1mill
Binary search for Greedy problem

```
int fn( vector<int> &arr){
   int left= MINIMUM_POSSIBLE_ANSWER;
   int right =  MAXIMUM_POSSIBLE_ANSWER;

   while( left <= right) {
      int mid = left + (right - left)/2;
      if( check(mid)) {
          right = mid-1;
      }
      else{
          left = mid+1;
      }
   }

   return left;

}
```

This is for minimum

many times this logic is very handy

```
bool check( int x) {
   //this function is implemented
   //depending on the problem
   return BOOLEAN;
}
```

Similarly, Greedy maximum using Binary search

```
int fn(vector<int> &arr){
   int left = MINIMUM_POSSIBLE_ANSWER;
   int right =  MAXIMUM_POSSIBLE_ANSWER;

   while( left <= right) {
     int mid = left + (right - left)/2;
     if(check(mid)) left = mid+1;
     else right = mid-1;
   }

   return right;
}
```

Core logic method

```
bool check ( int x) {
   // this logic implemented base on
   // Problem statement
   return   bool
}
```

Utility method

Code Template for Backtracking

```
int backtrack(STATE curr, OTHER_ARGUMENSTS...){
    if(BASE_CASE){
        //modify the answer
        return 0;
    }

    int ans = 0;
    for(ITERATE_OVER_INPUT){
        //modify the current state
        ans += backtrack(curr, OTHER_ARGUMENTS...)
        //undo the modification in current state
    }

    return ans;
}
```

BASE Condition/ recursion termination

forward step

Backtrack step

Binary search – duplicate elements, left most insertion point

```
int binarySearch(vector<int> &arr, int target){
    int left = 0;
    int right = arr.

    while(left < right){
        int mid = left + (right - left)/2;
        if(arr[mid] >= target) right = mid;
        else left = mid+1;
    }

    return left;
}
```

@Note: right is here last index+1

As soon as left = right this iteration ends.

arr : {3, 7, 9, 9, 9, 9, 11, 15}

target : 9

left, m, m, right, m, right

Binary search: duplicate elements, right most insertion point

Example

arr: { 1, 5, 7, 7, 7, 7, 9, 10, 11 }

  0 1 2 3 4 5 6 7 8

$\boxed{Ans = 6}$

So your template code here is

```
int binarysearch(vector<int>&arr, int target){
    int left = 0;
    int right = arr.size();
    while (left < right){
        int mid = left + (right - left)/2;
        if(arr[mid] > target) right = mid;
        else left = mid+1;
    }
    return left;
}
```

l   r   m

~~0~~ ~~5~~, 6   ~~9~~ ~~7~~ ~~7~~ ~~6~~ 5 6

✓ l<r

m = 4

5

$\boxed{l = 6, \ r = 6}$

Comeout from while loop

$\boxed{Ans = 6}$

and your Answer becomes 6

Code template for Binary Search

```
int bs(vector<int> &arr , int target){
    int left = 0;
    int right = arr.size() -1 ; // right index
    while( left <= right) {
        int mid = left+ (right - left)/2 ;
        if(arr[mid] == target){
            //do something
            return mid ;
        }

        if (arr[mid] > target) right = mid -1;
        else left = mid+1 ;

    }

    return left ;

}
```

Dry Run your code on example

arr : {1, 5, 7, 11, 12, 13, 15, 25} _

target : ~~13~~ 27

l	r	m	while true	if	else if		
				equal	greater	less	
0	7	3	true	f	greater f	t	
4	7	5	true	f	f	t	
6	7	6	true	f	f	t	
7	7	7	true	f	f	t	
			false				
8	7						

return 8

l	r	m	Condition
~~0~~ 4	7	~~3~~ 5	true

(11, 13)

~~equal~~ ~~greater~~ Small ✓

(13, 13)

Equal greater less

return 5

find top K elements from heap

```
vector<int> fn(vector<int> & arr, int K) {
→ Priority_queue < int, CRITERIA > heap;
    for (int num : arr) {
        heap.push(num);
        if(heap.size() > K) heap.pop();
    }

    vector<int> ans;

    while(heap.size() > 0) {
        ans.push_back(heap.top());
        heap.pop()
    }

    return ans;
}
```

you are just maintaining K element in heap

you are populating your ans vector to be returned.

Graph: DFS (Recursive)

Few things you have to consider here before moving ahead.

⇨ Assume that you have graph in terms nodes & edges and you also have adjacency list prepared.

Let's assume that you as an example of this template, accomplishing to sum the all node's value in this graph using DFS (Recursion)

Code Template below

```
unordered_set<int> visited;

int fn( vector<vector<int>> &graph){
    visited.insert(START_NODE);   } startnode marking visited true.
    return dfs(START_NODE, graph);
}


int dfs(int node, vector<vector<int>> &graph){
    int ans =0;
    // do some logic
    for( int neighbour : graph[node]){
        if(visited.find(neighbour) == visited.end()) {
            visited.insert(neighbour);
            ans += dfs(neighbour, graph);
        }
    }
    return ans;
}
```

you are iterating to see all the child one by one which is not yet visited.

DFS (iterative) || you will compare your iterative approach
|| with recursive approach.

```
/// graph is representing as an adjacency list.
int fn( vector <vector<int> > &graph) {
  stack <int> s;
  unordered_set<int> visited;

  s.push(START_NODE);
  visited.insert (START_NODE);
```

for starting your code

```
  int ans=0;

  while (s.empty() ==false) {
    int node = s.top(); s.pop();
    ans += node;
```

you are accumulating the value here

```
    for (int neighbor : graph[node]) {
      if(visited.find(neighbor) == visited.end()) {
        visited.insert(neighbor);
        s.push(neighbor);
      }
    }
  }
  return ans;
```

returning the output

```
}
```

If you compare the recursive and iterative version only difference is System stack Vs own stack

This could be sometime very handy because System stack has 1MB memory limitation.