## Contents

## Arrays

### Binary Search

#### Introduction

- Refer Binary Search Notes
- Linear search worst case: n comparisons, that is O(n) complexity. And the worst case is when the element is not present in the array.
- **Binary search only applies in monotonic functions.**
- Monotonic functions are those functions which are either in increasing order or either in decreasing order.
- If start = 2^31-1 and end = 2^31-1 which is the highest limit of the integer, then in line no.8 when we do $(start + end)/2$ , in the operation of $(start + end)$ , our integer can go out of bound and may give an error.
- So we can modify this formula as
$$s + \frac{(e - s)}{2}$$
which will result in $(2s + e - s)/2$ and in return it gives $(s + e)/2$ which is the correct original formula.
- Advantage of Binary Search
  1. Suppose if there's an array of size 1000.
  2. In the worst case, in linear case there will be 1000 comparisons.
  3. In binary search, first the array is of size 1000, the, 500–>250->125->62->31->15->7->3->1->0, that is just 10 comparisons.
  4. In linear search, the complexity is O(n) whereas in binary search, the complexity is O(logn).
  5. Binary Search Complexity Analysis

#### Problems

1. Peak Index in a Mountain Array.
   - There are three parts to this. This sum is based on the concept of binary search.
   - One being that when the array is increasing, the element will be greater than the element which is behind it. In this case, we still haven't reached the peak element, so we need to take the next half of the array.
   - When the array is decreasing, the element will be lesser than the one which is behind it.
   - The peak element will be the element which is greater than the previous element as well as greater than the next element as well.

2. Find Pivot element in an array
    - Pivot element is an element where sum of all the numbers strictly to the left side of the array is equal to the sum of all the numbers strictly to the index's right.
    - Here, we can apply a logic of binary search where, we can take a mid element and check if the sum of left and right are equal.
    - If they are equal, we will return the index.
    - If the left side has a greater sum, we will move the index to the left side.
    - If the right side has a greater sum, then we will move the index to the right side.
3. Find Pivot element in an array (Love Babbar)
    - This is a problem of a rotated array.
    - Pivot element is an element through which rotation of an array has occurred.
    - When plotting on a graph, we will be having two lines one and two and the element we want to find will be the beginning of the second line with the second line being the line where small numbers are present.
    - We will check if the mid element belongs to the first line or not by checking if it is greater than or equal to the first element of the array and if it is, we will update the value of start to mid+1 as it lies on the first line.
    - In case it is not greater than the first element, then we will update the value of end to mid.
    - The reason we are updating the end to mid and not mid-1 is because, if our mid element turns out to be the pivot element then we won't be doing mid-1 and thus limiting it to first line thereby writing wrong logic. We will be doing this until start=end in which case loop won't be executed and then we will return start.
    - Also we are doing start '<' end and not '<=' because the loop will then be infinite as start will be equal to end and the element will continue to be less than first element and we will keep on updating end to mid thereby selecting same element as start and end infinitely and comparing it to the first element.
    - We can do start<=end in case we have additional if where we specify if start==end, return start;
4. Search in Rotated Sorted array
    - This problem is similar to the 'Find Pivot Element' one, except we will be given an element to find and we have to find if the element exists in the array or not and return the index.
    - Here, first we will find the pivot element.
    - We will then find if the element belongs to line 1 or line 2, if it belongs to line 1, then we will apply binary search on line 1, that is from arr[0] to arr[pivot-1].
    - If the element belongs to line 2, then we will apply binary search on line 2, that is from arr[pivot] to arr[n-1].

- The reason we are not doing if(k>=arr[0]) is because in case of a single element we will be passing the start and end as 0 and pivot-1 = 0-1 = -1 in the binary search function and we won't get the correct answer.
- Therefore, we are doing if(k>=arr[pivot] && k<=arr[n-1]) so that we will be sending start and end as pivot and n-1 = 1-1 = 0, due to which the binary search function will execute once where the start=end, and we will get the index of the single element.

5. Square Root of a Number using Binary Search

    - The range in which the answer may lie is known as search space.

## Sorting

### References

- Stability in Sorting Algorithms.

### Selection Sort

- Selection Sort Coding Ninjas Problem
- It is unstable.
- Flowchart

### Bubble Sort

- Bubble Sort Coding Ninjas Problem
- It is stable.
- We can further optimize the bubble sort by checking if we did any swaps in particular iteration or not.
- That is suppose at any iteration, the number of swaps done is 0, then that means that the array is already sorted.
- This is because, suppose if there are 5 elements a,b,c,d,e and if no swaps are done, then that means a less than b, b less than c, c less than d, d less than e, then in such case, after getting out of the inner loop, we can directly break out of the outer loop.
- After optimizing, if the array is already sorted, then the time complexity for bubble sort is O(n) and the worst case is O(n^2).

### Insertion Sort.

- Insertion Sort Coding Ninjas Problem.
- Let us consider the example of sorting cards.
- Suppose if we are given some number of cards one by one and after getting each card, we need to place them in increasing order.
- So after getting each card, we will compare it with the cards we have sorted previously and then place that card accordingly.
- For example, let us consider the cards with numbers 8, 9, 4, 6, 2 and we are getting each of them one by one. Below are each of the iterations.
- In each iteration, for each element we will decide if the element should go to the left of the element or not, if it does go to the left, we will first right shift all the elements ahead of it, which will create space for that element, and we will then assign that index to the element.

- 8
- 8, 9.
- 4, 8, 9.
- 4, 6, 8, 9.
- 2, 4, 6, 8, 9.

## Strings

### Char array

- A char datatype can only store single character.
- In order to store multiple characters, we need to make use of character arrays or strings.
- Strings are 1-dimensional char arrays.
- In character array, at the end of the array, an extra character is added by itself which is known as the null character (‘\0’).
- This null character is used as a terminator.
- This helps to know where the string is ending. It will print the characters before the null character.
- When having a space or tab in input, the cin will stop the input when we have space.
- The character array does allow having space in it, but cin limits that.
- The character array however, stops the execution at a null character which can be understood by the example given in the code.

  ```
  char namewithnull[20];
  cout << "Enter your name" << endl;
  cin >> namewithnull; //Yash
  namewithnull[2] = '\0'; // Null character
  cout << namewithnull; //Ya
  ```

- When you initialize a char array with the length longer than that of required, the indexes which aren’t used are filled with garbage values.
- When you initialize a char array with the length less than that of required, the inputted string which is longer than the length will be printed correctly, however it will be with a segmentation fault error which signifies that we are accessing restricted area of memory.
- Problems

  1. Reverse String Leetcode Problem
  2. Palindrome String

  - Palindrome string is a string which when reversed gives the same output.
  - In order to check whether a string is palindrome or not, we can create a new string which is reverse of the original string. Then with the help of a for loop, we can compare the indexes of both the strings, and check if they are equal or not.

- However, this results in extra space as we are having an extra string which is reverse of the original string.
- The second approach is that, we need to compare the first letter with that of the last letter, then second letter with that of the second last letter and so on.

### Strings

- The storage of strings is done in the same way as that of the character arrays, which is null terminated.
- However, the difference is that in normal usage, we cannot access the null terminator.
- TODO: Key differences between character arrays and strings.(from 47:00 in lecture 22 video)

## CPP-STL

### References

1. Love Babbar Whimsical
2. Backup Whimsical
3. STL Data Structures Iterators

### Array

- The implementation of the STL array is same as that of the basic array.
- That is, the STL array is also static which is the reason that it is not mostly used.
- Refer Array Notes
- Array Operations:
    1. `arr.size()`- Helps to find the size of the array. (Complexity : O(1))
    2. `arr.at(index)` - Helps to find the value present in the array at the given index. (Complexity : O(1))
    3. `arr.empty()` - Helps to find if the array is empty or not. (Complexity : O(1))
    4. `arr.front()` - Returns the first element of the STL array. (Complexity : O(1))
    5. `arr.back()` - Returns the last element of the STL array. (Complexity : O(1))

### Vector

- Refer Vector Notes
- Vector is simply a dynamic array.
- Just like array, elements in vector are also stored in contiguous memory locations.

- The difference is that when you add an element in a vector which is full, the vector will double its size.
- The way it does this is that, it will create another vector which is double the size of the current vector and then it will copy all the elements of the current vector to the another vector and will dump the current vector. Can also decrease the size using to fit.
- Initialising a vector.
  - We can create a vector without specifying the size.
  - The size of the vector during its creation in such case is 0.
  - If we know the size of the vector, we can specify the size of the array as shown in the code and can initialise all the elements of the vector with a given value.
  - The size of the vector denotes how many elements are present in the vector, and the capacity denotes for how many elements, the vector contains space.
  - In order to create a vector with the values of another vector, we can just specify the vector to be copied in brackets when initialising new array. For example `vector<int> b(a)`
  - Here, the elements of the vector a will be copied into b.
  - We can also initialise it as `vector<int> a(5,1)`, here the size of the vector a is 5, and all the elements are initialised with 1.
- Vector Operations:
  1. `vector.capacity()` - Indicates the vector capacity, that is, for how many elements the vector contains space.
  2. `vector.size()` - Indicates the vector size, that is, how many elements are actually present in the vector.
  3. `vector.push_back(ele)` - Is used to push the element(ele) into the vector.
  4. `vector.at(index)` - Helps to find the value present in the vector at the given index.
  5. `vector.front()` - Returns the first element of the STL vector.
  6. `vector.back()` - Returns the last element of the STL vector.
  7. `vector.push_back(ele)` enters the element to the last index in the vector
  8. `vector.pop_back()` - removes the element present in the last index of the vector.
  9. `vector.clear()` - It is used to clear the vector, that is, remove all the elements present in the vector. One thing to note here is that when the vector is cleared, only the size becomes 0, but the capacity remains the same as that of before clearing.
  10. `vector.begin()`- If starting iterator of the vector is required, we can use the begin.
  11. `vector.end()`- If ending iterator of the vector is required, we can use the end.
- The iterator methods are used to point to the first and the last element.

**Deque**

- Also known as Doubly ended queue.
- Can perform push-pop, that is insertion-deletion at both the ends, that is at starting as well ending of the deque.
- Unlike array and vector, values are not stored in contagious memory locations in deque.
- Here, there are multiple fixed static arrays and tracking is done that data is present in which array.
- Deque is also dynamic, random access is also possible (using at).
- Deque operations:

  1. `deque.begin()` - If starting iterator of the deque is required, we can use the begin.
  2. `deque.end()`- If ending iterator of the deque is required, we can use the end.
  3. `deque.push_back()` - Is used to push the element at the back of the deque.
  4. `deque.push_front()` - Is used to push the element at the front of the deque.
  5. `deque.pop_back()` - Is used to pop the element at the back of the deque.
  6. `deque.pop_front()` - Is used to pop the element at the front of the deque.
  7. `deque.at(index)`
  8. `deque.front()`
  9. `deque.back()`
  10. `deque.empty()`
  11. `deque.size()`
  12. `deque.erase(iterator1, iterator2)` - Is used to erase the specific elements from the deque. In case of just specifying one iterator, it will delete one element, and in case of providing two iterators, it will delete range of elements where the element at the first iterator is included and the the element at (iterator2-1)th index is included. Refer Deque Clear and erase

7. After emptying, in deque as well, the size will become 0, however, the capacity will remain the same.

**List**

- The list STL is created with the help of doubly linked list.
- In doubly linked list, there are two pointers, one for the front and other for the back.
- Direct access of the elements with the help of square brackets or with the help of at function as in arrays, vectors, deque is not possible in list.
- In list case, we need to travel to the fourth element, in case we want the fourth element.
- List Operations

  1. `list.begin().`- O(1)
  2. `list.end().`- O(1)

3. `list.empty().` - O(1)
4. `list.front().` - O(1)
5. `list.back().` - O(1)
6. `list.push_back().` - O(1)
7. `list.pop_back().` - O(1)
8. `list.push_front()` - O(1)
9. `list.pop_front()` - O(1)
10. `list.erase(l.begin() or l.end()).` - O(n)
11. `list.size()` - O(1)

**Stack**

- Last In, First Out.
- In stack, there are two main operations, one being push and the other being pop.
- The element which is entered at the last is the element which first comes out when the pop operation is applied on the stack.
- Stack Operations.

  1. `stack.push(element).`
  2. `stack.pop().`
  3. `stack.size().`
  4. `stack.empty().`
  5. `stack.top()` - Returns the top element that is the element which is entered latest into the stack.

**Queue**

- First in, First out.
- It can just be considered a normal queue as in case of theatre ticket queue, bouquet queue or just any other queue.
- The element which is entered at the first is the element which first comes out when the pop operation is applied on the queue.
- Queue operations.

  1. `queue.push(element).`
  2. `queue.pop()`
  3. `queue.size()`
  4. `queue.empty()`
  5. `queue.front()` - Returns the first element of the queue.

6. `queue.back()` - Returns the last element of the queue.

- All the operations have a complexity of O(1).

**Priority Queue**

- In priority queue, the first element is always the greatest just as in case of max heap.
- The default priority queue is max heap.
- In case of max-heap and priority queue, whenever you fetch the element, the element will be of largest value.
- In case of min-heap, whenever you fetch the element, the element will be of minimum value.
- When you initialise priority queue as `priority_queue<int> maxi`, it is based on max heap, which means when we fetch the element, the element will be greatest from the elements stored in the priority_queue.
- In order to create min heap, we can declare priority queue as `priority_queue<int, vector<int>, greater<int>> mini`.
- Refer the code, when printing the top element of the priority queue and popping the element of the priority queue, we first store the size of the priority queue in a variable rather than doing it in the for loop as `i<maxi.size()`, this is because at each iteration the size of the 'maxi' will keep on changing as in each iteration, we are popping the element out of the queue.
- Priority Queue operations

  1. `pq.push(ele)`
  2. `pq.pop()`
  3. `pq.top()`
  4. `pq.size()`
  5. `pq.empty()`

**Set**

- Refer Set Notes
- Refer Set Iterators
- The important property of a set is that it just has all the unique elements stored inside it.
- For example, if you store a single element, let's say 5, for 5 times, then too 5 will be stored just for a single time.
- It stores all the elements just for a single time and its implementation in the backend is done using BST.
- Once the element is inserted, the element cannot be modified, either enter the element or delete the element, but you cannot modify the element.

- Elements are returned in sorted order.
- The difference between unordered set and set is that set is little slow than unordered set and in unordered set elements are returned in unsorted order.
- Set operations
    1. `set.insert(ele)` - Is used to insert an element into a set. Complexity - (Ologn)
    2. `set.erase(iterator)` - It is used to erase or delete a particular element in the set with the help of an iterator.
    3. `set.begin()` - Iterator pointing to 0.
    4. `set.count(ele)` - Lets us know whether the element is present in the set or not. The output may be 0 or 1.
    5. `set.find(ele)` - After finding the element, it returns us the iterator/index of the element.

**Map**

- Refer Map Notes
- Map is a data structure in which the data is stored in the form of key-value pair.
- Each and every key is unique and a single key points to only one value. A key points to just a single value and it then cannot point to any other value.
- There can be 2 keys pointing to same value, however there cannot be 2 values pointing to one key.
- The output of the map is in sorted order and in case of unordered map, it is not in sorted order.
- The complexity for insert, erase, find, count, etc. in case of map is O(logn).
- The ordered map is implemented with the help of red-black tree or balanced tree and the complexity for search here is O(logn).
- In case of unordered map, the implementation is done using hash table and the complexity for search in that case is O(1).
- Map Operations
    1. `map.insert({key,value})` - It is used to insert a key-value pair in a map.
    2. `map.erase(key)` - In the case of map, we need to specify the key to erase the key-value pair instead of the iterator as seen in the other data structures.
    3. `map.begin()` - Already known.
    4. `map.end()` - In both, the begin and end, we need to specify the first or second in order to print the key or value respectively.
    5. `map.find(key)` - Returns the iterator of the key.

**Algorithms**

- We can find a particular element with the help of **binary search**, that is O(logn) complexity with the help of STL binary search.
- The syntax for the same is:

  ```
  binarySearch(vector.begin(),vector.end(), ele);
  ```

- Here, vector.begin() and vector.end() are the starting and ending iterators respectively and ele is the element which is to be found in the vector.
- Finding upper bound and lower bound, refer code and GeeksforGeeks
- Max and min of two elements.
- Swapping of two elements.
- Rotation of a vector (We need to specify the start iterator, how many elements we want to rotate and the end iterator).
- Sorting of a vector. It is based on intro sort which is the combination of three algorithms which are quick sort, heap sort and insertion sort. All these three algorithms together make up to intro sort which is behind the scenes of the sorting.

**Array Questions**

1. Reverse the Array
   - There are two approaches to this as seen in the code, the **first approach** is to go from `i=0` to `i<n/2`. This is because, in case of 5 elements, we won't be reversing the 3rd element and leave it as it is as it will stay in the middle itself after reversing as well.
   - Also in case of n=4 as well, we will need to do the reverse operation 2 times for 4 elements. i=0 to 4/2=2, i will run for 0 and 1.
   - And then we will reverse the ith element with respect to (n-i-1) for the last and second last elements and so on.
   - The **second approach** is to have start and end, and go on till start<end, and swap these elements. We don't consider start==end as it will be the same element and we don't need to swap the same element with itself.
   - At each iteration in the second approach, we will increment the start and decrement the end after swapping.
   - The start and the end approach is clean and better approach.
2. Reverse the Array after a particular key.
   - In this case, a key m is given after which we need to reverse the rest of the elements in the array.

- Here, we can simply follow the second approach of reverse the array where we can have start as `(key+1)th` element and end as `(n−1)th` and then follow the same steps of reversing the array for the subarray in this scenario.

3. Merge Sorted Array - Love Babbar

   - The input given to us will be two sorted arrays and their sizes and we have to create a third array which is the combination of both of these arrays.
   - Here, first we will be taking inputs for the size and the elements of both the arrays.
   - We will then add the size of both the elements which will then become the third array and initialize all the elements of the third array with `0`.
   - Now, as these elements are sorted, we will start the iterators of both the arrays by `0`, and then compare the elements of both these arrays.
   - If the element of the first array is greater than that of the second array or vice-versa, then we will insert that element into the third array and increase the iterator of the first array(or second array in case of vice versa) as well as the third array in which we are inserting the elements.
   - Then we will be comparing the second element of the first array to that of the first element of the second array and so on we will continue the above process.
   - We can do this until both the arrays are in bounds, if the iterator of any array goes beyond that of the array, then we will break out of the loop and copy the rest of the elements of the other array to the third array.

4. Merge Sorted Array - Leetcode

   1. Self Approach

      - Refer Dry Run in Notebook.
      - We will be having two iterators in this case which will be i and j both initialized with `0`.
      - We will run the loop till `i<m && j<n`.
      - If there are no elements in the second array that is n=0, then we will leave the array as it is and thus break out of the loop. (Outside this loop, we are having additional condition where `j<n` will be followed to replace all the `0` elements of nums1 with the remaining elements of nums2, however this will not run as the condition will become `0<0` which is false.)
      - Else if the nums1 element is less than that of the nums2 element, then we will increment the nums1 iterator, as the nums1 element is at the correct position in itself and so we need to compare the next nums1 element to that of the nums2.
      - If nums2 element is less than that of the nums1 element, then we will right shift all of the initial non-zero elements and assign the ith position of nums1 to nums2 element.
      - The way we are right shifting is we are taking the mth index which is the first element

after the series of non-zero elements and we are assigning it with the previous value and we are doing it till k>i as the ith+1 index will have the ith value. We will then increment the m as we now need to do additional comparisons as the nums1 elements have been right shifted.

- Then at the ith index, we will be assigning nums2 element.
- Then we will increase the ith iterator and jth iterator for further comparisons.
- In the end, if there are remaining j elements, then we will copy those elements and replace the nums1 0's with those elements as 0's will be the only one's remaining.

5. Move Zeroes - Leetcode

    1. Self Approach.

        - We will be having two pointers in this case, i and j both initialized with 0.
        - We will be first traversing through the array using i pointer.
        - If the value of the element in the array is non-zero, then we will assign the jth position of the array and increment the jth iterator.
        - We will repeat it till the end of the array.
        - Then for the further j elements till the end of the array, we will substitute 0 for the further elements as they need to be at the end of the array.

    2. Love Babbar Approach.

        1. Here, as well, we will be having two pointers, both initialized with 0.
        2. We will be first traversing through the array using ith pointer.
        3. If the value of the element in the array is non-zero, then we will swap the jth element and the ith element which is non zero and then increment the jth iterator.
        4. We will repeat it till the end of the array.

6. Rotate Array

    1. Self Approach

        - So the first step is to first store the last element into the temp variable.
        - Then starting from the last index as j, we will be storing (j-1)th index till 1st index which will store the value of nums[0].
        - And then we will be allocating the value of temp in nums[0].
        - Repeat these steps for k number of times which is the number of rotations.

    2. Love Babbar Approach.

        - First of all, we will be creating a temp array where we can store variables after rotation from the nums array.
        - Now any number modded with n gives the output from between 0 to (n-1).
        - Example `0%3=0, 1%3=1, 2%3=2, 3%3=0, 4%3=1, ......`

- Now in the problem, we need to shift the elements in the array by k.
- So we will add k to the ith element which we want to shift to the right. In case of the last element, that is (n-1)th element, we need to rotate it in cyclic manner that is from the start of the array.
- So in case of k=2, the last element would become

$$(n-1+2)\%n = (n+1)\%n = 1$$

- Therefore, it would be shifted at 1st index which is the correct answer as we are rotating it by 2 times.