C++ :

Heap

In [ ]:
```
priority_queue<T, Container, Compare> heap; // Wrapper that uses heap operations
// if Compare is less<T> then heap is a max heap (less<T> = {lhs < rhs})
// if Compare is greater<T> then heap is a min heap (greater<T> = {lhs > rhs})
// if compare returns true then its first argument will be before its second argumen
// End of the heap is the top of it
heap.top(); // access the top element O(1)
heap.push(element); // push element to the heap O(lgn)
heap.pop(); // remove the top element from the heap O(lgn)
heap.size(); // access the top element O(1)
```

Stack

In [ ]:
```
stack<T> stack;
stack.top(); // top element in the stack
stack.push(element); // push element to the stack
stack.pop(); // Remove top element from the stack
stack.size(); // Returns the size of the stack
```

Queue

In [ ]:
```
queue<T> queue;
queue.push(element); // Add
queue.front(); // First element in the queue (The next element on the list)
queue.pop(); // Remove the front from the queue
queue.back(); // Access last element in the heap
stack.size(); // Returns the size of the queue
```

set/unordered_set

In [ ]:
```
set.insert(element);
set.erase(element); // erase the element completely from the set
set.count(element); // returns the number of elements matching specific key
set.size(); // How many elements in the set
```

map/unordered_map

In [ ]:
```
pair<iterator, bool> emplace(key, value); // return true if insertion happens, false
// iterator->first => is the key
// iterator->second => is the value
iterator find(key); // returns iterator which is == to map.end() if the element is n
void erase(element); // erase the element completely from the map
```

map

In [ ]:

```
auto b = map.begin(); // returns iterator to the smallest element in the map (This i
b++; // next bigger element in the list
auto e = map.rbegin(); // returns iterator to the biggest element in the map (This i
e--; // next smaller element in the list
```

Utilities

- prev(itr) -> returns the previous iterator used with end usually
- advance(itr, i) -> advance iterator by i'th element
- Lower and upper bound :
  - std::lower_bound - returns iterator to first element in the given range which is EQUAL_TO or Greater than val.
  - std::upper_bound - returns iterator to first element in the given range which is Greater than val.
  - **Example :** lower_bound(data.begin(), data.end(), i); // Find the lower bound of i in data
  - These functions return data.end() if nothing found
- distance(itr1, itr2) return the distance between two iterators

In [ ]:
```
// value a a a b b b c c c
// index 0 1 2 3 4 5 6 7 8
// bound       l     u
// l and u are respectivelly the lower and upper bound of b
```

- nth_element(RandomIt first, RandomIt nth, RandomIt last);
- nth_element is a partial sorting algorithm that rearranges elements in [first, last) such that:
  - The element pointed at by nth is changed to whatever element would occur in that position if [first, last) were sorted.
  - All of the elements before this new nth element are less than or equal to the elements after the new nth element. (Meaning that it puts all elements smaller in the front and all elements larger in the back)

In [ ]:
```
std::vector<int> v{5, 10, 6, 4, 3, 2, 6, 7, 9, 3};
auto m = v.begin() + v.size()/2;
std::nth_element(v.begin(), m, v.end());
// The median is : v[v.size()/2]
// v is now {3, 2, 3, 4, 5, 6, 10, 7, 9, 6, }
```

Arrays/Strings :

Sliding Window :

- Useful to find the longest/shortest string, subarray or desired value. (Have to be somewhat contigious)
- The window size can shrink or grow depending on the problem
- **Note** : Consider prefix sum when the values in the array can be negative
- **Useful links** : https://medium.com/leetcode-patterns/leetcode-pattern-2-sliding-windows-for-strings-e19af105316b

- **Template :**

In [ ]:
```
int max_sum = 0, window_sum = 0;

/* calculate sum of 1st window */
for (int i = 0; i < k; i++)
    window_sum += arr[i];

/* slide window from start to end in array. */
for (int i = k; i < n; i++){
    window_sum += arr[i] - arr[i-k];    // saving re-computation
    max_sum = max(max_sum, window_sum);
}
```

Prefix Sum :

- The idea is to use an array/matrix of prefix sums/products/etc
- Useful when we are searching for a continuous subarray, sub-matrix or a path in a binary tree/graph
- **Link on prefix sum** : https://leetcode.com/problems/path-sum-iii/solution/

Two pointers :

- Usually used on sorted array
- Two pointers low and high
- Move low and/or high and/to test/satisfy some criteria

Kadane's algorithm :

- To find a contigious sub-array with the largest sum
- **Algorithm template** :
- Initialize:
  - max_so_far = INT_MIN
  - max_ending_here = 0
- Loop for each element of the array
  - max_ending_here = max_ending_here + a[i]
  - if(max_so_far < max_ending_here)
    - max_so_far = max_ending_here
  - if(max_ending_here < 0)
    - max_ending_here = 0
- return max_so_far
- **Code template** :

In [ ]:
```
int maxSubArray(vector<int>& arr) {
    int n = arr.size();
    int res = arr[0], maxEle = arr[0];

    for(int i=1; i<n; i++) {
        maxEle = max(maxEle + arr[i], arr[i]);
```

```
        res = max(maxEle, res);
    }

    return res;
}
```

Most frequent element (Boyer-Moore voting algorithm) :

- Used to find the most frequent element in the array in O(n) time and O(1) space
- **Code template** :

In [ ]:

```
int majorityElement(vector<int>& nums) {
    // Boyer-Moore Voting Algorithm
    int candidate = 0;
    int count = 0;

    for (int n : nums) {
        if (count == 0) {
            candidate = n;
        }

        count += (candidate == n) ? +1 : -1;
    }

    return candidate;
}
```

Cyclic Sort :

- Useful when we have numbers in a certain **range** and asked to find **duplicates** or **missing** ones
- **Template :**

In [ ]:

```
int start = 0;

while (start < nums.size()) {
    int n = nums[start];
    if (n < nums.size() && n != start) {
        swap(nums[start], nums[n]);
    }else{
        start++;
    }
}
```

Dutch flag problem :

- Sort in O(n) time and O(1) space an array of 3 disctinct elements (could be repeated multiple times)

In [ ]:

```
// nums can be [2,0,2,1,1,0] or  [2,0,1]
// Output : [0,0,1,1,2,2] and [0,1,2] respectively
void sortColors(vector<int>& nums) {
    int s = 0;
```

```
    int e = nums.size() - 1;

    for (int i = 0; i <= e;) {
            if (nums[i] == 0)
            swap(nums[s++], nums[i++]);
        else if (nums[i] == 2)
            swap(nums[e--], nums[i]);
        else
            i++;
    }
}
```

Binary Search

- Used in a sorted array to find an element **Ordinary binary search**

In [ ]:

```
int search(vector<int>& nums, int target) {
    int pivot, left = 0, right = nums.size() - 1;
    while (left <= right) {
      pivot = left + (right - left) / 2; // avoid overflow
      if (nums[pivot] == target) return pivot;
      if (target < nums[pivot]) right = pivot - 1;
      else left = pivot + 1;
    }
    return -1;
}
```

- How to find the first element in a rotated array

In [ ]:

```
int get_first(vector<int>& nums) {
    int s = 0;
    int e = nums.size() - 1;

    while (s < e) {
        int m = s + (e-s)/2;
        if (nums[m] > nums[e]) {
            s = m+1;
        } else {
            e = m;
        }
    }

    return s;
}
```

- Binary search in a rotated array

In [ ]:

```
int search(vector<int>& nums, int target) {
    int start = 0, end = nums.length - 1;
    while (start <= end) {
      int mid = start + (end - start) / 2;
      if (nums[mid] == target)
          return mid;
      else if (nums[mid] >= nums[start]) {
        if (target >= nums[start] && target < nums[mid])
            end = mid - 1;
        else
```

```
            start = mid + 1;
        }else {
            if (target <= nums[end] && target > nums[mid])
                start = mid + 1;
            else
                end = mid - 1;
        }
    }
    return -1;
}
```

Two Intervals :

- Useful to detect overalapping intervals
- Useful to merge overlapping intervals
- Different cases with intervals

```
1) [...A...]
           [...B...]
2) [...A...]
     [...B...]
3) [...A...]
   [.B.]
4)    [...A...]
 [.B.]
5) [...B...]
   [.A.]
1) [...B...]
           [...A...]
```

- If we sort a.start <= b.start => only 1,2 and 3 cases will be possible
- To merge the overlapping intervals in scnearios 2 and 3

```
2) [...A...]
     [...B...]      ---->      C : [A.start .... B.end]
3) [...A...]
   [.B.]             ---->      C : [A.start .... A.end]
C the merged interval is going to have the following start and end :
c.start = a.start
c.end = max(a.end, b.end)
```

Example

In [ ]:

```
vector<vector<int>> merge(vector<vector<int>>& intervals) {
    vector<vector<int>> merged;

    sort(intervals.begin(), intervals.end(), [](const auto& l, const auto& r){
        return l[0] < r[0];
    });

    int start = intervals[0][0];
    int end = intervals[0][1];

    for (int i = 1; i < intervals.size(); i++) {
        auto& interval = intervals[i];

        if (interval[0] <= end) { // overlap
            end = max(end, interval[1]);
        }else{
            merged.push_back({start, end});
```

```
            start = interval[0];
            end = interval[1];
        }
    }

    merged.push_back({start, end}); // dont forget to push the last interval
    return merged;
}
```

Subsets

- Useful to get all the elements of the powerset of a given set ## Approach 1**
- Start with empty set [[]]
- For each element in the array
  - Copy the existing sets in the powerset and add the current element
  - Add the result to the power set ## Approach 2**
- Use a bitset to indicate wether an element should be in the powerset or not
- For each element with index i in the given set
  - For j = 0 to 1 << n (where n is the size of the initial set)
    - if ((i >> j) & 1)
      - Add the j'th element into the powerset[i]

Heaps

Two Heaps

- Useful to find a median in a data stream for example
- Use two heaps (min heap and max heap)
- We can store the smaller part of the list in a max_heap. We are using max_heap because we are only interested in knowing the largest number in the first half of the list.
- We can store the larger part of the list in a min_heap. We are using min_heap because we are only interested in knowing the smallest number in the second half of the list.
- Inserting a number in a heap will take O(log N) (better than the brute force approach)
- The median of the current list of numbers can be calculated from the top element of the two heaps.

- An alternative approach chould be using C++ multiset (Which is red-black tree)

- Keep track of the middle element
- Update it upon insertion/deletion
- **Example** :

In [ ]:

```
vector<double> medianSlidingWindow(vector<int>& nums, int k) {
    vector<double> medians;
    multiset<int> window(nums.begin(), nums.begin() + k);
    auto mid = next(window.begin(), k/2);
    for (int i =k;;i++) {
        // Push the current median
        medians.push_back(((double)(*mid) + *next(mid, k%2-1)) * 0.5);
        // If all done, break
```

```
        if (i == nums.size())
            break;
        // Insert incoming element
        window.insert(nums[i]);
        if (nums[i] < *mid)
            mid--;
        // Remove outgoing element
        if (nums[i-k] <= *mid)
            mid++;
        window.erase(window.lower_bound(nums[i-k]));
    }
    return medians;
}
```

K-way merge

- Useful when we want to merge K sorted linked-lists/arrays
- We can push the smallest (first) element of each sorted array in a Min Heap to get the overall minimum.
- After this, we can take out the smallest (top) element from the heap and add it to the merged list.
- After removing the smallest element from the heap, we can insert the next element of the same list into the heap.
- We can repeat steps 2 and 3 to populate the merged list in sorted order. ## Top K Numbers
- Find the K'th largest element in an unsorted array
- For each element in the array
  - If element > min_heap.top()
    - min_heap.pop() // Eject smallest element
    - min_heap.push(element) // Push the element bigger than the previous smallest element

Linked Lists :

Fast and Slow Pointer :

To find a cycle :

- Slow pointer moves one step at a time
- Fast pointer moves twice the speed of slow
  - While (fast and slow can advance)
  - Advance fast 2 times
  - Advance slow once
  - If (slow == fast)
    - Cycle exists!
- If we reach here then there is no cycle

To find the node where the cycle occurred :

- Slow pointer moves one step at a time

- Fast pointer moves twice the speed of slow
  - While (fast and slow can advance)
  - Advance fast 2 times
  - Advance slow once
  - If (slow == fast)
    - Cycle exists!
- If fast reached the end ⇒ No cycle found
- Otherwise
  - Reset slow to the beginning
  - While (fast != slow)
    - Advance slow 1 step
    - Advance fast 1 step // Reducing hence the speed of fast
  - Return Slow

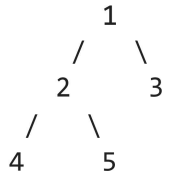
Explanation : Once the cycle is detected if we reset slow to the beginning and reduce the speed of slow both pointers will eventually meet at the start of the cycle (At the cycle node)

Find middle of a linked list :

- Set slow pointer at the beginning
- Set fast pointer two steps after
- While (fast && slow && can advance)
  - Advance slow 1 step
  - Advance fast 2 steps
- When fast reaches the end, Slow will be already in the middle ### In place reversal of linked list : **Template (Recursive) :**

In [ ]:
```
ListNode* reverseList(ListNode* head) {
    if (head == NULL || head->next == NULL)
                return head;
    ListNode* tempNode = reverseList(head->next);
    head->next->next = head;
    head->next = NULL;
    return tempNode;
}
```

Trees :



- **Inorder** (Left, Root, Right) : 4 2 5 1 3
- **Preorder** (Root, Left, Right) : 1 2 4 5 3
- **Postorder** (Left, Right, Root) : 4 5 2 3 1
- **BFS** Traversal : 1 2 3 4 5

- **Uses of inorder in a BST**: gives nodes in non-decreasing order
- **Uses of preorder** : create a copy of the tree or create prefix expression
- **Uses of postorder** : Postorder traversal is used to delete the tree or postfix expression

Graphs

DFS Iterative

- **When to use** : Shortest path
- **Algorithm :**

In [ ]:
```
unordered_set<Node*> visited = {};
stack<Node*> toVisit = { s };

while (toVisist.size()) {
    current = toVisit.top();
    visited.insert(current);

    for (Node* neighbour : current.getNeighbours()) {
        if (visited.count(neighbour ) == 0)
            toVisist.push(neighbour);
    }
}
```

BFS Iterative :

- **When to use** : Shortest path
- **Algorithm :**

In [ ]:
```
unordered_set<Node*> visited = {};
queue<Node*> toVisit = { s };

while (toVisit.size()) {
    int size = toVisit.size();

    for (int i = 0; i < size; i++) {
        auto current = toVisit.front();
        toVisit.pop();
        visited.insert(current);

        for (Node* neighbour : current.getNeighbours()) {
            if (visited.count(neighbour) == 0)
                toVisit.push(neighbour);
        }
    }

    // This happens each layer
}
```

Topological sort :

- **When to use** : Dependencies between nodes
- **Algorithm (using BFS)** :

In [ ]:

```
vector<Node*> sorted;
queue<Node*> toVisit;
// Push all nodes with incoming degree of 0 to toVisit

while (toVisit()) {
    Node* curr = toVisit.front();
    toVisit.pop();
    sorted.emplace_back(curr);

    for (Node* next : curr.getNeighbors()) {
        if (--inDegrees[next] == 0) {
            toVisit.push(next);
        }
    }
}

// If sorted.size() ==  to the number of the nodes in the graph ⇒ then there is no
```

- **Algorithm (using DFS)** :
  - Apply normal DFS
  - When popping from the dfs stack, add these elements to a list
  - Reverse the list
  - List should contain the nodes in their topological order