Algorithms: Greedy Algorithms

Amotz Bar-Noy

CUNY
Greedy Algorithms

Motivation

- **Greedy algorithms** make decisions that “seem” to be the best following some **greedy** criteria.

Off-Line settings

- The whole input is known in advance.
- It is possible to do some preprocessing based on the input.
- Decisions are done in rounds and are irrevocable.

Real-Time and On-Line settings

- Current decisions cannot change past decisions.
- Current decisions cannot rely on the un-known future input.
How and When to use Greedy Algorithms?

Initial solutions
- Establish trivial solutions for a problem of a small size.
  - Usually \( n = 0 \) or \( n = 1 \).

Top-down procedure
- For a problem of size \( n \), look for a greedy decision that reduces the size of the problem to some \( k < n \) and then, apply recursion.

Bottom-up procedure
- Construct the solution for a problem of size \( n \) based on some greedy criteria applied on the solutions to the problems of sizes smaller than \( n \).
The Coin Changing Problem

Input
- \( n \geq 1 \) integer coin denominations:
  \[ d_n > \cdots > d_2 > d_1 = 1 \]
- An integer amount to pay: \( A \).

Output
- Number of coins \( n_i \) for each denomination \( d_i \) to pay \( A \) exactly assuming there are infinite number of coins from each denomination:
  \[ A = n_n d_n + n_{n-1} d_{n-1} + n_2 d_2 + n_1 d_1. \]

Optimization goal
- Minimize total number of coins:
  \[ N = n_n + \cdots + n_2 + n_1. \]

Observation
- Since \( d_1 = 1 \), there is always a solution with \( N = n_1 = A \).
Examples

Current USA currency system

- Dollar, Half Dollar, Quarter, Dime, Nickel, Penny (Cent):
  
  \[ 100 > 50 > 25 > 10 > 5 > 1 \]
  
  \* \( A = 98 = 1 \cdot 50 + 1 \cdot 25 + 2 \cdot 10 + 3 \cdot 1 \)

  \* \( N = 1 + 1 + 2 + 3 = 7 \)

Old British currency system

- Pound, Crown, A half-crown, Florin, Shilling, Sixpence (Tanner), Threepence (Bit), Penny:
  
  \[ 240 > 60 > 30 > 24 > 12 > 6 > 3 > 1 \]

  \* \( A = 149 = 2 \cdot 60 + 1 \cdot 24 + 1 \cdot 3 + 2 \cdot 1 \)

  \* \( N = 2 + 1 + 1 + 2 = 6 \)
The Coin Changing Problem

Greedy Solution

**Greedy criterion**
- Use the largest possible denomination and update $A$.

**Implementation**

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

for $i = n$ downto 1

$n_i = \lfloor A / d_i \rfloor$

$A = A \mod d_i = A - n_id_i$

Return ($N = n_n + \cdots + n_2 + n_1$)

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

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

Examples
- Greedy is optimal for the USA system and the old British system.

Greedy is not always optimal
- \( d_3 = 4, \ d_2 = 3, \ d_1 = 1 \) and \( A = 6 \):
  - **Greedy:** \( 6 = 1 \cdot 4 + 2 \cdot 1 \implies N = 3 \).
  - **Optimal:** \( 6 = 2 \cdot 3 \implies N = 2 \).

Greedy could be very far from optimal
- \( d_3 = x + 1, \ d_2 = x, \ d_1 = 1 \) and \( A = 2x \):
  - **Greedy:** \( 2x = 1 \cdot (x + 1) + (x - 1) \cdot 1 \implies N = x \).
  - **Optimal:** \( 2x = 2 \cdot x \implies N = 2 \).
  - For a very large \( x \), greedy is using too many coins instead of only two coins.
Efficiency of Optimal Algorithms

**Exhaustive optimal algorithm**
- Check all possible combinations.
- This is not a polynomial time algorithm since there are exponential in \( n \) possible combinations.

**A dynamic programming optimal algorithm**
- Find the optimal combination for all the values 1, 2, \ldots, \( A \).
- This algorithm is polynomial in both \( n \) and \( A \).
- It is only a *weakly polynomial time algorithm* because \( A \) could be very large.

**Open problem**
- Design a *strongly polynomial time algorithm*, one that is polynomial only in \( n \).
- Probably impossible!?
The Knapsack Problem

Input
- A thief enters a store and finds \( n \geq 1 \) items \( l_1, \ldots, l_n \).
- For \( 1 \leq i \leq n \), item \( l_i \) is associated with two positive integer parameters \( \langle w_i, v_i \rangle \):
  - The weight of item \( l_i \) is \( w_i \).
  - The value of item \( l_i \) is \( v_i \).

Constraints
- The thief can carry at most integer weight \( W \geq 1 \).
- The thief either takes all of item \( l_i \) or does not take item \( l_i \).

Goal
- Carry items with maximum total value.
  - Which are these items?
  - What is their total value?
The Knapsack Problem

Example

Input

\[
\begin{array}{|c|c|c|}
\hline
W &= 10 & \text{Weight} & \text{Value} \\
\hline
I_1 & 3 & 20 \\
I_2 & 3 & 30 \\
I_3 & 4 & 10 \\
I_4 & 5 & 40 \\
\hline
\end{array}
\]

Non-optimal output

- The thief carries \( \{I_1, I_2, I_3\} \) for a profit of 60 = 20 + 30 + 10.

Optimal output

- The thief carries \( \{I_2, I_4\} \) for a profit of 70 = 30 + 40.
A General Greedy Scheme

Preprocessing

- Order the \( n \) items according to some greedy criterion.
  - Assume this order is \( J_1, J_2, \ldots, J_n \).
  - Assume \( J_1 \) is the most desired item and \( J_n \) is the least desired item.

Rules

- If \( J_1 \) is not too heavy (\( w_1 \leq W \)):
  * Take item \( J_1 \).
  * Update the maximum weight to \( W = W - w_1 \).
  * Continue recursively with \( J_2, J_3, \ldots, J_n \).

- If \( J_1 \) is too heavy (\( w_1 > W \)):
  * Ignore item \( J_1 \).
  * Do not update the maximum weight to \( W \).
  * Continue recursively with \( J_2, J_3, \ldots, J_n \).
A General Greedy Scheme – Implementation

**Implementation**

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

Let \(J_1, \ldots, J_n\) be the new order on the items.

\[ S = \emptyset \quad (* \text{the set of items the thief takes} *) \]

\[ V = 0 \quad (* \text{the value of these items} *) \]

for \(i = 1\) to \(n\) 

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

\[ S = S \cup \{J_i\} \]

\[ V = V + v_i \]

\[ W = W - w(J_i) \]

Return \((S, V)\)
Greedy Criteria

Criterion I
- Order the items by their value from the most expensive to the cheapest.

Criterion II
- Order the items by their weight from the lightest to the heaviest.

Criterion III
- Order the items by their ratio of value over weight from the largest ratio to the smallest ratio.
Greedy-by-Value is Not Optimal

A counter example
- Three items and maximum weight \( W = 10 \).
- Weights and values: \( I_1 = \langle 6, 10 \rangle \), \( I_2 = \langle 5, 6 \rangle \), and \( I_3 = \langle 5, 6 \rangle \).
- **Optimal** takes items \( I_2 \) and \( I_3 \) for a profit of 12.
- **Greedy-by-Value** takes only item \( I_1 \) for a profit of 10.

A very bad counter example
- \( n \) items and maximum weight \( W \geq n - 1 \).
- Weights and values: \( I_1 = \langle W, 2 \rangle \), \( I_2 = \langle 1, 1 \rangle \), \ldots , \( I_n = \langle 1, 1 \rangle \).
- **Optimal** takes items \( I_2, \ldots , I_n \) for a profit of \( n - 1 \).
- **Greedy-by-Value** takes only item \( I_1 \) for a profit of 2.
- The ratio is \( (n - 1)/2 \) that is \( \Omega(n) \).
Greedy-by-Weight is Not Optimal

A counter example
- Three items and maximum weight $W = 10$.
- Weights and values: $I_1 = \langle 6, 13 \rangle$, $I_2 = \langle 5, 6 \rangle$, and $I_3 = \langle 5, 6 \rangle$.
- **Optimal** takes only item $I_1$ for a profit of 13.
- **Greedy-by-Weight** takes items $I_2$ and $I_3$ for a profit of 12.

A very bad counter example
- Two items and maximum weight $W = 2$.
- Weights and values: $I_1 = \langle 1, 1 \rangle$ and $I_2 = \langle 2, x \rangle$.
- **Optimal** takes item $I_2$ for a profit of $x$.
- **Greedy-by-Weight** takes item $I_1$ for a profit of 1.
- The ratio is $x$ that could be very large.
Greedy-by-Ratio is Not Optimal

A counter example
- Three items and maximum weight $W = 10$.
- Weights and values: $I_1 = \langle 6, 10 \rangle$, $I_2 = \langle 5, 6 \rangle$, and $I_3 = \langle 5, 6 \rangle$.
- **Optimal** takes items $I_2$ and $I_3$ for a profit of 12.
- **Greedy-by-Ratio** takes only item $I_1$ for a profit of 10.

A very bad counter example
- Two items and maximum weight $W$.
- Weights and values: $I_1 = \langle 1, 2 \rangle$ and $I_2 = \langle W, W \rangle$.
- **Optimal** takes item $I_2$ for a profit of $W$.
- **Greedy-by-Ratio** takes item $I_1$ for a profit of 2.
- The ratio is $W/2$ that could be very large.
A Better “Almost Greedy” Algorithm

Algorithm
- Select either the output of Greedy-by-Ratio or the output of Greedy-by-Value,
- In particular, the algorithm guarantees the profit of the most valuable item whose weight is at most $W$.

The algorithm is almost optimal
- The algorithm is a $1/2$ guaranteed approximation algorithm.
- The profit of the thief is guaranteed to be at least half of the optimal profit.
The Fractional Knapsack Problem

Input
- A thief enters a store and finds $n \geq 1$ items $I_1, \ldots, I_n$.
- For $1 \leq i \leq n$, item $I_i$ is associated with a positive integer weight $w_i$ and a positive integer value $v_i$.

Constraints
- The thief can carry at most integer weight $W \geq 1$.
- The thief can take portions of items. If the thief takes a fraction $0 < p_i \leq 1$ of item $I_i$:
  * Its value is $p_i v_i$.
  * Its weight is $p_i w_i$.

Goal
- Carry portions of items with maximum total value.
The Knapsack Problem

Optimal Greedy Criterion

**Theorem**

- **Greedy-by-Ratio** is optimal for the fractional Knapsack problem.

**Proof Sketch**

- Assume that **Greedy-by-Ratio** is not optimal on $I_1, \ldots, I_n$ and $W$.
- Let the portions taken by **Optimal** be $0 \leq p_1, \ldots, p_n \leq 1$.
- Since **Greedy-by-Ratio** is not optimal, there exist $I_i$ and $I_j$ for which $p_i < 1$ and $p_j > 0$ such that
  \[
  \frac{v_i}{w_i} > \frac{v_j}{w_j}
  \]
- Because each unit of weight of item $I_i$ generates more profit than each unit of weight of item $I_j$, it is more profitable to take more of item $I_i$ and less of item $I_j$.
- A **contradiction** to the optimality of **Optimal**.
The 0-1 Knapsack Problem

**The exhaustive algorithm**
- Check all possible sets of items.
  * Not a polynomial time algorithm because there are $2^n$ such sets.

**Another optimal algorithm**
- A dynamic programming based algorithm.
  * Polynomial in both $n$ and $W$.
  * Not a strongly polynomial time algorithm.

**Open problem**
- Find an algorithm that is polynomial only in $n$.
- Probably impossible!?
- There are “good” theoretical and practical solutions based mainly on Greedy-by-Ratio.
The Activity-Selection Problem

Input
- Activities $A_1, \ldots, A_n$ 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

- Three activities $A_1 = [1, 4)$, $A_2 = [3, 6)$, $A_3 = [5, 8)$.

A graphical representation:

Optimal solution
Static vs. Dynamic Greedy Strategies

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

**Dynamic greedy approach**
- The *greedy* criterion may be modified during the execution of the algorithm based on prior decisions.

**Remark**
- A static criterion is by definition also a dynamic criterion but not vice versa.
A General Static Greedy Scheme

Preprocessing
- Order the activities following some greedy criterion.

Data structure
- Maintain a set $S$ of the activities that have been selected so far.
- Initially $S = \emptyset$ and at the end $S$ is the output.

Scheme
- Let $A$ be the current considered activity. If $A$ is compatible with all the activities already in $S$:
  - Then add $A$ to $S$.
  - Else ignore $A$.
- Repeat the above until there are no activities to consider.
A General Dynamic Greedy Scheme

Data structure
- **Maintain** two sets of activities:
  - $S$: activities that have been selected so far.
  - $R$: activities not in $S$ that are compatible with all the activities in $S$.
  - Initially, $S = \emptyset$ and $R = \{A_1, \ldots, A_n\}$.
  - At the end, $S$ is the output and $R = \emptyset$.

Scheme
- **Select** a “good” activity $A$ from $R$, following some greedy criterion.
- **Add** $A$ to $S$.
- **Delete** $A$ and the activities that are not compatible with $A$ from $R$.
- **Repeat** the above until $R$ is empty.
Greedy Criteria

Four possible criteria
- Prefer short activities.
- Prefer activities that intersect 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 has a dynamic version.
The Activity-Selection Problem

An Optimal Greedy Algorithm

**Preprocessing** \((A_1, \ldots, A_n)\)

Sort the activities according to their finish time
Let this order be \(A_1, \ldots, A_n\) \((^* i < j \Rightarrow f_i \leq f_j *\)

**Greedy-Activity-Selector** \((A_1, \ldots, A_n)\)

\[ S = \{A_1\} \quad (^* A_1 \text{ terminates the earliest } *) \]
\[ j = 1 \quad (^* A_j \text{ is the current selected activity } *) \]

for \(i = 2\) to \(n\) \((^* \text{ scan all the activities } *)\)

if \(s_i \geq f_j\) \((^* \text{ check compatibility } *)\)

then

\[ S = S \cup \{A_i\} \quad (^* \text{ select } A_i \text{ that is compatible with } S *) \]
\[ j = i \]

else \((^* A_i \text{ is not compatible } *)\)

Return\((S)\)
The Activity-Selection Problem

Example – Input

A_{11}  A_{10}  A_9  A_8  A_7  A_6  A_5  A_4  A_3  A_2  A_1

activities

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

time
Example – Output

The Activity-Selection Problem
The Activity-Selection Problem

Example – Output

activities

$A_1$

$A_4$

$A_8$

$A_{11}$

time

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

Amotz Bar-Noy (CUNY)
Correctness and Complexity

Correctness

- The selection rule guarantees that a new activity $A_i$ is added to $S$ only if it is compatible with activity $A_j$ that was the last activity added to $S$.
- Therefore, $A_i$ is compatible with all the activities in $S$ because $A_j$ terminates the latest among these activities.
- As a result, all the activities in the output set are compatible with each other.

Complexity

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

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

**Proof sketch**
- $A_i \notin S \Rightarrow \exists A_j \in S$ that is not in $T$ in which $j < i$.
- $A_j$ is compatible with all the activities in $T$ that finish before it since they are all in $S$.
- $A_j$ is compatible with all the activities in $T$ that finish after $A_i$ since it finishes before $A_i$.
- Therefore, $T \cup \{A_j\} \setminus \{A_i\}$ is a solution with the same size as $T$ and hence optimal.
- Continue this way until $T$ becomes $S$. 
Another optimal solution with 4 activities.
Example

A third optimal solution: after the first transformation.
The Activity-Selection Problem

Example

The greedy solution: after the second transformation.
Huffman Codes

Input
- An alphabet of \( n \) symbols \( a_1, a_2, \ldots, a_n \).
- A File \( \mathcal{F} \) containing \( L \) symbols from the alphabet.

Notations
- For \( 1 \leq i \leq n \), the symbol \( a_i \) appears \( n_i \) times in \( \mathcal{F} \).
- For \( 1 \leq i \leq n \), the frequency of \( a_i \) is \( f_i = n_i / L \).

Observation
\[
\sum_{i=1}^{n} n_i = L \quad \Rightarrow \quad \sum_{i=1}^{n} f_i = 1
\]

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 – Goals

**Optimization**
- Minimize $L'$ the length of $F'$.

**Efficiency**
- Design an efficient algorithm to find the $n$ codewords.
  - An algorithm with a “good” polynomial running time: $O(n \log n)$.
- Efficient **encoding** and **decoding** procedures.
  - Should be done in $O(B)$-time where $B$ is the size of the original file in bits.
**Example**

**Input**
- 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: $300 = 3 \cdot 100$.

**Code II**
- $w_a = 0$, $w_b = 101$, $w_c = 100$, $w_d = 111$, $w_e = 1101$, $w_f = 1100$.
- Length of encoded file: $224 = 1 \cdot 45 + 3 \cdot 13 + 3 \cdot 12 + 3 \cdot 16 + 4 \cdot 9 + 4 \cdot 5$.

**Remark**
- Code II is optimal, $\approx 25\%$ better than code I.
Encoding Codes

Encoding

- Straightforward using tables.

Example: Code II

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>101</td>
</tr>
<tr>
<td>c</td>
<td>100</td>
</tr>
<tr>
<td>d</td>
<td>111</td>
</tr>
<tr>
<td>e</td>
<td>1101</td>
</tr>
<tr>
<td>f</td>
<td>1100</td>
</tr>
</tbody>
</table>

“fedcba”  ⇒  1100|1101|111|100|101|0

“abbfcccde”  ⇒  0|101|101|1100|100|100|100|111|1101
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.

Encoding and Decoding

- Simple with one pass of the encoded and decoded files.

Theorem

- An optimal prefix free code always exists.
Prefix Free Codes as Binary Trees

**Definition**

- A prefix free 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.

**Proposition**

- The binary tree represents a prefix free code since a path to a leaf cannot be a prefix of any other path.
Example: Code II

Illustration

- 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.
Decoding with Prefix Free Codes

Decoding

- Follow the tree.

Example: Code II

```
110011011111001010  ⇒  1100|1101|111|100|101|0
                         f e d c b a
010110111001001001001111101  ⇒  “abbfcccde”
```
Binary Trees Cost

Notations

- \( T \): a binary tree with \( n \) leaves representing a code for a File \( F \) with an alphabet with \( n \) symbols.
- \( n(x) \): the number of appearances of a leaf \( x \) in \( F \).
- \( f(x) = \frac{n(x)}{L} \): the frequency of a leaf \( x \).
- \( \ell(x) \) the length of the path from the root to a leaf \( x \).

Cost

- The length of the encoded file
  \[
  B(T) = \sum_{\text{a leaf } x} n(x)\ell(x)
  \]
- The average length of a codeword is
  \[
  \sum_{\text{a leaf } x} f(x)\ell(x) = \sum_{\text{a leaf } x} \frac{n(x)}{L}\ell(x) = \frac{B(T)}{L}
  \]
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 outline
- Assume that an internal node $z$ has only one child $y$.
- Eliminate $z$ to create a tree with a smaller cost.
- In the new tree, $\ell(x)$ of all the leaves in the sub-tree rooted at $z$ is reduced by 1.
- The cost of the tree is improved since, these are the only changes.
- A contradiction to the optimality of the code.

Two cases to complete the proof
- Case I: $z$ is the root.
- Case II: $z$ is not the root.
Lemma

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

\[
B(T) = (45 + 13 + 12 + 16 + 9 + 5)3 = 300
\]
Lemma

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

$$B(T) = (45 + 13 + 12 + 16)3 + (9 + 5)2 = 286$$
**A Structural Claim**

**Case I**
- \( z \) is the root: Make \( y \) the new root.

![Diagram showing structural change from \( z \) to \( y \) as the root]
A Structural Claim

Case II

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

Diagram:

Before:

```
    p
   / \  
  z   A
   \ /
    y B
   / \  
  C   
```

After:

```
    p
   / \  
  A   y
   \ /
    B C
```
**Huffman Code Algorithm**

**Goal**
- Construct a coding tree bottom-up.

**Constructing the tree outline**
- Maintain a forest with total of $n$ leaves in all of its trees.
- 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 its leaves.
- In each round reduce the number of trees in the forest by one.
- Terminate when there is only one tree in the forest.

**Greedy step**
- Combine two trees with the minimum frequencies into one tree.
- The frequency of the new tree is the sum of the frequencies of the two combined trees.
Example

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

- f:5
- e:9

- c:12
- b:13

- 14

- d:16
- a:45
Example

Huffman Codes

- c:12
- b:13
- 14
- d:16
- a:45
- f:5
- e:9
- 25
- 14
- d:16
- a:45
- f:5
- e:9
- 25
- c:12
- b:13
Example

Huffman Codes

Amotz Bar-Noy (CUNY)
Example
Example

- a: 45
- c: 12
- b: 13
- e: 9
- 30
- 25
- 14
- d: 16
- f: 5
- 55
- 100

Huffman Codes

Greedy Algorithms
Another Example

- A small example with animation:
  
Correctness and Complexity

Correctness

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

Complexity

- The Huffman code algorithm can be implemented with complexity $\Theta(n \log(n))$. 
**Implementation – Data Structure**

**Forest of trees**
- Initially, the forest contains $n$ singleton trees.
- At the end, the forest contains one tree.

**Priority queue of frequencies**
- 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 1 the sum of all original frequencies.
Huffman Codes

**Implementation – Procedure**

Huffman\((\langle a_1, f_1 \rangle, \ldots, \langle a_n, f_n \rangle)\)

Build-Queue\((\{f_1, \ldots, f_n\}, Q)\)

for \(i = 1\) to \(n - 1\)  (* the combination loop *)

\(z = \text{Allocate-Node}()\)  (* creating a new root *)

\(x = \text{left}(z) = \text{Extract-Min}(Q)\)

(* the lightest tree is the left sub-tree *)

\(y = \text{right}(z) = \text{Extract-Min}(Q)\)

(* the second lightest tree is the right sub-tree *)

\(f(z) = f(x) + f(y)\)  (* the frequency of the new root *)

Insert\((Q, f(z))\)  (* inserting the new root to the queue *)

return Extract-Min\((Q)\)  (* the last tree is the Huffman code *)
Complexity

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

Lemma I

- 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.
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.
Lemma II

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 $\mathcal{A}'$. 
Lemma II – Proof Sketch

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

- The Huffman code is an optimal code.

Proof by induction outline

- **Lemma I** implies that the first greedy step is a first step towards an optimal solution.
- **Lemma II** justifies the inductive steps that apply again and again the first lemma.