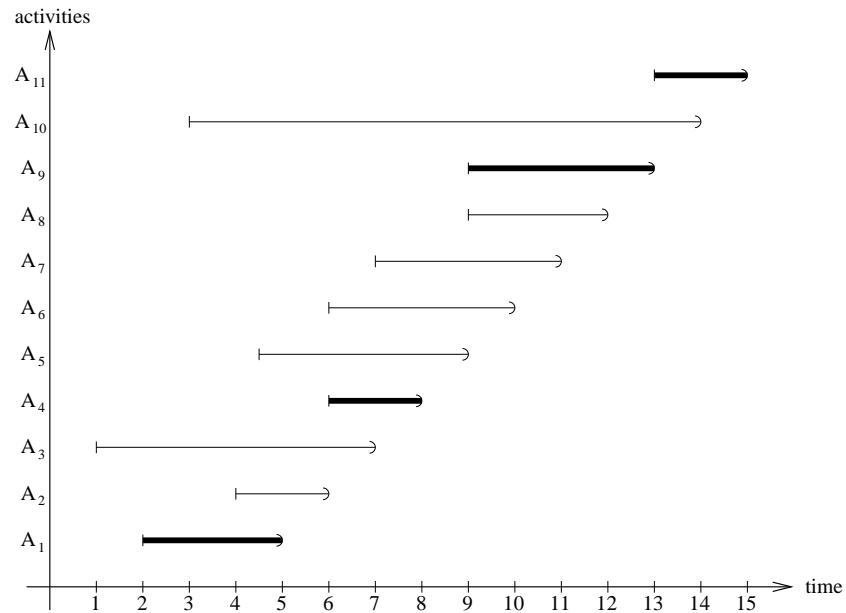
- ★  $A_i \notin \mathcal{S} \Rightarrow \exists A_j \in \mathcal{S}$  that is not in  $\mathcal{T}$  in which  $j < i$ .
- ★  $A_j$  is compatible with all the activities in  $\mathcal{T}$  that finish **before** it since they are all in  $\mathcal{S}$ .
- ★  $A_j$  is compatible with all the activities in  $\mathcal{T}$  that finish **after**  $A_i$  since it finishes before  $A_i$ .
- ★ Therefore,  $\mathcal{T} \cup \{A_j\} \setminus \{A_i\}$  is a solution with the **same** size as  $\mathcal{T}$  and hence **optimal**.
- ★ **Continue** this way until  $\mathcal{T}$  **becomes**  $\mathcal{S}$ .

# Example



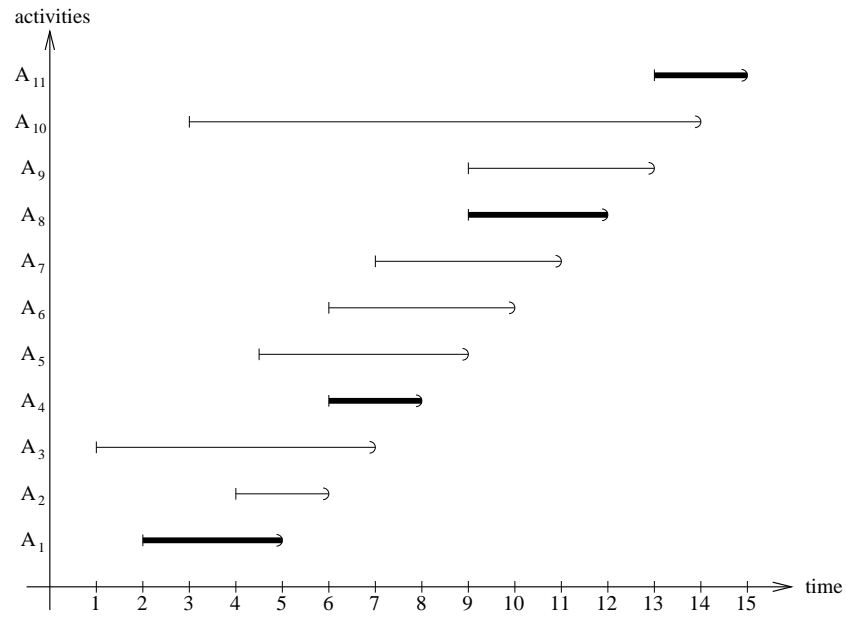
Another optimal solution with 4 activities.

## Example



A **third** optimal solution: after the **first** transformation.

# Example



The **greedy** solution: after the **second** transformation.

# Huffman Codes

## Input:

- ★ An alphabet of  $n$  **symbols**  $a_1, \dots, a_n$ .
- ★ A **frequency**  $f_i$  for each symbol  $a_i$ :
  - $\sum_{i=1}^n f_i = 1$ .
- ★ A File  $\mathcal{F}$  containing  $L$  symbols from the alphabet.
  - $a_i$  appears **exactly**  $n_i = f_i \cdot L$  times in  $\mathcal{F}$ .

## Output:

- ★ For symbol  $a_i$ ,  $1 \leq i \leq n$ :
  - A binary **codeword**  $w_i$  of length  $\ell_i$ .
- ★ A **compressed (encoded)** binary file  $\mathcal{F}'$  of  $\mathcal{F}$ .

## Huffman Codes – Goal

- ★  $L'$  the length of  $\mathcal{F}'$  should be **minimal**.
- ★ An **efficient** algorithm to find the  $n$  codewords.
  - **Good** polynomial running time ( $O(n \log n)$ ).
- ★ Efficient **encoding** and **decoding** of the file
  - Should be done in  $O(B)$ -time.
  - $B$  is the size of the original file in **bits**.

## Example

- ★ A file with the alphabet  $a, b, c, d, e, f$  containing 100 symbols.
  - $n_a = 45, n_b = 13, n_c = 12, n_d = 16, n_e = 9, n_f = 5$ .
- ★ Code  $I$ :
  - $w_a = 000, w_b = 001, w_c = 010, w_d = 011, w_e = 100, w_f = 101$ .
  - Length of **encoded** file is 300.
- ★ Code  $II$ :
  - $w_a = 0, w_b = 101, w_c = 100, w_d = 111, w_e = 1101, w_f = 1100$ .
  - Length of **encoded** file is  $1 \cdot 45 + 3 \cdot 13 + 3 \cdot 12 + 3 \cdot 16 + 4 \cdot 9 + 4 \cdot 5 = 224$ .
- ★ Code  $II$  is **optimal**,  $\approx 25\%$  **better** than code  $I$ .

## Prefix Free Codes

**Definition:** A **prefix free** code is a code in which no codeword is a prefix of another codeword.

**Examples:** Both code  $I$  and code  $II$  are prefix free.

**Proposition:** A code in which the lengths of all the codewords is the **same** is a prefix free code.

**Theorem:** **Always** exists an **optimal** prefix free code.

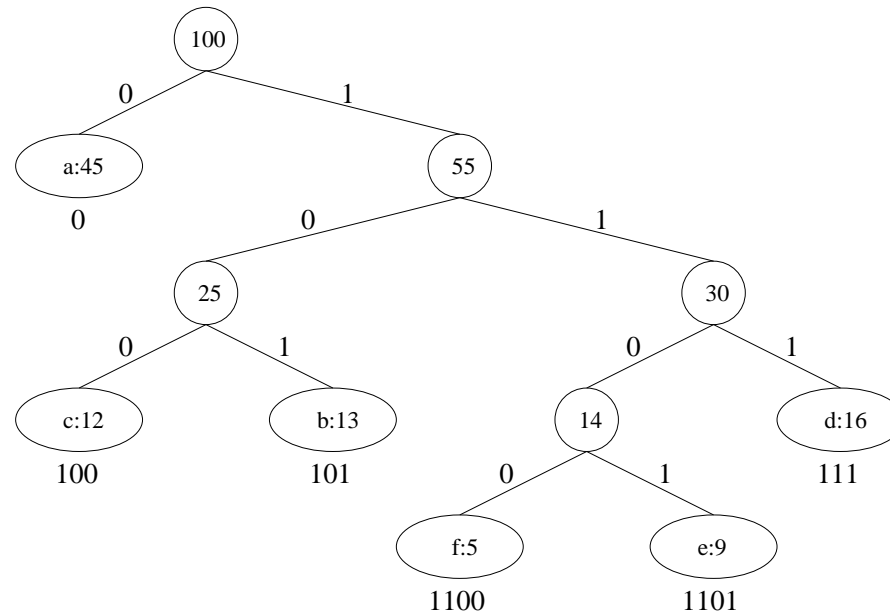
**Encoding:** “**Easy**”. Using tables.

**Decoding:** By **scanning** the coded text once.

## Binary Tree Representation for Prefix Free Codes

- ★ A code can be **represented** by a rooted and ordered binary tree with  $n$  leaves.
- ★ Each leaf **stores** a codeword.
- ★ The codeword corresponding to a leaf is defined by the **unique path** from the root to the leaf:
  - 0 for going **left**.
  - 1 for going **right**.

## Example: Code *II*



- ★ A **leaf** is represented by the symbol and its frequency.
- ★ An **internal node** is labelled by the sum of the frequencies of all the leaves in its subtree.

## Binary Tree Representation

**Proposition:** The binary tree represents a **prefix free code** since a path to a leaf **cannot** be a prefix of any other path.

### Complexity:

- ★  $f(x)$  the frequency of a leaf  $x$ .
- ★  $\ell(x)$  the length of the path from the root to  $x$ .
- ★ The **cost** of the tree is:  $B(T) = \sum_{\text{a leaf } x} (f(x) \cdot \ell(x))$ .
  - $B(T)$  is the **average** length of a codeword.
- ★ The **length** of the encoded file:  $\sum_{\text{a leaf } x} (n(x) \cdot \ell(x))$ .

## A Structural Claim

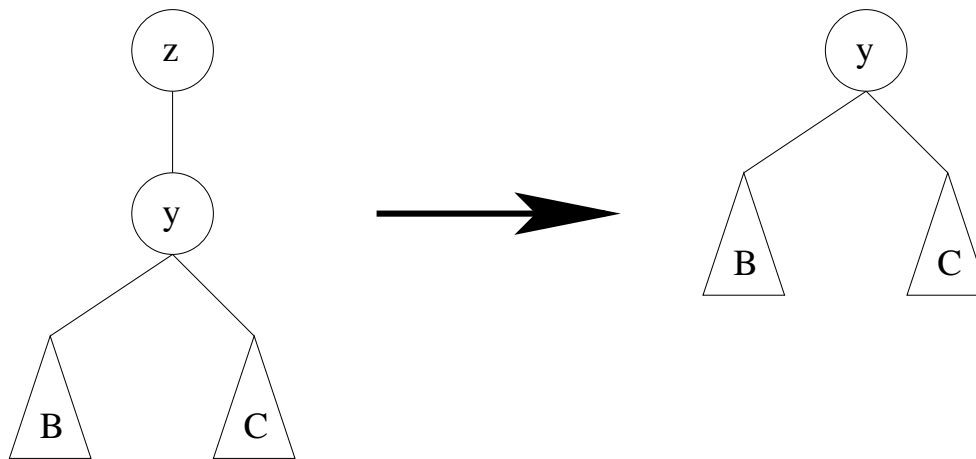
**Lemma:** Let  $T$  be a tree that represents an **optimal** code.  
Then each **internal** node in the tree has **two** children.

### Proof:

- ★ Let  $z$  be an internal node with only one child  $y$ .
- ★ There are 2 cases:
  - **Case I:**  $z$  is the root.
  - **Case II:**  $z$  is not the root.

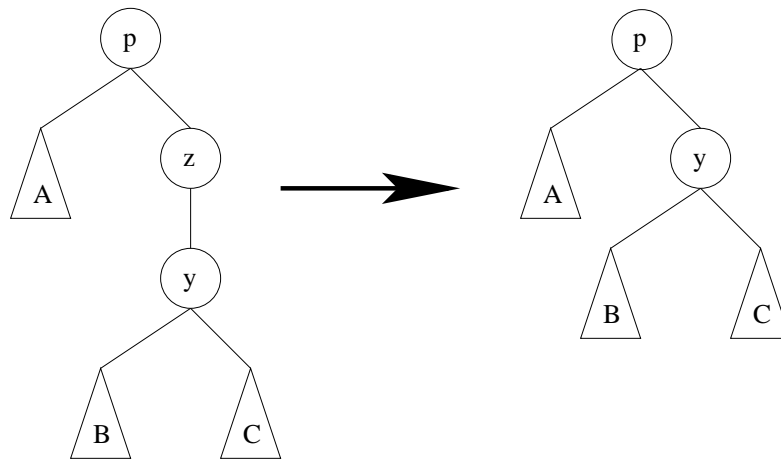
## Case I

- ★  $z$  is the root:
  - **Make**  $y$  the new root.



## Case II

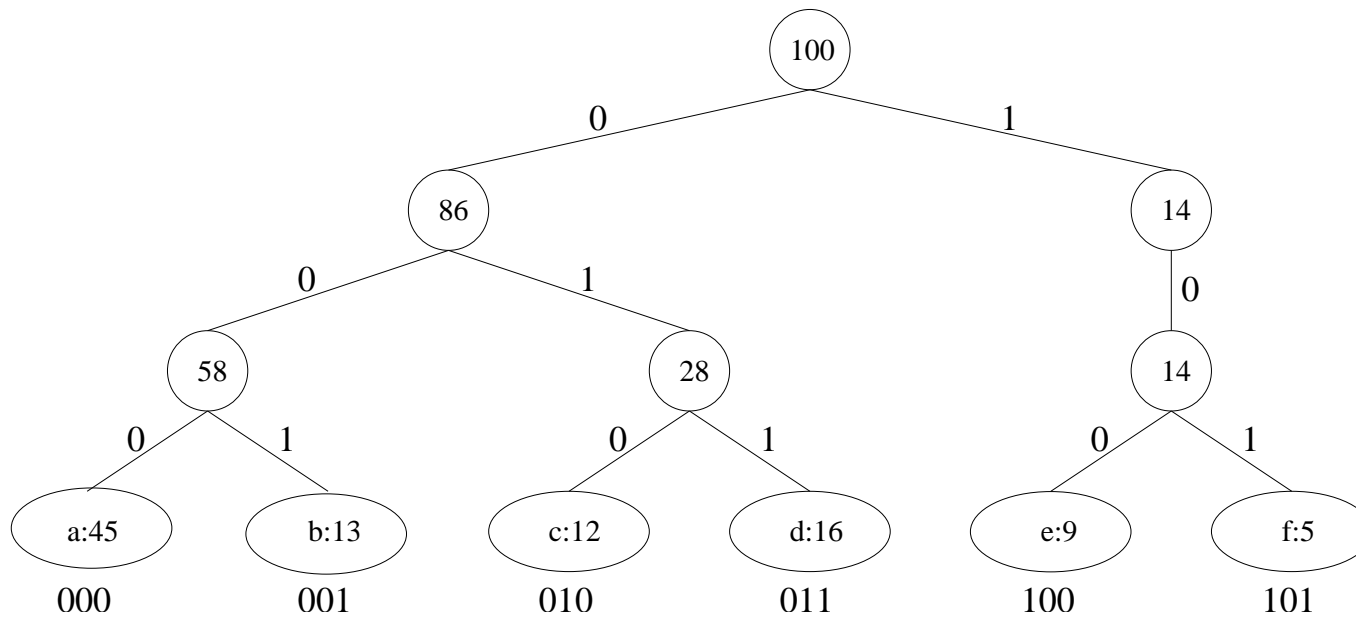
- ★  $z$  is not a root and  $p$  is its parent:
  - **Bypass**  $z$  by making  $y$  the child of  $p$ .



## Proof

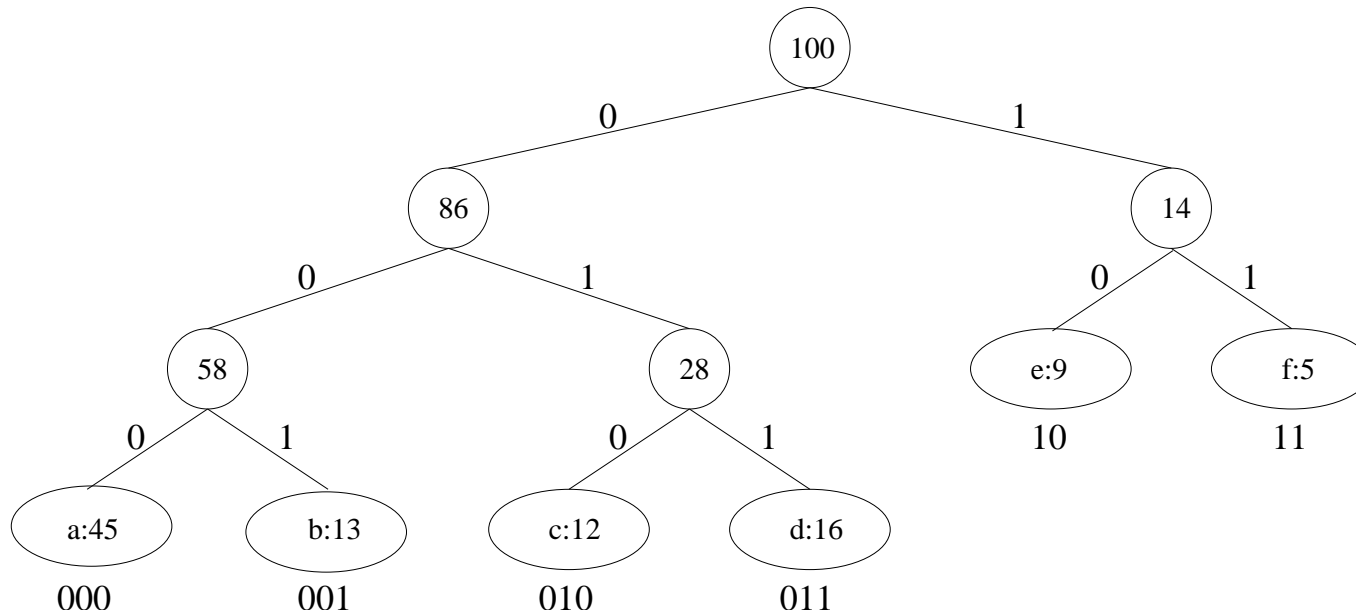
- ★ In both cases:
  - $\ell(x)$  of **all** the leaves in the sub-tree rooted at  $z$  is **reduced** by 1.
  - These are the **only** changes.
  - As a result the cost of the tree is **improved**.
  - A **contradiction** to the optimality of the code.

## Example: Code *I*



$$B(T) = 300$$

## Example: Improving Code *I*

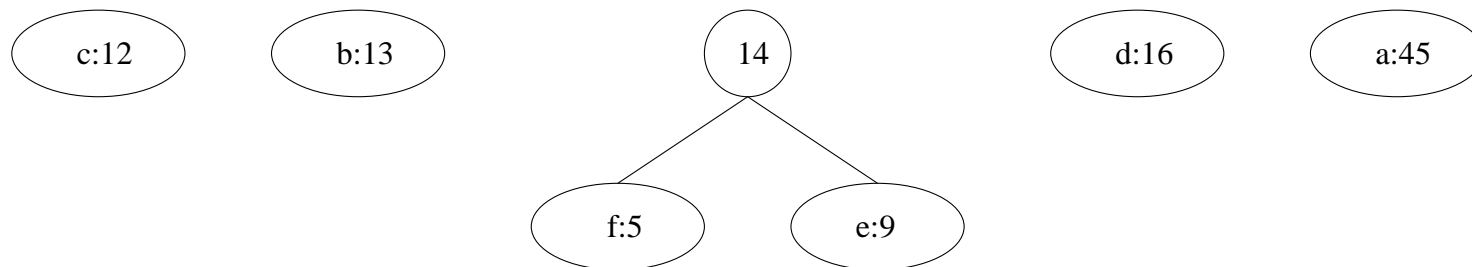
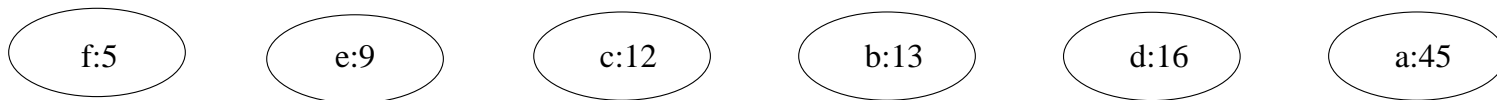


$$B(T) = 3 \cdot 86 + 2 \cdot 14 = 286$$

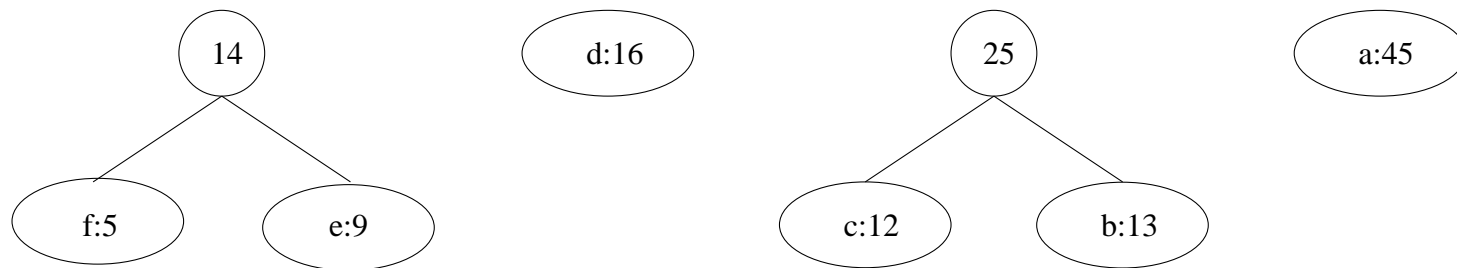
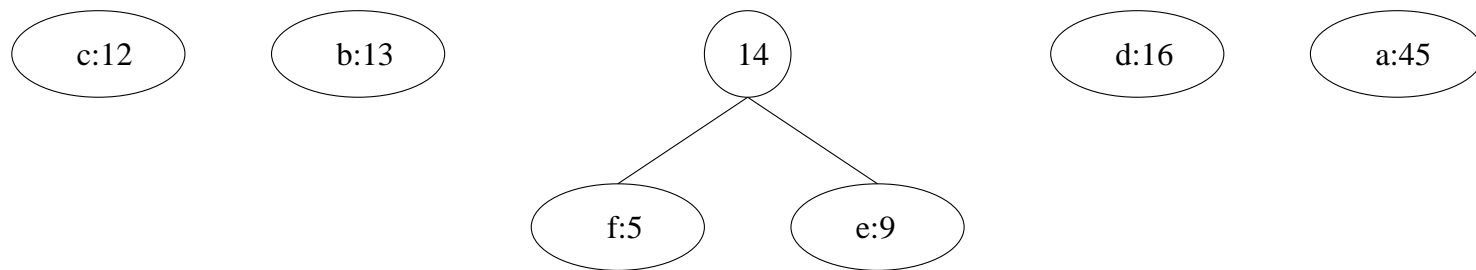
# Huffman Algorithm

- ★ **Construct** a coding tree **bottom-up**.
- ★ **Maintain** a **forest** with  $n$  leaves in all of its trees.
  - Each tree is **optimal** for its leaves.
- ★ Initially, there are  $n$  **singleton** trees in the forest.
  - Each tree is a leaf.
- ★ The **frequency** of a tree is the sum of the frequencies of all of its leaves.
- ★ **Greedy** step:
  - **Find** the two trees with the **minimum** frequencies.
  - **Combine** them together into one tree.
  - The frequency of the new tree is the **sum** of the frequencies of the two combined trees.
- ★ **Terminate** when there is **only one** tree in the forest.

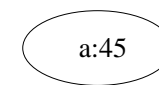
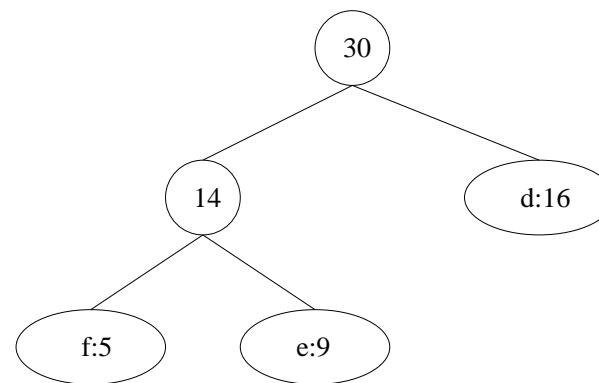
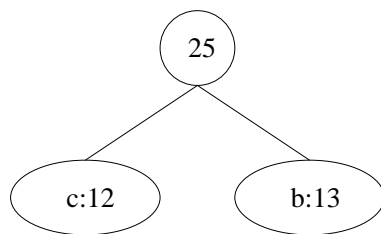
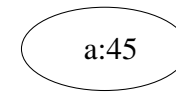
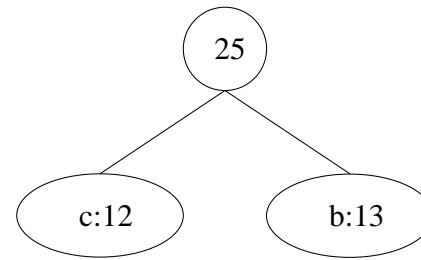
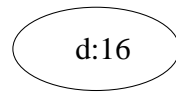
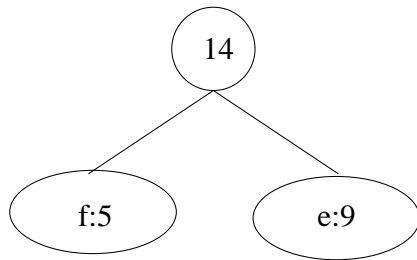
# Example



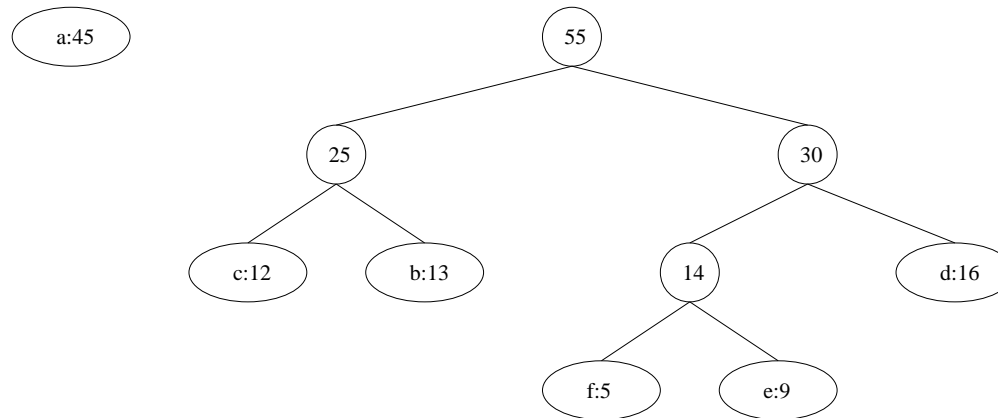
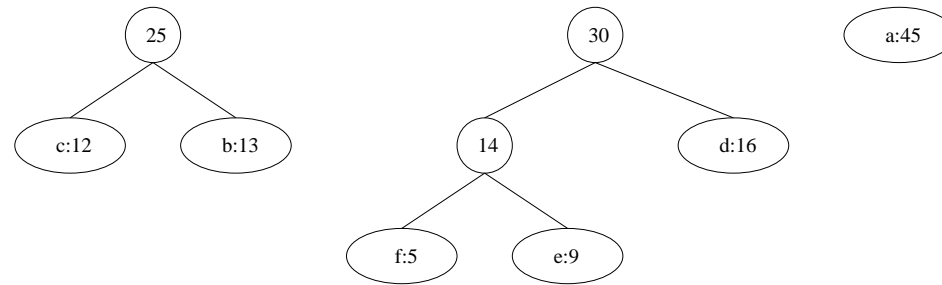
# Example



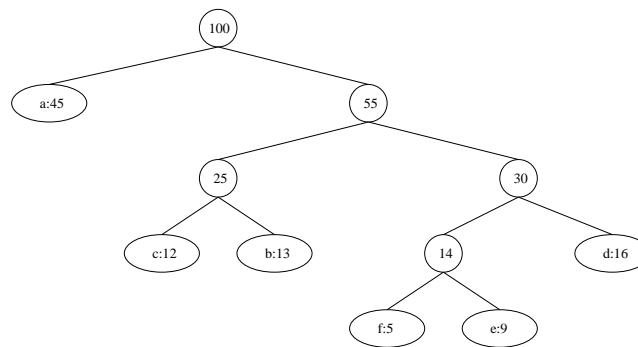
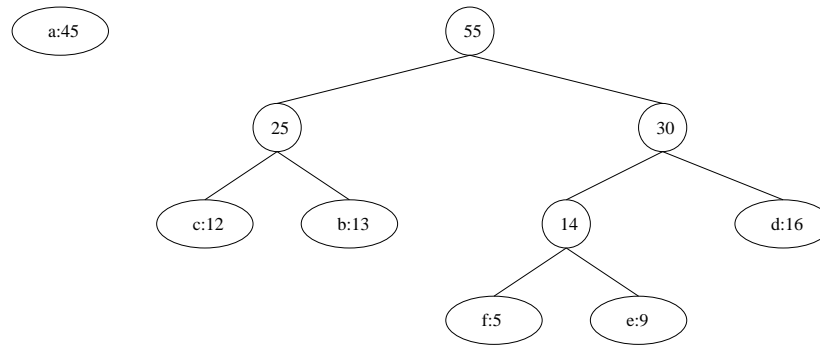
# Example



# Example



# Example



## Correctness

- ★ Huffman algorithm **generates** a binary tree with  $n$  leaves.
- ★ A binary tree **represents** a prefix free code.

## Implementation – Data Structure

- ★ A forest of **binary trees**.
  - Initially, the forest contains  $n$  singleton trees.
  - At the end, the forest contains one tree.
- ★ The frequencies of the trees in the forest are maintained in a **priority queue**  $Q$ .
  - Initially, the queue contains the  $n$  original frequencies.
  - At the end, the queue contains one frequency which is the sum of all original frequencies.

## Implementation – Procedure

```
Huffman( $\langle a_1, f_1 \rangle, \dots, \langle a_n, f_n \rangle$ )
  Build-Queue( $\{f_1, \dots, f_n\}, Q$ )
  for  $i = 1$  to  $n - 1$  (* the combination loop *)
     $z = \text{Allocate-Node}()$  (* creating a new root *)
     $x = \text{left}(z) = \text{Extract-Min}(Q)$ 
      (* lightest tree is the left sub-tree *)
     $y = \text{right}(z) = \text{Extract-Min}(Q)$ 
      (* second lightest tree is the right sub-tree *)
     $f(z) = f(x) + f(y)$  (* frequency of new root *)
    Insert( $Q, f(z)$ ) (* inserting the new root to the queue *)
  return Extract-Min( $Q$ ) (* last tree is the Huffman code *)
```

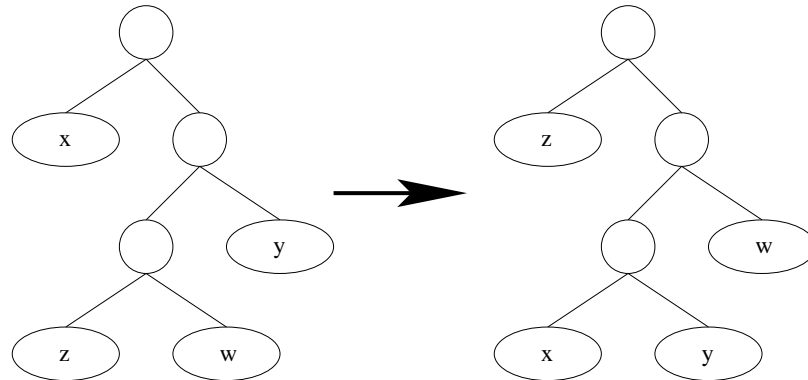
## Complexity

- ★ Implement the priority queue with a **Binary Heap**.
- ★ The complexity of **Build-Queue** is  $O(n)$ .
- ★ The complexity of **Extract-Min** and **Insert** is  $O(\log n)$ .
- ★ The loop is executed  $O(n)$  times.
- ★ The complexity of all the **Extract-Min** and the **Insert** operations is  $O(n \log n)$ .
- ★ The **total** complexity is:  $O(n \log n)$ .

## Optimality - First Lemma

- ★ Let  $\mathcal{A}$  be an alphabet.
- ★ Let  $x$  and  $y$  be the two symbols in  $\mathcal{A}$  with the **smallest** frequencies.
- ★ Then, there exists an **optimal** tree in which:
  - $x$  and  $y$  are **adjacent** leaves (differ only in their last bit).
  - $x$  and  $y$  are the **farthest** leaves from the root.

## Proof

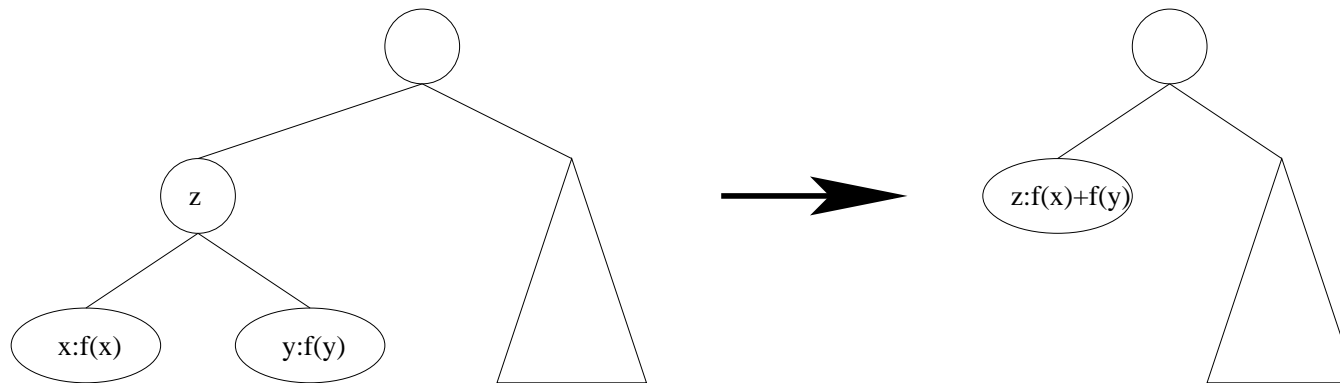


- ★ Let  $z$  and  $w$  be **adjacent** leaves in an optimal tree that are the **farthest** from the root.
- ★ **Exchanging**  $z$  and  $w$  with  $x$  and  $y$  yields a tree with a **smaller or equal** cost.

## Optimality - Second Lemma

- ★ Let  $T$  be an **optimal** tree for the alphabet  $\mathcal{A}$ .
- ★ Let  $x, y$  be **adjacent leaves** in  $T$  and let  $z$  be their **parent**.
- ★ Let  $\mathcal{A}'$  be  $\mathcal{A}$  with a **new** symbol  $z$  **replacing**  $x$  and  $y$  with frequency:  $f(z) = f(x) + f(y)$ .
- ★ Let  $T'$  be the tree  $T$  **without** the leaves  $x$  and  $y$  and with  $z$  as a **new** leaf.
- ★ Then  $T'$  is an **optimal** tree for the alphabet  $\mathcal{A}'$ .

## Proof



- ★ Let  $T''$  be an optimal tree with **smaller** cost than  $T'$ .
- ★ **Replacing**  $z$  in  $T''$  with the two leaves  $x$  and  $y$  **creates** a tree with a **smaller** cost than  $T$ .
- ★ A **contradiction** to the optimality of  $T$ .

## Optimality

**Theorem:** Huffman code is **optimal**.

**Proof:**

- ★ By **induction**.
- ★ The first lemma implies that the first **greedy** step is a **first step** towards an optimal solution.
- ★ The second lemma justifies the **inductive** steps, applying again and again the first lemma.