**Optimize I/O**

#define optimize() ios\_base::sync\_with\_stdio(0);cin.tie(0);cout.tie(0);

#define fraction() cout.unsetf(ios::floatfield); cout.precision(10); cout.setf(ios::fixed,ios::floatfield);

#define file() freopen("input.txt","r",stdin);freopen("output.txt","w",stdout);

**Vector:**

**vector <data\_type> vector\_name;** // 1D Vector vector < vector<data\_type> > vector\_name; // 2D Vector

**begin()** *Returns an iterator to the first element.*

**end()** *Returns an iterator to the theoretical element after the last element.*

**size()** *Returns the number of elements present.*

**empty()** *Returns true if the vector is empty, false otherwise.*

**push\_back()** *Adds an element to the back of the vector.*

**pop\_back()** *Removes an element from the end.*

**insert(vector\_name.begin() + 1, 500)** *Insert an element at the specified position.*

**erase(iterator)** *Delete the elements at a specified position or range.*

**clear()** *Removes all elements.*

**assign(length,value)** *Assign a new value to the vector elements. E.g.* **v.assign(7, 100);**

For fixed sized vector with value **vector<int> v(10, 100);**  here 10 is size and 100 is value.

**reverse(a.begin(), a.end());** to reverse a vector. **sort(v.begin(), v.end());**  to sort the vector

**Stack:**

**stack<int> s;** **empty()** – Returns whether the stack is empty.

**size()** – Returns the size of the stack.

**top()** – Returns a reference to the top most element of the stack.

**push(g)** – Adds the element ‘g’ at the top of the stack.

**pop()** – Deletes the most recent entered element of the stack. **It doesn’t return the last element**

// Sorting

    vector<int> nums = {3, 1, 4, 1, 5, 9, 2, 6};

    sort(nums.begin(), nums.end()); // Sort in ascending order

    sort(nums.rbegin(), nums.rend()); // Sort in descending order

    // Maximum and Minimum

    int max\_num = \*max\_element(nums.begin(), nums.end());

    int min\_num = \*min\_element(nums.begin(), nums.end());

    // Reverse

    reverse(nums.begin(), nums.end()); // Reverse the vector

    // Rotate

    rotate(nums.begin(), nums.begin() + 3, nums.end()); // Rotate the vector by 3 positions

    // Find

    auto it = find(nums.begin(), nums.end(), 4); // Find the first occurrence of 4

    // Remove

    nums.erase(remove(nums.begin(), nums.end(), 1), nums.end()); // Remove all occurrences of 1

    // Accumulate

    int sum = accumulate(nums.begin(), nums.end(), 0); // Sum of elements

    // Count

    int count\_1 = count(nums.begin(), nums.end(), 1); // Count occurrences of 1

    // Generate

    generate(nums.begin(), nums.end(), []() { return rand() % 10; }); // Generate random numbers

    // Unique

    sort(nums.begin(), nums.end());

    auto last = unique(nums.begin(), nums.end()); // Remove duplicates

    nums.erase(last, nums.end());

    // For Each

    for\_each(nums.begin(), nums.end(), [](int x) { cout << x << " "; }); // Apply a function to each element

**Pair:**

defining a pair **pair<int, char> PAIR1;** first part of the pair **PAIR1.first = 100;** second part of the pair **PAIR1.second = 'G';**

**pair = {1, 'a'};**  is also valid.

**HashMap:**

**unordered\_map<string, int> umap;** // inserting values by using [] operator **umap["GeeksforGeeks"] = 10;** **umap["Practice"] = 20;**

**unordered\_map<string, double> umap = {** //inserting element directly in map

**{"One", 1},**

**{"Two", 2},**

**{"Three", 3}**

**};**

**at()** This function in C++ unordered\_map returns the reference to the value with the element as key k

**begin()** Returns an iterator pointing to the first element in the container in the unordered\_map container

**end()** Returns an iterator pointing to the position past the last element in the container in the unordered\_map container.

**bucket()** Returns the bucket number where the element with the key k is located in the map

**bucket\_count** Bucket\_count is used to count the total no. of buckets in the unordered\_map. No parameter is required to pass into this function

**bucket\_size** Returns the number of elements in each bucket of the unordered\_map

**count()** Count the number of elements present in an unordered\_map with a given key

**equal\_range** Return the bounds of a range that includes all the elements in the container with a key that compares equal to k

**find()** Returns iterator to the element

**empty()** Checks whether the container is empty in the unordered\_map container

**erase()** Erase elements in the container in the unordered\_map container

**Set:**

**set<int> val;** // defining an empty set **set<int> val = {6, 10, 5, 1};** // defining a set with values **val.insert('G');**

By default, the std::set is sorted in ascending order. However, we have the option to change the sorting order by using the following syntax. **set <data\_type, greater<data\_type>> set\_name; val.erase(const g)** Removes the value ‘g’ from the set.

**empty()** – Returns whether the stack is empty **size()** – Returns the size of the stack **top()** – Returns a reference to the top most element of the stack **push(g)** – Adds the element ‘g’ at the top of the stack **pop()** – Deletes the most recent entered element of the stack

**Queue:**

**empty()** Returns whether the queue is empty. It return true if the queue is empty otherwise returns false.

**size()** Returns the size of the queue. **swap()** Exchange the contents of two queues but the queues must be of the same data type, although sizes may differ. **emplace()** Insert a new element into the queue container, the new element is added to the end of the queue. **front()** Returns a reference to the first element of the queue. **back()** Returns a reference to the last element of the queue. **push(g)** Adds the element ‘g’ at the end of the queue. **pop()** Deletes the first element of the queue.

**Math**

**int** **nCr**(**int** n, **int** r) **return** fact(n) **/** (fact(r) **\*** fact(n **-** r));

**int** **nPr**(**int** n, **int** r) **return** fact(n)**/**fact(n**-**r);

abs(x); sqrt(x); cbrt(x); pow(x, y); log10(x)

**GCD and LCM**

**long** **long** **gcd**(**long** **long** a, **long** **long** b) { **return** \_\_gcd(a, b); }

**long** **long** **lcm**(**long** **long** a, **long** **long** b)

{ **return** a \* (b / gcd(a, b)); }

**Char**

toupper(), tolower(), isxdigit(), isupper(), isspace(), ispunct(), islower(), isdigit(), isblank(), isalpha()

**String**

str1.substr(position,length); // return sub string

str1.replace(position,length,str2);

**Tokenize String:**

string input = "ABC,EF";

stringstream **ss**(input);

vector<string> tokens;

string token;

**while** (getline(ss, token, ','))

tokens.push\_back(token);

The **tokens**  vector will hold {“ABC”, “EF”}

**Binary Search**

binary\_search(arr.begin(), arr.end(), 15); // **1 or 0**

lower\_bound(arr1.begin(), arr1.end(), 20) **-** arr1.begin();

upper\_bound(arr1.begin(), arr1.end(), 20) **-** arr1.begin();

int binSearch(vector<int> arr, int l, int r, int x) {

if (x < arr[l] || x > arr[r - 1]) return -1;

int mid = (l + r) / 2;

if (arr[mid] < x) return binSearch(arr, mid, r, x);

else if (arr[mid] > x) return binSearch(arr, l, mid, x);

else return mid + 1;

}

**LL Reverse**

**struct** ListNode {

**int** val;

ListNode **\***next;

ListNode() **:** val(0), next(nullptr) {}

ListNode(**int** x) **:** val(x), next(nullptr) {}

ListNode(**int** x, ListNode **\***next) **:** val(x), next(next) {}

};

**public** **class** **Solution** {

**public** ListNode ReverseList(ListNode head) {

**if**(head **==** null) **return** head;

ListNode prev\_node **=** null;

**while**(head **!=** null){

ListNode next\_node **=** head.next;

head.next **=** prev\_node;

prev\_node **=** head;

head **=** next\_node;

}

**return** prev\_node;

}

}

**DFS + memo example**

unordered\_map<**int**, **long** **long**> map;

**int** **minCostClimbingStairs**(vector<**int**> &cost)

{

**return** min(dfs(cost, **0**), dfs(cost, **1**));

}

**long** **long** **dfs**(vector<**int**> &cost, **int** i)

{

**if** (i >= cost.size())

**return** **0**;

**if** (map[i])

**return** map[i];

map[i] = min(dfs(cost, i + **1**), dfs(cost, i + **2**)) + cost[i];

**return** map[i];

}

**BFS + memo example**

unordered\_map<int, vector<int>> graph = {

    {0, {1, 2}},

    {1, {3}},

    {2, {4}},

    {3, {5}},

    {4, {5}},

    {5, {}}

};

unordered\_map<int, int> memo;

int bfs\_shortest\_path(int start, int target) {

    queue<pair<int, int>> q; // (node, distance)

    q.push({start, 0});

    while (!q.empty()) {

        int node = q.front().first;

        int dist = q.front().second;

        q.pop();

        if (node == target) return dist;

        for (int neighbor : graph[node]) {

            if (memo.find(neighbor) == memo.end() || memo[neighbor] > dist + 1) {

                memo[neighbor] = dist + 1;

                q.push({neighbor, dist + 1});

            }

        }

    }

    return -1; // Target node not reachable

}

**BFS Tree Right Side:**

**struct** TreeNode {

**int** val;

TreeNode **\***left;

TreeNode **\***right;

TreeNode() **:** val(0), left(nullptr), right(nullptr) {}

TreeNode(**int** x) **:** val(x), left(nullptr), right(nullptr) {}

TreeNode(**int** x, TreeNode **\***left, TreeNode **\***right) **:** val(x), left(left), right(right) {}

};

vector**<int>** rightSideView(TreeNode**\*** root) {

vector**<int>** result;

**if** (**!**root) **return** result;

queue**<**TreeNode**\*>** q;

q.push(root);

**while**(**!**q.empty()) {

vector**<int>** local;

**int** s **=** q.size();

**for** (**int** i **=** 0; i **<** s; i**++**) {

TreeNode**\*** node **=** q.front();

q.pop();

**if**(node**->**left **!=** NULL) q.push(node**->**left);

**if**(node**->**right **!=** NULL) q.push(node**->**right);

local.push\_back(node**->**val);

}

result.push\_back(local[local.size() **-** 1]);

}

**for** (**auto** i **:** result)

cout **<<** i **<<** endl;

**return** result;

}

**Prime Sieve:**

*// Given n, return all primes up to and including n.*

vector**<int>** GeneratePrimes(**int** n) {

**if** (n **<** 2) {

**return** {};

}

**const** **int** size **=** floor(0.5 **\*** (n **-** 3)) **+** 1;

vector**<int>** primes;

primes.emplace\_back(2);

*// is\_prime[i] represents whether (2i + 3) is prime or not.*

*// For example, is\_prime[0] represents 3 is prime or not, is\_prime[1]*

*// represents 5, is\_prime[2] represents 7, etc.*

*// Initially, set each to true. Then use sieving to eliminate nonprimes.*

deque**<bool>** is\_prime(size, true);

**for** (**int** i **=** 0; i **<** size; **++**i) {

**if** (is\_prime[i]) {

**int** p **=** (i **\*** 2) **+** 3;

primes.emplace\_back(p);

*// Sieving from p^2, whose value is (4i^2 + 12i + 9). The index in*

*// is\_prime is (2i^2 + 6i + 3) because is\_prime[i] represents 2i + 3.*

*//*

*// Note that we need to use long long for j because p^2 might overflow.*

**for** (**long** **long** j **=** 2LL **\*** i **\*** i **+** 6 **\*** i **+** 3; j **<** size; j **+=** p) {

is\_prime[j] **=** false;

}

}

}

**return** primes;

}