8/June/025 (Complexity)

Time Complexity: TC != Time taken

↳ Rate at which the time taken increases with respect to the IP Size.

Big-oh Notation ⇒ O(↑)
time taken

↳ Points to be taken –
→ TC, worst case scenario
→ avoid constraints
→ avoid lower values

Big-oh (O)	Theta (Θ)	Omega (Ω)
↓	↓	↓
Worst case		
[Upper bound]	[Average Complexity]	[Lower Bound]

Space Complexity: → Memory space
→ Big-oh Notation

→ Auxilliary space → space that you take to solve the problem

\+

Input Space → space that you take to store the Input

Note:- code → Server
↓
[1s ≅ 10^8 operations]

(C++ STL)

→ Algorithms
→ Containers
→ Functions
→ Iterators

STL → Standard Template Library

⇒ STL → STL is basically compilation of containers, algorithms, iterators, fc^n's in a minimized version.

* Pairs :-

```
void explainPairs() {

    pair<int, int> p = {1, 3};
    cout << p.first << " " << p.second;
    pair<int, pair<int, int>> p = {1, {3, 4}};
    cout << p.first << " " << p.second.second << " " << p.second.fir
    pair<int, int> arr[] = {{1,2}, {2,5}, {5,1}};
    cout << arr[1].second;

}
```

⇒ Containers & Iterator :-

1) Vector :- vector is a container which stores elements in a similar fashion as the array does.

Declaration :- `vector<int> v;`

`vector<int> v(5);` → size

`vector<int> v(5, 20);` → {20, 20, 20, 20, 20}

`vector<pair<int, int>> vec;`

⇒ method- (To insert element)

v.push_back(value); and v.emplace_back(value);

⇒ method - (To access element)

↓

with the help of iterator

vector<int>:: iterator it = v.begin();

" " = v.end();

" " = v.rend();

" " = v.rbegin();

Note:-

end() points to the memory location that is right after the last element.

v.back() → last element

Iterator:- points to the memory where the element is lying

auto variable_name → automatically takes data type

⇒ method - (To delete element)

v.erase (iterator) → e.g : v.erase(v.begin()+1);

v.erase (start, end)

⇒ Insert function :-

v.insert (v.begin(), 300) // {300, 100, 100}

v.insert (v.begin()+1, 2, 10); // {300, 10, 10, 100, 100}

⇒ other functions :-

v.size();

v.pop_back(); In {10, 20} returns {10}

v1.swap(v2);

v.clear();

v.empty();

2) List :- container → dynamic size

Declaration :- List<int> ls;

→ functions :-

```
ls.push_back(2);      // {2}
ls.emplace_back(4);   // {2, 4}
ls.push_front(5);     // {5, 2, 4}
ls.emplace_front(6);  // {6, 5, 2, 4}
```

↳ rest fcn same as vector

3) Deque :- similar to list & vector

Declaration :- deque<int> dq;

→ functions :-

```
dq.push_back(1);      // {1}
dq.emplace_back(2);   // {1, 2}
dq.push_front(4);     // {4, 1, 2}
dq.emplace_front(3);  // {3, 4, 1, 2}

dq.pop_back();        // {3, 4, 1}
dq.pop_front();       // {4, 1}

dq.back();
dq.front();
```

↳ rest function's same as vector

4) Stack :- (LIFO)

Declaration :- stack<int> st;

functions :-

```
st.push(1);
st.emplace(5);
st.top();
st.pop();
st.size();
st.empty();
st1.swap(st2);
```

→ Complexity of stack opr^n : O(1)

5) Queue :- (FIFO)

Declaration :- queue <int> q;

functions :-

```
q.push(1);
q.emplace(4);
q.back();
q.front();
q.pop(); → deletes front end element
```

→ Complexity of queue opr^n ? O(1)

6) priority queue :-

// Max Heap -

Declaration :- priority_queue <int> pq;

functions :-

```
pq.push(5);
pq.pop();
pq.top();
```

// Min Heap —

Declaration :- priority_queue <int, vector<int>, greater<int>> pq;

functions :-

```
pq.push(5);
pq.top();
pq.pop();
```

→ complexity of - push - O(log n)

pop - ~~O(1)~~ O(log n)

top - O(1)

7) Set :- sorted & unique

Declaration : set<int> st;

functions :-

st.insert(1);
st.find(3);
st.erase(5);
st.upper_bound(2);
st.count(2);
st.erase(start, end)
st.lower_bound(4);

→ rest fcn same as vector

↳ complexity of each fcn = O(log n)

but st.erase(5) → takes O(log n)

while auto it = st.find(3);

st.erase(it) → takes constant time

↳ address

8) Multiset :- sorted

Declaration : multiset<int> ms;

functions :-

ms.insert(1);
ms.erase(1); → all 1's erased
ms.find(1);

→ rest functions same as set

9) Uset :- unique

Declaration :- unordered set<int> st;

↳ lower bound & upper bound fcn doesn't works, rest fcn are same as above, it does not stores in any particular order it has a better complexity than set most cases, except some when collision happens.

d) Map :- {key, value} [map stores unique key in sorted order.]

Declaration :- map<int, int> mpp;
map<int, pair<int, int>> mpp;
map<pair<int, int>, int> mpp;

[Complexity ↓ O(log n)]

functions :- mpp.emplace({3, 1});
mpp.insert({2, 4});
mpp[1] = 2;
mpp[{2, 3}] = 10;

mpp.find(3); → key
mpp.lower_bound(2);
mpp.upper_bound(3);

→ rest fcn same as set

) Multimap :- everything same as map, only it can store multiple keys i.e. [duplicate keys in sorted order]
only mpp[key] cannot be used here

) Unordered Map :- ~~same~~ same as set & unordered set difference i.e. [duplicate keys in unsorted order]

→ complexity :- O(1) but in worst case O(N)

: Algorithms :-

→ sorting :-

sort(start, end) e.g: sort(a, a+n); → in ascending
sort(start, end, greater<int>); → in descending
sort(a, a+n, comp) → in your way
↳ self written comparator (a boolean fcn)

↳ sort in your way :- (comparator)

Prob: sort it according to second element if secon element is same, then sort it according to first element but in descending

```
pair<int, int> a[] = {{1,2}, {2,1}, {4,1}};
sort(a, a+n, comp);

bool comp(pair<int, int> p1, pair<int, int> p2) {
    if(p1.second < p2.second) return true; (swap intem
    if(p1.second > p2.second) return false;
    // they are same
    if(p1.first > p2.first) return true;
    return false;
}
```

↳ builtin_popcount :- returns no. of set bits (no. of 1'

Ex:-
```
int num=7;
int cnt = __builtin_popcount(); // return 3

long long num= 165786578687;
int cnt= __buitin_popcountLL();
```

↳ next_permutation :- (get all permutations)

Ex:-
```
string s="123";
do {
    cout << s << endl;
} while(next_permutation(s.begin(), s.end()));
```

⇒ prints in dictionary order

↳ max_element :- To get maximum element

Ex:- int maxi = *max_element (a, a+n);

⇒ Similarly min_element is also there.

9/June/25

(Basic Maths)

Extracting digits of a number :-

⇒ 7789 → N = N%10 → Digits
N = N/10 → remaining digits
↳ Complexity: $O(\log_{10}(N))$

⇒ ~~int cnt~~ = To count no. of digits -

int cnt = (int)[log10(N)+1];

Reverse of a number :-

reverse num = reversenum x 10 + Lastnum

Print all factors of a number :-

first method by loop if (N%i==0) but time Complexity of this method is O(n).

⇒ another method -

```
vector<int> v;
for(int i=1; i<sqrt(n); i++){ → O(sqrt(n))
    if(n%i==0){
        v.push_back(i);
```

```
        if((n/i) != i) {
            v.push_back(n/i);
        }
    }
}
```

sort(v.begin(), v.end()); → O(n log n) where n → no. of factors
for(auto it : v) cout << it << " "; → O(n)

⇒ u can use sqrt(n) for prime no. also.

GCD / HCF :-

```
for(i = min(n1, n2); i >= 1; i--) {      // (optimal code & many cases)
    if(N1 % i == 0 && N2 % i == 0) {
        print(i);
        break;
    }
}
```

Eudedian Algorithm :-

$gcd(N_1, N_2) = gcd(N_1 - N_2, N_2)$ if $N_1 > N_2$

$gcd(a, b) = gcd(a-b, b)$ where $a > b$

Ex :- $N_1 = 15$, $N_2 = 20$

$gcd(20, 15) = gcd(5, 15)$

$gcd(15, 5) = gcd(10, 5)$

$gcd(10, 5) = gcd(5, 5)$

$gcd(5, 5) = gcd(0, 5)$

↓

gcd

Better way –

$$gcd(a, b) = gcd(a \% b, b) \quad \text{where } a > b$$

```
while(a>0 && b>0) {
    if(a>b)   a=a%b
    else      b=b%a
}
if(a==0)  print(b)
else      print(a)
```

// optimal code for GCD/HCF

Complexity: $O(\log_{\phi}(\min(a, b)))$