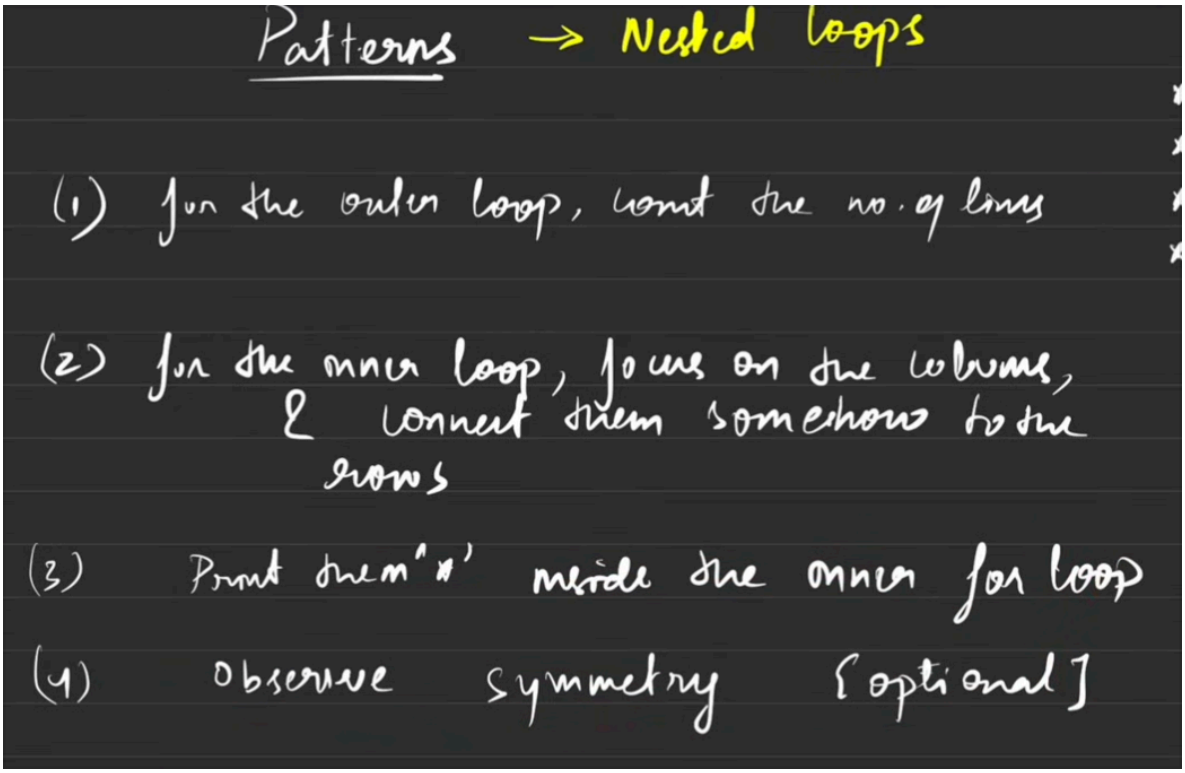
DSA (BASICS)

TIME COMPLEXITY
Rate at which the time taken increases with respect to input size.
Computed in worst case scenario, avoid constants and lower powers

SPACE COMPLEXITY
Auxiliary space — operations performed while using new variables therefore space.
Input Space — Data given through taking input.
“Do not manipulate input data”



STL

1. Pairs

```cpp
// Pairs
void explainPair() {

    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.first;


    pair<int, int> arr[] = { {1, 2}, {2, 5}, {5, 1}};


    cout << arr[1].second;

}
```

2. Vectors

```cpp
vector<int> v;

v.push_back(1);
v.emplace_back(2);

vector<pair<int, int>>vec;

v.push_back({1, 2});
v.emplace_back(1, 2);
```

.emplace_back is faster and for pairs the syntax is slightly different

```cpp
vector<int> v(5, 100);


vector<int> v(5);

vector<int> v1(5, 20);
vector<int> v2(v1);
```

vector<int> v(size, default value)
Last line here is for copying the vectors

Accessing elements in the vector —>
First is normal way like v[2]
Next is using iterators

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

it++;
cout << *(it) << " ";


it = it + 2;
cout << *(it) << " ";

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

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

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

Here iterator it is a pointer
.end()—> points to the location adjacent to the last element
.rend()—>points to the location before the first element (Reverse end)
.r begin()—> point to the last element and on it++ the pointer moves towards the start instead of the end.
.back()—> points at the last element

```cpp
for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
    cout << *(it) << " ";
}


for (auto it = v.begin(); it != v.end(); it++) {
    cout << *(it) << " ";
}

for (auto it : v) {
    cout << it << " ";
}
```

auto automatically defines the datatype
In the last one we do not use * (for each loop)

```cpp
// {10, 20, 12, 23}
v.erase(v.begin()+1);

// {10, 20, 12, 23, 35}
v.erase(v.begin() + 2, v.begin() + 4); // // {10, 20, 35} [start, end)
```

.erase to remove elements
Endpoint is not included

```
vector<int>v(2, 100); // {100, 100}
v.insert(v.begin(), 300); // {300, 100, 100};
v.insert(v.begin() + 1, 2, 10); // {300, 10, 10, 100, 100}

vector<int> copy(2, 50); // {50, 50}
v.insert(v.begin(), copy.begin(), copy.end()); // {50, 50, 300, 10, 10, 100, 100}
```

startpoint , Endpoint

```
cout << v.size(); // 2

//{10, 20}
v.pop_back(); // {10}

// v1 -> {10, 20}
// v2 -> {30, 40}
v1.swap(v2); // v1 -> {30, 40} , v2 -> {10, 20}

v.clear(); // erases the entire vector

cout << v.empty();
```

.empty returns true or false based on the state of the vector

3. Lists

```
void explainList() {
    list<int> ls;

    ls.push_back(2); // {2}
    ls.emplace_back(4); // {2, 4}

    ls.push_front(5); // {5, 2, 4};

    ls.emplace_front(); {2, 4};

    // rest functions same as vector
    // begin, end, rbegin, rend, clear, insert, size, swap
}
```

Here difference between vector and list is that list allows operations on the front. We can use insert in case of vector to add elements in the beginning of the vector but the time complexity of insert function is costlier that’s why we use list.

4. Deque

```
void explainDeque() {

    deque<int>dq;
    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 functions same as vector
    // begin, end, rbegin, rend, clear, insert, size, swap
}
```

5. Stack (last in first out)

```
void explainStack() {
    stack<int> st;
    st.push(1); // {1}
    st.push(2); // {2, 1}
    st.push(3); // {3, 2, 1}
    st.push(3); // {3, 3, 2, 1}
    st.emplace(5); // {5, 3, 3, 2, 1}

    cout << st.top(); // prints 5  "** st[2] is invalid **"

    st.pop(); // st looks like {3, 3, 2, 1}

    cout << st.top(); // 3

    cout << st.size(); // 4

    cout << st.empty();

    stack<int>st1, st2;
    st1.swap(st2);

}
```

Pop removes the element whereas top does not remove the element
All operations complexity is O(1)

6. Queue (First in first out)

```
void explainQueue() {
    queue<int> q;
    q.push(1); // {1}
    q.push(2); // {1, 2}
    q.emplace(4); // {1, 2, 4}

    q.back() += 5

    cout << q.back(); // prints 9

    // Q is {1, 2, 9}
    cout << q.front(); // prints 1

    q.pop(); // {2, 9}

    cout << q.front(); // prints 2

    // size swap empty same as stack
}
```

Complexity is O(1)

7. Priority Queue

```
void explainPQ() {
    priority_queue<int>pq;

    pq.push(5); // {5}
    pq.push(2); // {5, 2}
    pq.push(8); // {8, 5, 2}
    pq.emplace(10); // {10, 8, 5, 2}

    cout << pq.top(); // prints 10

    pq.pop(); // {8, 5, 2}

    cout << pq.top(); // prints 8

    // size swap empty function same as others

    // Minimum Heap
    priority_queue<int, vector<int>, greater<int>> pq;
    pq.push(5); // {5}
    pq.push(2); // {2, 5}
    pq.push(8); // {2, 5, 8}
    pq.emplace(10); // {2, 5, 8, 10}

    cout << pq.top(); // prints 2

}
```

Min Heap & Max Heap
Push and pop —> O(log n)
Top —> O(1)

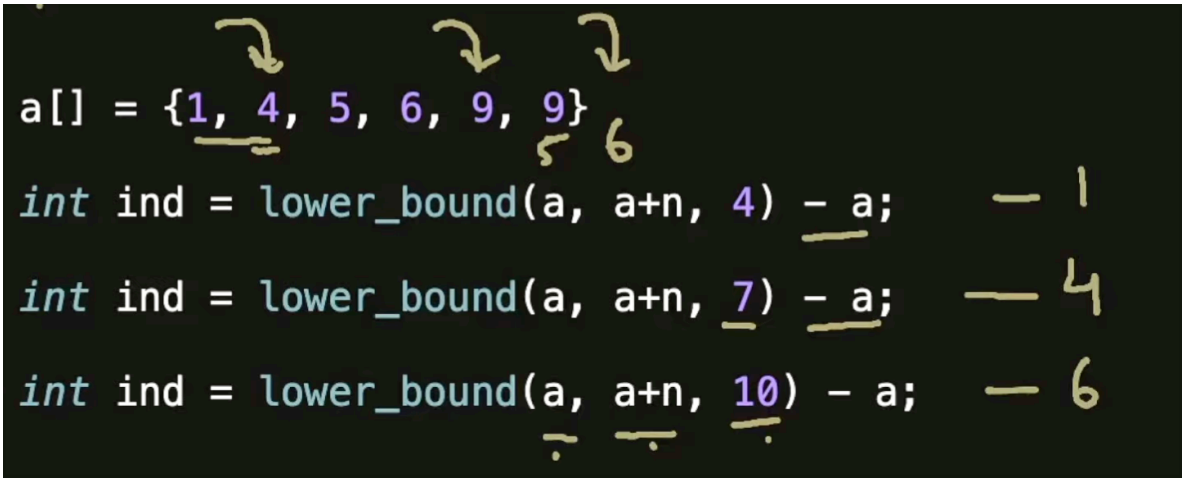
8.Sets(sorted and Unique)

```
void explainSet() {
    set<int>st;
    st.insert(1); // {1}
    st.emplace(2); // {1, 2}
    st.insert(2); // {1, 2}
    st.insert(4); // {1, 2, 4}
    st.insert(3); // {1, 2, 3, 4}

    // Functionality of insert in vector
    // can be used also, that only increases
    // efficiency

    // begin(), end(), rbegin(), rend(), size(),
    // empty() and swap() are same as those of above
```

```
  // {1, 2, 3, 4, 5}
  auto it = st.find(3);

  // {1, 2, 3, 4, 5}
  auto it = st.find(6);

  // {1, 4, 5}
  st.erase(5); // erases 5 // takes logarithmic time


  int cnt = st.count(1);


  auto it = st.find(3);
  st.erase(it); // it takes constant time

  // {1, 2, 3, 4, 5}
  auto it1 = st.find(2);
  auto it2 = st.find(4);
  st.erase(it1, it2); // after erase {1, 4, 5} [first, last)

  // lower_bound() and upper_bound() function works in the same way
  // as in vector it does.

  // This is the syntax
  auto it = st.lower_bound(2);

  auto it = st.upper_bound(3);
```

If an element does not exist it returns the pointer after the last element.
Every task in logarithmic time complexity.

Lower-Bound Function—>




```
Q. Find the first occurrence of a X in a
   sorted array. If it does not exits, print -1.

A[] = {1, 4, 4, 4, 4, 9, 9, 10, 11}


int ind = lower_bound(a, a+n, X) - a;

if(ind != n && a[ind] == X) cout << ind;
else cout << -1;
```

```
Q. Find the largest number smaller than X in
   a sorted array. If no number exists print -1.

A[] = {1, 4, 4, 4, 4, 9, 9, 10, 11}


int ind = lower_bound(a, a+n, X) - a;
ind--;

if(ind>=0) cout << a[ind];
else cout << -1;
```

Upper-Bound Function—>

```
upper_bound function:

a[] = {1, 4, 5, 6, 9, 9}

int ind = upper_bound(a, a+n, 4) - a;

int ind = upper_bound(a, a+n, 7) - a;

int ind = upper_bound(a, a+n, 10) - a;
```

```
Q. Find the last occurrence of a X in a
   sorted array. If it does not exits, print -1.

A[] = {1, 4, 4, 4, 4, 9, 9, 10, 11}


int ind = upper_bound(a, a+n, X) - a;
ind--;

if(ind >= 0 && a[ind] == X) cout << ind;
else cout << -1;
```

```
Q. Find the smallest number greater than X in
   a sorted array. If no number exists print -1.

A[] = {1, 4, 4, 4, 4, 9, 9, 10, 11}


int ind = upper_bound(a, a+n, X) - a;

if(ind<n) cout << a[ind];
else cout << -1;
```

9. Multi set (Sorted but not unique)