# Greedy Algorithm

KOMS120403 / KOMS119602 - Design and Analysis of Algorithm (2021/2022)

09 - Greedy Algorithm

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 4-8 April 2022

# Table of contents

# Optimization problem

An optimization problem is the problem of finding the best solution from all feasible solutions.

**Types of optimization problems**

- Maximization
  - Example: Integer knapsack problem
- Minimization
  - Example: Graph coloring problem, TSP

**Standard form**:

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 0, \ i = 1, \ldots, m \\ & h_j(x) = 0, \ j = 1, \ldots, p \end{aligned}$$

**Definition**

Greedy algorithm is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.

This is the most popular and the simplest algorithm to solve optimization problems (maximization and minimization).

- Greedy principal: **Take what you can get now!**
- Greedy algorithm builds the solution step-by-step.
- In each step, there are many possible choices. We take the best decision in each step, i.e. we choose the local optimum, in order to reach the global optimum.

# Greedy algorithm

## Example (Coin exchange problem)

We have a check of 42 in hand. There are coins of nominal 1, 5, 10, and 25. We want to exchange the money with the coins. What is the minimum number of coins needed in the exchange.

There are many possible combinations of coins. With brute-force algorithm, we can simply list all possibilities:

- $42 = 1 + 1 + \cdots + 1 \rightarrow 42$ coins
- $42 = 5 + 1 + 1 + \cdots + 1 \rightarrow 38$ coins
- ... etc.

With greedy algorithm, at each step, we take a coin with maximum value as many as possible.

$$42 = 25 + 10 + 5 + 1 + 1$$

So 5 coins are enough.

**Components of greedy algorithm:**

- A candidate set - consists of candidate of solution that is created from the set.
- A selection function - used to choose the best candidate to be added to the solution.
- A feasibility function - used to determine whether a candidate can be included in the solution (feasible/unfeasible).
- An objective function - used to assign a value to a solution or a partial solution (maximizing or minimizing).
- A solution function - used to indicate whether a complete solution has been reached.

# Scheme of greedy algorithm

**Components analysis of the coin exchange problem:**

- Candidate set: the set of coins $\{1, 5, 10, 25\}$.
- Selection function: choose the coin that has maximum value.
- Feasibility function: check if the sum of coins after taking a new coin does not exceed the amount of money.
- Objective function: minimizing the number of coins used.
- Solution function: the set of selected coins $\{1, 10, 25\}$.

# Scheme of greedy algorithm

**Algorithm 1** General scheme of greedy algorithm

**procedure** GREEDY($C$: candidate set)
    $S \leftarrow \{\}$ ▷ *$S$ is the solution function*
    **while** (**not** SOLUTION($S$)) **and** ($C \neq \{\}$) **do**
        $x \leftarrow$ SELECTION($C$)
        $C \leftarrow C - \{x\}$
        **if** FEASIBLE($S \cup \{x\}$) **then**
            $S \leftarrow S \cup \{x\}$
        **end if**
    **end while**
    **if** SOLUTION($S$) **then** **return** $S$
    **else print**('No solution exists')
    **end if**
**end procedure**

- At the end of iteration ("if condition"), we have a local optimum solution.
- At the end of the while loop, we obtain the global optimum solution (if any)

# Solving problems with greedy algorithm

**Some examples:**

1. Coin exchange problem
2. Activity selection problem
3. Time minimization in the system
4. Integer knapsack problem
5. Fractional knapsack problem
6. Huffman coding
7. Traveling Salesman Problem

# 1. Coin exchange problem

**Problem formulation:**

- The amount of money want to be exchanged: $M$
- The set of available coins: $\{c_1, c_2, \ldots, c_n\}$
- The solution set: $X = \{x_1, x_2, \ldots, x_n\}$, where $x_i = 1$ if $a_i$ is chosen and $x_i = 0$ otherwise

The objective function:

$$\text{Minimize } F = \sum_{i=1}^{n} x_i$$

$$\text{subject to } \sum_{i=1}^{n} c_i x_i = M$$

# 1. Coin exchange problem (2)

**Solution by exhaustive search**

- Since $X = \{x_1, x_2, \ldots, x_n\}$ and $x_i \in \{01\}$, then there are $2^n$ possible solutions.
- To evaluate the objective function for each solution candidate, we need $\mathcal{O}(n)$-time
- So, the time complexity of the exhaustive search is: $\mathcal{O}(n \cdot 2^n)$.

# 1. Coin exchange problem (3)

**Algorithm 2** General scheme of greedy algorithm

```
procedure CoinExchange(C: coin set, M: integer)
    S ← {}
    while (∑(all coins in S) ≠ M) and (C ≠ {}) do
        x ← coin of maximum value
        C ← C − {x}
        if ∑(all coins in S) + value(x) ≤ M then
            S ← S ∪ {x}
        end if
    end while
    if ∑(all coins in S) = M then
        return S
    else
        print('No feasible solution')
    end if
end procedure
```

# 1. Coin exchange problem (4)

**Time complexity:**

- Choosing a coin with maximum value: $\mathcal{O}(n)$ (using brute-force to get max).
- The while loop is repeated $n$ times (maximum), so the overall complexity is: $\mathcal{O}(n^2)$.
- If the coin is ordered in descending order, the complexity becomes $\mathcal{O}(n)$, because choosing a coin with max value only takes $\mathcal{O}(1)$-time.

# 2. Activity selection problem

**Problem:** given $n$ activities $S = \{1, 2, \ldots, n\}$, that will use a resource (for instance, meeting room, studio, processor, etc.).

Suppose that a resource can only be used to do one activity at one time. Each time an activity occupies a resource, then the other activities cannot use it until the first activity is finished.

Each activity $i$ starts at time $s_i$ and ends at time $f_i$, where $s_i \leq f_i$. Two activities $i$ and $j$ is called *compatible* if the interval $[s_i, f_i]$ and $[s_j, f_j]$ do not intersect.

Our goal is to choose as many activities as possible that can be served by a resource.

# 2. Activity selection problem (2)

## Example (Activity selection problem)

Given $n = 11$ activities with the starting-ending time as given in the following table:

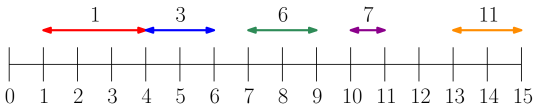
| $i$ | $s_i$ | $f_i$ |
|---|---|---|
| 1 | 1 | 4 |
| 2 | 3 | 5 |
| 3 | 4 | 6 |
| 4 | 5 | 7 |
| 5 | 3 | 8 |
| 6 | 7 | 9 |
| 7 | 10 | 11 |
| 8 | 8 | 12 |
| 9 | 8 | 13 |
| 10 | 2 | 14 |
| 11 | 13 | 15 |

**Solution with exhaustive search:**

1. List all subsets of the set of $n$ activities.
2. Evaluate each subset, check if the solution is compatible.
3. If yes, then the subset is a solution candidate.
4. Choose the solution candidate with the maximum number of activities
5. The time complexity of the algorithm is $\mathcal{O}(n \cdot 2^n)$. Why?

**The greedy approach:**

1. Order the activities based on the end-time in ascending order.
2. At each step, choose the activity whose starting time is greater than or equal to the end-time of the activity that was chosen before




**Strategy:** at each step, we choose the activity of the smallest index that can be done in the available time-slot.

Set of solution: $\{1, 3, 6, 7, 11\}$

**Algorithm 3** Greedy Activity Selector

**procedure** $\textsc{ActvtySlctr}((s_1, \ldots, s_n), (f_1, \ldots, f_n))$<br>
    $n \leftarrow \text{length}(s)$ ▷ *$s = (s_1, \ldots, s_n)$*<br>
    $A \leftarrow \{1\}$ ▷ *A is the solution function*<br>
    $j \leftarrow 1$<br>
    **for** $i \leftarrow 2$ **to** $n$ **do**<br>
        **if** $s_i \geq f_j$ **then** ▷ *$s_i$: starting time of activity $i$, $f(j)$: finishing time of activity $j$*<br>
            $A \leftarrow A \cup \{i\}$<br>
            $j \leftarrow i$<br>
        **end if**<br>
    **end for**<br>
**end procedure**

Time complexity: $\mathcal{O}(n)$. Why?

**Theorem**

*The greedy algorithm gives an optimal solution for the activity selection problem.*

**Proof idea.**

- We wanted to show that the schedule, $A$, chosen by greedy was optimal.
- To do this, we showed that the number of activities in $A$ was at least as large as the number of activities in any other non-overlapping set of activities.
- To show this, we considered any arbitrary, non-overlapping set of activities, $B$. We showed that we could replace each activity in $B$ with an activity in $A$.

# 2. Activity selection problem (7)

**Proof.**

Let $S = \{1, 2, \ldots, n\}$ be the set of activities that are ordered based on the finishing time, $A$ be the optimal solution, and $B$ be the output of the greedy algorithm. Moreover, they are ordered based on the finishing time.

Let $a_x$ be the first activity in $A$ that is different than an activity in $B$. So:

- $A = a_1, a_2, \ldots, a_{x-1}, a_x, a_{x+1}, \ldots$
- $B = a_1, a_2, \ldots, a_{x-1}, b_x, b_{x+1}, \ldots$

Since $B$ was chosen by the Greedy algorithm, $b_x$ must have a finishing time earlier than the finishing time of $a_x$.

So, $A' = A - \{a_x\} \cup \{b_x\} = a_1, a_2, \ldots, a_{x-1}, b_x, b_{x+1}, \ldots$ (i.e. replacing $a_x$ with $b_x$ in $A$) is also a feasible solution.

Continuing this process, we see that we can replace each activity in $A$ with an activity in $B$. $\square$

**Another proof alternatives** (*as explained in class*)

### Proof.

Let $S = \{1, 2, \ldots, n\}$ be the set of activities that are ordered based on the finishing time.

Suppose that $A \subseteq S$ be an optimal solution, where the elements in $A$ are also ordered based on the finishing time, and the first element of $A$ is $k_1$.

If $k_1 = 1$, then $A$ is begun with a greedy choice (as in the algorithm). Otherwise, we show that $B = (A - \{k_1\}) \cup \{1\}$ (a solution begin with the greedy choice 1 is another optimal solution).

Since $f_1 \leq f_{k_1}$ (bcs 1 is the first element of $S$), and the activities in $A$ are compatible, then the activities in $B$ are also compatible. Since $|A| = |B|$, then $B$ is an optimal solution. Hence, there is also an optimal solution that is begun with a greedy choice. $\square$

# 3. Time minimization in the system

# 3. Time minimization in the system (1)

**Problem:** a server (processor, cashier, customer service, etc.) has $n$ clients that must be served. The time to serve client $i$ is $t_i$. How to minimize the total time in the system (including the waiting time)?

$$T = \sum_{i=1}^{n} \text{time spent in the system}$$

*Remark.* This problem is equivalent to minimizing the average time of clients in the system.

# 3. Time minimization in the system (2)

**Example**

There are three clients with serving time: $t_1 = 5$, $t_2 = 10$, and $t_3 = 3$.

**Solution**

*The possible order of the clients:*

- **1,2,3:** $5 + (5 + 10) + (5 + 10 + 3) = 38$
- **1,3,2:** $5 + (5 + 3) + (5 + 3 + 10) = 31$
- **2,1,3:** $10 + (10 + 5) + (10 + 5 + 3) = 43$
- **2,3,1:** $10 + (10 + 3) + (10 + 3 + 5) = 41$
- **3,1,2:** $3 + (3 + 5) + (3 + 5 + 10) = 29$
- **3,2,1:** $3 + (3 + 10) + (3 + 10 + 5) = 34$

# 3. Time minimization in the system (3)

- Other problems similar to minimizing time in the system is optimal storage on tapes or music storage in a cassette tape (analog system with sequential storage system).
- The programs/musics are saved in the tape sequentially. The length of every song $i$ is $t_i$ (in second/minute). To retrieve and play a song, the tape is initially placed in the beginning.
- If the songs in the tape are saved in the order $X = \{x_1, x_2, \ldots, x_n\}$, then the time needed to play the song $x_j$ to the end is $T_j = \sum_{1 \le k \le j} t_{x_k}$ .
- If all songs are often played, the *mean retrieval time / MRT* is $\frac{1}{n} \sum_{1 \le j \le n} T_j$.
- Here we are asked to find a permutation of $n$ songs s.t. if the songs are saved in the tape, then the MRT is minimum. Minimizing MRT is equivalent to minimizing the following:

$$d(X) = \frac{1}{n} \sum_{1 \le j \le n} \sum_{1 \le k \le j} t_{x_k} \quad \text{or} \quad d(X) = \sum_{1 \le j \le n} \sum_{1 \le k \le j} t_{x_k}$$

**Solution with the exhaustive search**

- The order of clients in the system is a permutation. The number of permutation of $n$ elements is $n!$.
- To evaluate the objective function of a permutation needs $\mathcal{O}(n)$-time
- The time complexity of exhaustive search is $\mathcal{O}(n \cdot n!)$.

# 3. Time minimization in the system (5)

## Solution with the greedy algorithm

**Strategy:** At each step, choose the client that needs the minimum serving time among all clients that have not been served.

**Algorithm 4** Clients Scheduling

```
procedure CLIENTSSCHEDULING(n)
    S ← {}                                         ▷ S is the solution function
    while C ≠ {} do                                ▷ C is the solution candidate
        i ← client with minimum t[i] in C          ▷ t_i is the serving time of client i
        C ← C − {i}
        S ← S ∪ {i}
    end while
    return S
end procedure
```

Time complexity: $\mathcal{O}(n^2)$. Why?

# 3. Time minimization in the system (6)

If the clients are ordered based on the serving time (in ascending order), then the complexity of the greedy algorithm is $\mathcal{O}(n)$.

**Algorithm 5** Clients Scheduling

```
procedure ClientsScheduling2(n)
    input: clients (1, 2, ..., n) ordered ascendingly based on t_i
    for i ← 1 to n do
        print(i)
    end for
end procedure
```

*Remark.* The greedy algorithm for clients scheduling based on the serving time with ascending order always produces an optimal solution.

## 3. Time minimization in the system (7)

**Theorem**

*If $t_1 \leq t_2 \leq \cdots \leq t_n$, then the order $i_j = j$, $1 \leq j \leq n$ minimizes:*

$$T = \sum_{k=1}^{n} \sum_{j=1}^{k} t_{i_j}$$

*for all possible permutations of $i_j$, $1 \leq j \leq n$.*

Proof available in Ellis Horrowitz & Sartaj Sahni, Computer Algorithms, 1998). See next slide.

**Proof:** Let $I = i_1, i_2, \ldots, i_n$ be any permutation of the index set $\{1, 2, \ldots, n\}$. Then

$$d(I) = \sum_{k=1}^{n} \sum_{j=1}^{k} t_{i_j} = \sum_{k=1}^{n} (n-k+1) t_{i_k}$$

If there exist $a$ and $b$ such that $a < b$ and $t_{i_a} > t_{i_b}$, then interchanging $i_a$ and $i_b$ results in a permutation $I'$ with

$$d(I') = \left[ \sum_{\substack{k \\ k \neq a \\ k \neq b}} (n-k+1) t_{i_k} \right] + (n-a+1) t_{i_b} + (n-b+1) t_{i_a}$$

Subtracting $d(I')$ from $d(I)$, we obtain

$$\begin{aligned} d(I) - d(I') &= (n-a+1)(t_{i_a} - t_{i_b}) + (n-b+1)(t_{i_b} - t_{i_a}) \\ &= (b-a)(t_{i_a} - t_{i_b}) \\ &> 0 \end{aligned}$$

Hence, no permutation that is not in nondecreasing order of the $t_i$'s can have minimum $d$. It is easy to see that all permutations in nondecreasing order of the $t_i$'s have the same $d$ value. Hence, the ordering defined by $i_j = j, 1 \leq j \leq n$, minimizes the $d$ value. □

# 4. Integer (1/0) knapsack problem

# 4. Integer (1/0) knapsack problem (1)

**Problem:** given $n$ objects and a knapsack with capacity $K$. Every object has weight $w_i$ and profit $p_i$.

How to chose the objects to be included in the knapsack s.t. the total profit is maximum? The total weight of the objects should not exceed the capacity of the knapsack.

The mathematical formulation of 1/0 knapsack problem:

$$\text{Maximize} \quad F = \sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq K$$

$$\text{and } x_i = 0 \text{ or } x_i = 1, \text{ for } i = 1, 2, \ldots, n$$

# 4. Integer (1/0) knapsack problem (2)

Recall that the time complexity with exhaustive search is $\mathcal{O}(n \cdot 2^n)$. Why?

**The greedy approach:**

- Include the object one-by-one to the knapsack. Once it is included, it cannot be undone.
- Some greedy-heuristically strategies that can be used to choose the objects in the knapsack:
  1. Greedy by profit: at each step, choose the object with maximum profit
  2. Greedy by weight: at each step, choose the object of minimum weight
  3. Greedy by density: at each step, choose the object with the maximum value of $p_i/w_i$
- However, none of the strategies above guarantees an optimal solution.

# 4. Integer (1/0) knapsack problem (3)

**Example**

Given four objects as follows, and a knapsack of capacity $M = 16$.

$$(w_1, p_1) = (6, 12);\ (w_2, p_2) = (5, 15)$$

$$(w_3, p_3) = (10, 50);\ (w_4, p_4) = (5, 10)$$

| Object properties | | | | Greedy by | | | Optimal solution |
|---|---|---|---|---|---|---|---|
| $i$ | $w_i$ | $p_i$ | $p_i/w_i$ | profit | weight | density | |
| 1 | 6 | 12 | 2 | 0 | 1 | 0 | 0 |
| 2 | 5 | 15 | 3 | 1 | 1 | 1 | 1 |
| 3 | 10 | 50 | 5 | 1 | 0 | 1 | 1 |
| 4 | 5 | 10 | 2 | 0 | 1 | 0 | 0 |
| Solution set | | | | $\{3, 2\}$ | $\{2, 4, 1\}$ | $\{3, 2\}$ | 15 |
| Total weight | | | | 20 | 20 | 20 | 15 |
| Total profit | | | | 28.2 | 31.0 | 31.5 | 65 |

# 4. Integer (1/0) knapsack problem (4)

**Example**

Given six objects as follows:

$$(w_1, p_1) = (100, 40);\ (w_2, p_2) = (50, 35);\ (w_3, p_3) = (45, 18)$$

$$(w_4, p_4) = (20, 4);\ (w_5, p_5) = (10, 10);\ (w_6, p_6) = (5, 2)$$

and a knapsack of capacity $M = 100$.

# 4. Integer (1/0) knapsack problem (4)

**Example**

Given six objects as follows:

$$(w_1, p_1) = (100, 40);\ (w_2, p_2) = (50, 35);\ (w_3, p_3) = (45, 18)$$

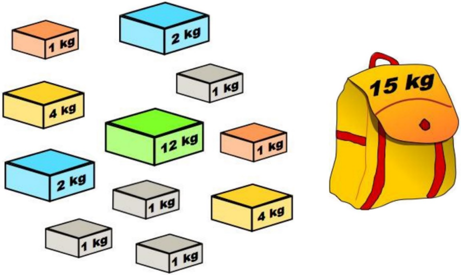
$$(w_4, p_4) = (20, 4);\ (w_5, p_5) = (10, 10);\ (w_6, p_6) = (5, 2)$$

and a knapsack of capacity $M = 100$.

| Object properties | | | | Greedy by | | | Optimal solution |
|---|---|---|---|---|---|---|---|
| $i$ | $w_i$ | $p_i$ | $p_i/w_i$ | profit | weight | density | |
| 1 | 100 | 40 | 0.4 | 1 | 0 | 0 | 0 |
| 2 | 50 | 35 | 0.7 | 0 | 0 | 1 | 1 |
| 3 | 45 | 18 | 0.4 | 0 | 1 | 0 | 1 |
| 4 | 20 | 4 | 0.2 | 0 | 1 | 1 | 0 |
| 5 | 10 | 10 | 1.0 | 0 | 1 | 1 | 0 |
| 6 | 5 | 2 | 0.4 | 0 | 1 | 1 | 0 |
| | Total weight | | | 100 | 80 | 85 | 100 |
| | Total profit | | | 40 | 34 | 51 | 55 |

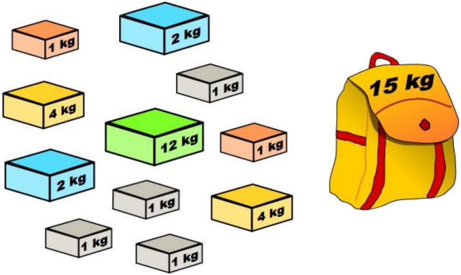
**Conclusion:**

Is greedy algorithm always able to find an optimal solution for the integer knapsack problem?



**Conclusion:**

Is greedy algorithm always able to find an optimal solution for the integer knapsack problem?




NO! **Homework:** Find an example where the three approaches do not give an optimum solution!

# 5. Fractional knapsack problem

# 5. Fractional knapsack problem (1)

The fractional knapsack problem is a variant of knapsack problem, but the solution is not necessarily integer, it can be in fraction.

**Problem formulation:**

$$\text{Maximize } F = \sum_{i=1}^{n} p_i x_i$$

$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \leq K$$

$$\text{and } 0 \leq x_i \leq 1, \text{ for } i = 1, 2, \ldots, n$$

**Question:** is it possible to solve the problem with exhaustive search?

# 5. Fractional knapsack problem (2)

**Question:** is it possible to solve the problem with exhaustive search?

Since $0 \leq x_i \leq 1$, then there are an infinite number of possibilities of $x_i$.

This problem is not discrete, but a continuous problem, so it is not possible to solve with exhaustive search.

# 5. Fractional knapsack problem (3)

**Question:** is it possible to solve the problem with the greedy approach?

**Example**

Given three objects as follows:

$$(w_1, p_1) = (18, 25);\ (w_2, p_2) = (15, 24);\ (w_3, p_3) = (10, 15)$$

and a knapsack of capacity $M = 20$.

| Object properties | | | | Greedy by | | |
|---|---|---|---|---|---|---|
| $i$ | $w_i$ | $p_i$ | $p_i/w_i$ | profit | weight | density |
| 1 | 18 | 25 | 1.4 | 1 | 0 | 0 |
| 2 | 15 | 24 | 1.6 | 2/15 | 2/3 | 1 |
| 3 | 10 | 15 | 1.5 | 0 | 1 | 1/2 |
| | Total weight | | | 20 | 20 | 20 |
| | Total profit | | | 28.2 | 31.0 | 31.5 |

# 5. Fractional knapsack problem (4)

**Theorem (Greedy by density gives an optimal solution)**

*If $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$, then the greedy algorithm with the strategy of choosing the maximum $\frac{p_i}{w_i}$ gives an optimal solution.*

**Proof.**

Homework! (give a similar taste of proof as for the "Activity Selector Problem") $\square$

**Algorithm:**

- Compute $\frac{p_i}{w_i}$ for $i = 1, 2, \ldots, n$
- For this strategy to work, the $\frac{p_i}{w_i}$'s are ordered in descending order.

# 5. Fractional knapsack problem (5)

```
procedure FractionalKnapsack(C: objects set, K: real)
    for i ← 1 to n do
        x[i] ← 0                                        ▷ x is the solution set
    end for
    i ← 0;  totalwt ← 0;  intFrac ← True    ▷ 'totalwt': total weight, and 'intFrac':
boolean var indicating if current object can be included fully
    while (i ≤ n) and intFrac do
        i ← i + 1
        if totalwt + w[i] ≤ K then
            x[i] ← 1                                    ▷ Include object i to knapsack
            totalwt ← totalwt + w[i]    ▷ Include the fraction of object i to total weight
        else
            intFrac ← False
            x[i] ← (K − totalwt) / w[i]    ▷ Only a fraction of object i can be included to the knapsack
        end if
    end while
    return x
end procedure
```

# 6. Huffman coding

**The principal of encoding and decoding**

Encoding/decoding is the translation of a message that is easily understood.

**Encoding**: the way any character is understood within the computer storage or transmission from one machine to another machine.

**Decoding**: the process of turning back an encoded message to the original message.

# 7. Huffman coding (2)

**Fixed-length versus Variable-length codes**

Fixed length encoding scheme uses a fixed number of bytes to represent different characters.

Variable length encoding scheme uses different number of bytes to represent different characters

Huffman coding is used for data compression.

**Fixed-length code**

Given a message of length 100,000 characters with frequency of a letter appears in the message as the following:

| **Character** | *a* | *b* | *c* | *d* | *e* | *f* |
|---|---|---|---|---|---|---|
| **Frequency** | 45% | 13% | 12% | 16% | 9% | 5% |
| **Encoding** | 000 | 001 | 010 | 011 | 100 | 111 |

**Example:** the encoding of 'bad' is 001000011

With this method, the encoding of 100,000 characters needs 300,000 bits.

# 6. Huffman coding (3)

Huffman coding is used for data compression.

**Fixed-length code**

Given a message of length 100,000 characters with frequency of a letter appears in the message as the following:

| **Character** | *a* | *b* | *c* | *d* | *e* | *f* |
|---|---|---|---|---|---|---|
| **Frequency** | 45% | 13% | 12% | 16% | 9% | 5% |
| **Encoding** | 000 | 001 | 010 | 011 | 100 | 111 |

**Example:** the encoding of 'bad' is 001000011

With this method, the encoding of 100,000 characters needs 300,000 bits.

**The principal of Huffman coding**:

- the more often a character appears, the shortest its *encoding*, and vice versa.

**Variable-length code (Huffman code)**

| **Character** | *a* | *b* | *c* | *d* | *e* | *f* |
|---|---|---|---|---|---|---|
| **Frequency** | 45% | 13% | 12% | 16% | 9% | 5% |
| **Encoding** | 0 | 101 | 100 | 111 | 1100 | 1100 |

**Example:** the encoding of bad is 1010111

With this method, the encoding of 100,000 characters needs:

$$(0.45 \times 1 + 0.13 \times 3 + 0.12 \times 3 + 0.16 \times 3 + 0.09 \times 4 + 0.05 \times 4) \times 10^5$$
$$= 224,000 \text{ bits}$$

Ratio of compression $= \frac{300{,}000 - 224{,}000}{300{,}000} \times 100\% = 25,5\%$.

# 6. Huffman coding (5)

- The greedy algorithm to form Huffman coding aims to minimize the length of binary code for all characters in the message $(M_1, M_2, \ldots, M_n)$.
- We build a weighted binary tree. Every *leave node* indicates the character in the message, and *internal nodes* indicate the merging of those characters.
- Every edge in the tree is given label 0 or 1 consistently (e.g.: left given '0' and right given '1').
- Minimizing the binary code for every character is equivalent to minimizing the length of path from the root to the leaves.

**Algorithm:**

1. Compute the frequency of every character in the message. Represent every character by a tree with a single node, and every node is assigned with the frequency of the corresponding character.
2. We apply the greedy strategy: at each step, merge two trees that have the smallest frequencies in a root. The new root has frequency equals to the sum of the frequencies of the two trees that composed it.
3. We repeat the 2nd step until we finally obtain a single Huffman tree. It forms a binary tree.
4. We label every edge of the tree by 0 or 1 (e.g. left-oriented edge is labeled 0 and right-oriented edge is labeled 1.
5. Every path from the root the each leaf of the tree represents the binary string for every character, with frequency as indicated on the corresponding leaf.

# 6. Huffman coding (7)

**What is the time complexity?**
$\mathcal{O}(n \log n)$.

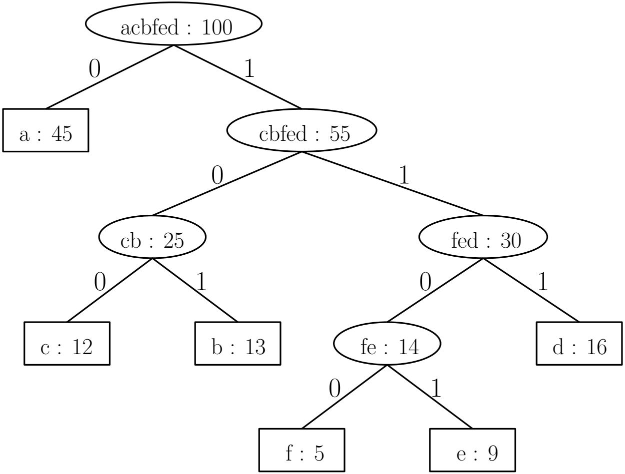
- Use a heap to store the weight of each tree, each iteration requires $\mathcal{O}(\log n)$-time to determine the cheapest weight and insert the new weight.
- There are $\mathcal{O}(n)$ iterations, one for each item.

**Exercise:** Given a message of length 100. The message is composed with letters $a, b, c, d, e, f$. The frequency of each letter in the message is as follows:

| **Character** | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ |
|---|---|---|---|---|---|---|
| **Frequency** | 45% | 13% | 12% | 16% | 9% | 5% |

Find the Huffman code for every character in the message.

# 6. Huffman coding (10)



## 6. Huffman coding (11)

**Huffman code**:

- a : 0
- b : 101
- c : 100
- d : 111
- e : 1101
- f : 1100