

mini 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 logic  
    // Problem statement  
    return bool  
}
```

Utility
method

Code Template for Backtracking

```
int backtrack (STATE curr, OTHER_ARGUMENTS...) {  
    if (BASE_CASE) {  
        // modify the answer  
        return 0;  
    }  
    int ans = 0;  
    for (ITERATE_OVER_INPUT) {  
        // modify the current state  
        ans += backtrack (curr, OTHER_ARGUMENTS...) } forward step  
        // undo the modification in current state } Backtrack step  
    }  
    return ans;  
}
```

Binary search — duplicate elements, left most insertion point

```
int binarySearch (vector<int> &arr, int target) {  
    int left = 0;  
    int right = arr.size() - 1;  
    @Note: right is here  
    // last index + 1
```

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

```
    return left;  
}
```

As soon as $left = right$ this iteration ends.

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

target: 9
↑ left m

↑ m m 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

So your template code here is

Ans = 6

```
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 1 2 3 4 5 6 7 8
~~0 1 2 3 4 5~~
6, 6, 6, 6, 6, 6, 9, 10, 11
✓ l < r
m = 4
l = 6, r = 6
Come out from while loop
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

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

l	r	m	Condition
0	7	3	equal
4	7	5	greater
6	7	6	greater
7	7	7	equal
8	7		return 8

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();  
    }
```

you are just maintaining K element in heap

```
    vector<int> ans;
```

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

you are populating your ans vector to be returned.

```
    return ans;
```

```
}
```

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);  
    return dfs(START_NODE, graph);  
}
```

} start node marking visited true.

```
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.