Team Reverence Document

ISTT\_ SHADOW CodeBook

Institute of Science Trade and Technology

Plot: 1/9, Road: 2, Block: D, Section: 15, Mirpur, Dhaka-1216, Bangladesh

* Template and Functions………………………… 02, 03

**Template:**

#pragma GCC optimize("O3,unroll-loops")

#include<bits/stdc++.h>

#include<ext/pb\_ds/assoc\_container.hpp>

#include<ext/pb\_ds/tree\_policy.hpp>

using namespace std;

using namespace chrono;

using namespace \_\_gnu\_pbds;

#define fastio() ios\_base::sync\_with\_stdio(false);cin.tie(NULL);cout.tie(NULL)

#define MOD 1000000007

// #define MOD1 998244353

#define INF 1e18

#define nline "\n"

#define pb push\_back

#define ppb pop\_back

#define mp make\_pair

#define ff first

#define ss second

#define PI 3.141592653589793238462

#define set\_bits \_\_builtin\_popcountll

#define sz(x) ((int)(x).size())

#define all(x) (x).begin(), (x).end()

#define uni(a) (a).erase(unique(all(a)), (a).end())

#define prec(n) fixed<<setprecision(n)

#define rep(a,b) for(auto &a : b)

#define rep1(n) for(ll i=0; i<(ll)(n); ++i)

#define rep2(i,n) for(ll i=0; i<(ll)(n); ++i)

#define rep3(i,a,b) for(ll i=(ll)(a); i<(ll)(b); ++i)

#define rep4(i,a,b,c) for(ll i=(ll)(a); i<(ll)(b); i+=(c))

#define per1(n) for(ll i=((ll)n)-1; i>=0; --i)

#define per2(i,n) for(ll i=((ll)n)-1; i>=0; --i)

#define per3(i,a,b) for(ll i=((ll)a)-1; i>=(ll)(b); --i)

#define per4(i,a,b,c) for(ll i=((ll)a)-1; i>=(ll)(b); i-=(c))

const int MOD1 = 998244353;

const int N = 1e6+10;

#ifdef RASHEDUL

#define debug(x) cerr << #x<<" "; \_print(x); cerr << endl;

#else

#define debug(x);

#endif

typedef long long ll;

typedef unsigned long long ull;

typedef long double lld;

typedef \_\_int128 ell;

typedef tree<pair<ll, ll>, null\_type, less<pair<ll, ll>>, rb\_tree\_tag, tree\_order\_statistics\_node\_update > pbds; // find\_by\_order, order\_of\_key

void \_print(ll t) {cerr << t<<',';}

void \_print(int t) {cerr << t<<',';}

void \_print(string t) {cerr << t<<',';}

void \_print(char t) {cerr << t<<',';}

void \_print(lld t) {cerr << t<<',';}

void \_print(double t) {cerr << t<<',';}

void \_print(ull t) {cerr << t<<',';}

template <class T, class V> void \_print(pair <T, V> p);

template <class T> void \_print(vector <T> v);

template <class T> void \_print(set <T> v);

template <class T, class V> void \_print(map <T, V> v);

template <class T> void \_print(multiset <T> v);

template <class T, class V> void \_print(pair <T, V> p) {cerr << "{"; \_print(p.ff); cerr << ","; \_print(p.ss); cerr << "}\n";}

template <class T> void \_print(vector <T> v) {cerr << "[ "; for (T i : v) {\_print(i); cerr << " ";} cerr << "]\n";}

template <class T> void \_print(set <T> v) {cerr << "[ "; for (T i : v) {\_print(i); cerr << " ";} cerr << "]\n";}

template <class T> void \_print(multiset <T> v) {cerr << "[ "; for (T i : v) {\_print(i); cerr << " ";} cerr << "]\n";}

template <class T, class V> void \_print(map <T, V> v) {cerr << "[ "; for (auto i : v) {\_print(i); cerr << " ";} cerr << "]\n";}

void \_print(pbds v) {cerr << "[ "; for (auto i : v) {\_print(i); cerr << " ";} cerr << "]\n";}

mt19937 rng(chrono::steady\_clock::now().time\_since\_epoch().count());

/\*---------------------------------------------------------------------------------------------------------------------------\*/

ll gcd(ll a, ll b) {if (b > a) {return gcd(b, a);} if (b == 0) {return a;} return gcd(b, a % b);}

ll expo(ll a, ll b, ll mod) {ll res = 1; while (b > 0) {if (b & 1)res = (res \* a) % mod; a = (a \* a) % mod; b = b >> 1;} return res;}

void extendgcd(ll a, ll b, ll\*v) {if (b == 0) {v[0] = 1; v[1] = 10; v[2] = a; return ;} extendgcd(b, a % b, v); ll x = v[1]; v[1] = v[0] - v[1] \* (a / b); v[0] = x; return;} //pass an array of size1 3

ll mod\_add(ll a, ll b, ll m) {a = a % m; b = b % m; return (((a + b) % m) + m) % m;}

ll mod\_mul(ll a, ll b, ll m) {a = a % m; b = b % m; return (((a \* b) % m) + m) % m;}

ll mod\_sub(ll a, ll b, ll m) {a = a % m; b = b % m; return (((a - b) % m) + m) % m;}

ll mminv(ll a, ll b) {ll arr[3]; extendgcd(a, b, arr); return mod\_add(arr[0], 0, b);} //for non prime b

ll mminvprime(ll a, ll b) {return expo(a, b - 2, b);}

bool revsort(ll a, ll b) {return a > b;}

ll mod\_div(ll a, ll b, ll m) {a = a % m; b = b % m; return (mod\_mul(a, mminvprime(b, m), m) + m) % m;} //only for prime m

ll combination(ll n, ll r, ll m, ll \*fact, ll \*ifact) {ll val1 = fact[n]; ll val2 = ifact[n - r]; ll val3 = ifact[r]; return (((val1 \* val2) % m) \* val3) % m;}

void google(int t) {cout << "Case #" << t << ": ";}

vector<ll> sieve(int n) {int\*arr = new int[n + 1](); vector<ll> vect; for (int i = 2; i <= n; i++)if (arr[i] == 0) {vect.push\_back(i); for (int j = 2 \* i; j <= n; j += i)arr[j] = 1;} return vect;}

ll phin(ll n) {ll number = n; if (n % 2 == 0) {number /= 2; while (n % 2 == 0) n /= 2;} for (ll i = 3; i <= sqrt(n); i += 2) {if (n % i == 0) {while (n % i == 0)n /= i; number = (number / i \* (i - 1));}} if (n > 1)number = (number / n \* (n - 1)) ; return number;} //O(sqrt(N))

ll getRandomNumber(ll l, ll r) {return uniform\_int\_distribution<ll>(l, r)(rng);}

/\*--------------------------------------------------------------------------------------------------------------------------\*/

bool multiTest = true;

void solve(int testCase)

{ }

int main()

{

#ifdef RASHEDUL

freopen("Error.txt", "w", stderr);

freopen("in.in", "r", stdin);

freopen("out.out", "w", stdout);

auto start1 = high\_resolution\_clock::now();

#endif

fastio();

int testCase = 1;

int T{1};

if (multiTest)

cin >> T;

while (T--)

solve(testCase++);

#ifdef RASHEDUL

auto stop1 = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(stop1 - start1);

cerr << "Time: " << duration.count() / 1000 << endl;

#endif

}**Command:**

"cpp": "clear && cd $dir && g++ -DRASHEDUL $fileName -std=c++2b -o zzExecutable && ./zzExecutable" // Linux

"clear; cd $dir && g++ -DRASHEDUL $fileName -std=c++2b -o zzExecutable && .\\zzExecutable.exe" // windows

**Sublim Setup Windows:**

{

"cmd": ["g++.exe","-std=c++14", "${file}", "-o", "${file\_base\_name}.exe", "&&" , "${file\_base\_name}.exe<inputf.in>outputf.in"],

"selector":"source.cpp",

"shell":true,

"working\_dir":"$file\_path"

}

Linux:

{

"cmd" : ["g++ -std=c++14 $file\_name -o $file\_base\_name && timeout 4s ./$file\_base\_name<inputf.in>outputf.in"],

"selector" : "source.c",

"shell": true,

"working\_dir" : "$file\_path"

}

**Find Factors of a number:**

void findFactors(int num) {

std::cout << "Factors of " << num << " are: ";

int limit = std::sqrt(num); // Iterate up to the square root

for (int i = 1; i <= limit; ++i) {

if (num % i == 0) {

std::cout << i << " ";

if (i != num / i) { // Avoid duplicates for perfect squares

std::cout << num / i << " ";

}}}

std::cout << std::endl;

}

**Stack (std::stack)**

s.push(element): Inserts an element at the top of the stack.

s.pop(): Removes and returns the element at the top of the stack. Throws an exception if the stack is empty.

s.top(): Returns a reference to the element at the top of the stack without removing it. Throws an exception if the stack is empty.

s.empty(): Checks if the stack is empty. Returns true if empty, false otherwise.

s.size(): Returns the number of elements in the stack.

**Queue (std::queue)**

q.push(element): Inserts an element at the back of the queue.

q.pop(): Removes and returns the element at the front of the queue. Throws an exception if the queue is empty.

q.front(): Returns a reference to the element at the front of the queue without removing it. Throws an exception if the queue is empty.

q.back(): Returns a reference to the element at the back of thequeue without removing it. Throws an exception if the queue is empty (not as common for queues).

q.empty(): Checks if the queue is empty. Returns true if empty, false otherwise.

q.size(): Returns the number of elements in the queue.

**Vector (std::vector)**

v.push\_back(element): Appends an element to the end.

v.pop\_back(): Removes the last element.

v.front(): Access the first element.

v.back(): Access the last element.

v.size(): Returns the number of elements.

v.empty(): Checks if the vector is empty.

v.clear(): Removes all elements.

v.insert(iterator, element): Inserts an element before the specified iterator.

v.erase(iterator): Removes the element at the specified iterator.

numbers.assign(2, 3); // Now numbers will be {3, 3}

// Filling with a value (assign 3 as the value, 2 elements)

**List (std::list)**

l.push\_front(element): Inserts an element at the front.

l.pop\_front(): Removes the first element.

l.push\_back(element): Inserts an element at the back.

l.pop\_back(): Removes the last element.

l.front(): Access the first element.

l.back(): Access the last element.

l.size(): Returns the number of elements.

l.empty(): Checks if the list is empty.

l.clear(): Removes all elements.

l.insert(iterator, element): Inserts an element before the specified iterator.

l.erase(iterator): Removes the element at the specified iterator.

**Deque (std::deque)**

d.push\_front(element): Inserts an element at the front.

d.pop\_front(): Removes the first element.

d.push\_back(element): Inserts an element at the back.

d.pop\_back(): Removes the last element.

d.front(): Access the first element.

d.back(): Access the last element.

d.size(): Returns the number of elements.

d.empty(): Checks if the deque is empty.

d.clear(): Removes all elements.

**Set (std::set)**

s.insert(element): Inserts an element (no duplicates).

s.erase(element): Removes an element.

s.find(element): Returns an iterator to the element if found, otherwise end().

s.size(): Returns the number of elements.

s.empty(): Checks if the set is empty.

s.clear(): Removes all elements.

**Map (std::map)**

m[key] = value: Inserts or updates a key-value pair.

m.erase(key): Removes a key-value pair by key.

m.find(key): Returns an iterator to the key-value pair if found, otherwise end().

m.size(): Returns the number of key-value pairs.

m.empty(): Checks if the map is empty.

m.clear(): Removes all key-value pairs.

m.count('k')

return 1 if key 'k' exixts in the map, else returns 0

**Unordered\_set (std::unordered\_set)**

Similar to std::set, but with faster average lookup time.

Same common methods as std::set.

**Unordered\_map (std::unordered\_map)**

Similar to std::map, but with faster average lookup time.

Same common methods as std::map.

std::vector<std::vector<int>> ans;

void **getAllPermutations**(std::vector<int> vec, int pos)//pos = 0

{

if (pos == vec.size())

{

ans.push\_back(vec);

return;

}

for (int i = pos; i < vec.size(); i++)

{

std::swap(vec[i], vec[pos]);

getAllPermutations(vec, pos + 1);

std::swap(vec[i], vec[pos]);

}

}

**int binarySearch**(int arr[], int n, int key)

{

int start = 0, end = n - 1;

while (start <= end)

{

int mid = start + (end - start) / 2;

if (arr[mid] == key)

{

return mid;

}

else if (arr[mid] > key)

{

end = mid - 1;

}

else

{

start = mid + 1;

}

}

return -1;

}

long long **countInversionMerge**(std::vector<int> &array, int start, int mid, int end)

{

long long int inversion = 0;

int firstLen = mid - start + 1;

int secondLen = end - mid;

std::vector<int> firstArray(firstLen);

std::vector<int> secondArray(secondLen);

for (int i = 0; i < firstLen; i++)

{

firstArray[i] = array[i + start];

}

for (int i = 0; i < secondLen; i++)

{

secondArray[i] = array[i + mid + 1];

}

int firstIndex = 0, secondIndex = 0;

int mergeIndex = start;

while (firstIndex < firstLen && secondIndex < secondLen)

{

if (firstArray[firstIndex] <= secondArray[secondIndex])

{

array[mergeIndex] = firstArray[firstIndex];

firstIndex++;

mergeIndex++;

}

else

{

array[mergeIndex] = secondArray[secondIndex];

secondIndex++;

mergeIndex++;

inversion += (firstLen - firstIndex);

}

}

while (firstIndex < firstLen)

{

array[mergeIndex] = firstArray[firstIndex];

firstIndex++;

mergeIndex++;

}

while (secondIndex < secondLen)

{

array[mergeIndex] = secondArray[secondIndex];

secondIndex++;

mergeIndex++;

}

return inversion;

}

long long countInversionMergeSort(std::vector<int> &array, int start, int end)

{

long long int inversion = 0;

if (start < end)

{

int mid = start + (end - start) / 2;

inversion += countInversionMergeSort(array, start, mid);

inversion += countInversionMergeSort(array, mid+1, end);

inversion += countInversionMerge(array, start, mid, end);

}

return inversion;

}

int main()

{

int n;

std::cin >> n;

std::vector<int> array(n);

for (int i = 0; i < n; i++)

{

std::cin >> array[i];

}

std::cout << countInversionMergeSort(array, 0, n - 1);

return 0;

}

**int knapSack**(int value[], int wt[], int n, int W)

{

if (n == 0 || W == 0)

{

return 0;

}

if (wt[n - 1] >= W)

{

return knapSack(value, wt, n - 1, W);

}

return std::max(knapSack(value, wt, n - 1, W - wt[n - 1]) + value[n - 1], knapSack(value, wt, n - 1, W));

}

#include <iostream>

using namespace std;

void **merge(**int \*arr, int start, int mid, int end)

{

const int firstLen = (mid - start) + 1;

const int secondLen = (end - mid);

int \*firstArr = new int[firstLen];

int \*secondArr = new int[secondLen];

for (int i = 0; i < firstLen; i++)

{

firstArr[i] = arr[start + i];

}

for (int i = 0; i < secondLen; i++)

{

secondArr[i] = arr[mid + 1 + i];

}

int firstIndex = 0, secondIndex = 0;

int mergeIndex = start; // we used start because it is the main array passed by pointer or reference

// we can't use 0 because it passed fully in recursive call but with different start and end value

while (firstIndex < firstLen && secondIndex < secondLen)

{

if (firstArr[firstIndex] <= secondArr[secondIndex])

{

arr[mergeIndex] = firstArr[firstIndex];

firstIndex++;

mergeIndex++;

}

else

{

arr[mergeIndex] = secondArr[secondIndex];

secondIndex++;

mergeIndex++;

}

}

while (firstIndex < firstLen)

{

arr[mergeIndex] = firstArr[firstIndex];

firstIndex++;

mergeIndex++;

}

while (secondIndex < secondLen)

{

arr[mergeIndex] = secondArr[secondIndex];

secondIndex++;

mergeIndex++;

}

delete[] firstArr, secondArr;

return;

}

void mergeSort(int \*arr, int start, int end)

{

if (start >= end)

{

return;

}

int mid = start + (end - start) / 2;

mergeSort(arr, start, mid);

mergeSort(arr, mid + 1, end);

merge(arr, start, mid, end);

}

int main()

{

int n;

cin >> n;

int arr[n];

for (int i = 0; i < n; i++)

{

cin >> arr[i];

}

cout << "given array is : ";

for (int i = 0; i < n; i++)

{

cout << arr[i] << " ";

}

mergeSort(arr, 0, n - 1);

cout << endl;

cout << "Sorted array is : ";

for (int i = 0; i < n; i++)

{

cout << arr[i] << " ";

}

return 0;

}

void **sieve**(int n)

{

vector<int> vec(n, 0);

for (int i = 2; i <= n; i++)

{

if (vec[i] == 0)

{

for (int j = i \* i; j <= n; j += i)

{

vec[j] = 1;

}

}

}

for (int i = 2; i <= n; i++)

{

if (vec[i] == 0)

{

cout << i << " ";

}

}

}

**Sliding Window:**

int main()

{

// time complexity O(n)

int arraySize, windowSize;

cin >> arraySize >> windowSize;

vector<int> array(arraySize);

for (auto &it : array)

{

cin >> it;

}

deque<int> q;

vector<int> ans;

for (int i = 0; i < arraySize; i++)

{

if (!q.empty() and q.front() == i - windowSize)

q.pop\_front();

while (!q.empty() and array[q.back()] <= array[i])

{

q.pop\_back();

}

q.push\_back(i);

if (i >= windowSize - 1)

ans.push\_back(array[q.front()]);

}

for (auto it : ans)

{

cout << it << " ";

}

return 0;

}

**maxSum:**

int main()

{

int n;

cin >> n;

vector<int> vec(n);

int maxSum = 0, currentSum = 0;

for (auto &it : vec)

{

cin >> it;

}

for (auto it : vec)

{

currentSum += it;

if (currentSum < 0)

currentSum = 0;

maxSum = max(maxSum, currentSum);

}

cout << "MAX Sum is: " << maxSum << endl;

return 0;

}  
**Kadane Circular MaxSum**

int kadane(const vector<int> &vec)

{

int currSum = 0;

int maxSum = INT\_MIN;

for (auto it : vec)

{

currSum += it;

if (currSum < 0)

{

currSum = 0;

}

maxSum = max(currSum, maxSum);

}

return maxSum;

}

int main()

{

int n;

cin >> n;

vector<int> vec(n);

for (auto &it : vec)

{

cin >> it;

}

int wrapSum = 0;

int nonWrapSum = 0;

int totalSum = 0;

nonWrapSum = kadane(vec);

for (auto &it : vec)

{

totalSum += it;

it = -it;

}

wrapSum = totalSum + kadane(vec);

cout << max(wrapSum, nonWrapSum) << endl;

return 0;

}

std::vector<std::vector<int>> &**threeSum(**std::vector<int> &arr, int target)

{

static std::vector<std::vector<int>> result;

if (arr.size() < 3)

return result;

for (int i = 0; i < arr.size() - 2; i++)

{

if (arr[i] == arr[i + 1])

continue;

int left = i + 1;

int right = arr.size() - 1;

while (left < right)

{

int sum = arr[i] + arr[left] + arr[right];

if (sum == target)

{

result.push\_back({arr[i], arr[left], arr[right]});

while (left < right and arr[left] == arr[left + 1])

left++;

while (left < right and arr[right] == arr[right - 1])

right--;

left++;

right--;

}

else if (sum < target)

left++;

else

right--;

}

}

return result;

}

**BEST SUM:**

std::unordered\_map<int, std::vector<int>> memo;

std::vector<int> bestSum(int targetSum, std::vector<int> &numbers)

{

if (memo.find(targetSum) != memo.end())

return memo[targetSum];

if (targetSum == 0)

return {};

if (targetSum < 0)

return {-1};

std::vector<int> shortestCombination = {-1};

for (auto it : numbers)

{

std::vector<int> tempResult = bestSum(targetSum - it, numbers);

if (tempResult.empty() || tempResult[0] != -1)

{

tempResult.push\_back(it);

if (shortestCombination[0] == -1 || shortestCombination.size() > tempResult.size())

{

shortestCombination = tempResult;

}

}

}

memo[targetSum] = shortestCombination;

return shortestCombination;

}

**PrefixSum 2D:**

#include <iostream>

const int N = 1e3 + 10;

int matrix[N][N];

long long prefMatrix[N][N];

int main()

{

int n, m;

std::cin >> n >> m;

for (int i = 1; i <= n; i++)

for (int j = 1; j <= m; j++)

std::cin >> matrix[i][j];

for (int i = 1; i <= n; i++)

for (int j = 1; j <= m; j++)

prefMatrix[i][j] = matrix[i][j] + prefMatrix[i - 1][j] + prefMatrix[i][j - 1] - prefMatrix[i - 1][j - 1];

int query;

std::cin >> query;

while (query--)

{

int a, b, c, d;

std::cin >> a >> b >> c >> d;

std::cout << prefMatrix[c][d] - prefMatrix[a - 1][d] - prefMatrix[c][b - 1] + prefMatrix[a - 1][b - 1] << std::endl;

}

return 0;

}

**Common STL Containers characteristics:**

Vector (std::vector)

Dynamically resizable array.

Fast random access (O(1)). Efficient insertions/deletions at the end (O(1) amortized). Not efficient for insertions/deletions in the middle (O(n)).

Use cases: Storing large datasets with frequent access by index, implementing dynamic arrays for algorithms.

List (std::list)

Doubly-linked list. Efficient insertions/deletions at any position (O(1)). Slower random access compared to vectors (O(n) in the worst case).

Use cases: Implementing stacks or queues (though std::stack and std::queue might be better for simplicity), representing graphs or other linked data structures.

Deque (std::deque)

Similar to vector, but supports efficient insertions/deletions at both ends (O(1) amortized). Slower random access compared to vectors (O(n) in the worst case).

Use cases: Frequent insertions/deletions at both ends with less important random access, implementing queues or double-ended queues.

Set (std::set)

Stores unique elements, sorted in ascending order. Fast average lookup time (O(log n)). No duplicates allowed.

Use cases: Representing sets with fast lookups and uniqueness requirements, implementing disjoint-set forests (union-find).

Map (std::map)

Associative container for key-value pairs. Keys are unique and sorted in ascending order. Fast average lookup, insertion, and deletion (O(log n)).

Use cases: Implementing dictionaries or hash tables with fast access by key, representing adjacency lists for graphs.

Unordered\_set (std::unordered\_set)

Similar to std::set with faster average lookup time (O(1) on average, but can vary) using hashing. No duplicates allowed, may not maintain specific element order.

Use cases: When fast average lookup is the priority and order doesn't matter (e.g., frequency counters).

Unordered\_map (std::unordered\_map)

Similar to std::map with faster average lookup time (O(1) on average, but can vary) using hashing. Keys are unique, may not maintain specific key-value pair order.

Use cases: When fast average lookup of key-value pairs is the priority and order doesn't matter (e.g., counting element occurrences in a string).

**DataTypes and limits**

char:

Variable: CHAR\_MAX for maximum value, CHAR\_MIN for minimum value.

unsigned char:

Variable: UCHAR\_MAX for maximum value.

short:

Variable: SHRT\_MAX for maximum value, SHRT\_MIN for minimum value.

unsigned short:

Variable: USHRT\_MAX for maximum value.

int:

Variable: INT\_MAX for maximum value, INT\_MIN for minimum value.

unsigned int:

Variable: UINT\_MAX for maximum value.

long:

Variable: LONG\_MAX for maximum value, LONG\_MIN for minimum value.

unsigned long:

Variable: ULONG\_MAX for maximum value.

long long:

Variable: LLONG\_MAX for maximum value, LLONG\_MIN for minimum value.

unsigned long long:

Variable: ULLONG\_MAX for maximum value.

void **bitsNumber**(vector<bool>&givenNumber, ll num)

{

while(num> 0)

{

givenNumber.push\_back(num&1);

num >>= 1;

}

reverse(all(givenNumber));

}

**Character Functions**

isalpha(c) It returns True if C is an uppercase letter and False if c is lowercase.

isdigit(c) It returns True if c is a digit (0 through 9) otherwise False.

isalnum(c) It returns True if c is a digit from 0 through 9 or an alphabetic character (uppercase or lowercase)

islower(c) It returns True if C is a lowercase letter otherwise False.

isupper(c) It returns True if C is an uppercase letter otherwise False.

toupper(c) It converts c to uppercase letter.

tolower(c) It converts c to lowercase

**STL Algorithms and Functions**

Sorting:

std::sort:

How to use: std::sort(first, last)

first and last define the range of elements to sort.

What it does: Sorts the elements in the range [first, last) in ascending order.

std::stable\_sort:

How to use: std::stable\_sort(first, last)

first and last define the range of elements to sort.

What it does: Sorts the elements in the range [first, last) in ascending order, preserving the relative order of elements with equivalent values.

std::partial\_sort:

How to use: std::partial\_sort(first, middle, last)

first, middle, and last define the range of elements.

middle points to the element in the range where the sorted sequence should end.

What it does: Partially sorts the range [first, last) such that the elements before middle are sorted in ascending order, and the rest of the elements are unordered.

std::nth\_element:

How to use: std::nth\_element(first, nth, last)

first, nth, and last define the range of elements.

nth points to the element that should appear in its sorted position if the range were fully sorted.

What it does: Reorders the elements in the range [first, last) such that the element at the position pointed to by nth is the one that would be in that position if the range was fully sorted.

Binary Search:

std::binary\_search:

How to use: std::binary\_search(first, last, value)

first and last define the range of elements.

value is the value to search for.

What it does: Checks if the sorted range [first, last) contains the specified value, returning true if found, false otherwise.

std::lower\_bound:

How to use: std::lower\_bound(first, last, value)

first and last define the range of elements.

value is the value to search for.

What it does: Returns an iterator pointing to the first element in the sorted range [first, last) that is not less than value.

std::upper\_bound:

How to use: std::upper\_bound(first, last, value)

first and last define the range of elements.

value is the value to search for.

What it does: Returns an iterator pointing to the first element in the sorted range [first, last) that is greater than value.

std::equal\_range:

How to use: std::equal\_range(first, last, value)

first and last define the range of elements.

value is the value to search for.

What it does: Returns a range of iterators [lower\_bound, upper\_bound) that represent the subrange of elements in the sorted range [first, last) that are equivalent to value.

Permutations:

std::next\_permutation:

How to use: std::next\_permutation(first, last)

first and last define the range of elements.

What it does: Rearranges the elements in the range [first, last) into the next lexicographically greater permutation.

std::prev\_permutation:

How to use: std::prev\_permutation(first, last)

first and last define the range of elements.

What it does: Rearranges the elements in the range [first, last) into the previous lexicographically smaller permutation.

Merge Operations:

std::merge:

How to use: std::merge(first1, last1, first2, last2, result)

first1 and last1 define the first sorted range.

first2 and last2 define the second sorted range.

result is the output iterator where the merged range will be stored.

What it does: Merges two sorted ranges [first1, last1) and [first2, last2) into a single sorted range in non-descending order.

std::inplace\_merge:

How to use: std::inplace\_merge(first, middle, last)

first, middle, and last define the range of elements.

middle points to the element dividing the two sorted subranges to merge.

What it does: Merges two consecutive sorted ranges [first, middle) and [middle, last) into one sorted range in [first, last).

Set Operations:

std::set\_union:

How to use: std::set\_union(first1, last1, first2, last2, result)

first1 and last1 define the first sorted range.

first2 and last2 define the second sorted range.

result is the output iterator where the union of the two ranges will be stored.

What it does: Computes the union of two sorted ranges [first1, last1) and [first2, last2) and stores the result in the output range.

std::set\_intersection:

How to use: std::set\_intersection(first1, last1, first2, last2, result)

first1 and last1 define the first sorted range.

first2 and last2 define the second sorted range.

result is the output iterator where the intersection of the two ranges will be stored.

What it does: Computes the intersection of two sorted ranges [first1, last1) and [first2, last2) and stores the result in the output range.

std::set\_difference:

How to use: std::set\_difference(first1, last1, first2, last2, result)

first1 and last1 define the first sorted range.

first2 and last2 define the second sorted range.

result is the output iterator where the difference of the two ranges will be stored.

What it does: Computes the difference of two sorted ranges [first1, last1) and [first2, last2) and stores the result in the output range.

std::set\_symmetric\_difference:

How to use: std::set\_symmetric\_difference(first1, last1, first2, last2, result)

first1 and last1 define the first sorted range.

first2 and last2 define the second sorted range.

result is the output iterator where the symmetric difference of the two ranges will be stored.

What it does: Computes the symmetric difference of two sorted ranges [first1, last1) and [first2, last2) and stores the result in the output range. The symmetric difference of two sets A and B is the set of elements that are in A or B but not in their intersection.

Heap Operations:

std::make\_heap:

How to use: std::make\_heap(first, last)

first and last define the range of elements.

What it does: Rearranges the elements in the range [first, last) into a max heap.

Example:

cpp

Copy code

std::vector<int> vec = {3, 1, 4, 1, 5, 9};

std::make\_heap(vec.begin(), vec.end());

std::push\_heap:

How to use: std::push\_heap(first, last)

Assumes the last element of the range [first, last-1] is a valid heap and adds the new element to the heap.

What it does: Inserts the element at the end of the heap and maintains the heap property.

Example:

cpp

Copy code

std::vector<int> vec = {3, 1, 4, 1, 5, 9};

vec.push\_back(6);

std::push\_heap(vec.begin(), vec.end());

std::pop\_heap:

How to use: std::pop\_heap(first, last)

Assumes the first element of the range [first, last) is the maximum element and moves it to the end.

What it does: Swaps the maximum element with the last element and rebuilds the heap with the range [first, last-1).

Example:

cpp

Copy code

std::vector<int> vec = {3, 1, 4, 1, 5, 9};

std::pop\_heap(vec.begin(), vec.end());

vec.pop\_back(); // Removing the maximum element

std::sort\_heap:

How to use: std::sort\_heap(first, last)

Assumes the range [first, last) forms a valid heap and sorts it into non-decreasing order.

What it does: Repeatedly calls std::pop\_heap on the range [first, last) to sort it.

Example:

cpp

Copy code

std::vector<int> vec = {3, 1, 4, 1, 5, 9};

std::make\_heap(vec.begin(), vec.end());

std::sort\_heap(vec.begin(), vec.end());

Min/Max Operations:

std::min:

How to use: std::min(a, b)

What it does: Returns the smaller of the two arguments.

Example:

cpp

Copy code

int a = 5, b = 10;

int min\_val = std::min(a, b); // min\_val will be 5

std::max:

How to use: std::max(a, b)

What it does: Returns the larger of the two arguments.

Example:

cpp

Copy code

int a = 5, b = 10;

int max\_val = std::max(a, b); // max\_val will be 10

std::min\_element:

How to use: std::min\_element(first, last)

first and last define the range of elements.

What it does: Returns an iterator to the smallest element in the range [first, last).

Example:

cpp

Copy code

std::vector<int> vec = {3, 1, 4, 1, 5, 9};

auto min\_it = std::min\_element(vec.begin(), vec.end());

std::max\_element:

How to use: std::max\_element(first, last)

first and last define the range of elements.

What it does: Returns an iterator to the largest element in the range [first, last).

Example:

cpp

Copy code

std::vector<int> vec = {3, 1, 4, 1, 5, 9};

auto max\_it = std::max\_element(vec.begin(), vec.end());

Counting Operations:

std::count:

How to use: std::count(first, last, value)

first and last define the range of elements.

value is the value to count occurrences of.

What it does: Returns the number of elements equal to value in the range [first, last).

Example:

cpp

Copy code

std::vector<int> vec = {3, 1, 4, 1, 5, 1};

int count\_one = std::count(vec.begin(), vec.end(), 1); // count\_one will be 3

std::count\_if:

How to use: std::count\_if(first, last, pred)

first and last define the range of elements.

pred is a unary predicate function or functor.

What it does: Returns the number of elements in the range [first, last) for which pred returns true.

Example:

cpp

Copy code

std::vector<int> vec = {3, 1, 4, 1, 5, 1};

int count\_odd = std::count\_if(vec.begin(), vec.end(), [](int x) { return x % 2 != 0; }); // count\_odd will be 5  
Searching:

std::find:

How to use: std::find(first, last, value)

first and last define the range of elements.

value is the value to search for.

What it does: Returns an iterator to the first occurrence of value in the range [first, last) or last if not found.

Example:

cpp

Copy code

std::vector<int> vec = {3, 1, 4, 1, 5, 9};

auto it = std::find(vec.begin(), vec.end(), 4);

std::find\_if:

How to use: std::find\_if(first, last, pred)

first and last define the range of elements.

pred is a unary predicate function or functor.

What it does: Returns an iterator to the first element in the range [first, last) for which pred returns true, or last if not found.

Example:

cpp

Copy code

std::vector<int> vec = {3, 1, 4, 1, 5, 9};

auto it = std::find\_if(vec.begin(), vec.end(), [](int x) { return x % 2 == 0; });

std::find\_if\_not:

How to use: std::find\_if\_not(first, last, pred)

first and last define the range of elements.

pred is a unary predicate function or functor.

What it does: Returns an iterator to the first element in the range [first, last) for which pred returns false, or last if not found.

Example:

cpp

Copy code

std::vector<int> vec = {3, 1, 4, 1, 5, 9};

auto it = std::find\_if\_not(vec.begin(), vec.end(), [](int x) { return x % 2 == 0; });

std::search:

How to use: std::search(first1, last1, first2, last2)

Searches for the first occurrence of the sequence [first2, last2) in the range [first1, last1).

What it does: Returns an iterator to the beginning of the found sequence in the range [first1, last1), or last1 if not found.

Example:

cpp

Copy code

std::vector<int> vec = {1, 2, 3, 4, 5};

std::vector<int> pattern = {3, 4};

auto it = std::search(vec.begin(), vec.end(), pattern.begin(), pattern.end());

std::search\_n:

How to use: std::search\_n(first, last, count, value)

first and last define the range of elements.

count is the number of consecutive occurrences to search for.

value is the value to search for.

What it does: Searches for count consecutive occurrences of value in the range [first, last).

Example:

cpp

Copy code

std::vector<int> vec = {1, 2, 2, 2, 3, 4, 5};

auto it = std::search\_n(vec.begin(), vec.end(), 3, 2);

Accumulation Operations:

std::accumulate:

How to use: std::accumulate(first, last, init)

first and last define the range of elements.

init is the initial value for accumulation.

What it does: Computes the sum of the elements in the range [first, last) starting with the initial value init.

Example:

cpp

Copy code

std::vector<int> vec = {1, 2, 3, 4, 5};

int sum = std::accumulate(vec.begin(), vec.end(), 0);

std::partial\_sum:

How to use: std::partial\_sum(first, last, result)

first and last define the range of elements.

result is the output iterator where the partial sums will be stored.

What it does: Computes the partial sums of the elements in the range [first, last) and stores the result in the output range.

Example:

cpp

Copy code

std::vector<int> vec = {1, 2, 3, 4, 5};

std::vector<int> result(5);

std::partial\_sum(vec.begin(), vec.end(), result.begin());

std::exclusive\_scan:

How to use: std::exclusive\_scan(first, last, result, init)

first and last define the range of elements.

result is the output iterator where the exclusive scan results will be stored.

init is the initial value for the exclusive scan.

What it does: Computes the exclusive scan (prefix sum) of the elements in the range [first, last) and stores the result in the output range.

Example:

cpp

Copy code

std::vector<int> vec = {1, 2, 3, 4, 5};

std::vector<int> result(5);

std::exclusive\_scan(vec.begin(), vec.end(), result.begin(), 0);

std::inclusive\_scan:

How to use: std::inclusive\_scan(first, last, result)

first and last define the range of elements.

result is the output iterator where the inclusive scan results will be stored.

What it does: Computes the inclusive scan (inclusive prefix sum) of the elements in the range [first, last) and stores the result in the output range.

Example:

cpp

Copy code

std::vector<int> vec = {1, 2, 3, 4, 5};

std::vector<int> result(5);

std::inclusive\_scan(vec.begin(), vec.end(), result.begin());

Random Shuffle:

std::shuffle:

How to use: std::shuffle(first, last, g)

first and last define the range of elements.

g is the random number generator engine.

What it does: Randomly shuffles the elements in the range [first, last) using the provided random number generator.

Example:

cpp

Copy code

std::vector<int> vec = {1, 2, 3, 4, 5};

std::shuffle(vec.begin(), vec.end(), std::mt19937(std::random\_device()()));

Other Utilities:

std::swap:

How to use: std::swap(a, b)

Swaps the values of a and b.

What it does: Exchanges the values of the two arguments.

Example:

cpp

Copy code

int a = 5, b = 10;

std::swap(a, b);

std::move:

How to use: std::move(x)

Casts x to an rvalue reference.

What it does: Indicates that the object x can be safely moved from or treated as a temporary.

Example:

cpp

Copy code

std::string str = "Hello";

std::string moved\_str = std::move(str);

std::minmax:

How to use: std::minmax(a, b)

Returns a pair containing the minimum and maximum of a and b.

What it does: Determines both the minimum and maximum of the two arguments without redundant comparisons.

Example:

cpp

Copy code

int a = 5, b = 10;

auto minmax\_val = std::minmax(a, b);

std::distance:

How to use: std::distance(first, last)

first and last define the range of elements.

What it does: Computes the number of elements in the range [first, last).

Example:

cpp

Copy code

std::vector<int> vec = {1, 2, 3, 4, 5};

int size = std::distance(vec.begin(), vec.end());

std::begin / std::end:

How to use: std::begin(container) / std::end(container)

Returns iterators to the beginning and one past the end of the container, respectively.

What it does: Provides a convenient way to obtain iterators to the beginning and end of a container, regardless of its type.

Example:

cpp

Copy code

std::vector<int> vec = {1, 2, 3, 4, 5};

auto first = std::begin(vec);

auto last = std::end(vec);

String Manipulation:

std::getline:

How to use: std::getline(input\_stream, str)

input\_stream is the input stream to read from.

str is the string where the input will be stored.

What it does: Reads a line of input from the input stream and stores it in the string str, discarding the newline character.

Example:

cpp

Copy code

std::string line;

std::getline(std::cin, line);

std::stoi, std::stol, std::stoll, std::stoul, std::stoull:

How to use: std::stoi(str), std::stol(str), std::stoll(str), std::stoul(str), std::stoull(str)

str is the string to convert.

What it does: Converts the string representation of a number to an integer (stoi), long (stol), long long (stoll), unsigned long (stoul), or unsigned long long (stoull).

Example:

cpp

Copy code

std::string num\_str = "123";

int num = std::stoi(num\_str);

std::to\_string:

How to use: std::to\_string(value)

value is the value to convert to a string.

What it does: Converts a numerical value to its string representation.

Example:

cpp

Copy code

int num = 123;

std::string num\_str = std::to\_string(num);

Math Functions:

std::abs, std::fabs:

How to use: std::abs(x), std::fabs(x)

x is the input value.

What it does: Returns the absolute value of x. std::abs works for integral types, while std::fabs works for floating-point types.

Example:

cpp

Copy code

int num = -5;

double val = -3.14;

int abs\_num = std::abs(num);

double abs\_val = std::fabs(val);

std::pow, std::sqrt, std::log, std::exp:

How to use: std::pow(base, exponent), std::sqrt(x), std::log(x), std::exp(x)

base and exponent are the base and exponent values for power function.

x is the input value.

What it does: Computes the power, square root, natural logarithm, and exponential of the input value, respectively.

Example:

cpp

Copy code

double result\_pow = std::pow(2, 3); // result\_pow will be 8.0

double result\_sqrt = std::sqrt(16); // result\_sqrt will be 4.0

double result\_log = std::log(10); // result\_log will be approximately 2.30259

double result\_exp = std::exp(1); // result\_exp will be approximately 2.71828

std::ceil, std::floor, std::round:

How to use: std::ceil(x), std::floor(x), std::round(x)

x is the input value.

What it does: Rounds x to the smallest integer greater than or equal to x (ceil), largest integer less than or equal to x (floor), or to the nearest integer (round).

Example:

cpp

Copy code

double val = 3.6;

double ceil\_val = std::ceil(val); // ceil\_val will be 4.0

double floor\_val = std::floor(val); // floor\_val will be 3.0

double round\_val = std::round(val); // round\_val will be 4.0

Time Functions:

std::chrono (for precise timing):

How to use: Various utilities are available in the <chrono> header for precise timing, such as std::chrono::high\_resolution\_clock, std::chrono::duration, etc.

What it does: Provides a high-resolution clock and utilities for measuring time durations.

Example:

cpp

Copy code

#include <chrono>

auto start = std::chrono::high\_resolution\_clock::now();

// Code to measure time

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> duration = end - start;

std::cout << "Time taken: " << duration.count() << " seconds" << std::endl;

Memory Management:

std::vector::reserve:

How to use: std::vector::reserve(n)

n is the number of elements to allocate memory for.

What it does: Reserves memory to accommodate at least n elements without reallocation.

Example:

cpp

Copy code

std::vector<int> vec;

vec.reserve(100); // Reserve memory for at least 100 elements

std::vector::resize:

How to use: std::vector::resize(n)

n is the new size of the vector.

What it does: Resizes the vector to contain n elements. If n is greater than the current size, additional elements are default-inserted. If n is less than the current size, elements are removed from the end.

Example:

cpp

Copy code

std::vector<int> vec = {1, 2, 3};

vec.resize(5); // Resize vector to contain 5 elements

std::vector::shrink\_to\_fit:

How to use: std::vector::shrink\_to\_fit()

What it does: Reduces the capacity of the vector to fit its size.

Example:

cpp

Copy code

std::vector<int> vec = {1, 2, 3};

vec.shrink\_to\_fit(); // Reduce capacity to fit size

std::vector::assign

How to use:

std::vector::assign allows you to assign new values to the vector, replacing its current contents.

There are several overloads for assign:

void assign(size\_type count, const T& value): Assigns count copies of value value to the vector.

template<class InputIterator> void assign(InputIterator first, InputIterator last): Assigns values from the range [first, last) to the vector.

void assign(std::initializer\_list<T> ilist): Assigns values from the initializer list ilist to the vector.

What it does:

Replaces the current contents of the vector with new values specified either by count, a range, or an initializer list.

Examples:

Assigning a specified number of copies:

cpp

Copy code

std::vector<int> vec;

vec.assign(5, 10); // Assigns 5 copies of value 10 to the vector

// vec now contains {10, 10, 10, 10, 10}

Assigning values from a range:

cpp

Copy code

std::vector<int> vec;

std::vector<int> new\_values = {1, 2, 3, 4, 5};

vec.assign(new\_values.begin(), new\_values.end());

// vec now contains {1, 2, 3, 4, 5}

Assigning values from an initializer list:

cpp

Copy code

std::vector<int> vec;

vec.assign({1, 2, 3, 4, 5});

// vec now contains {1, 2, 3, 4, 5}  
// Function to convert a long long integer to binary string

string longLongToBinary(long long number) {

return bitset<64>(number).to\_string(); // Assuming long long is 64 bits on your platform

}

// Function to convert a binary string to long long integer

long long binaryToLongLong(string binary) {

return bitset<64>(binary).to\_ullong(); // Assuming long long is 64 bits on your platform

}

ll integerToBinary(ll number)

{

if (number == 0)

{

return 0; // Special case for 0

}

string binary = "";

while (number > 0)

{

binary = to\_string(number % 2) + binary;

number /= 2;

}

return stoll(binary);

}

**Now here Tree and Graph algorithms will take place**