STL Containers

Deque/Vector:

- *deque<data type> d;*
- push_front(x): pushes to front of deque
- push_back(x): pushes to back of deque/vector
- pop_front(x): removes element from front of deque
- pop_back(x): removes element from back of deque/vector
- deque.front()/back(): returns element at front/back of the deque
- deque.begin()/end(): iterators for begin and end of deque (rbegin, rend are their reverse)
- deque.size(): returns size of deque
- deque.erase(deque.begin() + index): erases element at index from the beginning of the deque
- deque.insert(deque.begin() + index, input): inserts input at this position
- deque.empty(): checks if deque is empty
- Traversal either with array method or (auto i : vector) method

Priority Queue:

- Stores a pointer to the element with the highest value in the queue
- *priority_queue<data type> pq;*
- *priority_queue<data type, vector<data type>, greater<>> pq:* Points to lowest element instead of highest element
- pq.push(x): adds x to the queue
- pq.pop(x): removes element with the highest value from the queue
- pq.top(x): get element with the highest value
- Traversal only with top > pop

Set:

- Useful for storing values uniquely in a sorted manner
- *set<data type> set;*
- set.insert(x): inserts x to the set
- set.erase(x): deletes x from the set, also works with iterators like begin()
- set.find(x): returns set.end() if x doesn't exist in the set
- Traversal with the auto method
- Multiset is the same but allows multiple values in a sorted manner
- multiset.count(x): counts number of occurrences of x

Map:

- Useful for storing pairs of elements in a unique, sorted manner by key, value pairs.
- *map<key type, value type> map;*
- map.insert({key, value}): inserts the pair into the map
- map[key] = value: if key doesn't exist, make the key and assign the value. If the key exists, update the value directly.
- map.erase(key): erases the key and its associated value from the map
- map[key]++: Fancy way of counting number of occurrences of an element while trying to insert it to the map
- map.find(x): attempts to find x in the map, returns end() on failure
- Traversal is by the auto method with *i.first and *i.second
- Multimap allows multiple keys to exist in a map, erasing a key erases all instances of that key. If there are multiple keys, they can't be called directly and count(x) can be used to count the number of occurrences of a key value.

Frequency Array/Map:

- Useful for counting the number of occurrences of elements in an array.
- If the input space is small enough, create an array from smallest to highest input and increase that index's count by 1 if it occurs.
- For more general cases, use map with <input, frequency> instead.

```
int main() {
    int n, mx = 0;
    string mostOccuring;
    cin >> n;
    unordered_map<string, int> freq;
    for (int i = 0; i < n; ++i) {
        string s;
        cin >> s;
        freq[s]++;
    }
    for (auto &str: freq) {
        if (str.second > mx) {
            mx = str.second;
            mostOccuring = str.first;
        }
    }
    cout << mostOccuring;
}
```

Prefix Array:

Index	0	1	2	3	4	5	6	7	8
Element	1	6	10	8	18	26	32	31	33

```
const int N = 1e6 + 1;
int A[N], prefix_sum[N];
int main() {
    int n, q;
    cin >> n;
    for (int i = 0; i < n; ++i) {
        cin >> A[i];
        prefix_sum[i] = A[i];
        if (i) prefix_sum[i] += prefix_sum[i - 1];
    }
    cin >> q;
    while (q--) {
        int x;
        cin >> x;
        cout << prefix_sum[x] <<"\n";
    }
}
```

- We can get *Sum[L, R]* using:
  - prefix sum array:

$$Sum[L, R] = Prefix\_Sum[R] - Prefix\_Sum[L - 1]$$

```
const int N = 1e6 + 1;
int A[N], prefix_min[N];

int main() {
    int n, q;
    cin >> n;
    for (int i = 0; i < n; ++i)
        cin >> A[i];
    prefix_min[0] = A[0];
    for (int i = 1; i < n; ++i)
        prefix_min[i] = min(prefix_min[i - 1], A[i]);
    cin >> q;
    while (q--) {
        int x;
        cin >> x;
        cout << prefix_min[x] <<"\n";
    }
}
```

Greedy:

- To solve a problem optimally, try solving many local problems optimally hoping to solve the global problem optimally

```
int main() {
    int n, freeTime;
    cin >> n >> freeTime;
    vector<int> completionTime(n);
    for (int i = 0; i < n; ++i) {
        cin >> completionTime[i];
    }
    sort(completionTime.begin(), completionTime.end());
    int curTime = 0, maxTasksCnt = 0;
    for (int i = 0; i < n; ++i) {
        curTime += completionTime[i];
        if (curTime > freeTime)
            break;
        maxTasksCnt++;
    }
    cout << maxTasksCnt;
}
```

Two-Pointers:

- An optimization for bruteforce problems

```
bool twoSum(vector<int>& A, int T) {
    int j = A.size() - 1;
    for (int i = 0; i < j; ++i) {
        while (j > i && A[i] + A[j] > T)
            --j;
        if (A[i] + A[j] == T)
            return true;
    }
    return false;
}
```

```
bool subarraySum(vector<int>& A, int T) {
    int S = 0, E = 0, sum = 0;
    while (S < A.size()) {
        // keep expanding E as long as the new sum <= T
        while (E < A.size() && sum + A[E] <= T) {
            sum += A[E];
            ++E;
        }
        // we found a solution
        if (sum == T) {
            cout<< S <<" "<< E - 1 <<"\n";
            return;
        }
        sum -= A[S++]; // remove the front element, move S one step forward
    }
    cout<< -1 <<"\n"; // no solution
}
```

Code

Binary-Search:

```
int BinarySearch(vector<int>& A, int T) {
    int n = (int)A.size();
    int l = 0, r = n - 1, mid;
    while (l <= r) {
        mid = (l + r) / 2;
        if (A[mid] == T) return mid;
        else if (A[mid] < T) l = mid + 1;
        else r = mid - 1;
    }
    return -1; // T does not exist
}
```

```
int findMinimum(vector<int>& A, int T) {
    int n = (int)A.size();
    int l = 0, r = n - 1, mid;
    while (l < r) {
        mid = (l + r) / 2;
        if (A[mid] > A[n - 1]) l = mid + 1; // first part
        else r = mid; // second part
    }
    // l & r are now pointing to the minimum value
    return l;
}
```

Sieve:

Code

```
const int N = 1e7;

bool is_prime[N + 1];
void sieve() {
    fill(is_prime, is_prime + N, true);
    is_prime[0] = is_prime[1] = false;
    for (int i = 2; i <= N; ++i) {
        if (is_prime[i]) {
            // if i is prime mark all its multiples as composite
            for (int j = i + i; j <= N; j += i)
                is_prime[j] = false;
        }
    }
}
```

IsPrime:

```
bool isPrime(int n) {
 for (int i = 2; i * i <= n; ++i) {
   if (n % i == 0) return false;
 }
 return n > 1;
}

int main() {
 int n;
 cin >> n;
 cout << (isPrime(n) ? "Prime" : "Not a Prime");
}
```