# GDSC: Algorithms & Data Structures

Sergey Kopelevich, Vladislav Artiukhov

February 1, 2024

## Contents

# 1. Big-O Notations

## 1.1 Definitions of $O, o, \Omega, \omega, \Theta$

## 1.2 Asymptotic types: linear, quadratic, polylog, exponential

For now refer to **GDSC Competitive Programming Abstract (topics 1-5), Fall 2023**.

# 2. Basic data structures

## 2.1 Arrays, doubly linked list, singly linked list

## 2.2 std::vector and how it works internally

## 2.3 stack, queue, deque

## 2.4 Keeping minimum for $O(1)$: min-stack, min-queue implementation

For now refer to **GDSC Competitive Programming Abstract (topics 1-5), Fall 2023**.

# 3. Master Theorem

## 3.1 Master Theorem

Algorithms that are written in a recursive manner oftentimes utilize *divide-and-conquer* technique which implies devision of the task into smaller subtasks that are processed by further recusive calls of the algorithm; once the subtask is small enough it is considered as a base case and processed manually. Some examples include **Merge sort algorithm**, **Binary search tree traversal**, etc.

For such algorithms we need to divide their asymptotics. **Master Theorem** is a generalized method that yields asymptotically tight bounds for divide and conquer algorithms [wiki].

**Theorem 3.1. Master Theorem**

Consider the following recurrence relation: $T(n) = a \cdot T(\frac{n}{b}) + f(n)$ where $f(n) = n^c$ for constants $a > 0, b > 1, c \geq 0$; let $k = \log_b n$ be the recursion depth. Then the following holds:

$$\begin{cases} T(n) = \Theta(a^k) = \Theta(n^{\log_b a}), \quad a > b^c \\ T(n) = \Theta(f(n)) = \Theta(n^c), \quad a < b^c \\ T(n) = \Theta(k \cdot f(n)) = \Theta(n^c \cdot \log n), \quad a = b^c \end{cases} \tag{1}$$

**Proof**.

$$T(n) = f(n) + a \cdot T(\frac{n}{b}) = f(n) + af(\frac{n}{b}) + a^2 f(\frac{n}{b^2}) + ... + a^k f(\frac{n}{b^k}) \quad | \ f(n) = n^c$$

$$T(n) = n^c + a \cdot (\frac{n}{b})^c + a^2 \cdot (\frac{n}{b^2})^c + ... + a^k \cdot (\frac{n}{b^k})^c$$

$$T(b) = n^c \cdot (1 + \frac{a}{b^c} + (\frac{a}{b^c})^2 + ... + (\frac{a}{b^c})^k)$$

Let $q = \frac{a}{b^c}$ and $S(q) = 1 + q + ... + q^k$:

1. If $q = 1$: $S(q) = 1 + 1 + ... + 1 = k + 1 = \log_b n + 1 \implies T(n) = \Theta(f(n) \cdot \log n)$.

2. If $q < 1$: $S(q)$ is a geometric progression, thus it is equal to $S(q) = \frac{1-q^{k+1}}{1-q} = const = \Theta(1) \implies T(n) = \Theta(f(n))$.

3. If $q > 1$: $S(q) = q^k + \frac{q^k-1}{q-1} = \Theta(q^k) \implies T(n) = \Theta(a^k \cdot (\frac{n}{b^k})^c) = \Theta(a^k)$.

**Note**: $f(n)$ could be $O(n^c)$; it does not violate the proof ($f(n) = O(n^c) = C \cdot n^c$).

□

## 3.2 Generalized Master Theorem

**Theorem 3.2. Generalized Master Theorem**

In the case of $f(n) = n^c \cdot \log_d n$ Master Theorem still holds:

$T(n) = a \cdot T(\frac{n}{b}) + n^c \cdot \log_d n, \ a > 0, \ b > 1, \ c \geq 0, \ d \geq 0.$

$$\begin{cases} T(n) = \Theta(a^k) = \Theta(n^{\log_b a}), \quad a > b^c \\ T(n) = \Theta(f(n)) = \Theta(n^c \cdot \log^d n), \quad a < b^c \\ T(n) = \Theta(k \cdot f(n)) = \Theta(n^c \cdot \log^{d+1} n), \quad a = b^c \end{cases} \tag{2}$$

## 3.3 Algorithm for recurrence relations

There are also recurrence relations with the following form:

$T(n) = a_0 \cdot T(n - p_0) + a_1 \cdot T(n - p_1) + ... + a_k \cdot T(n - p_k) \quad a_i, p_i > 0,\ \sum p_i > 1$

There exists an algorithm of how to find asymptotics for such relations:

**Theorem 3.3. Algorithm for recurrence relations**

Given $T(n)$ of the above form with the above constants, then the following holds:

$T(n) = \Theta(\alpha^n)$, such that $\alpha > 1$ and it is the **only root** of the equation: $\alpha^n = a_0 \cdot \alpha^{n-p_0} + ... + a_k \cdot \alpha^{n-p_k}$

**Example. Use of Master Theorem**

1. $T(n) = 4 \cdot T(\frac{n}{2}) + 20 \cdot n^{\frac{3}{2}}$

$a = 4,\ b = 2,\ c = \frac{3}{2},\ f(n) = 20 \cdot n^{\frac{3}{2}} \implies a = 4 > b^c = \sqrt{8} \implies T(n) = \Theta(n^{log_b a}) = \Theta(n^2).$

2. Merge sort algorithm recurrence relation: $T(n) = 2 \cdot T(\frac{n}{2}) + C \cdot n^1$

$a = 2,\ b = 2,\ c = 1 \implies a = 4 = b^c = 2^1 \implies T(n) = \Theta(n^1 \cdot \log n)$

**Example. Use of Algorithm for recurrence relations**

1. $T(n) = T(n - 1) + 6 \cdot T(n - 2)$

$T(n) = \Theta(\alpha)$, notice that $\alpha = 3$ satisfies the equation: $3^n = 3^{n-1} + 6 \cdot 3^{n-2}$.

2. $T(n) = T(n - 1) + T(n - 2) + T(n - 3)$

$1 = \alpha^{-1} + \alpha^{-2} + \alpha^{-3} \implies \alpha \approx 1.839$

# 4. Amortized Analysis

## 4.1 Definition and general perception of the concept

Remember that we were talking about **std::vector** we said that its **push_back** operation works for an average of $\Theta(1)$. It was due to presence of 2 **distinct states**:

1. vector has enough capacity to fit the next pushed element.
2. vector does not have enough capacity and has to make itself twice bigger (i.e. reallocating a memeory buffer of size $2 \cdot N$).

There are definetely more complicated senarios where number of such *interesting states* is much greater, thus we need a unified approach of how to define this average, or **amortized**, time for an operation.

***Definition* 4.1. The amortized analysis**

The amortized analysis is an approach that allows to determine an average running time (time complexity) of operations $o_1$, $o_2$, ..., $o_k$ in a sequence $S$ over that sequence $S$.

There are several methods that are referred to as amortized analysis (wiki). We are going to discuss **Potential method**.

## 4.2 Amortized analysis: Potential method

***Definition* 4.2. Potential method**

Introduce a **potential function** called $\Phi : \mathbf{S} \to \mathbf{R_0^+}$ where $\mathbf{S}$ is a set of states of the considered data structure and $R_0^+ = [0, +\inf)$, i.e. the potential function maps states of the data structure to some non-negavite values. As an important edge case for the initial state $S_{init}$: $\Phi(S_{init}) = 0$.

Let $o_i$ be an individual operation within some sequence of operations named $Q$. Let $S_{i-1}$ be the state of the considered data structure before the execution of the operation $o_i$ and $S_i$ be the state after the execution of $o_i$. Let $\Phi$ be a chosen potential function, then the amortized time for an operation $o_i$ is defined as follows:

$T_a(o_i) = T_r(o_i) + (\Phi(S_i) - \Phi(S_{i-1}))$ where:

1. $T_a(o)$ - amortized time of the operation.
2. $T_r(o)$ - real/actual time spent on the operation.

**Theorem 4.1. Potential method yields an upper bound**

The amortized time of a sequence of operations always yields an **upper bound** of the the real/actual time for the considered sequence of operations, i.e.:

$\forall O = o_1, o_2, ..., o_n$ be a sequence of operations, define:

1. $T_a(O) = \Sigma_{i=1}^{n} T_a(o_i)$ - amortized time of the sequence $O$
2. $T_r(O) = \Sigma_{i=1}^{n} T_r(o_i)$ - real time of the sequence $O$

Then: $T_r(O) \leq T_a(O)$

**Proof**. As definition for $T_a(O)$ suggests:

$$T_a(O) = \Sigma_{i=1}^{n}(T_r(o_i) + \Phi(S_i) - \Phi(S_{i-1})) = T_r(O) + \Phi(S_n) - \Phi(S_0)$$
$$T_a(O) = T_r(O) + \underbrace{\Phi(S_n)}_{\cdot \geq 0} - \underbrace{\Phi(S_0)}_{\cdot = 0} \geq T_r(O)$$

□

Note: Generally speaking, $\Phi$ could be any function that satisfies the constraints but the idea is to find such $\Phi$ that would represent the closest upper bound of the real/actual time for the considered sequence of operations.

**Example. Amortized analysis for std::vector::push_back operation**

1. Analyse push_back operation that expands the vector:

Let $\Phi = 2 \cdot s - N$ where the $s$ is the actual size of a vector and $N$ is its capacity. Remember that the expansion occurrs once $s == N$.

Let's consider a push_back operation that doubles the size of the vector:

1. $T_r = s + 1 = N + 1$ - because we need to copy all the existing elements in a new buffer + place a new element; here $s = N$ because the extention could only happen once $s = N$.

2. $\Phi_0 = 2 \cdot s - N \geq 0$ - value of the potential function before the operation.

3. $\Phi_1 = 2 \cdot (s + 1) - 2N \geq 0$ - value of the potential function after the operation.

Thus, we have:

$T_a = T_r + \Phi_1 - \Phi_0 = N + 1 + (2s + 2 - 2N - 2s + N) = N + 1 + (2 - N) = N - N + 3 = \Theta(1)$.

2. Analyse push_back operation that does not expand the vector:

Notice that in this case $\Delta\Phi = \Phi_1 - \Phi_0 = const$ and $T_r = const$, thus $T_a = const = \Theta(1)$.

For more examples check the wiki page.

# 5. Binary Search

## 5.1 Definition, use cases

## 5.2

## 5.3 STL implementation