GDSC: Algorithms & Data Structures

Vladislav Artiukhov, based on materials of Sergey Kopelevich

February 11, 2024

Contents

1. Preamble

1.1 Credits

The further work is mostly based on the **Algorithms & Data Structures course** held by Professor **Sergey Kopelevich** in **Higher School of Ecomonics, Saint Petersburg**, **Applied Mathematics & Computer Science Bachelor's Program**, 1-3 semesters, 2021-2022.

<u>**The materials**</u>:

1. SPb HSE, 1st-2nd semesters, Fall 2021/22, Algorithms Lectures Abstract
2. SPb HSE, 3rd semester, Fall 2021/22, Algorithms Lectures Abstract

Many thanks to Professor **Sergey Kopelevich**!

2. Big-O Notations

2.1 Definitions of $O, o, \Omega, \omega, \Theta$

2.2 Asymptotic types: linear, quadratic, polylog, exponential

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

3. Basic data structures

3.1 Arrays, doubly linked list, singly linked list

3.2 std::vector and how it works internally

3.3 stack, queue, deque

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

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

4. Master Theorem

4.1 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 define their asymptotics. **Master Theorem** is a generalized method that yields asymptotically tight bounds for divide and conquer algorithms [wiki].

Theorem 4.1. 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:

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

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 ($f(n) = O(n^c) = C \cdot n^c$).

□

4.2 Generalized Master Theorem

Theorem 4.2. Generalized Master Theorem

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

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

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

4.3 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 4.3. 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. $T(n) = T(n-1) + T(n-2) + T(n-3)$

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

5. Amortized Analysis

5.1 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 memory 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* 5.1. 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**.

5.2 Amortized analysis: Potential method

***Definition* 5.2. 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 Φ 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 5.1. 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, Φ could be any function that satisfies the constraints but the idea is to find such Φ 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.

6. Binary Search

6.1 Definition, use cases

6.1.1 Simplest version

Given a sorted array of size n. We need to find and element x in this array for $O(\log n)$:

```
int find(int l, int r, int x) { // [l,r]
    // a is sorted in the ascending order
    while(l <= r) {
        int m = (l + r) / 2;
        if (a[m] == x) return m;
        else if (a[m] < x) l = m + 1;
        else r = m - 1;
    }
    return -1; // i.e., not found
}
```

The time complexity is indeed $O(\log n)$ since on each iteration the $|r-l|$ decreases its value in a half ($|r-l| \to \frac{|r-l|}{2}$).

The problem of such an implementation is that if there are multiple occurrences of x in the array, the algorithm will return **some index** in a range $[l_{ans},\ r_{ans})$ where $\forall i = l_{ans}, ..., r_{ans} - 1 :\ a[i] == x$, although we would like to get either of both sides of the range, i.e. either l_{ans} or r_{ans}.

6.1.2 Lower bound / Upper bound

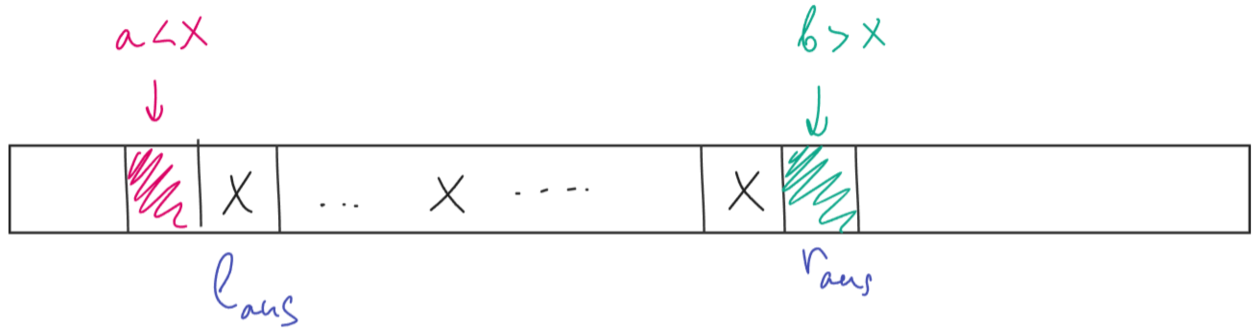
The above problem leads to the following implementations of the binary search algorithm that searches for the forementioned indicies:

1. $\min i : a_i \geq x$, i.e. l_{ans}:

```
int lower_bound(int l, int r, int x) {
    while (r - l > 1) {
        int m = (l + r) / 2;
        if (a[m] < x) l = m; // Notice that we keep the invariant: a[l] < x
        else r = m; // => a[r] >= x
    }
    // (a[l] < m) && (a[r] >= m) => r is a minimum index
    return r;
}
```

2. $\min i : a_i > x$, i.e. r_{ans}:

```
int upper_bound(int l, int r, int x) {
    while(r - l > 1) {
        int m = (l + r) / 2;
        if (a[m] > x) r = m;
        else l = m;
    }
    // Now a[r] > x => l is the last index for which a[l] <= x
    return r;
}
```



6.1.3 STL implementation

In C++ language we are already provided with the template functions that do the same thing:

```
/* returns iterator (for int* it is a pointer) */
int i = std::lower_bound(a, a + n, x) - a;
int i = std::upper_bound(a, a + n, x) - a;

// For other containers (e.g., std::vector, std::set) that define begin()/end() operations use the following:
std::lower_bound(
    std::begin(container),
    std::end(container),
    element
) //  e.g., for std::vector<T> the return type is std::vector<T>::iterator, i.e. pointer to the element
```

6.2 Use of a predicate

We could abstract the implementation even further via searching for a predicate. Predicate is such a function $f : S \to \{0, 1\}$. Then let's consider the predicate $f(i) = if(a_i < x)then0else1$ - in this case the binary search will find such indicies l and r that satisfy the following:

1. $l + 1 = r$
2. $f(l) = 0$ (i.e. $a_l < x$)
3. $f(r) = 1$ (i.e. $a_r >= x$)

```
int binary_search(int l, int r, int x) {
    while(r - l > 1) {
        int m = (l + r) / 2;
        if (f(m)) r = m; // invariant: a[r] >= x
        else l = m; // invariant a[l] < x
    }
    return r;
}

// If you want to parameterize predicate as well:
// #1: provide a 4th argument as a pointer to a function
// (see: https://www.cprogramming.com/tutorial/function-pointers.html)
int binary_search(int l, int r, int x, bool(*f)(int)) {
    ...
}

// #2: template parameter
template<typename /* or class */ Func>
```

```
int binary_search(int l, int r, int x, Func f) {
    ...
}
// possible call:
binary_search(l, r, x, []() { /* predicate impl */ });
```

6.3 Correctness

You might already noticed that the functions (arrays are also akin functions, i.e. $a : \{0, 1, .., n-1\} : \mathbb{N}$) over which we apply the binary search algorithm are all monotonic functions, i.e. they comply to $x < y : \ f(x) < f(y)$ (strict monotonically increasing functions; the rest are alike).

Lemma. *The binary search algorithm over some range [x, y) is correct iff the considered function f is monotonic over [x, y).*

6.4 Binary search for functions over $\mathbb{R}$

It is possible to use the binary search with real numbers as well. Let's consider a problem of *finding square root of x*:

Problem statement: Given $x \in \mathbb{R}$. Find a value $y \in \mathbb{R}$ so that $y^2 == x$ (for us it is $|y^2 - x| < \varepsilon$):

```
double my_sqrt(double x /* never use 'float' */) {
    double l = 0.0;
    double r = x + 1;

    const dobule EPS = 1e-9; // 10^(-9)

    while(r - l > EPS) {
        double m = (r + l) / 2;
        // Preserve invariant: l^2 <= x, r^2 > x
        if (m*m > x) r = m;
        else l = m;
    }

    return (l + r) / 2;
}
```

In the above notice: if $0 < y < 1$ then $y^2 < y$, and if $y \geq 1$ then $y^2 \geq 1$. Thus, for $0 < x < 1$ the corresponding y is greater than x, i.e. we select $r = x + 1$.

Example. Root of a polynomial $\mathbf{P(x)}$

Given a polynomial $P(x)$ of **an odd degree**, i.e. $\deg P = 2k + 1$ with the coerfficient for x^{2k+1} be equal to 1. There exists a root $x_0 \in \mathbb{R}$, and we need to find it.

Solution:

We could do that using binary search with any precision ε. First, we need to find points l and r, such that: $P(l) < 0$ and $P(r) > 0$ (e.g., $l = -\infty, \ r = +\infty$ - **MAX_INT** and **MIN_INT** in C++):

```
    for (l = -1; P(l) >= 0; l *= 2);
    for (r = 1 ; P(r) <= 0; r *= 2);
```

And finally the root search:

```
        while (r - l > EPS) {
            double m = (l + r) / 2;
            if (P(m) < 0) l = m; // P(l) < 0
            else r = m; // P(r) >= 0
```

```
    }
    return (l + r) / 2;
```

Actually we need exactly $k := \frac{r-l}{\varepsilon}$ iterations, thus the while-loop could be changed by the for-loop.

7. Ternary Search

7.1 Definition, use cases

Imagine that we need to find a maximum of a function $f(x)$ in the interval $[l, r]$ but now the function is no longer monotonic. Is there a fast way (i.e. as fast as logarithmic asymptotic as in the binary search algorithm) of finding a point x_0 such that $f(x_0) \to \max$?

The answer is yes if our function satisfies some constraint.

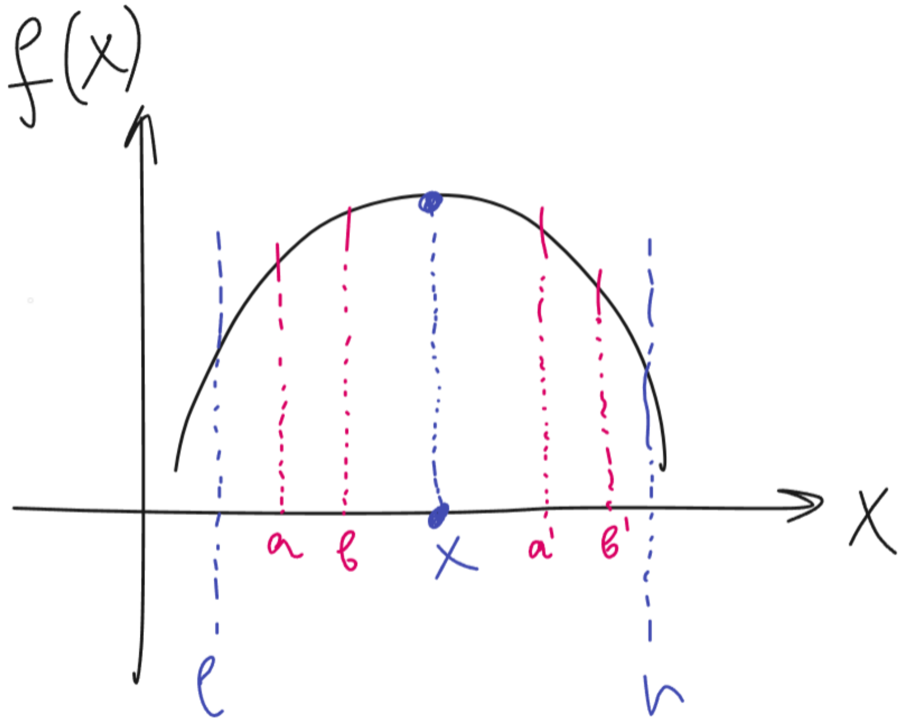
***Definition* 7.1. Ternary search algorithm**

Given a function $f(x)$ and a range $[l, r]$ in which the function is unimodal and convex, and we need to find maximum in the range (maximum for convex functions, minimum for concave functions).

Since the function is unimodal over $[l, r]$ the following holds:

1. $\forall a, b : l \leq a < b \leq x_0 : \ f(a) < f(b)$
2. $\forall a, b : x_0 \leq a < b \leq r : \ f(a) > f(b)$

Where x_0 is a point where $f(x) = \max$

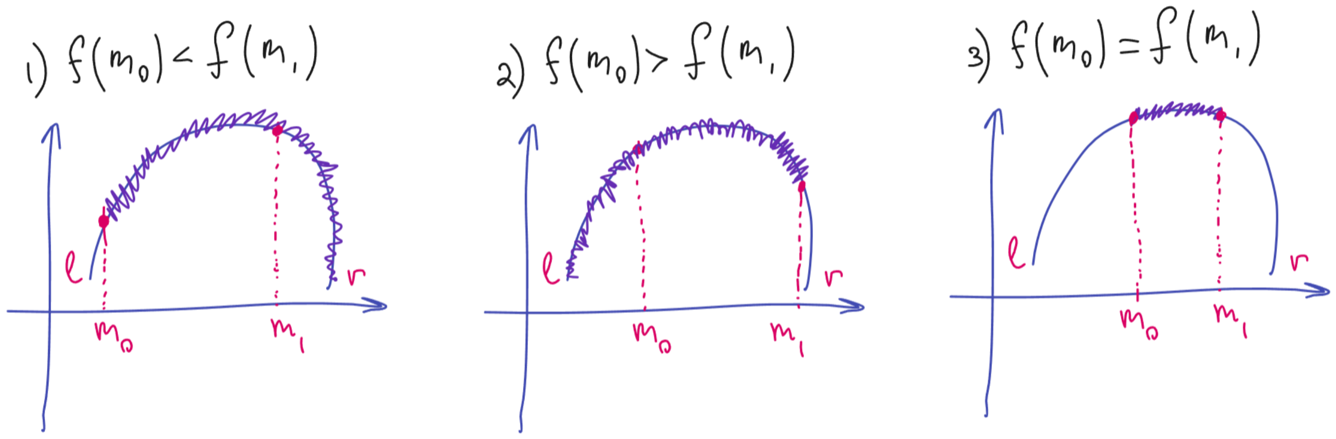


For the algorithm let's consider two points m_0 and m_1 such that $l < m_0 < m_1 < r$, there are several cases:

1. $f(m_0) < f(m_1)$, then the maximum of the function f cannot be located in the interval $[l, m_0]$ since there is an interval $[m_0, r]$ on which there are points where function takes greater values.

2. $f(m_0) > f(m_1)$, then the maximum of the function f cannot be located in the interval $[m_1, r]$ since the interval $[l, m_1]$ contains points in which the function takes greater values.

3. $f(m_0) = f(m_1)$, in this case the maximum will be located in the interval $[m_0, m_1]$ but for the implementation simplisify this condition branch can be merged with any of the above.



As for the choice of the points m_0 and m_1 we need to select something that would trancate the remaining seraching area in a manner that would give us a logarithmic asymptotic. Let's make m_0 be the first 3rd of the interval and m_1 be the second 3rd of the interval:

1. $m_0 = l + \frac{r-l}{3}$
2. $m_1 = l + 2 \cdot \frac{r-l}{3} = \frac{3l+2r-2l}{3} = \frac{2r+l}{3} = \frac{3r-r+l}{3} = r - \frac{r-l}{3}$

Notice that according to the above statement the searching interval becomes $\frac{2}{3}$ of the initial interval, thus:

$T(n) = T(\frac{2}{3}n) + 1$

According to **Master Theorem**:

$a = 1,\ b = \frac{2}{3}, c = 0,\ f(n) = n^c = 1 \implies a = b^c \implies \Theta(n^c \cdot \log n) = \Theta(\log n) \implies$
$T(n) = T(\frac{2}{3}n) + 1 = \Theta(\log n)$

7.2 Implementation

Here is the implementation for a function which is an array:

```
int ternary_search(vector<int>& arr) {
    // 'arr' is a unimodal function N -> Z
    int l = 0;
    int r = arr.size();

    while(r - l > 2 /* not 1 because m0 = l, m1 = r possible */) {
        int m0 = l + (r - l) / 3;
        int m1 = r - (r - l) / 3;

        if (arr[m0] < arr[m1]) {
            l = m0;
        }
        else {
            r = m1;
        }
    }

    int ans = l;

    if (r < arr.size() && arr[r] > arr[ans]) {
        ans = r;
    }

    return ans;
```

```
}
```

Here is the implementation for a function f over floating-point numbers:

```
int ternary_search(double l, double r, double (*f)(double)) {
    while (r - l > EPS) {
        double m0 = l + (r - l) / 3;
        double m1 = r - (r - l) / 3;

        if (f(m0) < f(m1)) {
            l = m0;
        }
        else {
            r = m1;
        }
    }

    return (l + r) / 2;
}
```

8. Two pointers and Set operations

8.1 What a set is

We are going to understand a set as a data structure which contains **distinct** elements (otherwise **multiset**) sorted in the **ascending order**.

STL library contains a data structure that complies to the above definition: **std::set<T>**. In the following subsections we are going to look at fundamental operations over sets. We assume that every time when we are given a set, it is an ascendingly sorted array of distinct elements.

8.2 Intersection of sets

Given two sets A and B and we want to create a set $C = A \cap B$ in $O(|A| + |B|)$:

```
vector<int> intersection(vector<int> A, vector<int> B) {
    int i = 0;
    int j = 0;
    vector<int> C;

    for (; i < |A| && j < |B|; ++j) {
        while (i < |A| && A[i] < B[j]) ++i;
        // last condition means "either 1st element of B or a distinct element"
        // condition can be removed since all elements of bith A and B are distinct
        if (i < |A| && A[i] == B[j] && (j == 0 || B[j - 1] != B[j])) {
            C.push_back(B[j]);
        }
    }

    return C;
}
```

The above algorithm can be extended to an intersection of an arbitrary number of sets $A_0, A_1, ..., A_k$. You can try to do that in the following LeetCode problem: 2248. Intersection of Multiple Arrays.

8.3 Union of sets

Given two sets A and B, we want to find $C = A \cup B$ in $O(|A| + |B|)$:

```
vector<int> union(vector<int> A, vector<int> B) {
    int i = 0;
    int j = 0;
    vector<int> C;

    for (; j < B.size(); ++j) {
        while (i < A.size() && A[i] <= B[j]) {
            C.push_back(A[i++]);
        }
        // in the case if last added element A[i] is equal to B[j]
        if (C.empty() || C.back() != B[j]) {
            C.push_back(B[j]);
        }
    }

    while (i < A.size()) {
        C.push_back(A[i++]);
    }

    return C;
}
```

8.4 Difference of sets

Given two sets A and B, we want to find $C = A \setminus B$ in $O(|A| + |B|)$:

```
vector<int> difference(vector<int> A, vector<int> B) {
    int i = 0;
    int j = 0;
    vector<int> C;

    for (; i < A.size(); ++i) {
        // skipping elements less than A[i]
        while(j < B.size() && B[j] < A[i]) {
            ++j;
        }
        // if A[i] not present in B add A[i] to the answer
        if (j >= B.size() || B[j] != A[i]) {
            C.push_back(A[i]);
        }
    }

    return C;
}
```

You may solve this task by yourself of LeetCode: 2215. Find the Difference of Two Arrays

8.5 STL implementations

STL already has implementations of all the operations mentioned above: std::set_difference, std::set_union, std::set_intersection, and std::merge (the latter one is the same as std::set_union but it does not remove the duplicate elements). All the mentioned STL functions are called in the following manner:

```
int k = std::merge(A, A + |A|, B, B + |B|, C) - C;
// C - pointer to where to store the result (memory allocation is your responsibility)
// k - number of elements in the result, i.e. k == |C|
```