GDSC: Algorithms & Data Structures

Vladislav Artiukhov, based on materials of Sergey Kopelevich

March 22, 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.

The materials:

1. SPb HSE, 1st-2nd semesters, Fall 2021/22, Algorithms Lectures Abstract
2. SPb HSE, 1st-2nd semesters, Spring 2021/22, Algorithms Lectures Abstract
3. 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 $R_0^+ = [0, +\infty)$, 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:

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

$$T_a(O) = T_r(O) + \underbrace{\Phi(S_n)}_{\cdot \geq 0} - \underbrace{\Phi(S_0)}_{\cdot = 0} \geq T_r(O)$$

Thus, the amortized time provides an accurate upper bound on the actual time of a **sequence of operations**, even though the amortized time for an individual operation may vary widely from its actual time.

□

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

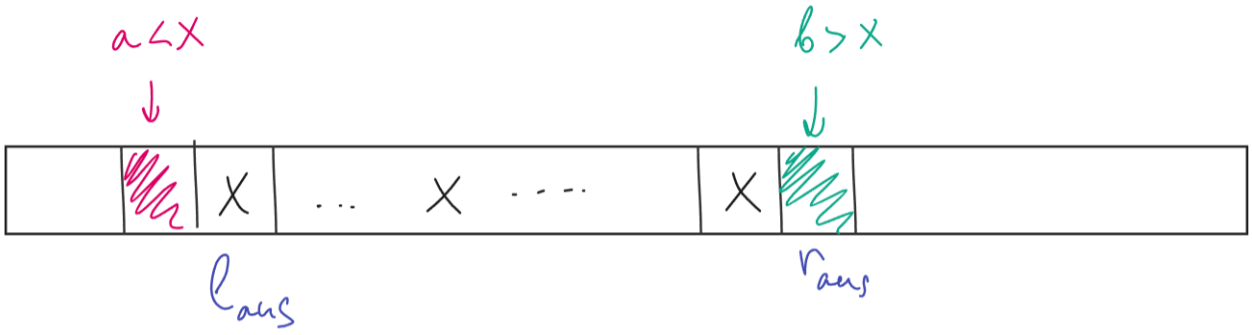
The above problem leads to the following implementations of the binary search algorithm that searches for the forementioned indicies:

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:

```cpp
/* 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 $f(i) = if(a_i < x)\ then\ 0\ else\ 1$ - 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. $a_r >= x$)

```cpp
int binary_search(int l, int r, int x) {
    while(r - l > 1) {
        int m = (l + r) / 2;
        if (f(m)) r = m; // invariant: f(r) >= x
        else l = m; // invariant f(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 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
```

```
5           }
6           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.

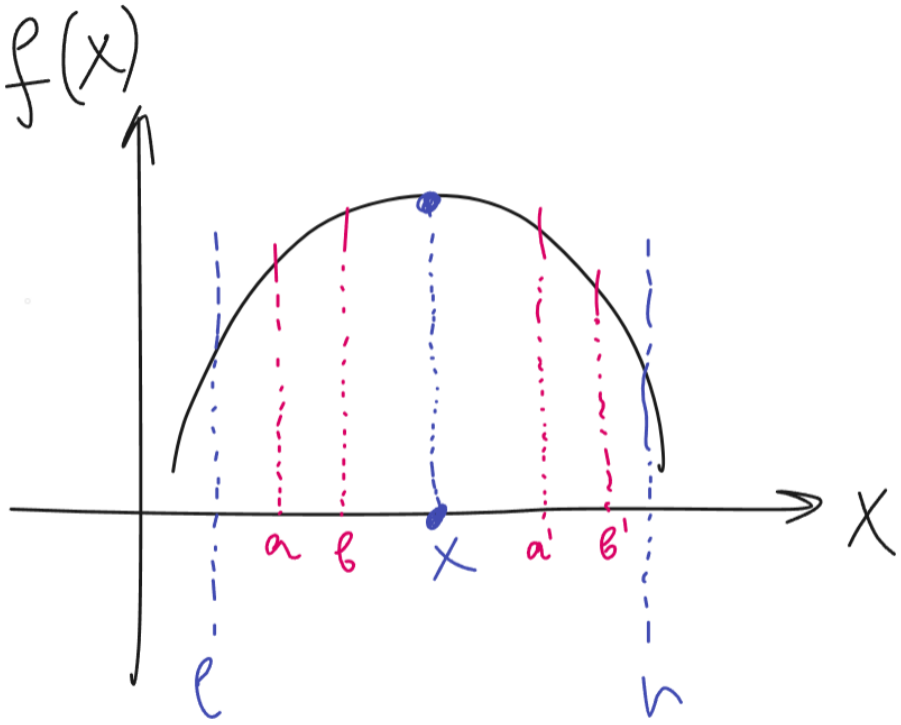
***Definition* 7.1. Ternary search algorithm**

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 $f(x) = \max$

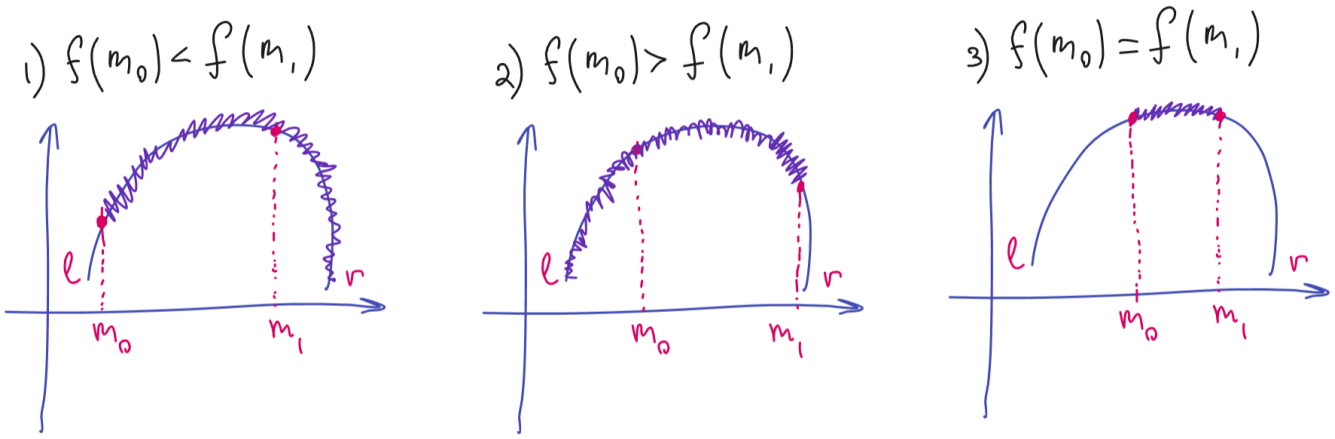


For the algorithm let's consider two points m_0 and m_1 such that $l < m_0 < m_1 < r$, there are several cases:

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 searching 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:

```
1  int ternary_search(vector<int>& arr) {
2      // 'arr' is a unimodal function N -> Z
3      int l = 0;
4      int r = arr.size();
5
6      while(r - l > 2 /* not 1 because m0 = l, m1 = r possible */) {
7          int m0 = l + (r - l) / 3;
8          int m1 = r - (r - l) / 3;
9
10         if (arr[m0] < arr[m1]) {
11             l = m0;
12         }
13         else {
14             r = m1;
15         }
16     }
17
18     int ans = l;
19
20     if (r < arr.size() && arr[r] > arr[ans]) {
21         ans = r;
22     }
23
24     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|
```

8.6 Problem example

9. Hash table

9.1 Problem statement

We are given an empty set of integers, and we want to be able to quickly add, remove, find elements.

Approach 1: The easiest (and slow) implementation could be to store all the elements in std::vector, then:

1. add = push_back = $O(1)$.
2. find = $O(n)$.
3. remove = find + $O(1)$ (swap the found element with the last one and perform pop_back).

Approach 2 (with additional constraint): if all the elements are in the range of $[0, N)$, then we can allocate an array **is[N]**; then $is[x]$ indicates whether the element x is present in the set or not, in this case all the required operations will work in $O(1)$.

Ideal solution: the hash table is a data structure that allows to perform add, remove, and find in randomized $O(1)$.

9.2 Hash table via std::list

```
1  std::list<int> table[N];
2
3  void add(int x) {
4      table[x % N].push_back(x); // O(1)
5  }
6
7  auto /* = std::list<int>::iterator */ find(int x) {
8      const auto& list = table[x % N];
9      return std::find(list.begin(), list.end(), x); // works for O(list.size())
10 }
11
12 void remove(int x) {
13     table[x % N].erase(find(x)); // works for find + O(1)
14 }
```

Instead of std::list you can use any container-like data structure, e.g., std::vector.

If there are n elements present in the set and all of them are uniformly distributed, then each list will have on average $\frac{n}{N}$ elements $\implies$ if $n \leq N +$ elements are uniformly distributed, then all the operations will work in $O(1)$.

The uniform distribution of the elements is something we can potentially adjust via selecting a good hash function $f(x)$ for the elements (e.g., in the above we have used $f(x) = x \mod N$).

9.3 Hash functions with probability of collision $O(\frac{1}{m})$

Statement 9.1. Functions $f(x) = x \mod N$ where N is a prime number are good enough.

In the following theorem we will show that there exist a special type of functions whose probability of collision is $O(\frac{1}{m})$ where m is the capacity of the underlying hash table.

***Definition* 9.1. Universal family of hash functions**

Given a universe of elements U.

Let a hash function be defined as: $h : U \to \{0, 1, 2, ..., m-1\}$ for some m, or $h : U \to [m]$ ($[m] := \{0, 1, 2, ..., m-1\}$).

$H = \{h : U \to [m]\}$ is called a **universal family** if:

$\forall x, y \in U: \ x \neq y: \ |S| \leq \frac{|H|}{m}$ where $S := \{h \in H : h(x) = h(y)\}$. Perceive the $|S|$ as the number of hash functions that collide x and y, then the inequation can be rewritten as:

$\forall x, y \in U: \ x \neq y: \ \frac{|S|}{|H|} \leq \frac{1}{m}$ where $\frac{|S|}{|H|}$ is the probability of collision of x and y for the universal family H.

"In other words, any two different keys of the universe collide with probability at most $\frac{1}{m}$ when the hash function h is drawn uniformly at random from H." (wiki).

Theorem 9.2. Integer hashing

Let's consider a case where the values from the universe U are finite (e.g., integer numbers in 64-bit computers), i.e. $U := \{0, 1, 2, ..., |U| - 1\}$.

Select a prime p which is $p \geq |U|$ and define:

$h_{a,b}(x) := ((ax + b) \mod p) \mod m$ where a, b are randomly chosen integers modulo p, and $a \neq 0$.

The set $H := \{h_{a,b}, \forall a \neq 0, b\}$ is a **universal family**, i.e.:

$\forall x, y \in U : x \neq y: \ Pr[\text{collision of } x \text{ and } y \text{ occurred}] \leq \frac{1}{m}$.

Proof. Notice that the collision occurrs once $h_{a,b}(x) = h_{a,b}(y)$, $x \neq y$ which holds only when:

$(ax + b)\%p \equiv_m (ay + b)\%p$

$ax + b \equiv_p ay + b + im$ where $im \leq p - 1$ because the right part of the equation is in $\{0, 1, 2, ..., p - 1\}$, thus $i = 1, 2, ..., \frac{p-1}{m}$.

$a(x-y) \equiv_p im$ since $x \neq y$, $x, y \in U = \{0, 1, ..., |U|-1\}$ and $p \geq |U|$, and p - prime, then $(x-y) \not\equiv_p 0$, thus the $(x - y)^{-1}$ exists:

$a \equiv_p im \cdot (x - y)^{-1}$ - collision happens once the right and the left parts are equal. For every $i = 1, 2, ..., \frac{p-1}{m}$ the equality takes place in exactly single value of a because $a = 1, 2, ..., p - 1$, thus the probability of the collision is $Pr = \frac{\#i}{\#a} = \frac{\frac{p-1}{m}}{p-1} = \frac{1}{m}$.

Thus, we have $Pr[\text{collision of } x \text{ and } y \text{ occurred}] \leq \frac{1}{m} \implies H$ is a universal family.

□

9.4 Open addressing collision resolution

The implementation utilizes the cyclic array. A hash function is used to get the initial position in the array of the element x, next we move strictly right (modulo the capacity of the cyclic array) until we find a position in which either no element present or the element x located:

```
1  int h[N]; // hash table
2
3  // h[i] = 0  : an empty cell
4  // h[i] = -1 : deleted element
5  // h[i] > 0  : there is an element
6
7  int getIndex(int x) { // get index by an element, require x > 0
8      int i = x % N;
9      while (h[i] && h[i] != x) {
10         i = (i + 1) % N;
11     }
12     return i;
13 }
14
15 void add(int x) {
16     h[getIndex(x)] = x;
17 }
```

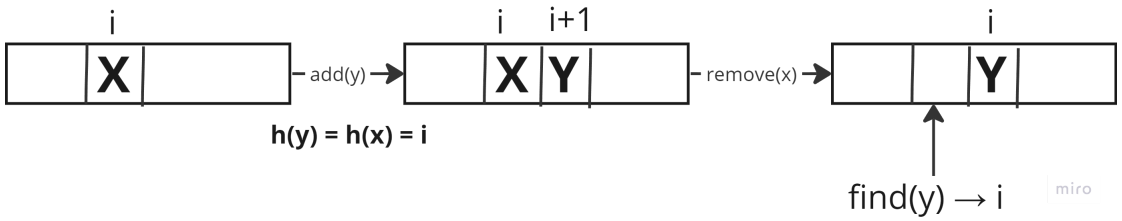
```
18
19 void remove(int x) {
20     // notice that the cell does not become empty!
21     // marking a cell as empty will break the algorthm! see below
22     h[getIndex(x)] = -1; // requires x != -1
23 }
24
25 int find(int x) {
26     return h[getIndex(x)] != x;
27 }
```

Why marking cells as removed is important:

Suppose we mark a cell as **empty** (not **removed**) once element is removed, then let's consider the following scenario:

You have 2 elements x and y which are about to be added into the hash table T, and it turns out that the hash function h over the elements results in $h(x) = h(y)$. Let's execute the following sequence of operations:

1. Suppose T is empty.
2. $add(x)$ - adds x at the position i.
3. $add(y)$ - adds y at the position $i + 1$ because $h(x) = h(y)$, thus the algorithm will traverse right until a free cell is found.
4. $remove(x)$ - marks the position i as empty.
5. $find(y)$ - returns **false** because $h(y) = h(x) = i$, the cell $T[i]$ is marked to be empty, thus the find method says no y found in the T.



Lemma. If the hash table with open addressing of size N has αN occupied cells ($\alpha < 1$), then the mathematical expectation E of the time of **getIndex** $\leq \frac{1}{1-\alpha}$.

For example, if there are only half of the elements occupied, i.e. $\alpha = \frac{1}{2}$, then on average each operation will require $\leq 2 = O(1)$ of time.

Proof. Suppose that the free cells are distributed uniformly due to the hash function being good enough (no proof for it; oftentimes holds if you do not encounter an anti-test for your solution where all elements would have the same hash value).

The worst case is when the element x is not present in the table. In this case, on each iteration of the **while**-loop of **getIndex** the probability to **not stop** equals to $\frac{\alpha \cdot N}{N} = \alpha$ (due to the supposed uniform distribution).

The probability to not stop even after k iterations is α^k, i.e. we will make k-th step with the probability of α^k. Then, the execution time will be:

T = mathematical expectation E of the number of steps made =

$= 1 + \sum_{k=1}^{+\infty} Pr[\text{the } k\text{-th step was made}] = 1 + \alpha + \alpha^2 + \alpha^3 + ... = \frac{1}{1-\alpha}$

□

9.4.1 Hash table overflow

If α becomes too big the operations start to work slower:

1. $\alpha = 1$: no free cells, **getIndex** will infinitely search for a free cell. How to resolve this problem?

Once $\alpha > \frac{2}{3}$ let's double the size of the underlying cyclic array and in linear time insert the elements into the new allocated buffer. During copying we will of course skip the cells marked as removed (i.e. $T[i] = -1$) $\implies$ the removed cells occupy the additional space only until the next size doubling takes place (i.e. there will be no bloating with removed cells inside the hash table).

9.5 Comparison of the approaches

Overall, we have discussed two types of the hash table, let's compare them in terms of the required memory and time:

Suppose we store x bytes for an object (element) and a pointer requires 8 bytes, then our hash tables use for n elements:

1. List-based approach (exactly n lists): $8n$(pointers to n lists) $+ n \cdot (x + 8) = n \cdot (x + 16)$ bytes ($x + 8$ because every list has an element and a pointer to next).

2. Open addressing approach (having an additional capacity of $\frac{3}{2}$): $\frac{3}{2} \cdot n \cdot x$ bytes.

Time complexity:

1. List-based solution: requires to make on average 2 lookups, one for a list and one for an element inside the selected list.

2. Open addressing solution: requires on average 1 lookup.

9.6 STL implementation

STL implementation uses list-based approach, thus if you write your own solution with open addressing it will work almost twice faster on average.

1. **std::unordered_set<int> h** - hash table that stores the set of integers.

Use:

1. **std::unordered_set<int> h(N)** - create a hash table with the capacity of N cells.

2. **h.count(x)** - check whether x present.

3. **h.insert(x)** - add an element x (no-op if already present).

4. **h.erase(x)** - remove an element x (no-op if element not present).

2. **std::unordered_map<int, int> h** - hash table that stores **std::pair<int, int>**.

Use:

1. **std::unordered_map<int, int> h(N)** - create a hash table with the capacity of N cells.

2. **h[i] = x** - i-th cell can be used as an array index, stores an association $i \to x$.

3. **h.count(i)** - check whether a pair with the first component being equal i present (i is a key).

4. **h.erase(i)** - remove a pair with the first component being equal i if present (i os a key).

You may use **std::unordered_map** as an array with arbitrary indices, it is oftentimes called an **associative array** because for each key i there is an association with a value x.

Depending on the initial capacity of the **std::unordered_map** there exists an anti-test which makes the structure to work in linear time for an operation, thus, if there is a chance that a testing

system might have an anti-tests for the initial capacity N, it may be helpful to set the initial capacity to be a random value:

```
int64_t randomTime() {
    std::random_device rd;  // Obtain a random number from hardware
    std::mt19937 eng(rd()); // Seed the generator

    // Define the range of time (in seconds)
    std::uniform_int_distribution<> distr(0, 86400); // 86400 seconds in a day

    // Get the current time
    std::time_t currentTime = std::time(nullptr);

    // Generate a random offset from the current time
    std::time_t randomTime = currentTime + distr(eng);
    return static_cast<int64_t>(randomTime);
}

unordered_map<int, int> h(N + randomTime() % N);
```

10. Binary heap

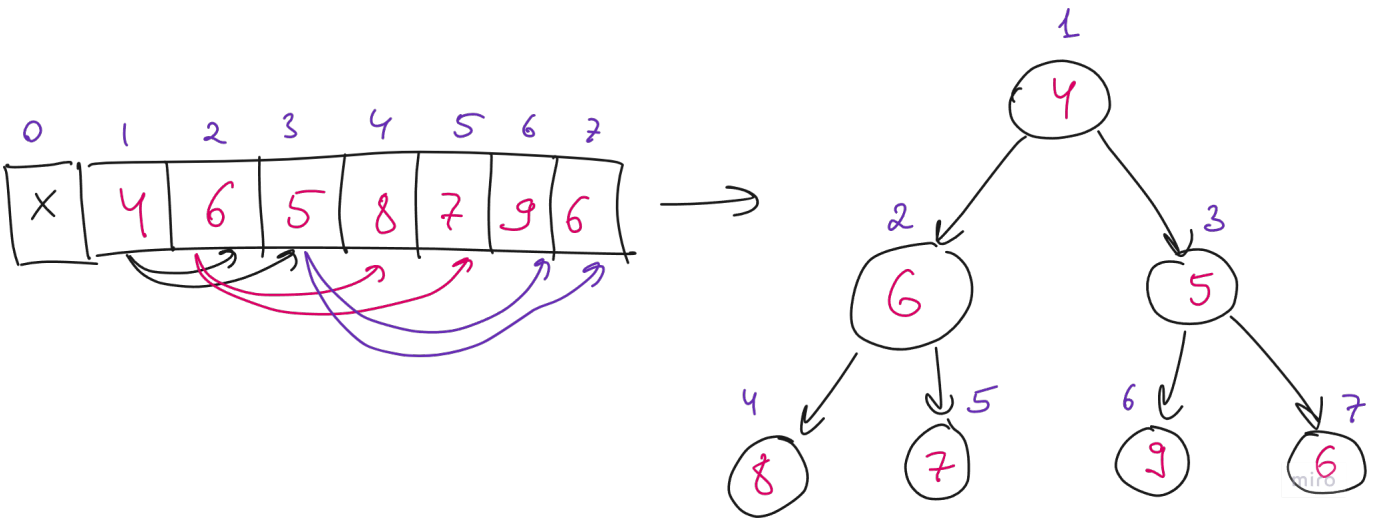
10.1 Definition

Consider an array $a[1..n]$. Its elements form a binary tree with a root at 1.

Children of a vertex at i are vertices at $2i$ and $2i+1$; the parent of a vertex at i is $\lfloor \frac{i}{2} \rfloor$.

***Definition* 10.1. Binary heap**

A binary heap is an array indicies of which form the described above tree which in turn holds the following property: $\forall$ vertex i the value $a[i]$ is a **minimum** in a subtree of i.



Lemma. The height of the binary heap is $\lfloor \log_2 n \rfloor$.

Proof. The height is equal to the length of the distance between the vertex at n and the root at 1. Notice, that by the definition $\forall k$: distance from the root to vertices from the range $[2^k, 2^{k+1})$ is equal to $k \implies$ for the vertex at n its corresponding $k := \lfloor \log_2 n \rfloor$ (it is a simple induction over k):

Let d_k be the distance between (in terms of the number of edges) the root and the vertices from the range $[2^k, 2^{k+1})$. We want to prove $d_k \equiv k$:

1. **Base**: $k = 1$

$[2^1, 2^2) = [2, 2^2 - 1] = [2, 3]$ - distance d_1 between the root and vertices 2 and 3 is indeed 1 by the definition, thus $d_1 = k$.

2. **Transition:** $k \to k + 1$:

Let's consider the range on the step k: $[2^k, 2^{k+1} - 1]$, now let's see what are the children for the vertex $v_1 = 2^k$ and the vertex $v_2 = 2^{k+1} - 1$ (children of all vertices between v_1 and v_2 will have numbers in between the obtained range):

1. v_1: $2^k \to 2^{k+1}, 2^{k+1} - 1$.
2. v_2: $2^{k+1} - 1 \to 2^{k+2} - 2, 2^{k+2} - 1$.

Thus, the obtained range will be $[2^{k+1}, 2^{k+2} - 1]$, i.e. the distance d_{k+1} from the root to the vertices in this range will be $d_{k+1} = d_k + 1 = k + 1 \implies d_{k+1} = k + 1$.

□

Interface:

The binary heap due to its height executes in $O(\log n)$ the following operations:

1. $GetMin()$ - find the minimum element (works in $O(1)$).
2. $Add(x)$ - add a new element.

3. $ExtractMin()$ - remove the current minimum element.

If there is a support of *reversed references* which allow access to the position i by a given element $a[i]$ in $O(1)$, then the binary heap will also support the following operations:

1. $DecreaseKey(x, c)$ - decrease a value x by c.
2. $Del(x)$ - remove element x from the heap.

10.2 siftUp, siftDown

siftUp(i) - push up (bubble up) an element at the position i.

siftDown(i) - push down an element at the position i.

Both operations require the heap to comply with its definition for all position except the position i, i.e. these operations place an element at the position i into such a position, so that the underlying property of a heap starts to hold for all elements.

```
1  void siftUp(int i) {
2      int p = i / 2;
3      // while i is not a root and parent's value is greater than value of i
4      while (i > 1 && a[p] > a[i]) {
5          swap(a[i], a[p]);
6          i /= 2;
7      }
8  }
9
10 void siftDown(int i) {
11     while (true) {
12         int l = 2 * i;
13         // select a smaller child of i
14         if (l + 1 <= n && a[l + 1] < a[l]) ++l;
15         // if both children are not less than i
16         if (l > n || a[l] >= a[i]) break;
17         // go to the child
18         swap(a[l], a[i]);
19         i = l;
20     }
21 }
```

Lemma. Both operations work in $O(\log n)$.

Proof. Both operations work in $O(\text{heap height}) = O(\log n)$.

□

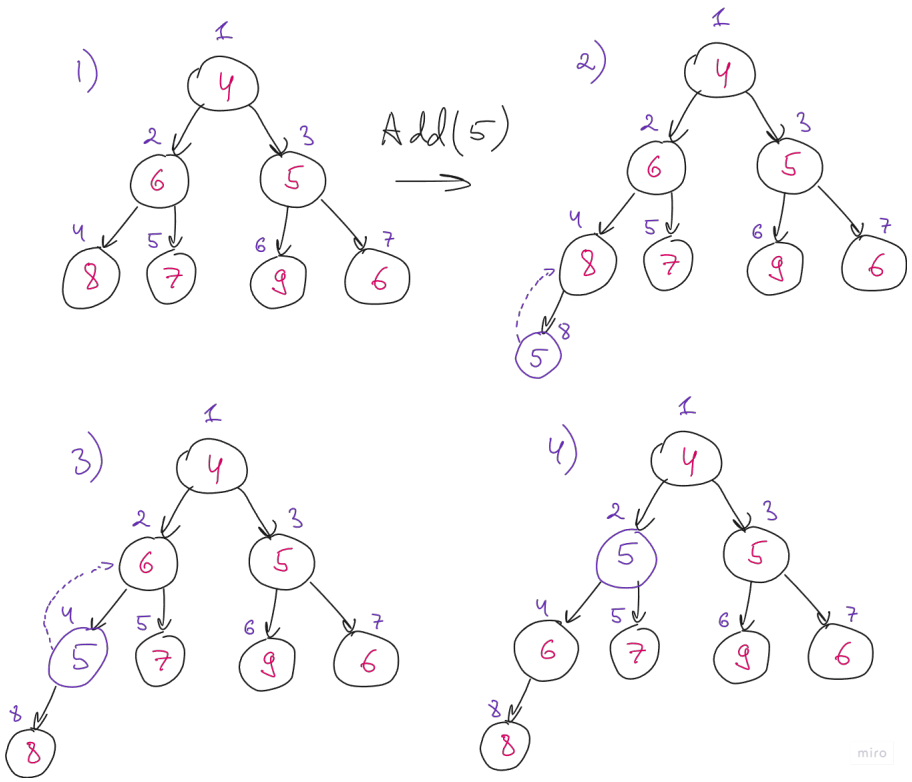
10.3 GetMin, Add, ExtractMin

All of these operations can be expressed via **siftUp/siftDown**.

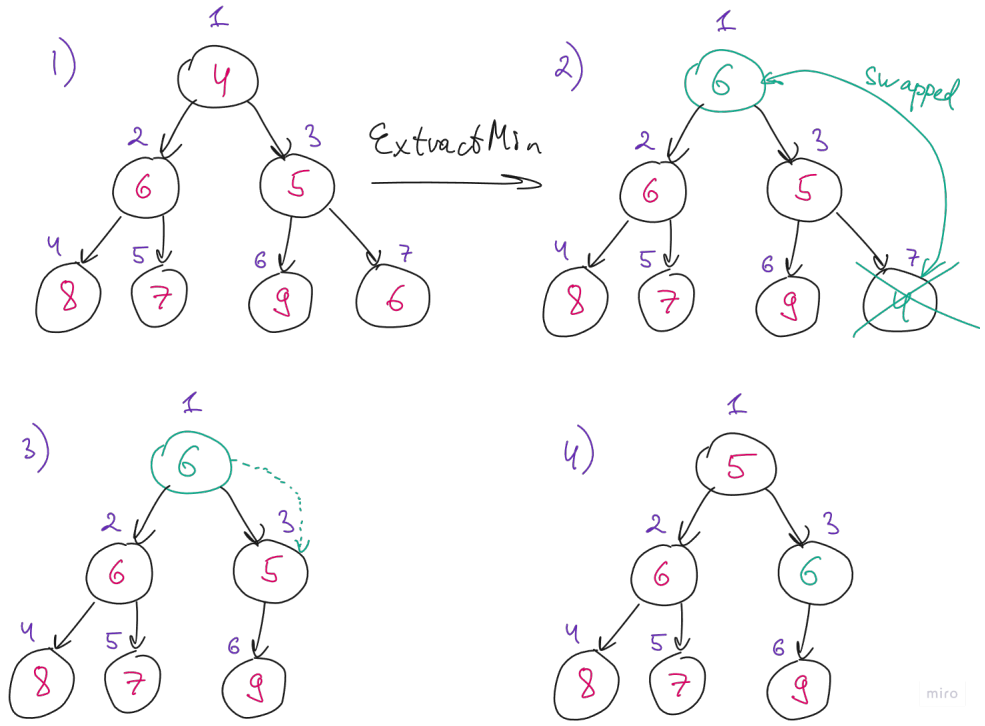
```
1  vector<int> h(N + 1);
2  int n = 0; // current size (i.e. the number of elements)
3
4  int GetMin() {
5      return a[1];
6  }
7
8  void Add(int x) {
9      ++n;
```

```
10      a[n] = x;
11      siftUp(n);
12  }
13
14  void ExtractMin() {
15      // extracting root
16      swap(a[1], a[n]);
17      n--;
18      siftDown(1);
19  }
```

Example of the **siftUp** operation during **Add**:



Example of the **siftDown** operation during **ExtractMin**:



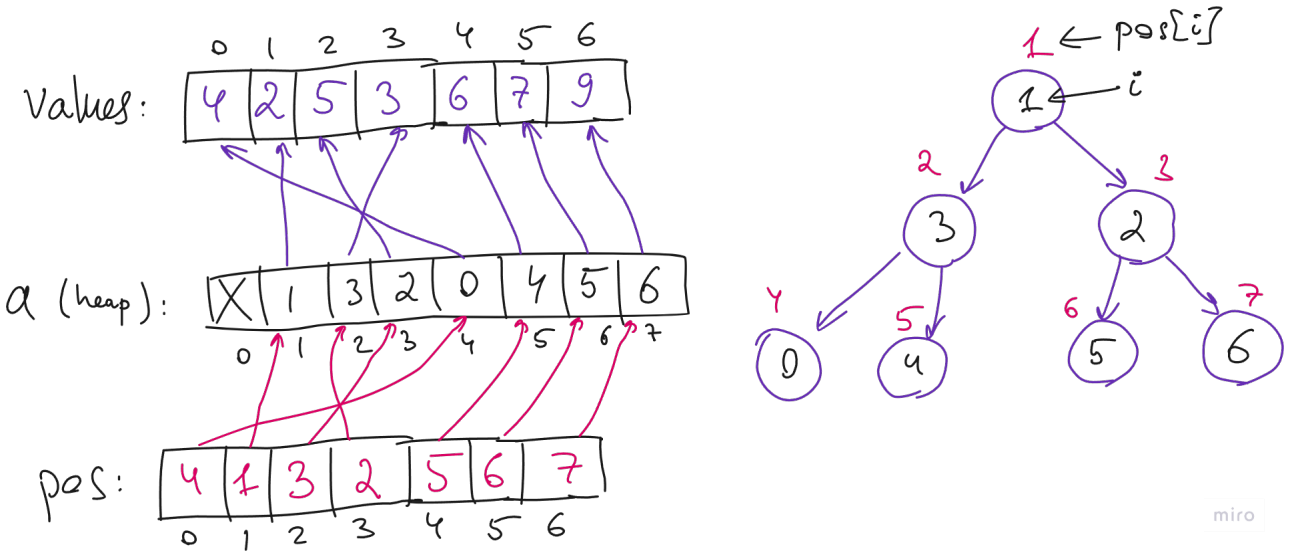
10.4 Reversed references and DecreaseKey

Suppose now we have an array **vector<int> values**. In the heap **a**[] we will store indicies of the array **values**. In this case all comparisons $a[i] < a[j]$ will be rewritten as values$[a[i]] <$ values$[a[j]]$. In order to add an element we need to add it inside **values**: **values.push_back(x)** and then add its index inside the heap: **Add(values.size() - 1)**. Since now we store the indicies inside the heap, we can for every i store an index inside the heap, i.e. **pos[i]** = position of the i-th element of **values** in **a**.

1. Given position in the heap $\to$ get its value: $i \to a[i] \to$ values$[a[i]]$.
2. Property of **pos**: $i \to \text{pos}[i] \to a[\text{pos}[i]] = i$.

The values of **pos**[] must be recalculated every time when we modify **a**[].

See the following image for clarifications:



Now we can implement removal and decrease of a key of an element inside **values** by its index i in $O(\log n)$ of time:

```
void Del(int i) {
    // i - index inside values[]
    int ind = pos[i]; // index inside heap a[]

    a[ind] = h[n]; n--;
    pos[a[ind]] = ind; // update pos[] for the 'a[ind]'-th element of values[]

    // both sift operations because new element can be
    // either greater than its children or less than its parent
    siftUp(ind);
    siftDown(ind);
}

void DecreaseKey(int i, int c) {
    // i - index inside values
    values[i] -= c;
    int ind = pos[i]; // index inside heap a[]

    // the updated element could become less than its parent
    siftUp(ind);
}
```

10.5 Build, HeapSort

We can construct a heap from a list of given elements inplace:

```
void Build(int n, int a[]) {
    for (int i = n; i >= 1; --i) {
        siftDown(i);
    }
}
```

Lemma. The **Build** operation will construct a correct binary heap from the array **a**[].

Proof. When we push down i, due to induction assumption, the left and the right subtrees are already correct binary heaps. Due to correctness of **siftDown** operation after its execution the subtree at i will become a correct binary heap. □

Lemma. **Build** works in $\Theta(n)$ of time.

Proof. Let $n = 2^k - 1$, then our heap is a full binary tree, because each element except for the leaves will have 2 children. The last level of the heap will contain 2^{k-1} elements, the pre-last level will contain 2^{k-2} elements, and so on.

siftDown(i) works in $O(\text{depth of the subtree of } i)$, thus its total time will be:

$\sum_{i=1}^{k} 2^{k-i} \cdot i = 2^k \cdot \sum_{i=1}^{k} \frac{i}{2^i} =_{(*)} 2^k \cdot \Theta(1) = (n+1) \cdot \Theta(1) = \Theta(n)$.

(*): Need to proof that $\sum_{i=1}^{k} \frac{i}{2^i} = \textbf{const}$;

Notice that $\sum_{i=1}^{k} \frac{i}{2^i} = 2 - \frac{k+2}{2^k}$ (I used wolframalpha :)). Let's prove it via induction:

1. **Base**: $k = 1$:

$\sum_{i=1}^{k} \frac{i}{2^i} = \frac{1}{2} = 2 - \frac{3}{2}$ - the assumption holds.

2. **Transition**: $k \to k+1$:

$\sum_{i=1}^{k+1} \frac{i}{2^i} = \sum_{i=1}^{k} \frac{i}{2^i} + \frac{k+1}{2^{k+1}} = 2 - \frac{k+2}{2^k} + \frac{k+1}{2^{k+1}} = 2 - \frac{k+3}{2^{k+1}} = 2 - \frac{(k+1)+2}{2^{k+1}}$.

□

The **HeapSort** implementation that works in $O(n \log n)$ and uses $O(1)$ of additional memory due to **Build** working inplace:

```
void HeapSort(int n, int a[]) {
    Build(n, a);

    for(int i = 0; i < n; ++i) {
        // extracting values in ascending order
        int x = GetMin();
        ExtractMin();
    }
}
```

11. Sorting Algorithms

11.1 Basics

***Definition* 11.1. Stable sorting algorithm**

The given sorting algorithm is called **stable** if equal elements are place in the same order (among these equal elements) as it was in the initial condition, i.e. $\forall i \neq j : a_i = a_j$, π - some permutation that sorts the elements, then $i < j \implies \pi(i) < \pi(j)$.

***Definition* 11.2. Inversion**

Inversion is a pair $(i,\ j)$, s.th.: $i < j : a_i > a_j$.

***Definition* 11.3.** I - number of inversions in the array.

***Definition* 11.4.** The array is sorted $\Leftrightarrow I = 0$.

Lemma. The sorting algorithm A based on the comparisons of the elements which does for all posible inputs of size n $o(n \log n)$ comparisons is **incorrect**, i.e. $\exists$ test T on which the result of sorting by A will be **incorrect**.

Proof. Notice that an input of size n can be expressed as a permutation of indicies $\pi : I \to I$ where I is the set of indicies, i.e. $I := \{0, 1, 2, .., n-1\}$, then there exist $n!$ inputs.

For the proof it is convinient for us to think that for every input the algorithm A does exactly k comparisons (if in fact it does less then let's just do some meaningless comparisons to get exactly k).

Since the sorting algorithm A is based on comparisons we can assign a binary value for each comparison. The algorithm does k comparisons, if on the i-th position the comparison yielded "less or equal" then we assign 0, if it yielded "greater" we assign 1 $\implies$ we end up having a binary string of size k. Notice that there is a bijection between the set of the produced strings of size k and the set of inputs of size n $\implies$ $2^k = n!$:

$o(n \log n) = 2^k = n!$ $\implies$ $n! = o(n \log n)$ but we know that $n! = \Theta(n \log n)$, thus there is a no bijection $\implies$ there exist a test on which the algorithm A produces a wrong result.

□

11.2 CountSort

This algorithm sorts a set of integers in range $[0, m)$ in $O(n + m)$ of time and $O(m)$ of memory where n is the size of the input array.

```
1  int n;
2  int arr[n];
3  int count[M] = {0};
4
5  for (int i = 0; i < n; ++i) { // O(n)
6      count[x]++;
7  }
8
9  for (int x = 0; x < M; ++x) { // O(n + M)
10     while(count[x]--) {
11         std::cout << x << " ";
12     }
13 }
```

Notice that this sorting algorithm is based on the counting not comparisons, thus it is correct and there is a speed boost.

11.3 MergeSort: recursive and iterative solutions

The main idea is to soft the left part, sort the right part and then merge two sorted parts into a single sorted array via two pointers approach.

11.3.1 Recursive implementation

```
void MergeSort(int l, int r, int* a, int* buffer) { // [l, r)
    // base case
    if (r - l <= 1) return;

    int m = (l + r) / 2;
    // sort [l, m)
    MergeSort(l, m, a, buffer);
    // sort [m, r)
    MergeSort(m, r, a, buffer);

    // merge two sorted halves via 2 pointers in O(r - l) time
    Merge(l, m, r, a, buffer);
}
```

buffer is an additional chunk of memory which is required for the **Merge** method. The **Merge** goes through the $[l, m)$ and $[m, r)$ and puts them according to the two pointers technique into the *buffer*.

Lemma. Time complexity is $O(n \log n)$.

Proof. The recursive formula is as follows: $T(n) = 2 \cdot T(\frac{n}{2}) + n$.

Note (Master Theorem): $T(n) = a \cdot T(\frac{n}{b}) + n^c$.

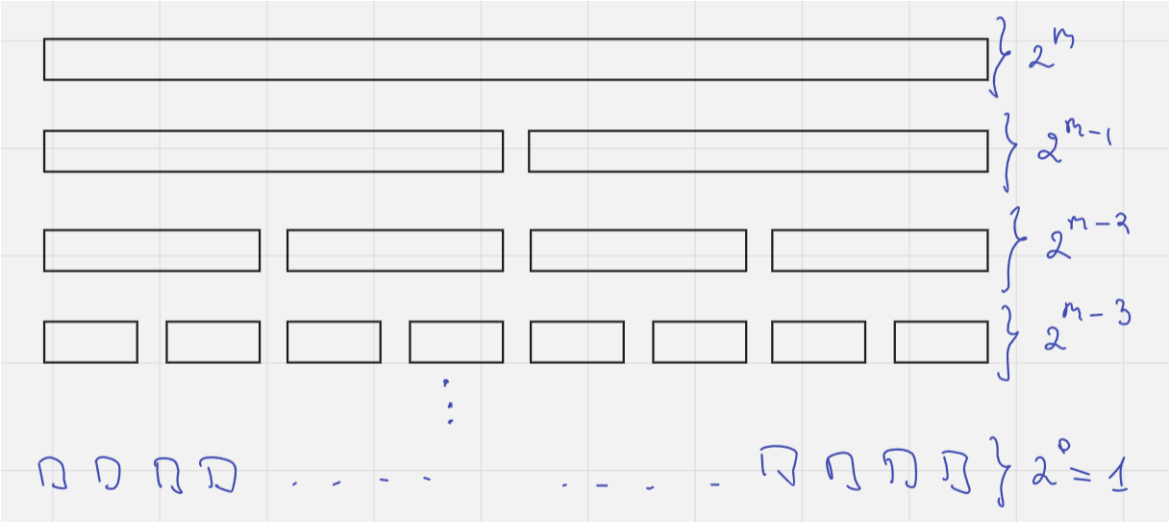
According to the Master Theorem: $a = 2, b = 2, c = 1 \implies a = b^c \implies O(n^c \log n) = O(n \log n)$.

□

11.3.2 Iterative implementation

Suppose that the size of the input array is $n = 2^m$ (if not then pad the array with **MAX_INT** until it becomes 2^m). In the recursive approach we traverse the recurse tree from top to bottom, here we traverse it from bottom up. At the bottom we have chunks of size 1, on the next level chunks of size 2^1, on the next 2^2 and so on.

```
int n;
vector<int> arr(n), buffer(n);
// In C++: (1 << k) == 2^k
for (int k = 0; (1 << k) < n; ++k) {
    // iterate through pairs of chunks of size 2^k
    for (int i = 0; i < n; i += 2 * (1 << k)) {
        int m = min(n, i + (1 << k));
        int r = min(n, i + 2 * (1 << k));
        Merge(i, m, r, a, buffer);
    }
    // now buffer has properly sorted chunks of size 2^(k+1),
    // we need arr to store the updated result after each iteration.
    swap(arr, buffer); // O(1)
}
// the answer is here because we always updated arr inside the for-loop
return arr;
```



11.4 QuickSort

The idea: select an x, split the given array a into 3 parts: $< x$, $= x$, $> x$, then make 2 recursive calls to sort the 1st and the 3rd parts.

```
1  def QuickSort(a):
2      if len(a) <= 1: return a
3      x = random.choice(a)
4      p1 = select (< x)
5      p2 = select (= x)
6      p3 = select (> x)
7      return QuickSort(p1) + p2 + QuickSort(p3)
```

The above is the general idea of the algorithm. In order to make it practically quick the following strategy is used to split the array into 3 parts:

```
1  void partition(int l, int r, int x, int* a, int &i, int &j) { // [l, r] - both
       inclusive
2      assert(x in a[l..r]); // x must be from a
3      i = l;
4      j = r;
5      while(i <= j) {
6          while (a[i] < x) i++;
7          while (a[j] > x) j--;
8          // a[i] >= x, a[j] <= x => swap elements
9          if (i <= j) swap(a[i++], a[j--]);
10     }
11 }
```

The above *partition* will split the array into 3 parts $[l, j](j, i)[i, r]$ where:

1. $a[l, j] \leq x$.
2. $a[j + 1, i - 1] = x$.
3. $a[i, r] \geq x$.

Lemma. The algorithm will not exceed the bounds $[l, r]$.

Proof. $x \in a[l, r] \implies$ there will be at least one **swap** operation executed (e.g. once $a[i] = x$ and $a[j] = x$). After the last **swap** the following holds: $l \leq i,\ i \geq j,\ j \leq r$. □

The implementation of the **QuickSort**:

```
void QuickSort(int l, int r, int* a) { // [l, r] - both inclusive
    if (l >= r) return;
    int i, j;
    int x = a[random in [l, r]];
    // partition accepts i and j by reference and updates them
    Partition(l, r, x, i, j);
    // now: a[l,j] < x, a(j, i) = x, a[i, r] > x
    // we need to sort the 1st and the 3rd parts
    QuickSort(l, j, a); // j < i
    QuickSort(i, r, a); // j < i
}
```

11.5 Comparison of the sorting algorithms

Name	Time	Space	Stable
HeapSort	O(n logn)	O(1)	-
MergeSort	O(n logn)	O(n)	+
QuickSort	O(n logn)	O(logn)	-

Notice that among the mentioned sorting algorithms no one is able to make a stable sorting in $O(1)$ of space. There exist such an algorithm, it is implemented via **inplace stable merge** in $O(n \log n)$ of time and $O(1)$ of space.

11.6 Integer Sorting

11.6.1 Stable CountSort

Let's take a look at somewhat altered sorting problem: we are given an array of pairs $(a_i,\ b_i)$ of integersm and we need to sort in a **stable** manner.

```
int pos[n] = {0};
int count[m] = {0};

void CountSort(int n, int* a, int* b) { // 0 <= a[i] < m
    for (int i = 0; i < n; ++i) {
        // count the number of occurrences of a[i]
        count[a[i]]++;
    }

    // pos[i] = "position of the start of the chunk which contains pairs <i, ?>"
    pos[0] = 0;
    for (int i = 0; i + 1 < m; ++i) {
        pos[i + 1] = pos[i] + count[i];
    }
    for (int i = 0; i < n; ++i) {
        int index = pos[a[i]]++;
        result[index] = {a[i], b[i]};
    }
}
```

Since the algorithm is **stable** the following algorithm in the next section can be implemented.

11.6.2 RadixSort

Let's sort n strings of length L where symbols of strings are integers in range $[0, k)$.

Algorithm: firstly, sort (via stable sorting algorithm) by the **last** symbol, then sort by the pre-last symbol, .., sort by the 1st symbol.

Correctness: we are sorting via a stable sorting algorithm some strings by a symbol on the position i. In this case, the strings are already sorted by the suffix $(i, L]$. Due to stability of the sorting algorithm, we can state that the strings that have equal i-th symbols will be sorted preserving their relative order after previous sorting iteratation, i.e. by the suffix $(i, L] \implies$ the strings are now sorted by suffix $[i, L]$.

Time complexity: L times we call the **CountSort** algorithm $\implies$ it will sort the strings in $\Theta(L \cdot (n + k))$.

Notice that $\forall b$ - the integer system base, any integer in $[0, m)$ is a string of length $\log_b m$ over the alphabet $[0, b) \implies$ we can sort integers over base b in $\Theta((n + b) \cdot \lceil \log_b m \rceil)$. Then once $b = n$ we have the optimum time of $\Theta(n \cdot \lceil \log_n m \rceil)$.

The idea of the integer sorting: we transform integers of the given array from the base of 10 into the base of n and apply the string sorting algorithm described above.

11.6.3 BucketSort

The idea is to place all of the elements x ($\min \le x \le \max$) into n buckets. In order to do that the numeric line is splitted into n segments of equal size where the i-th segment contains the elements from the following range:

$L := \max - \min + 1$, $[\min + \frac{i}{n}L,\ \min + \frac{(i+1)}{n}L]$

In this case, each element x_j will be placed into the segment of number $i_j = \lfloor \frac{x_j - \min}{L} \cdot n \rfloor$. The buckets themselves are already sorted, we only need to sort elements inside each bucket. It can be done either by calling the BucketSort recursively or by running any other integer sorting algorithm on each bucket.

```
1  void BucketSort(vector<int>& a) { // we update the provided array
2      if (a.empty()) return;
3      int n = a.size();
4      int min = min_element(a);
5      int max = max_element(a);
6      int L = max - min + 1;
7
8      if (min == max) return;
9      vector<int> buckets[n]; // array of n vectors
10
11     for (int x : a) {
12         int bucketIndex = n * (x - min) / L;
13         buckets[bucketIndex].push_back(x);
14     }
15     a.clear();
16     for (int i = 0; i < n; ++i) {
17         // sort the i-th bucket
18         BucketSort(buckets[i]);
19         // place the elements back into a
20         for (int x : buckets[i]) {
21             a.push_back(x);
22         }
23     }
24 }
```

Lemma. BucketSort works in $O(n \cdot \lceil \log \max - \min \rceil)$ of time.

Proof. The algorithm divides the numbers into buckets only if $n \geq 2$, thus the L is being reduced on each recursive call at least by 2 (in particular, $L \to \frac{L}{n}$) $\implies$ the depth of the recursion is not more than $\log L$, each recursion level has not more than n buckets. $\square$

12. Graphs and basic depth-first-search (dfs)

12.1 Definitions

***Definition* 12.1. Graph** Graph G is a pair of sets: the set of verticies V and the set of edges E represented as a set of pairs of verticies, $G := (V,\ E)$.

***Definition* 12.2. Basics**

1. If edges have directions then the graph is called **directed graph**.
2. If edges do not have any directions then the graph is called **undirected graph**.
3. If each edge has a corresponding weight assigned to it then the graph is called **weighted graph**.
4. If E is a *multiset*, i.e. some verticies may have several edges between them, then such edges are called **multiple edges** and the graph may be called **multi-gprah**.
5. For an edge $e = (a,\ b)$ we say that e is **incident** to the vertex a/b.
6. The **degree** of a vertex is the number of its incident edges, denoted as $\deg v$.
7. In the directed graph there are **incoming** and **outgoing** degrees: $\deg v = \deg_{in} v + \deg_{out} v$.
8. Two edges that share a common vertex are called **adjacent** edges.
9. Two verticies that are connected by an edge are also called **adjacent**.
10. If $\deg v = 0$ then v is an **isolated** vertex.
11. If $\deg v = 1$ then v is a **leaf** in a graph.
12. An edge $e = (v,\ v)$ is called a **loop**.
13. A **simple graph** is a graph that does not have more than one edge between any two vertices and no edge starts and ends at the same vertex (i.e. no multi-edges and no loops).

***Definition* 12.3. Path**

A path in a graph is an alternating sequence of vertices and edges in which neighboring elements are incident and the extreme ones are vertices. In the digraph, the direction of all edges is from **i** to **i+1**.

E.g., $v_0 \to e_0 \to v_1 \to e_1 \to v_2 \to e_2 \to v_3 \to ... \to e_{k-1} \to v_k$.

***Definition* 12.4.** 1. The path is **vertex simple/simple** if all its verticies are distinct.

2. The path is **edge simple** if all its edges are distinct.

Note: paths exist in both undirected and directed graphs; if there are no multi-edges in the graph then each path can be denoted merely via its verticies.

3. **Cycle** is a path with equal extreme verticies (i.e. both the start and the end are the same vertex).
4. Cycles also can be **vertex simple** and **edge simple**.
5. **Acyclic** graph is a graph with no cycles in there.
6. **Tree** is an acyclic undirected graph.

12.2 How to store a gprah programmically

We will assume that $|V| = n$ and $|E| = m$. Sometimes V and E could be used to denote their sizes.

12.2.1 List of edges

Here we merely store the vector of edges, i.e. pairs of vertex $(a_i,\ b_i)$ which denote an edge $e_i := a_i \to b_i$:

```
1 pair<int, int> edges[m];
2 // Or:
```

```
vector<pair<int, int>> edges(m);
// Or (if we want to store weights on edges):
struct Edge {int from, to, weight; };
vector<Edge> edges(m);
```

12.2.2 Adjacency matrix

For each pair of vertices we can store either the fact of having/not having an edge between them (i.e. 0 or 1), or the number of edges between them, or the weight of an edge between them.

```
bool c[n][n];
// c[a][b] = 0/1 - denotes whether vertices a and b have an edge between them
int c[n][n]; // here c[a][b] denote the weight of an edge between a and b
```

12.2.3 Adjacency Lists

For each vertex v we store a list of its adjacent edges:

1. $v_0 \to e_{i_0}, e_{i_1}, ..., e_{i_k}$
2. $v_1 \to e_{j_0}, e_{j_1}, ..., e_{j_p}$

...

3. $v_{n-1} \to e_{t_0}, e_{t_1}, ..., e_{t_q}$

```
// to make it remove/find fast replace vector with set/unordered_set
vector<Edge> c[n];
```

12.2.4 Required actions with a graph

1. **adjacent(v)** - traverse all incident edges of v.
2. **get(a, b)** - check the existence/weight of an edge between a and b.
3. **all()** - check all the edges of the graph.
4. **add(a, b)** - add an edge $a \to b$.
5. **del(a, b)** - remove an edge $a \to b$.

Here are the time complexities of the operations for the described storing methods:

	adjacent	get	all	add	del	memory
List of edges	O(E)	O(E)	O(1)	O(E)	O(E)	O(E)
Adjacency matrix	O(V)	O(1)	O(V^2)	O(1)	O(1)	O(V^2)
Adjacency Lists (vector)	O(deg)	O(deg)	O(V+E)	O(1)	O(deg)	O(E)
Adjacency Lists (hash-map)	O(deg)	O(1)	O(V+E)	O(1)	O(1)	O(E)

12.3 dfs: depth-first search

Task: mark all the vertices reachable from a given vertex a.

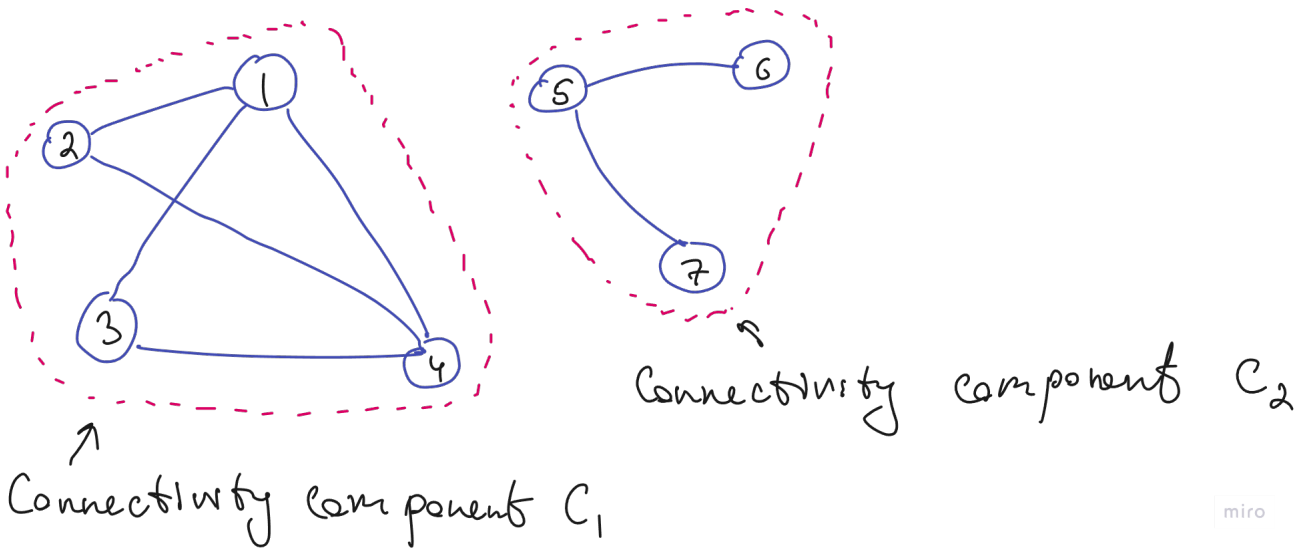
Solution: mark the adjacent verticies of the a and then call the procedure recursively.

```
vector<vector<int>> graph(n);
void dfs(int v) {
    // no need to traverse already visited verticies
    if(marked[v]) return;
    marked[v] = 1;
    for (int u : graph[v]) {
        dfs(u);
    }
}
dfs(a);
```

The time complexity is $O(E)$ because each edge is visited not more then once.

***Definition* 12.5. Connectivity components** Two verticies a and b are located in a single connectivity component iff $\exists$ a path between them.

In other words, a connectivity component is a equvalence class over a relation of reachableness (i.e. $a, b \in C \leftrightarrow \exists p - \text{path } a \to_p b$).



All connectivity components may be easily found via dfs. We need to start the dfs from a vertex v and it will cover all the verticies that are reachable from v, i.e. it will be a connectivity component.

```
vector<vector<int>> graph(n);
int connectivity_component = 0;

void dfs(int v) {
    if(marked[v] != 0) return;
    marked[v] = connectivity_component;
    for (int u : graph[v]) {
        dfs(u);
    }
}

for (int v = 0; v < n; ++v) {
    if (!marked[v]) {
        connectivity_component++;
        dfs(v);
```

```
16      }
17  }
```

Now each vertex is *colored* in the color of its connectivity component. The time complexity is $O(V + E)$ because each edge is traversed only once and we traverse each vertex in the for-loop.

12.3.1 Path recovery

We need to find a path from a vertex v to the vertex u and print the found path (assume it exists). We can collected the visited verticies as follows:

```
vector<int> path;

bool dfs(int v) {
    if (v == u) {
        path.push_back(v);
        return 1 /*path found*/;
    }
    marked[v] = 1;
    for (int x : graph[v]) {
        if (dfs(x) /*x is lying on the path*/) {
            // v -> .. -> x -> .. -> u  =>  x must in added to the path
            path.push_back(x);
            return 1 /*path found*/;
        }
    }
    return 0 /*path not found*/;
}
```

Now the **path** vector contains the verticies from the u to the v (i.e. the order is reversed since u was added first).

12.4 Topological sorting

***Definition* 12.6. Topological sorting** is an ordering of the verticies of **directed** (oriented) graph according to the partial order given by the graph's edges.

In order words, topological sorting is a mapping of integers to the verticies a_v, so that: $\forall e := v \to u : a_v < a_u$.

Lemma. Topological sorting exists $\leftrightarrow$ the graph is *acyclic*.

Proof. 1. $\bar{A} \leftarrow \bar{B}$: If there is a cycle $v_1 \to v_2 \to .. \to v_{k-1} \to v_k \to v_1$, then we have the sequence of inequalities:

$a_{v_1} < a_{v_2} < a_{v_3} < .. < a_{v_{k-1}} < a_{v_k} < a_{v_1} \implies a_{v_1} < a_{v_1}$ which is false.

2. $A \leftarrow B$: If there are no cycles in the graph then there exists a vertex v whose $\deg_{in}(v) = 0$, then let's assign $a_v := 0$ and remove the vertex. There still will be no cycles, do the same procedure until no verticies left in the graph $\implies$ topological sorting is constructed.

□

```
// non-recursive solution
vector<int> topSort(vector<vector<int>>& graph /*directed graph*/) {
    vector<int> ans;
    int start = find_vertex_of_zero_indeg(graph); // assert it exists

```

```
6       queue<int> q = { start };
7       while (!q.empty()) {
8           int v = q.front(); q.pop();
9           ans.push_back(v);
10          // we remove vertex v and update indegs of its neighbors
11          for (int u : graph[v]) {
12              if (--deg[u] == 0) {
13                  q.push(u);
14              }
15          }
16      }
17
18      return ans;
19  }
20
21  // recursive solution
22  vector<int> ans;
23  void dfs(int v) {
24      if (marked[v]) return;
25      marked[v] = 1;
26      for (int u : graph[v]) {
27          dfs(u);
28      }
29      ans.push_back(v);
30  }
31  std::reverse(std::begin(ans), std::end(ans));
32  for (int v : ans) {
33      std::cout << v << " ";
34  }
```

13. DFS (part 1)