```
                printf("Duplicates exist:%d", A[i]);
                return;
            }
            else    A[A[i]] = - A[A[i]];
        }
        printf("No duplicates in given array.");
}
```

Time Complexity: $O(n)$. Since, only one scan is required. Space Complexity: $O(1)$.

**Notes:**

- This solution does not work if the given array is read only.
- This solution will work only if all the array elements are positive.
- If the elements range is not in 0 to $n-1$ then it may give exceptions.

**Problem-5** Given an array of $n$ numbers. Give an algorithm for finding the element which appears maximum number of times in the array?

**Brute Force Solution:** One simple solution to this is, for each input element check whether there is any element with same value and for each such occurrence, increment the counter. Each time, check the current counter with the max counter and update it if this value is greater than max counter. This we can solve just by using two simple *for* loops.

```
int CheckDuplicatesBruteForce(int A[], int n) {
    int counter =0, max=0;
    for(int i = 0; i < n; i++) {
        counter=0;
        for(int j = 0; j < n; j++) {
            if(A[i] == A[j])
                counter++;
        }
        if(counter > max) max = counter;
    }
    return max;
}
```

Time Complexity: $O(n^2)$, for two nested *for* loops. Space Complexity: $O(1)$.

**Problem-6** Can we improve the complexity of Problem-5 solution?

**Solution: Yes.** Sort the given array. After sorting all the elements with equal values come adjacent. Now, just do another scan on this sorted array and see which element is appearing maximum number of times.

Time Complexity: $O(nlogn)$. (for sorting). Space Complexity: $O(1)$.

**Problem-7** Is there any other way of solving Problem-5?

**Solution: Yes,** using hash table. For each element of the input keep track of how many times that element appeared in the input. That means the counter value represents the number of occurrences for that element.

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

**Problem-8** For Problem-5, can we improve the time complexity? Assume that the elements range is 0 to $n-1$. That means all the elements are within this range only.

**Solution: Yes.** We can solve this problem in two scans. We *cannot* use the negation technique of Problem-3 for this problem because of number of repetitions. In the first scan, instead of negating add the value $n$. That means for each of occurrence of an element add the array size to that element. In the second scan, check the element value by dividing it with $n$ and return the element whichever gives the maximum value. The code based on this method is given below.





```
void MaxRepititions(int A[], int n) {
    int i = 0, max = 0, maxIndex;
    for(i = 0; i < n; i++)
        A[A[i]%n] +=n;
    for(i = 0; i < n; i++)
        if(A[i]/n > max) {  max = A[i]/n;
                            maxIndex =i;
        }
    return maxIndex;
}
```

**Notes:**

- This solution does not work if the given array is read only.
- This solution will work only if the array elements are positive.
- If the elements range is not in 0 to $n-1$ then it may give exceptions.

Time Complexity: $O(n)$. Since no nested *for* loops are required. Space Complexity: $O(1)$.

**Problem-9** Given an array of $n$ numbers, give an algorithm for finding the first element in the array which is repeated. For example, in the array, $A = \{3, 2, 1, 2, 2, 3\}$ the first repeated number is 3 (not 2). That means, we need to return the first element among the repeated elements.

**Solution:** We can use the brute force solution of Problem-1. For each element since it checks whether there is a duplicate for that element or not, whichever element duplicates first will be returned.

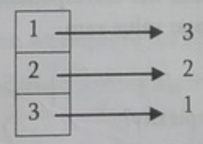
**Problem-10** For Problem-9, can we use sorting technique?

**Solution: No.** For proving the failed case, let us consider the following array. For example, $A = \{3, 2, 1, 2, 2, 3\}$. After sorting we get $A = \{1, 2, 2, 2, 3, 3\}$. In this sorted array the first repeated element is 2 but the actual answer is 3.

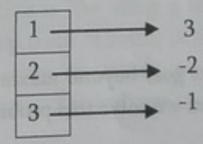
**Problem-11** For Problem-9, can we use hashing technique?

**Solution: Yes.** But the simple hashing technique which we used for Problem-3 will not work. For example, if we consider the input array as $A = \{3, 2, 1, 2, 3\}$, in this case the first repeated element is 3 but using our simple hashing technique we get the answer as 2. This is because of the fact that 2 is coming twice before 3. Now let us change the hashing table behavior so that we get the first repeated element.

Let us say, instead of storing 1 value, initially we store the position of the element in the array. As a result the hash table will look like (after inserting 3, 2 and 1):




Now, if we see 2 again, we just negate the current value of 2 in the hash table. That means, we make its counter value as $-2$. The negative value in the hash table indicates that we have seen the same element two times. Similarly, for 3 (next element in input) also, we negate the current value of hash table and finally the hash table will look like:




After processing the complete input array, scan the hash table and return the highest negative indexed value from it (i.e., $-1$ in our case). The highest negative value indicates that we have seen that element first (among repeated elements) and also repeating.

**What if the element is repeated more than two times?** In this case, just skip the element if the corresponding value $i$ already negative.

**Problem-12** For Problem-9, can we use Problem-3 technique (negation technique)?

**Solution: No.** As a contradiction example, for the array $A = \{3, 2, 1, 2, 2, 3\}$ the first repeated element is 3. But with negation technique the result is 2.

**Problem-13** **Finding the Missing Number:** We are given a list of $n - 1$ integers and these integers are in the range of 1 to $n$. There are no duplicates in list. One of the integers is missing in the list. Given an algorithm to find the missing integer. **Example**: I/P: $[1, 2, 4, 6, 3, 7, 8]$ O/P: 5

**Brute Force Solution:** One simple solution to this is, for each number in 1 to $n$ check whether that number is in the given array or not.

```
int FindMissingNumber(int A[], int n) {
        int i, j, found=0;
        for (i = 1; i <=n; i ++) {
                found = 0;
                for (j = 0; j < n; j ++)
                        if(A[j]==i)
                                found = 1;
                if(!found) return i;
        }
        return -1;
}
```

Time Complexity: $O(n^2)$. Space Complexity: $O(1)$.

**Problem-14** For Problem-13, can we use sorting technique?

**Solution: Yes.** Sorting the list will give the elements in increasing order and with another scan we can find the missing number.

Time Complexity: $O(nlogn)$, for sorting. Space Complexity: $O(1)$.

**Problem-15** For Problem-13, can we use hashing technique?

**Solution: Yes.** Scan the input array and insert elements into the hash. For inserted element keep *counter* as 1 (assume initially all entires are filled with zeros). This indicates that the corresponding element has occurred already. Now, scan the hash table and return the element which has counter value zero.

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

**Problem-16** For Problem-13, can we improve the complexity?

**Solution:** Using summation formula

1) Get the sum of numbers, $sum = n * (n + 1)/2$.
2) Subtract all the numbers from *sum* and you will get the missing number.

Time Complexity: $O(n)$, for scanning the complete array.

**Problem-17** In Problem-13, if the sum of the numbers goes beyond maximum allowed integer, then there can be integer overflow and we may not get correct answer. Can we solve this problem?

**Solution:**

1) *XOR* all the array elements, let the result of *XOR* be *X*.
2) *XOR* all numbers from 1 to $n$, let *XOR* be Y.
3) *XOR* of *X* and *Y* gives the missing number.





```
int FindMissingNumber(int A[], int n) {
        int i, X, Y;
        for (i = 0; i < n; i ++)
                X ^= A[i];
        for (i = 1; i <= n; i ++)
                Y ^= i;
        //In fact, one variable is enough.
        return X ^ Y;
}
```

Time Complexity: $O(n)$, for scanning the complete array. Space Complexity: $O(1)$.

**Problem-18** **Find the Number Occurring Odd Number of Times:** Given an array of positive integers, all numbers occurs even number of times except one number which occurs odd number of times. Find the number in $O(n)$ time & constant space. **Example**: I/P = $[1, 2, 3, 2, 3, 1, 3]$ O/P = 3

**Solution:** Do a bitwise *XOR* of all the elements. We get the number which has odd occurrences. This is because of the fact that, $A\ XOR\ A = 0$.

Time Complexity: $O(n)$. Space Complexity: $O(1)$.

**Problem-19** **Find the two repeating elements in a given array:** Given an array with $n + 2$ elements, all elements of the array are in range 1 to $n$ and also all elements occur only once except two numbers which occur twice. Find those two repeating numbers. For example: if the array is $4, 2, 4, 5, 2, 3, 1$ with $n = 5$. This input has $n + 2 = 7$ elements with all elements occurring once except 2 and 4 which occur twice. So the output should be 4 2.

**Solution:** One simple way is to scan the complete array for each element of the input elements. That means use two loops. In the outer loop, select elements one by one and count the number of occurrences of the selected element in the inner loop. For the below code assume that *PrintRepeatedElements* is called with $n + 2$ to indicate the size.

```
void PrintRepeatedElements(int A[], int n) {
        for(int i = 0; i < n; i++)
                for(int j = i+1; j <n; j++)
                        if(A[i] == A[j])
                                printf("%d", A[i]);
}
```

Time Complexity: $O(n^2)$. Space Complexity: $O(1)$.

**Problem-20** For the Problem-19, can we improve the time complexity?

**Solution:** Sort the array using any comparison sorting algorithm and see if there are any elements which contiguous with same value.

Time Complexity: $O(nlogn)$. Space Complexity: $O(1)$.

**Problem-21** For the Problem-19, can we improve the time complexity?

**Solution:** Use Count Array. This solution is like using a hash table. For simplicity we can use array for storing the counts. Traverse the array once and keep track of count of all elements in the array using a temp array *count*[] of size $n$. When we see an element whose count is already set, print it as duplicate. For the below code assume that *PrintRepeatedElements* is called with $n + 2$ to indicate the size.

```
void PrintRepeatedElements(int A[], int n) {
        int *count = (int *)calloc(sizeof(int), (n - 2));
        for(int i = 0; i < size; i++) {
                count[A[i]]++;
                if(count[A[i]] == 2)
```

```
            printf("%d", A[i]);
        }
}
```

Time Complexity: O($n$). Space Complexity: O($n$).

**Problem-22** Consider the Problem-19. Let us assume that the numbers are in the range 1 to $n$. Is there any other way of solving the problem?

**Solution: Using XOR Operation.** Let the repeating numbers be $X$ and $Y$, if we $XOR$ all the elements in the array and also all integers from 1 to $n$, then the result would be $X\ XOR\ Y$. The 1's in binary representation of $X\ XOR\ Y$ is corresponding to the different bits between $X$ and $Y$. If the $k^{th}$ bit of $X\ XOR\ Y$ is 1, we can $XOR$ all the elements in the array and also all integers from 1 to $n$, whose $k^{th}$ bits are 1. The result will be one of $X$ and $Y$.

```
void PrintRepeatedElements (int A[], int size) {
    int XOR = A[0];
    int i, right_most_set_bit_no, X= 0, Y = 0;
    for(i = 1; i < size; i++)              /* Compute XOR of all elements in A[]*/
        XOR ^= A[i];
    for(i = 1; i <= n; i++)                /* Compute XOR of all elements {1, 2 ..n} */
        XOR ^= i;
    right_most_set_bit_no = XOR & ~( XOR -1);   // Get the rightmost set bit in right_most_set_bit_no

    /* Now divide elements in two sets by comparing rightmost set */
    for(i = 0; i < size; i++) {
        if(A[i] & right_most_set_bit_no)
            X = X^ A[i];       /*XOR of first set in A[] */
        else    Y = Y ^ A[i];  /*XOR of second set inA[] */
    }
    for(i = 1; i <= n; i++) {
        if(i & right_most_set_bit_no)
            X = X ^ i;         /*XOR of first set in A[] and {1, 2, ...n }*/
        else    Y = Y ^ i;     /*XOR of second set in A[] and {1, 2, ...n } */
    }
    printf("%d and %d",X, Y);
}
```

Time Complexity: O($n$). Space Complexity: O(1).

**Problem-23** Consider the Problem-19. Let us assume that the numbers are in the range 1 to $n$. Is there yet other way of solving the problem?

**Solution:** We can solve this by creating two simple mathematical equations. Let us assume that two numbers which we are going to find are $X$ and $Y$. We know the sum of $n$ numbers is $n(n+1)/2$ and product is $n!$. Make two equations using these sum and product formulae, and get values of two unknowns using the two equations. Let the summation of all numbers in array be $S$ and product be $P$ and the numbers which are being repeated are $X$ and $Y$.

$$X + Y = \frac{n(n+1)}{2} - S$$

$$XY = n!/P$$

Using above two equations, we can find out $X$ and $Y$. There can be addition and multiplication overflow problem with this approach.

Time Complexity: O($n$). Space Complexity: O(1).





**Problem-24** Similar to Problem-19. Let us assume that the numbers are in the range 1 to $n$. Also, $n-1$ elements are repeating thrice and remaining element repeated twice. Find the element which repeated twice.

**Solution:** If we $XOR$ all the elements in the array and all integers from 1 to $n$, then the all the elements which are thrice will become zero. This is because, since the element is repeating thrice and XOR with another time from range makes that element appearing four times. As a result, output of $a\ XOR\ a\ XOR\ a\ XOR\ a = 0$. Same is case with all elements which repeated three times.

With the same logic, for the element which repeated twice, if we $XOR$ the input elements and also the range, then the total number of appearances for that element is 3. As a result, output of $a\ XOR\ a\ XOR\ a = a$. Finally, we get the element which repeated twice.

Time Complexity: O($n$). Space Complexity: O(1).

**Problem-25** Given an array of $n$ elements. Find two elements in the array such that their sum is equal to given element $K$.

**Brute Force Solution:** One simple solution to this is, for each input element check whether there is any element whose sum is $K$. This we can solve just by using two simple for loops. The code for this solution can be given as:

```
void BruteForceSearch[int A[], int n, int K) {
    for (int i = 0; i < n; i++) {
        for(int j = i; j < n; j++) {
            if(A[i]+A[j] == K) {
                printf("Items Found:%d %d", i, j);
                return;
            }
        }
    }
    printf("Items not found: No such elements");
}
```

Time Complexity: O($n^2$). This is because of two nested for loops. Space Complexity: O(1).

**Problem-26** For the Problem-25, can we improve the time complexity?

**Solution: Yes.** Let us assume that we have sorted the given array. This operation takes O($nlogn$). On the sorted array, maintain indices $loIndex = 0$ and hiIndex = $n-1$ and compute $A[loIndex] + A[hiIndex]$. If the sum equals $K$, then we are done with the solution. If the sum is less than $K$, decrement ~~$hiIndex$~~, if the sum is greater than $K$, increment $loIndex$.

```
void Search[int A[], int n, int K) {
    int i, j, temp;
    Sort(A, n);
    for(i = 0, j = n-1; i < j) {
        temp = A[i] + A[j];
        if(temp == K) {
            printf("Elements Found: %d %d", i, j);
            return;
        }
        else if(temp < K)
            i = i + 1;
        else    j = j - 1;
    }
    return;
}
```

Time Complexity: $O(nlogn)$. If the given array is already sorted then the complexity is $O(n)$. Space Complexity: $O(1)$.

**Problem-27** Does the solution of Problem-25 works even if the array is not sorted?

**Solution: Yes.** Since we are checking all possibilities, the algorithm ensures that we get the pair of numbers if they exist.

**Problem-28** Is there any other way of solving the Problem-25?

**Solution: Yes,** using hash table. Since our objective is to find two indexes of the array whose sum is $K$. Let us say those indexes are $X$ and $Y$. That means, $A[X] + A[Y] = K$.

What we need is, for each element of the input array $A[X]$, check whether $K - A[X]$ also exists in input array. Now, let us simplify that searching with hash table.

**Algorithm:**

- For each element of the input array, insert into the hash table. Let us say the current element is $A[X]$.
- Before proceeding to the next element we check whether $K - A[X]$ also exists in hash table or not.
- Existence of such number indicates that we are able to find the indexes.
- Otherwise proceed to the next input element.

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

**Problem-29** Given an array A of $n$ elements. Find three indices, $i, j$ & $k$ such that $A[i]^2 + A[j]^2 = A[k]^2$?

**Solution:**

**Algorithm:**

- Sort the given array in-place.
- For each array index $i$ compute $A[i]^2$ and store in array.
- Search for 2 numbers in array from 0 to $i-1$ which adds to $A[i]$ similar to Problem-25. This will give us the result in $O(n)$ time. If we find such sum return true otherwise continue.

```
Sort(A); // Sort the input array
for (int i=0; i < n; i++)
    A[i] = A[i]*A[i];
for (i=n; i > 0; i--) {
    res = false;
    if(res) {
        //Problem-11/12 Solution
    }
}
```

Time Complexity:Time for sorting + $n \times$ (Time for finding the sum) $= O(nlogn) + n \times O(n) = n^2$. Space Complexity: $O(1)$.

**Problem-30** **Two elements whose sum is closest to zero:** Given an array with both positive and negative numbers, find the two elements such that their sum is closest to zero. For the below array, algorithm should give $-80$ and 85. Example: 1 60 $-10$ 70 $-80$ 85

**Brute Force Solution:** For each element, find the sum of it with every other element in the array and compare sums. Finally, return the minimum sum.

```
void TwoElementsWithMinSum(int A[], int n) {
    int i, j, min_sum, sum, min_i, min_j, inv_count = 0;
    if(n < 2) {
        printf("Invalid Input");
        return;
```





```
    }
    /* Initialization of values */
    min_i = 0;
    min_j = 1;
    min_sum = A[0] + A[1];
    for(i= 0; i < n - 1; i ++)    {
        for(j = i + 1; j < n; j++)    {
            sum = A[i] + A[j];
            if(abs(min_sum) > abs(sum)) {
                min_sum = sum;
                min_i = i;
                min_j = j;
            }
        }
    }
    printf(" The two elements are %d and %d", arr[min_i], arr[min_j]);
}
```

Time complexity: $O(n^2)$. Space Complexity: $O(1)$.

**Problem-31** Can we improve the time complexity of Problem-30?

**Solution:** Use Sorting.

**Algorithm:**

1. Sort all the elements of the given input array.
2. Maintain two indexes one at the beginning ($i = 0$) and other at the ending ($j = n - 1$). Also, maintina two variables to keep track of smallest positive sum closest to zero and smallest negative sum closest to zero.
3. While $i < j$:
   a. If the current pair sum is > zero and < postiveClosest then update the postiveClosest. Decrement $j$.
   b. If the current pair sum is < zero and > negativeClosest then update the negativeClosest. Increment $i$.
   c. Else, print the pair

```
void TwoElementsWithMinSum(int A[], int n) {
    int i = 0, j = n-1, temp, postiveClosest = INT_MAX, negativeClosest = INT_MIN;
    Sort(A, n);
    while(i < j) {
        temp  = A[i] + A[j];
        if(temp  > 0) {
            if (temp < postiveClosest)
                postiveClosest = temp;
            j--;
        }
        else if (temp  < 0) {
            if (temp > negativeClosest)
                negativeClosest = temp;
            i++;
        }
        else printf("Closest Sum: %d ", A[i] + A[j]);
    }
    return (abs(negativeClosest)> postiveClosest: postiveClosest: negativeClosest);
}
```

Time Complexity: O($nlogn$), for sorting. Space Complexity: O(1).

**Problem-32** Given an array of $n$ elements. Find three elements in the array such that their sum is equal to given element $K$?

**Brute Force Solution:** The default solution to this is, for each pair of input elements check whether there is any element whose sum is $K$. This we can solve just by using three simple for loops. The code for this solution can be given as:

```
void BruteForceSearch[int A[], int n, int data) {
        for (int i = 0; i < n; i++) {
                for(int j = i+1; j < n; j++) {
                        for(int k = j+1; k < n; k++)          {
                                if(A[i] + A[j] + A[k]== data) {
                                        printf("Items Found:%d %d %d", i, j, k);
                                        return;
                                }
                        }
                }
        }
        printf("Items not found: No such elements");
}
```

Time Complexity: O($n^3$), for three nested $for$ loops. Space Complexity: O(1).

**Problem-33** Does the solution of Problem-32 works even if the array is not sorted?

**Solution: Yes.** Since we are checking all possibilities, the algorithm ensures that we can find three numbers whose sum is $K$ if they exist.

**Problem-34** Can we use sorting technique for solving Problem-32?

**Solution: Yes.**

```
void Search[int A[], int n, int data) {
        Sort(A, n);
        for(int k = 0; k < n; k++) {
                for(int i =  k + 1, j = n-1; i < j; ) {
                        if(A[k] + A[i] + A[j]  == data) {
                                printf("Items Found:%d %d %d", i, j, k);
                                return;
                        }
                        else if(A[k] + A[i] + A[j]  < data)
                                i =  i + 1;
                        else    j = j - 1;
                }
        }
        return;
}
```

Time Complexity: Time for sorting + Time for searching in sorted list = O($nlogn$) + O($n^2$) ≈ O($n^2$). This is because of two nested $for$ loops. Space Complexity: O(1).

**Problem-35** Can we use hashing technique for solving Problem-32?

**Solution: Yes.** Since our objective is to find three indexes of the array whose sum is $K$. Let us say those indexes are $X, Y$ and $Z$. That means, $A[X] + A[Y] + A[Z] = K$.





Let us assume that we have kept all possible sums along with their pairs in hash table. That means the key to hash table is $K - A[X]$ and values for $K - A[X]$ are all possible pairs of input whose sum is $K - A[X]$.

**Algorithm:**

- Before starting the searching, insert all possible sums with pairs of elements into the hash table.
- For each element of the input array, insert into the hash table. Let us say the current element is $A[X]$.
- Check whether there exists a hash entry in the table with key: $K - A[X]$.
- If such element exists then scan the element pairs of $K - A[X]$ and return all possible pairs by including $A[X]$ also.
- If no such element exists (with $K - A[X]$ as key) then go to next element.

Time Complexity: Time for storing all possible pairs in Hash table + searching = O($n^2$) + O($n^2$) ≈ O($n^2$). Space Complexity: O($n$).

**Problem-36** Given an array of $n$ integers, the $3 - sum\ problem$ is to determine find three integers whose sum is closest to $zero$.

**Solution:** This is same as that of Problem-32 with $K$ value is zero.

**Problem-37** Let A be an array of $n$ distinct integers. Suppose A has the following property: there exists an index $1 \le k \le n$ such that $A[1], \ldots, A[k]$ is an increasing sequence and $A[k+1], \ldots, A[n]$ is a decreasing sequence. Design and analyze an efficient algorithm for finding $k$.

**Similar question:** Let's assume that the given array is sorted but starts with negative numbers and ends with positive numbers [such functions are called monotonically increasing function]. In this array find the starting index of the positive numbers. Assume that we know the length of the input array. Design a O($logn$) algorithm.

**Solution:** Let us use a variant of the binary search.

```
int Search (int A[], int n, int first, int last) {
        int mid, first = 0, last = n-1;
        while(first <= last) {
                // if the current array has size 1
                if(first == last)
                        return A[first];
                // if the current array has size 2
                else if(first == last-1)
                        return max(A[first], A[last]);
                // if the current array has size 3 or more
                else {
                        mid = first + (last-first)/2;
                        if(A[mid-1] < A[mid] && A[mid] > A[mid+1])
                                return A[mid];
                        else if(A[mid-1] < A[mid] && A[mid] < A[mid+1])
                                first = mid+1;
                        else if(A[mid-1] > A[mid] && A[mid] > A[mid+1])
                                last = mid-1;
                        else    return INT_MIN ;
                } // end of else
        } // end of while
}
```

The recursion equation is $T(n) = 2T(n/2) + c$. Using master theorem, we get O($logn$).

**Problem-38** If we don't know $n$, how do we solve the Problem-37?

**Solution:** Repeatedly compute $A[1], A[2], A[4], A[8], A[16]$, and so on until we find a value of $n$ such that $A[n] > 0$.

Time Complexity: O($logn$), since we are moving at the rate of 2. Refer *Introduction to Analysis of Algorithms* chapter for details on this.

**Problem-39** Given an input array of size unknown with all $1's$ in the beginning and $0's$ in the end. Find the index in the array from where $0's$ start. Consider there are millions of $1's$ and $0's$ in the array. E.g. array contents 1111111.....1100000.....0000000.

**Solution:** This problem is almost similar to Problem-38. Check the bits at the rate of $2^K$where $k = 0, 1, 2$ .... Since we are moving at the rate of 2, the complexity is O($logn$).

**Problem-40** Given a sorted array of $n$ integers that has been rotated an unknown number of times, give a O($logn$) algorithm that finds an element in the array. **Example:** Find 5 in array (15 16 19 20 25 1 3 4 5 7 10 14) **Output:** 8 (the index of 5 in the array)

**Solution:** Let us assume that the given array is $A$[]. Using solution of Problem-37, with extension. The below function *FindPivot* returns the $k$ value (let us assume that this function return the index instead of value). Find the pivot point, divide the array in two sub-arrays and call binary search.

The main idea for finding pivot is – for a sorted (in increasing order) and pivoted array, pivot element is the only element for which next element to it is smaller than it. Using above criteria and binary search methodology we can get pivot element in O($logn$) time.

**Algorithm:**

1) Find out pivot point and divide the array in two sub-arrays.
2) Now call binary search for one of the two sub-arrays.
   a. if element is greater than first element then search in left subarray
   b. else search in right subarray
3) If element is found in selected sub-array then return index *else* return $-1$.

```
int FindPivot(int A[], int start, int finish) {
        if(finish - start == 0)
                return start;
        else if(start == finish - 1) {
                if(A[start] >= A[finish])
                        return start;
                else    return finish;
        }
        else {
                mid = start + (finish-start)/2;
                if(A[start] >= A[mid])
                        return FindPivot(A, start, mid);
                else    return FindPivot(A, mid, finish);
        }
}
int Search(int A[], int n, int x) {
        int pivot = FindPivot(A, 0, n-1);
        if(A[pivot] == x)
                return pivot;
        if(A[pivot] <= x)
                return BinarySearch(A, 0, pivot-1, x);
        else    return BinarySearch(A, pivot+1, n-1, x);
}
int BinarySearch(int A[], int low, int high, int x) {
        if(high >= low)  {
```





```
                int mid = low + (high - low)/2;
                if(x == A[mid])
                        return mid;
                if(x > A[mid])
                        return BinarySearch(A, (mid + 1), high, x);
                else    return BinarySearch(A, low, (mid -1), x);
        }
        return -1;      /*Return -1 if element is not found*/
}
```

Time complexity:O($logn$).

**Problem-41** For Problem-40, can we solve in one scan?

**Solution: Yes.**

```
int BinarySearchRotated(int A[], int start, int finish, int data) {
        int mid = start + (finish - start) / 2;
        if(start > finish)
                return -1;
        if(data == A[mid])
                return mid;
        else if(A[start] <= A[mid]) {          // start half is in sorted order.
                if(data >= A[start] && data < A[mid])
                        return BinarySearchRotated(A, start, mid - 1, data);
                else    return BinarySearchRotated(A, mid + 1, finish, data);
        }
        else {  // A[mid] <= A[finish], finish half is in sorted order.
                if(data > A[mid] && data <= A[finish])
                        return BinarySearchRotated(A, mid + 1, finish, data);
                else    return BinarySearchRotated(A, start, mid - 1, data);
        }
}
```

Time complexity:O($logn$).

**Problem-42** **Bitonic search:** An array is *bitonic* if it is comprised of an increasing sequence of integers followed immediately by a decreasing sequence of integers. Given a bitonic array A of n distinct integers, describe how to determine whether a given integer is in the array in O($logn$) steps.

**Solution:** This is same as Problem-37.

**Problem-43** Yet, other way of asking Problem-37?
Let $A$[] be an array that starts out increasing, reaches a maximum, and then decreases. Design an O($logn$) algorithm to find the index of the maximum value.

**Problem-44** Give an O($nlogn$) algorithm for computing the median of a sequence of $n$ integers.

**Solution:** Sort and return element at $\frac{n}{2}$.

**Problem-45** Given two sorted lists of size $m$ and $n$, find median of all elements in O($log\ (m + n)$) time.

**Solution:** Refer *Divide and Conquer* chapter.

**Problem-46** Given a sorted array $A$ of $n$ elements, possibly with duplicates, find the index of the first occurrence of a number in O($logn$) time.

**Solution:** To find the first occurrence of a number we need to check for the following condition. Return the position if any one of the following is true:

mid == low && A[mid] == data || A[mid] == data && A[mid-1] < data

```
int BinarySearchFirstOccurrence(int A[], int low, int high, int data) {
        int mid;
        if(high >= low) {
                mid = low + (high-low) / 2;
                if((mid == low && A[mid] == data) || (A[mid] == data && A[mid - 1] < data))
                        return mid;
                // Give preference to left half of the array
                else if(A[mid] >= data)
                        return BinarySearchFirstOccurrence (A, low, mid - 1, data);
                else    return BinarySearchFirstOccurrence (A, mid + 1, high, data);
        }
        return -1;
}
```

Time Complexity: O($logn$).

**Problem-47** Give a sorted array $A$ of $n$ elements, possibly with duplicates, find the index of the last occurrence of a number in O($logn$) time.

**Solution:** To find the last occurrence of a number we need to check for the following condition. Return the position if any one of the following is true:

mid == high && A[mid] == data || A[mid] == data && A[mid+1] > data

```
int BinarySearchLastOccurrence(int A[], int low, int high, int data) {
        int mid;
        if(high >= low) {
                mid = low + (high-low) / 2;
                if((mid == high && A[mid] == data) || (A[mid] == data && A[mid + 1] > data))
                        return mid;
                // Give preference to right half of the array
                else if(A[mid] <= data)
                        return BinarySearchLastOccurrence (A, mid + 1, high, data);
                else    return BinarySearchLastOccurrence (A, low, mod - 1, data);
        }
        return -1;
}
```

Time Complexity: O($logn$).

**Problem-48** Give a sorted array of $n$ elements, possibly with duplicates, find the number of occurrences of a number.

**Brute For Solution:** Do a linear search over the array and increment count as and when we find the element data in the array.

```
int LinearSearchCount(int A[], int n, int data) {
        int count = 0;
        for (int i = 0; i < n; i++)
                if(a[i] == k)
                        count++;
        return count;
```





```
}
```

Time Complexity: O($n$).

**Problem-49** Can we improve the time complexity of Problem-48?

**Solution: Yes.** We can solve this by using one binary search call followed by another small scan.

**Algorithm:**

- Do a binary search for the $data$ in the array. Let us assume its position be $K$.
- Now traverse towards left from K and count the number of occurrences of $data$. Let this count be $leftCount$.
- Similarly, traverse towards right and count the number of occurrences of $data$. Let this count be $rightCount$.
- Total number of occurrences = $leftCount + 1 + rightCount$

Time Complexity – O($logn + S$) where $S$ is the number of occurrences of $data$.

**Problem-50** Is there any alternative way of solving the Problem-48?

**Solution:**

**Algorithm:**

- Find first occurrence of $data$ and call its index as $firstOccurrence$ (for algorithm refer Problem-46)
- Find last occurrence of $data$ and call its index as $lastOccurrence$ (for algorithm refer Problem-47)
- Return $lastOccurrence - firstOccurrence + 1$

Time Complexity = O($logn + logn$) =O($logn$).

**Problem-51** What is the next number in the sequence 1, 11, 21 and why?

**Solution:** Read the given number loudly. This is just a fun problem.

One one
Two Ones
One two, one one→ 1211

So answer is, the next number is the representation of previous number by reading it loudly.

**Problem-52** Finding second smallest number efficiently.

**Solution:** We can construct a heap of the given elements using up just less than $n$ comparisons (Refer *Priority Queues* chapter for algorithm). Then we find the second smallest using $logn$ comparisons for the GetMax() operation. Overall, we get $n + logn + constant$.

**Problem-53** Is there any other solution for Problem-52?

**Solution:** Alternatively, split the $n$ numbers into groups of 2, perform $n/2$ comparisons successively to find the largest using a tournament-like method. The first round will yield the maximum in $n - 1$ comparisons. The second round will be performed on the winners of the first round and the ones the maximum popped. This will yield $logn - 1$ comparisons for a total of $n + logn - 2$. The above solution is called as *tournament problem*.

**Problem-54** An element is a majority if it appears more than $n/2$ times. Give and algorithm takes an array of $n$ element as argument and identifies a majority (if it exists).

**Solution:** The basic solution is to have two loops and keep track of maximum count for all different elements. If maximum count becomes greater than $n/2$ then break the loops and return the element having maximum count. If maximum count doesn't become more than $n/2$ then majority element doesn't exist.

Time Complexity: O($n^2$). Space Complexity: O(1).

**Problem-55** Can we improve the Problem-54 time complexity to O($nlogn$)?

**Solution:** Using binary search we can achieve this. Node of the Binary Search Tree (used in this approach) will be as follows.

```
        struct TreeNode {
```

```
        int element;
        int count;
        struct TreeNode *left;
        struct TreeNode *right;
} BST;
```

Insert elements in BST one by one and if an element is already present then increment the count of the node. At any stage, if count of a node becomes more than $n/2$ then return. The method works well for the cases where $n/2+1$ occurrences of the majority element is present in the starting of the array, for example $\{1, 1, 1, 1, 1, 2, 3, \text{and } 4\}$.

Time Complexity: If a binary search tree is used then worst time complexity will be $O(n^2)$. If a balanced-binary-search tree is used then $O(nlogn)$. Space Complexity: $O(n)$.

**Problem-56** Is there any other of achieving $O(nlogn)$ complexity for the Problem-54?

**Solution:** Sort the input array and scan the sorted array to find the majority element.

Time Complexity: $O(nlogn)$. Space Complexity: $O(1)$.

**Problem-57** Can we improve the complexity for the Problem-54?

**Solution:** If an element occurs more than $n/2$ times in $A$ then it must be the median of $A$. But, the reverse is not true, so once the median is found, we must check to see how many times it occurs in $A$. We can use linear selection which takes $O(n)$ time (for algorithm refer *Selection Algorithms* chapter).

```
int CheckMajority(int A[], in n) {
    1) Use linear selection to find the median m of A.
    2) Do one more pass through A and count the number of occurrences of m.
        a. If m occurs more than n/2 times then return true;
        b. Otherwise return false.
}
```

**Problem-58** Is there any other way of solving the Problem-54?

**Solution:** Since only one element is repeating, we can use simple scan of the input array by keeping track of count for the elements. If the count is 0 then we can assume that the element is coming first time otherwise that the resultant element.

```
int MajorityNum(int[] A, int n) {
    int count = 0, element = -1;
    for(int i = 0; i < n; i++) {
        // If the counter is 0 then set the current candidate to majority num and set the counter to 1.
        if(count == 0) {
            element = A[i];
            count = 1;
        }
        else if(element == A[i]) {
            // Increment counter If the counter is not 0 and element is same as current candidate.
            count++;
        }
        else {  // Decrement counter If the counter is not 0 and element is different from current candidate.
            count--;
        }
    }
    return element;
}
```





Time Complexity: $O(n)$. Space Complexity: $O(1)$.

**Problem-59** Given an array of $2n$ elements of which $n$ elements are same and the remaining $n$ elements are all different. Find the majority element.

**Solution:** The repeated elements will occupy half the array. No matter what arrangement it is, only one of the below will be true,

- All duplicate elements will be at a relative distance of 2 from each other. Ex: n, 1, n, 100, n, 54, n ...
- At least two duplicate elements will be next to each other

  Ex: $n, n, 1, 100, n, 54, n, \ldots$.
  $n, 1, n, n, n, 54, 100 \ldots$
  $1, 100, 54, n, n, n, n \ldots$.

In worst case, we need will two passes over the array,

- First Pass: compare $A[i]$ and $A[i+1]$
- Second Pass: compare $A[i]$ and $A[i+2]$

Something will match and that's your element. This will cost $O(n)$ in time and $O(1)$ in space.

**Problem-60** Given an array with $2n+1$ integer elements, $n$ elements appear twice in arbitrary places in the array and a single integer appears only once somewhere inside. Find the lonely integer with $O(n)$ operations and $O(1)$ extra memory.

**Solution:** Since except one element all other elements are repeated. We know that $A \; XOR \; A = 0$. Based on this if we *XOR* all the input elements then we get the remaining element.

```
int Solution(int* A) {
    int i, res;
    for (i = res = 0; i < 2n+1; i++)
        res = res ^ A[i];
    return res;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(1)$.

**Problem-61** **Throwing eggs from an n-story building:** Suppose that we have an $n$ story building and a set of eggs. Also assume that an egg breaks if it is thrown off floor $F$ or higher, and will not break otherwise. Devise a strategy to determine the floor $F$, while breaking $O(logn)$ eggs.

**Solution:** Refer *Divide and Conquer* chapter.

**Problem-62** **Local minimum of an array:** Given an array $A$ of $n$ distinct integers, design an $O(logn)$ algorithm to find a *local minimum*: an index $i$ such that $A[i-1] < A[i] < A[i+1]$.

**Solution:** Check the middle value $A[n/2]$, and two neighbors $A[n/2-1]$ and $A[n/2+1]$. If $A[n/2]$ is local minimum, stop; otherwise search in half with smaller neighbor.

**Problem-63** Give an $n \times n$ array of elements such that each row is in ascending order and each column is in ascending order, devise an $O(n)$ algorithm to determine if a given element $x$ in the array. You may assume all elements in the $n \times n$ array are distinct.

**Solution:** Let us assume that the given matrix is $A[n][n]$. Start with the last row, first column [or first row - last column]. If the element we are searching for is greater than the element at $A[1][n]$, then the column 1 can be eliminated. If the search element is less than the element at $A[1][n]$, then the last row can be completely eliminated. Once the first column or the last row is eliminated, start over the process again with left-bottom end of the remaining array. In this algorithm, there would be maximum $n$ elements that the search element would be compared with.

Time Complexity: $O(n)$. This is because we will traverse at most $2n$ points. Space Complexity: $O(1)$.

**Problem-64** Given an $n \times n$ array a of $n^2$ numbers, give an $O(n)$ algorithm to find a pair of indices $i$ and $j$ such that $A[i][j] < A[i+1][j], A[i][j] < A[i][j+1], A[i][j] < A[i-1][j]$, and $A[i][j] < A[i][j-1]$.

**Solution:** This problem is same as Problem-63.

**Problem-65** Given $n \times n$ matrix, and in each row all 1's are followed 0's. Find row with maximum number of 0's.

**Solution:** Start with first row, last column. If the element is 0 then move to the previous column in the same row and at the same time increase the counter to indicate the maximum number of 0's. If the element is 1 then move to next row in the same column. Repeat this process until we reach last row, first column.

Time Complexity: $O(2n) \approx O(n)$ (similar to Problem-63).

**Problem-66** Given an input array of size unknown with all numbers in the beginning and special symbols in the end. Find the index in the array from where special symbols start.

**Solution:** Refer *Divide and Conquer* chapter.

**Problem-67** **Separate Even and Odd numbers:** Given an array $A[\,]$, write a function that segregates even and odd numbers. The functions should put all even numbers first, and then odd numbers. **Example**: Input = {12, 34, 45, 9, 8, 90, 3} Output = {12, 34, 90, 8, 9, 45, 3}

**Note:** In the output, order of numbers can be changed, i.e., in the above example 34 can come before 12 and 3 can come before 9.

**Solution:** The problem is very similar to *Separate 0's and 1's (*Problem-68*)* in an array, and both of these problems are variation of famous *Dutch national flag problem*.

**Algorithm:** Logic is little similar to Quick sort.

1) Initialize two index variables left and right: $left = 0$, $right = n-1$
2) Keep incrementing left index until we see an odd number.
3) Keep decrementing right index until we see an even number.
4) If $left < right$ then swap $A[left]$ and $A[right]$

```
void DutchNationalFlag(int A[], int n) {
        /* Initialize left and right indexes */
        int left = 0, right = n-1;
        while(left < right) {
                /* Increment left index while we see 0 at left */
                while(A[left]%2 == 0 && left < right)
                        left++;
                /* Decrement right index while we see 1 at right */
                while(A[right]%2 == 1 && left < right)
                        right--;
                if(left < right) {
                        /* Swap A[left] and A[right]*/
                        swap(&A[left], &A[right]);
                        left++;
                        right--;
                }
        }
}
```

Time Complexity: $O(n)$.

**Problem-68** Other way of asking Problem-67 but with little difference.





**Separate 0's and 1's in an array:** We are given an array of 0's and 1's in random order. Separate 0's on left side and 1's on right side of the array. Traverse array only once.

**Input array** = [0, 1, 0, 1, 0, 0, 1, 1, 1, 0] **Output array** = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]

**Solution:** Counting 0's or 1's

1. Count the number of 0's. Let count be $C$.
2. Once we have count, put $C$ 0's at the beginning and 1's at the remaining $n - C$ positions in array.

Time Complexity: $O(n)$. This solution scans the array two times.

**Problem-69** Can we solve the Problem-68 in one scan?

**Solution: Yes.** Use two indexes to traverse: Maintain two indexes. Initialize first index left as 0 and second index right as $n-1$. Do following while $left < right$:

1) Keep incrementing index left while there are 0s at it
2) Keep decrementing index right while there are 1s at it
3) If left < right then exchange $A[left]$ and $A[right]$

```
/*Function to put all 0s on left and all 1s on right*/
void Separate0and1(int A[], int n) {
        /* Initialize left and right indexes */
        int left = 0, right = n-1;
        while(left < right) {
                /* Increment left index while we see 0 at left */
                while(A[left] == 0 && left < right)
                        left++;
                /* Decrement right index while we see 1 at right */
                while(A[right] == 1 && left < right)
                        right--;
                /* If left is smaller than right then there is a 1 at left
                and a 0 at right.  Swap A[left] and A[right]*/
                if(left < right) {
                        A[left] = 0;
                        A[right] = 1;
                        left++;
                        right--;
                }
        }
}
```

Time Complexity: $O(n)$. Space Complexity: $O(1)$.

**Problem-70** **Sort an array of 0's, 1's and 2's [or R's, G's and B's]:** Given an array A[] consisting 0's, 1's and 2's, give an algorithm for sorting $A[\,]$. The algorithm should put all 0's first, then all 1's and all 2's in last. **Example** **Input** = {0,1,1,0,1,2,1,2,0,0,0,1}, **Output** = {0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2}

**Solution:**

```
void Sorting012sDutchFlagProblem(int A[],int n){
        int low=0,mid=0,high=n-1;
        while(mid <=high){
                switch(A[mid]){
                        case 0:
                                swap(A[low],A[mid]);
                                low++;mid++;
                                break;
```

```
                case 1:
                        mid++;
                        break;
                case 2:
                        swap(A[mid],A[high]);
                        high--;
                        break;
            }
        }
}
```

Time Complexity: O($n$). Space Complexity: O(1).

**Problem-71** **Maximum difference between two elements:** Given an array $A[]$ of integers, find out the difference between any two elements such that larger element appears after the smaller number in $A[]$.

**Examples**: If array is [2, 3, 10, 6, 4, 8, 1] then returned value should be 8 (Diff between 10 and 2). If array is [ 7, 9, 5, 6, 3, 2 ] then returned value should be 2 (Difference between 7 and 9)

**Solution:** Refer *Divide and Conquer* chapter.

**Problem-72** Given an array of 101 elements. Out of them 25 elements are repeated twice, 12 elements are repeated 4 times and one element is repeated 3 times. Find the element which repeated 3 times in O(1).

**Solution:** Before solving this problem let us consider the following $XOR$ operation property: $a\ XOR\ a = 0$. That means, if we apply the $XOR$ on same elements then the result is 0.

**Algorithm:**

- $XOR$ all the elements of the given array and assume the result is $A$.
- After this operation, 2 occurrences of number which appeared 3 times becomes 0 and one occurrence will remain.
- The 12 elements which are appearing 4 times become 0.
- The 25 elements which are appearing 2 times become 0.
- So just *XOR'ing* all the elements give the result.

Time Complexity: O($n$), because we are doing only one scan. Space Complexity: O(1).

**Problem-73** Given a number $n$, give an algorithm for finding the number of trailing zeros in $n!$.

**Solution:**

```
int NumberOfTrailingZerosInNumber(int n) {
    int i, count = 0;
    if(n < 0)  return -1;
    for (i = 5; n / i > 0; i *= 5)
        count += n / i;
    return count;
}
```

Time Complexity: O($logn$).

**Problem-74** Given an array of $2n$ integers in the following format $a1\ a2\ a3 \ldots an\ b1\ b2\ b3 \ldots bn$. Shuffle the array to $a1\ b1\ a2\ b2\ a3\ b3 \ldots an\ bn$ without any extra memory.

**Solution:** A brute force solution involves two nested loops to rotate the elements in the second half of the array to the left. The first loop runs $n$ times to cover all elements in the second half of the array. The second loop rotates the elements to the left. Note that the start index in the second loop depends on which element we are rotating and the end index depends on how many positions we need to move to the left.





```
void ShuffleArray() {
    int n = 4;
    int A[] = {1,3,5,7,2,4,6,8};
    for (int i = 0, q =1, k = n; i < n; i++, k++, q++) {
        for (int j = k; j > i + q; j--) {
            int tmp = A[j-1];
            A[j-1] = A[j];
            A[j] = tmp;
        }
    }
    for (int i = 0; i  < 2*n; i++)
        printf("%d", A[i]);
}
```

Time Complexity: O($n^2$).

**Problem-75** Can we improve Problem-74 solution?

**Solution:** Refer Divide and Concur chapter. A better solution of time complexity O($nlogn$) can be achieved using *Divide and Concur* technique. Let us take an example

1. Start with the array: $a1\ a2\ a3\ a4\ b1\ b2\ b3\ b4$
2. Split the array into two halves: $a1\ a2\ a3\ a4 \cdot b1\ b2\ b3\ b4$
3. Exchange elements around the center: exchange $a3\ a4$ with $b1\ b2$ you get: $a1\ a2\ b1\ b2\ a3\ a4\ b3\ b4$
4. Split $a1\ a2\ b1\ b2$ into $a1\ a2 : b1\ b2$ then split $a3\ a4\ b3\ b4$ into $a3\ a4 : b3\ b4$
5. Exchange elements around the center for each subarray you get: $a1\ b1\ a2\ b2$ and $a3\ b3\ a4\ b4$

Note that this solution only handles the case when $n = 2^i$ where $i = 0, 1, 2, 3$ etc. In our example $n = 2^2 = 4$ which makes it easy to recursively split the array into two halves. The basic idea behind swapping elements around the center before calling the recursive function is to produce a smaller size problems. A solution with linear time complexity may be achieved if the elements are of specific nature for example if you can calculate the new position of the element using the value of the element itself. This is nothing but a hashing technique.

**Problem-76** Given an Aay A[], find the maximum j – i such that A[j] > A[i]. For example, Input: {34, 8, 10, 3, 2, 80, 30, 33, 1} and Output: 6 (j = 7, i = 1).

**Solution: Brute Force Approach:** Run two loops. In the outer loop, pick elements one by one from left. In the inner loop, compare the picked element with the elements starting from right side. Stop the inner loop when you see an element greater than the picked element and keep updating the maximum j-i so far.

```
int maxIndexDiff(int A[], int n){
  int maxDiff = -1;
  int i, j;
  for (i = 0; i < n; ++i){
    for (j = n-1; j > i; --j){
      if(A[j] > A[i] && maxDiff < (j - i))
        maxDiff = j - i;
    }
  }
  return maxDiff;
}
```

Time Complexity: O($n^2$). Space Complexity: O(1).

**Problem-77** Can we improve the complexity of Problem-76?

**Solution:** To solve this problem, we need to get two optimum indexes of A[]: left index $i$ and right index $j$. For an element A[i], we do not need to consider A[i] for left index if there is an element smaller than A[i] on left side of A[i]. Similarly, if there is a greater element on right side of A[j] then we do not need to consider this j for right index.

So we construct two auxiliary Aays LeftMins[] and RightMaxs[] such that LeftMins[i] holds the smallest element on left side of A[i] including A[i], and RightMaxs[j] holds the greatest element on right side of A[j] including A[j]. After constructing these two auxiliary arrays, we traverse both of these arrays from left to right.

While traversing LeftMins[] and RightMaxs[] if we see that LeftMins[i] is greater than RightMaxs[j], then we must move ahead in LeftMins[] (or do i++) because all elements on left of LeftMins[i] are greater than or equal to LeftMins[i]. Otherwise we must move ahead in RightMaxs[j] to look for a greater $j - i$ value.

```
int maxIndexDiff(int A[], int n){
  int maxDiff;
  int i, j;
  int *LeftMins = (int *)malloc(sizeof(int)*n);
  int *RightMaxs = (int *)malloc(sizeof(int)*n);
  LeftMins[0] = A[0];
  for (i = 1; i < n; ++i)
    LeftMins[i] = min(A[i], LeftMins[i-1]);
  RightMaxs[n-1] = A[n-1];
  for (j = n-2; j >= 0; --j)
    RightMaxs[j] = max(A[j], RightMaxs[j+1]);
  i = 0, j = 0, maxDiff = -1;
  while (j < n && i < n){
    if (LeftMins[i] < RightMaxs[j]){
      maxDiff = max(maxDiff, j-i);
      j = j + 1;
    }
    else
      i = i+1;
  }
  return maxDiff;
}
```

Time Complexity: O($n$). Space Complexity: O($n$).





# SELECTION ALGORITHMS [MEDIANS]

# Chapter-12

## 12.1 What are Selection Algorithms?

*Selection algorithm* is an algorithm for finding the $k^{th}$ smallest/largest number in a list (also called as $k^{th}$ order statistic). This includes, finding the minimum, maximum, and median elements. For finding $k^{th}$ order statistic, there are multiple solutions which provide different complexities and in this chapter we will try to enumerate those possibilities.

## 12.2 Selection by Sorting

Selection problem can be converted to sorting problem. In this method, we first sort the input elements and then get the desired element. It is efficient if we want to perform many selections.

For example, let us say we want to get the minimum element. After sorting the input elements we can simply return the first element (assuming the array is sorted in ascending order). Now, if we want to find the second smallest element, we can simply return the second element from the sorted list. That means, for the second smallest element we are not performing the sorting again. Same is case with subsequent queries too. Even if we want to get $k^{th}$ smallest element, just one scan of sorted list is enough for finding the element (or we can return the $k^{th}$-indexed value if the elements are in the array).

From the above discussion what we can say is, with the initial sorting we can answer any query in one scan, $O(n)$. In general, this method requires $O(nlogn)$ time (for *sorting*), where $n$ is the length of the input list. Suppose if we are performing $n$ queries then the average cost per operation is just $\frac{n\,logn}{n} \approx O(logn)$. This kind of analysis is called *amortized* analysis.

## 12.3 Partition-based Selection Algorithm

For algorithm check Problem-6. This algorithm works very much similar to Quick sort.

## 12.4 Linear Selection algorithm - Median of Medians algorithm

| | |
|---|---|
| Worst-case performance | $O(n)$ |
| Best case performance | $O(n)$ |
| Worst case space complexity | $O(1)$ auxiliary |

Refer Problem-11.

## 12.5 Finding the K Smallest Elements in Sorted Order

For algorithm check Problem-6. This algorithm works very much similar to Quick sort.

## 12.6 Problems on Selection Algorithms

**Problem-1** Find the largest element in an array A of size $n$.

**Solution:** Scan the complete array and return the largest element.

```
void FindLargestInArray(int n, const int A[]) {
        int large = A[0];
        for (int i = 1; i <= n-1; i++)
            if(A[i] > large)
                large = A[i];
        printf("Largest:%d", large);
}
```

Time Complexity - $O(n)$. Space Complexity - $O(1)$.

**Note:** Any deterministic algorithm that can find the largest of $n$ keys by comparisons of keys takes at least $n-1$ comparisons.

**Problem-2** Find the smallest and largest elements in an array $A$ of size $n$.

**Solution:**

```
void FindSmallestAndLargestInArray (int A[], int n) {
        int small = A[0];
        int large =A[0];
        for (int i = 1; i <= n-1; i++)
                if(A[i] < small)
                        small = A[i];
                else if(A[i] > large)
                        large = A[i];
        printf("Smallest:%d, Largest:%d", small, large);
}
```

Time Complexity - $O(n)$. Space Complexity - $O(1)$. The worst-ease number of comparisons are $2(n-1)$.

**Problem-3** Can we improve the previous algorithms?

**Solution:** **Yes.** Compare in pairs.

```
// n is assumed to be even. Compare in pairs.
void FindWithPairComparison (int A[], int n) {
        int large = small = -1;
        for (int i = 0; i <= n - 1; i = i + 2) {          // Increment i by 2.
                if(A[i] < A[i + 1]) {
                        if(A[i] < small)
                                small = A[i];
                        if(A[i + 1] > large)
                                large = A[i + 1];
                }
                else {  if(A[i + 1] < small)
                                small = A[i + 1];
                        if(A[i] > large)
                                large = A[i];
                }
        }
        printf("Smallest:%d, Largest:%d", small, large);
}
```

Time Complexity - $O(n)$. Space Complexity - $O(1)$.

Number of comparisons: $\begin{cases} \frac{3n}{2} - 2, & \text{if } n \text{ is even} \\ \frac{3n}{2} - \frac{3}{2} & \text{if } n \text{ is odd} \end{cases}$





**Summary:**

| |
|---|
| Straightforward comparison – $2(n-1)$ comparisons |
| Compare for min only if comparison for max fails |
| Best case: increasing order – $n-1$ comparisons |
| Worst case: decreasing order – $2(n-1)$ comparisons |
| Average case: $3n/2 - 1$ comparisons |

**Note:** For divide and conquer techniques refer Divide and Conquer chapter.

**Problem-4** Given an algorithm for finding the second largest element in the given input list of elements.

**Solution: Brute Force Method**

**Algorithm:**

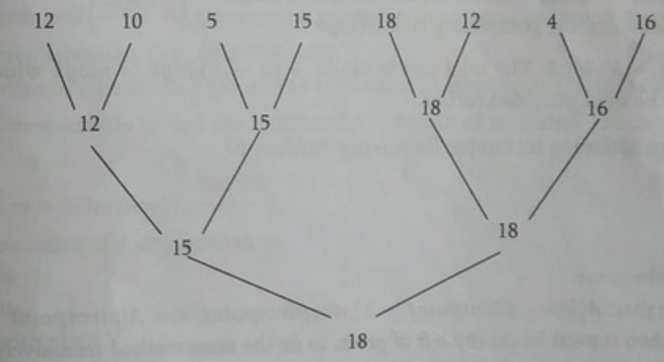
- Find largest element: needs $n-1$ comparisons
- Delete (discard) the largest element
- Again find largest element: needs $n-2$ comparisons

Total number of comparisons: $n-1+n-2 = 2n-3$

**Problem-5** Can we reduce the number of comparisons in Problem-4 solution?

**Solution: The Tournament method:** For simplicity, assume that the numbers are distinct and that $n$ is a power of 2. We pair the keys and compare the pairs in rounds until only one round remains. If the input has eight keys, there are four comparisons in the first round, two in the second, and one in the last. The winner of the last round is the largest key. Below figure shows the method. The tournament method directly applies only when $n$ is a power of 2. When this is not the case, we can add enough items to the end of the array to make the array size a power of 2. If the tree is complete then the maximum height of the tree is $logn$. If we construct the complete binary tree, we need $n-1$ comparisons to find the largest.

The second largest key has to be among the ones that lost in a comparison with the largest one. That means, the second largest element should be one of the opponents of largest element. The number of keys that lost to the largest key is the height of the tree, i.e. $logn$ [if the tree is a complete binary tree]. Then using the selection algorithm to find the largest among them takes $logn - 1$ comparisons. Thus the total number of comparisons to find the largest and second largest keys is $n + logn - 2$.




**Problem-6** Find the $k$-smallest elements in an array $S$ of $n$ elements using partitioning method.

**Solution:** **Brute Force Approach**: Scan through the numbers $k$ times to have the desired element. This method is the one used in bubble sort (and selection sort), every time we find out the smallest element in the whole sequence by comparing every element. In this method, the sequence has to be traversed $k$ times. So the complexity is $O(n \times k)$.

**Problem-7** Can we use sorting technique for solving Problem-6?

**Solution:** **Yes.** Sort and take first $k$ elements.

1. Sort the numbers.
2. Pick the first $k$ elements.

The complexity is very trivial. Sorting of $n$ numbers is of $O(nlogn)$ and picking $k$ elements is of $O(k)$. The total complexity is $O(nlogn + k) = O(nlogn)$.

**Problem-8** Can we use *tree sorting* technique for solving Problem-6?

**Solution: Yes.**

1. Insert all the elements to a binary search tree.
2. Do an InOrder traversal until and print $k$ elements which will be the smallest ones. So, we have the $k$ smallest elements.

The cost of creation of a binary search tree with $n$ elements is $O(nlogn)$ and the traversal upto $k$ elements is $O(k)$. Hence the complexity is $O(nlogn + k) = O(nlogn)$.

**Disadvantage**: If the numbers are sorted in descending order, we will be getting a tree which will be skewed towards left. In that case, construction of the tree will be $0 + 1 + 2 + \ldots + (n - 1) = \frac{n(n-1)}{2}$ which is $O(n^2)$. To escape from this, we can keep the tree balanced, so that the cost of constructing the tree will be only $nlogn$.

**Problem-9** Can we improve *tree sorting* technique for solving Problem-6?

**Solution: Yes.** Use a smaller tree to give the same result.

1. Take the first $k$ elements of the sequence to create a balanced tree of $k$ nodes (this will cost $klogk$).
2. Take the remaining numbers one by one, and
   a. If the number is larger than the largest element of the tree, return
   b. If the number is smaller than the largest element of the tree, remove the largest element of the tree and add the new element. This step is to make sure that a smaller element replaces a larger element from the tree. And of course the cost of this operation is $logk$ since the tree is a balanced tree of $k$ elements.

Once the step 2 is over, the balanced tree with $k$ elements will be having the smallest $k$ elements. The only remaining task is to print out the largest element of the tree.

Time Complexity:

1. For the first $k$ elements, we make the tree. Hence the cost is $klogk$.
2. For the rest $n - k$ elements, the complexity is of $O(logk)$.

Step 2 has a complexity of $(n - k)\ logk$. The total cost is $klogk + (n - k)\ logk = nlogk$ which is $O(nlogk)$. This bound is actually better than the ones provided earlier.

**Problem-10** Can we use partitioning technique for solving Problem-6?

**Solution: Yes.**

**Algorithm**

1. Choose a pivot from the array.
2. Partition the array so that: $A[low \ldots pivotpoint - 1] <= pivotpoint <= A[pivotpoint + 1..high]$.
3. if $k < pivotpoint$ then it must be on the left of pivot, so do the same method recursively on the left part.
4. if $k = pivotpoint$ then it must be the pivot and print all the elements from *low* to *pivotpoint*.
5. if $k > pivotpoint$ then it must be on the right of pivot, so do the same method recursively on the right part.

The top-level call would be kthSmallest = Selection(1, n, k).

```
int Selection (int low, int high, int k) {
    int pivotpoint;
    if(low == high)
            return S[low];
```





```
    else {  pivotpoint = Partition(low, high);
            if(k == pivotpoint)
                    return S[pivotpoint];          //we can print all the elements from low to pivotpoint.
            else if(k < pivotpoint)
                    return Selection (low, pivotpoint - 1, k);
            else    return Selection (pivotpoint + 1, high, k);
    }
}
void Partition (int low, int high) {
    int i, j, pivotitem;
    pivotitem = S[low];
    j = low;
    for (i = low + 1; i <= high; i++)
            if(S[i] < pivotitem) {
                    j++;
                    Swap S[i] and S[j];
            }
    pivotpoint = j;
    Swap S[low] and S[pivotpoint];
    return pivotpoint;
}
```

Time Complexity: $O(n^2)$ in worst case as similar to Quicksort. Although the worst case is the same as that of Quicksort, this performs much better on the average [$O(nlogk)$ – Average case].

**Problem-11** Find the $k^{th}$-smallest element in an array $S$ of $n$ elements in best possible way.

**Solution:** This problem is similar to Problem-6 and all the solutions discussed for Problem-6 are valid for this problem. The only difference is that instead of printing all the $k$ elements we print only the $k^{th}$element. We can improve the solution by using *median of medians* algorithm. Median is a special case of the selection algorithm. The algorithm Selection(A, k) to find the $k^{th}$ smallest element from the set $A$ of $n$ elements is as follows:

**Algorithm:** *Selection*(A, k)

1. Partition $A$ into $ceil\left(\frac{length(A)}{5}\right)$ groups, each group has five items (last group may have fewer items).
2. Sort each group separately (e.g., insertion sort).
3. Find the median of each of the $\frac{n}{5}$ groups and store them in some array (let us say $A'$).
4. Use *Selection*recursively to find the median of $A'$ (median of medians). Let us asay the median of medians is $m$.

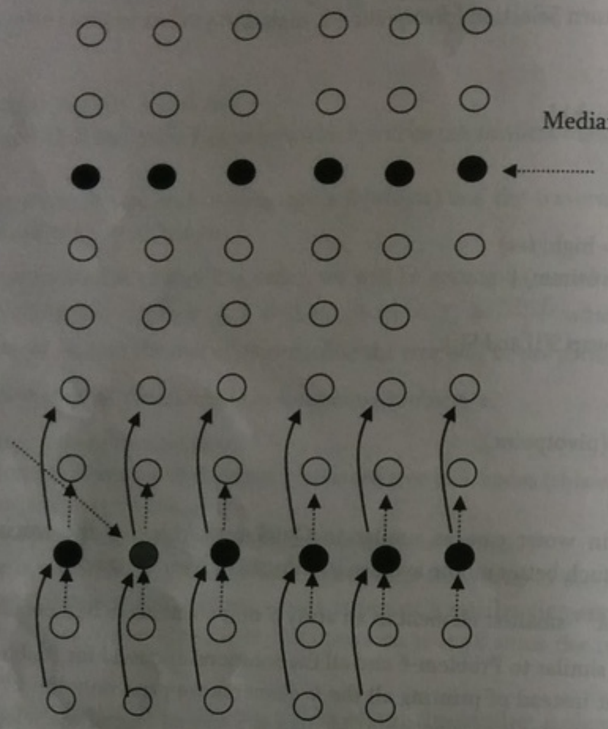
$$m = Selection(A', \frac{\frac{length(A)}{5}}{2});$$

5. Let $q$ = # elements of $A$ smaller than $m$;
6. If($k == q + 1$)
   return $m$;
   /* Partition with pivot */
7. Else partition $A$ into $X$ and $Y$
   - $X$ = {items smaller than $m$}
   - $Y$ = {items larger than $m$}

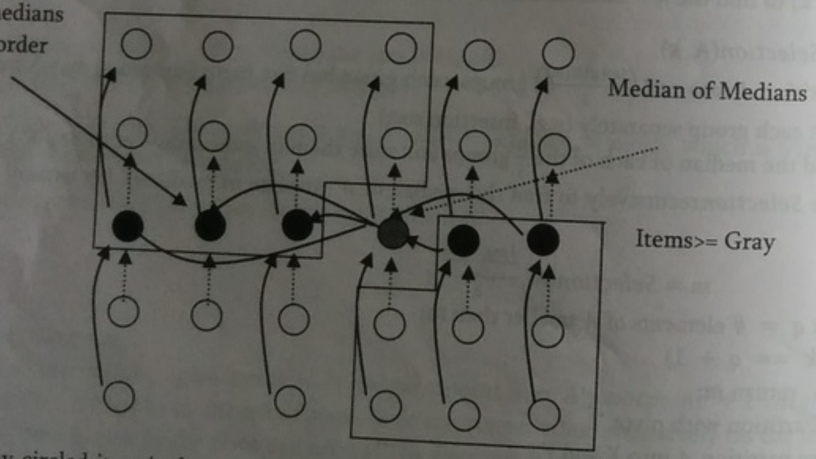
   /* Next,form a subproblem */
8. If($k < q + 1$)
   return Selection(X, k);
9. Else

return Selection(Y, k – (q+1));

Before developing recurrence, let us consider the below representation of the input. In the figure each circle is an element and each column is grouped with 5 elements. The black circles indicate the median in each group of 5 elements. As discussed, sort each column using constant time insertion sort.




After sorting rearrange the medians so that all medians will be in ascending order




In the above figure the gray circled item is the median of medians (let us call this as $m$). It can be seen that, at least 1/2 of 5 element group medians $\leq m$. Also, these 1/2 of 5 element groups contribute 3 elements that are $\leq m$ except 2 groups [last group which may contain fewer than 5 elements and other group which contains $m$]. Similarly, at least 1/2 of 5 element groups contribute 3 elements that are $\geq m$ as shown above. 1/2 of 5 element groups contribute 3 elements except 2 groups gives: $3(\lceil\frac{1}{2}\lceil\frac{n}{5}\rceil\rceil-2) \approx \frac{3n}{10} - 6$. The remaining are $n - \frac{3n}{10} - 6 \approx \frac{7n}{10} + 6$. Since $\frac{7n}{10} + 6$ is greater than $\frac{3n}{10} - 6$ we need to consider $\frac{7n}{10} + 6$ for worst.





**Components in recurrence:**

- In our selection algorithm, we chose $m$, which is the median of medians, to be a pivot and partition A into two sets $X$ and $Y$. We need to select the set which gives maximum size (to get the worst case).
- The time in function *Selection* when called from procedure *partition*. The number of keys in the input to this call to *Selection* is $\frac{n}{5}$.
- The number of comparisons required to partition the array. This number is $length(S)$, let us say $n$.

We have established the following recurrence: $T(n) = T\left(\frac{n}{5}\right) + \Theta(n) + Max\{T(X), T(Y)\}$

From the above discussion we have seen that, if we select median of medians m as pivot, the partition sizes are: $\frac{3n}{10} - 6$ and $\frac{7n}{10} + 6$. If we select the maximum of these, then we get:

$$\begin{aligned} T(n) &= T\left(\frac{n}{5}\right) + \Theta(n) + T\left(\frac{7n}{10} + 6\right) \\ &\approx T\left(\frac{n}{5}\right) + \Theta(n) + T\left(\frac{7n}{10}\right) + O(1) \\ &\leq c\frac{7n}{10} + c\frac{n}{5} + \Theta(n) + O(1) \end{aligned}$$

Finally, $T(n) = \Theta(n)$.

**Problem-12** In Problem-11, we divided the input array into groups of 5 elements. The constant 5 plays an important part in the analysis. Can we divide in groups of 3 which work in linear time?

**Solution:** In this case the modification causes the routine to take more than linear time. In the worst case, at least half the $\lceil\frac{n}{3}\rceil$ medians found in the grouping step are greater than the median of medians $m$, but two of those groups contribute less than two elements larger than $m$. So as an upper bound, the number of elements larger than pivotpoint is at least:

$$2(\lceil\frac{1}{2}\lceil\frac{n}{3}\rceil\rceil - 2) \geq \frac{n}{3} - 4$$

Likewise this is a lower bound. Thus up to $n - (\frac{n}{3} - 4) = \frac{2n}{3} + 4$ elements are fed into the recursive call to *Select*. The recursive step that finds the median of medians runs on a problem of size $\lceil\frac{n}{3}\rceil$, and consequently the time recurrence is:

$$T(n) = T(\lceil n/3 \rceil) + T(2n/3 + 4) + \Theta(n).$$

Assuming that $T(n)$ is monotonically increasing, we may conclude that $T(\frac{2n}{3} + 4) \geq T(\frac{2n}{3}) \geq 2T(\frac{n}{3})$, and so we can upper bound this as $T(n) \geq 3T(\frac{n}{3}) + \Theta(n)$, which is $O(nlogn)$. Therefore, we cannot select 3 as the group size.

**Problem-13** Similar to Problem-12, can we use groups of size 7?

**Solution:** Following a similar reasoning, we once more modify the routine, now to use groups of 7 instead of 5. In the worst case, at least half the $\lceil\frac{n}{7}\rceil$ medians found in the grouping step are greater than the median of medians $m$, but two of those groups contribute less than four elements larger than $m$. So as an upper bound, the number of elements larger than pivotpoint is at least:

$$4(\lceil\frac{1}{2}\lceil\frac{n}{7}\rceil\rceil - 2) \geq \frac{2n}{7} - 8.$$

Likewise this is a lower bound. Thus up to $n - (\frac{2n}{7} - 8) = \frac{5n}{7} + 8$ elements are fed into the recursive call to Select. The recursive step that finds the median of medians runs on a problem of size $\lceil\frac{n}{7}\rceil$, and consequently the time recurrence is

$$T(n) = T(\lceil\frac{n}{7}\rceil) + T(\frac{5n}{7} + 8) + O(n)$$

$$T(n) \leq c\lceil\frac{n}{7}\rceil + c(\frac{5n}{7} + 8) + O(n)$$

$$\leq c\frac{n}{7} + c\frac{5n}{7} + 8c + an, a \text{ is a constant}$$

$$= cn - c\frac{n}{7} + an + 9c$$
$$= (a + c)n - (c\frac{n}{7} - 9c).$$

which is bounded above by $(a + c)n$ provided that $c\frac{n}{7} - 9c \geq 0$. ∴ We can select 7 as the group size.

**Problem-14** Given two arrays each containing $n$ sorted elements, give an $O(logn)$-time algorithm to find the median of all $2n$ elements.

**Solution:** The simple solution to this problem is to merge the two lists and then take the average of the middle two elements (note the union always contains an even number of values). But, the merge would be $\Theta(n)$, so that doesn't satisfy the problem statement.

To get $logn$ complexity, let $medianA$ and $medianB$ be the medians of the respective lists (which can be easily found since both lists are sorted). If $medianA == medianB$, then that's the overall median of the union and we are done. Otherwise, the median of the union must be between $medianA$ and $medianB$. Suppose that $medianA < medianB$ (opposite case is entirely similar). Then we need to find the median of the union of the following two sets:

$$\{x\ in\ A \mid x >= medianA\}\{x\ in\ B \mid x <= medianB\}$$

So, we can do this recursively by resetting the *boundaries* of the two arrays. The algorithm tracks both arrays (which are sorted) using two indices. These indices are used to access and compare the median of both arrays to find where the overall median lies.

```
FindMedian(int A[], int alo , int ahi, int B[], int blo int bhi) {
        amid = alo + (ahi-alo)/2;
        amed = a[amid];
        bmid = blo + (bhi-blo)/2;
        bmed = b[bmid];
        if( ahi - alo + bhi - blo < 4) {
                Handle the boundary cases and solve it smaller problem in O(1) time.
                return;
        }
        else if(amed < bmed)
                FindMedian(A, amid, ahi, B, blo, bmid+1);
        else    FindMedian(A, alo, amid+1,B, bmid+1, bhi);
}
```

Time Complexity: $O(logn)$, since we are reducing the problem size by half every time.

**Problem-15** Let $A$ and $B$ be two sorted arrays of $n$ elements each. We can easily find the $k^{th}$ smallest element in $A$ in $O(1)$ time by just outputting $A[k]$. Similarly, we can easily find the $k^{th}$ smallest element in $B$. Give an $O(logk)$ time algorithm to find the $k^{th}$ smallest element overall { *i.e.*, the $k^{th}$ smallest in the union of $A$ and $B$.

**Solution:** It's just another way of asking Problem-14.

**Problem-16** **Finding the $k$ smallest elements in sorted order:** Given a set of $n$ elements from a totally-ordered domain, suppose we want to find the $k$ smallest elements, and list them in sorted order. For the following approach to this problem, analyze the worst-case running time of the best implementation of the approach.

**Solution:** Sort the numbers, and list the $k$ smallest.
$T(n)$ = Time complexity of sort + listing $k$ smallest elements $= \Theta(nlogn) + \Theta(n) = \Theta(nlogn)$.

**Problem-17** For Problem-16, if we follow the below approach then what is the complexity?

**Solution:** Using the priority queue data structure from heap sort, construct a min-heap over the set, and perform extract-min $k$ times. Refer *Priority Queues (Heaps)* chapter for more details.





**Problem-18** For Problem-16, if we follow the below approach then what is the complexity?
Find the $k^{th}$-smallest element of the set, partition around this pivot element, and sort the $k$ smallest elements.

**Solution:**
$T(n)$ = $Time\ complexity\ of\ kth - smallest + Finding\ pivot + Sorting\ prefix$
$= \Theta(n) + \Theta(n) + \Theta(klogk) = \Theta(n + klogk)$

Since, $k \leq n$, this approach is bettern than Problem-16 and Problem-17.

**Problem-19** Find $k$ nearest neighbors to the median of $n$ distinct numbers in $O(n)$ time.

**Solution:** Let us assume that the array elements are sorted. Now find the median of $n$ numbers and call its index as $X$ (since array is sorted, median will be at $\frac{n}{2}$ location). All we need to do is to select $k$ elements with the smallest absolute differences from the median moving from $X - 1$ to $0$ and $X + 1$ to $n - 1$ when the median is at index $m$.

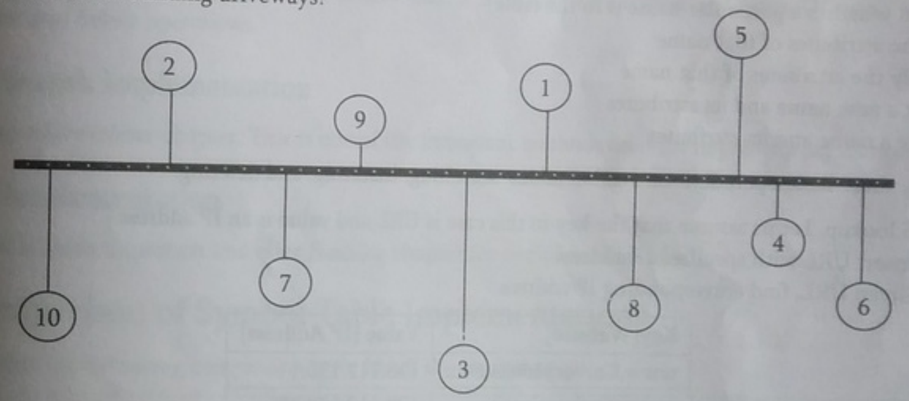
Time Complexity: Each step takes $\Theta(n)$. So that total time complexity of the algorithm is $\Theta(n)$.

**Problem-20** Is there any other way of solving the Problem-19?

**Solution:** Assume for simplicity that $n$ is odd and $k$ is even. If the set $A$ was in sorted order, the median is in position $n/2$ and the $k$ numbers in $A$ that closest to the median are in positions $(n - k)/2$ through $(n + k)/2$.

We first use linear time selection to find the $(n - k)/2, n/2$, and $(n + k)/2$ elements and then pass through the set A to find the numbers less than $(n + k)/2$ element, greater than the $(n - k)/2$ element, and not equal to the $n/2$ element. The algorithm takes $O(n)$ time as we use linear time selection exactly three times and traverse the $n$ numbers in $A$ once.

**Problem-21** Given $(x, y)$ coordinates of $n$ houses, where should you build road parallel to $x$-axis to minimize construction cost of building driveways?




**Solution:** The road costs nothing to build. It is the driveways that cost money. Driveway cost is proportional to its distance to road. Obviously, they will be perpendicular. Solution is to put street at median of $y$ coordinates.

**Problem-22** Given a big file containing billions of numbers. Find maximum 10 numbers from those file.

**Solution:** Refer *Priority Queues* chapter.

**Problem-23** Suppose there is a milk company. Daily they collect mail from all their agents. The agents are located at different places. To collect the milk, what is the best place to start so that the total distance travelled is minimum?

**Solution:** Starting at median reduces total distance travelled because it is the place which is center to all remaining places.

$$= cn - c\frac{n}{7} + an + 9c$$
$$= (a + c)n - (c\frac{n}{7} - 9c).$$

which is bounded above by $(a + c)n$ provided that $c\frac{n}{7} - 9c \geq 0$. ∴ We can select 7 as the group size.

**Problem-14** Given two arrays each containing $n$ sorted elements, give an $O(logn)$-time algorithm to find the median of all $2n$ elements.

**Solution:** The simple solution to this problem is to merge the two lists and then take the average of the middle two elements (note the union always contains an even number of values). But, the merge would be $\Theta(n)$, so that doesn't satisfy the problem statement.

To get $logn$ complexity, let $medianA$ and $medianB$ be the medians of the respective lists (which can be easily found since both lists are sorted). If $medianA == medianB$, then that's the overall median of the union and we are done. Otherwise, the median of the union must be between $medianA$ and $medianB$. Suppose that $medianA < medianB$ (opposite case is entirely similar). Then we need to find the median of the union of the following two sets:

$$\{x\ in\ A \mid x >= medianA\}\{x\ in\ B \mid x <= medianB\}$$

So, we can do this recursively by resetting the *boundaries* of the two arrays. The algorithm tracks both arrays (which are sorted) using two indices. These indices are used to access and compare the median of both arrays to find where the overall median lies.

```
FindMedian(int A[], int alo , int ahi, int B[], int blo int bhi) {
        amid = alo + (ahi-alo)/2;
        amed = a[amid];
        bmid = blo + (bhi-blo)/2;
        bmed = b[bmid];
        if( ahi - alo + bhi - blo < 4) {
                Handle the boundary cases and solve it smaller problem in O(1) time.
                return;
        }
        else if(amed < bmed)
                FindMedian(A, amid, ahi, B, blo, bmid+1);
        else    FindMedian(A, alo, amid+1,B, bmid+1, bhi);
}
```

Time Complexity: $O(logn)$, since we are reducing the problem size by half every time.

**Problem-15** Let $A$ and $B$ be two sorted arrays of $n$ elements each. We can easily find the $k^{th}$ smallest element in $A$ in $O(1)$ time by just outputting $A[k]$. Similarly, we can easily find the $k^{th}$ smallest element in $B$. Give an $O(logk)$ time algorithm to find the $k^{th}$ smallest element overall [ *i.e.*, the $k^{th}$ smallest in the union of $A$ and $B$.

**Solution:** It's just another way of asking Problem-14.

**Problem-16** **Finding the $k$ smallest elements in sorted order:** Given a set of $n$ elements from a totally-ordered domain, suppose we want to find the $k$ smallest elements, and list them in sorted order. For the following approach to this problem, analyze the worst-case running time of the best implementation of the approach.

**Solution:** Sort the numbers, and list the $k$ smallest.
$T(n)$ = Time complexity of sort + listing $k$ smallest elements $= \Theta(nlogn) + \Theta(n) = \Theta(nlogn)$.

**Problem-17** For Problem-16, if we follow the below approach then what is the complexity?

**Solution:** Using the priority queue data structure from heap sort, construct a min-heap over the set, and perform extract-min $k$ times. Refer *Priority Queues (Heaps)* chapter for more details.





**Problem-18** For Problem-16, if we follow the below approach then what is the complexity?
Find the $k^{th}$-smallest element of the set, partition around this pivot element, and sort the $k$ smallest elements.

**Solution:**
$T(n)$ = *Time complexity of kth − smallest + Finding pivot + Sorting prefix*
$= \Theta(n) + \Theta(n) + \Theta(klogk) = \Theta(n + klogk)$

Since, $k \leq n$, this approach is bettern than Problem-16 and Problem-17.

**Problem-19** Find $k$ nearest neighbors to the median of $n$ distinct numbers in $O(n)$ time.

**Solution:** Let us assume that the array elements are sorted. Now find the median of $n$ numbers and call its index as $X$ (since array is sorted, median will be at $\frac{n}{2}$ location). All we need to do is to select $k$ elements with the smallest absolute differences from the median moving from $X - 1$ to $0$ and $X + 1$ to $n - 1$ when the median is at index $m$.

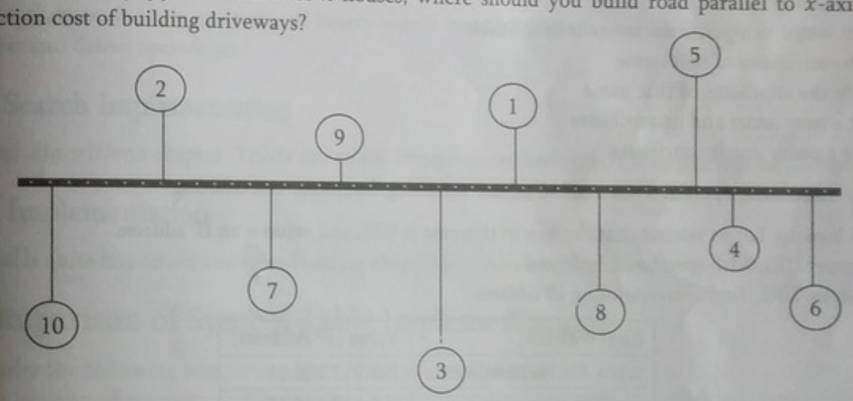
Time Complexity: Each step takes $\Theta(n)$. So that total time complexity of the algorithm is $\Theta(n)$.

**Problem-20** Is there any other way of solving the Problem-19?

**Solution:** Assume for simplicity that $n$ is odd and $k$ is even. If the set $A$ was in sorted order, the median is in position $n/2$ and the $k$ numbers in $A$ that closest to the median are in positions $(n - k)/2$ through $(n + k)/2$.

We first use linear time selection to find the $(n - k)/2, n/2$, and $(n + k)/2$ elements and then pass through the set A to find the numbers less than $(n + k)/2$ element, greater than the $(n - k)/2$ element, and not equal to the $n/2$ element. The algorithm takes $O(n)$ time as we use linear time selection exactly three times and traverse the $n$ numbers in $A$ once.

**Problem-21** Given $(x, y)$ coordinates of $n$ houses, where should you build road parallel to $x$-axis to minimize construction cost of building driveways?




**Solution:** The road costs nothing to build. It is the driveways that cost money. Driveway cost is proportional to its distance to road. Obviously, they will be perpendicular. Solution is to put street at median of $y$ coordinates.

**Problem-22** Given a big file containing billions of numbers. Find maximum 10 numbers from those file.

**Solution:** Refer *Priority Queues* chapter.

**Problem-23** Suppose there is a milk company. Daily they collect mail from all their agents. The agents are located at different places. To collect the milk, what is the best place to start so that the total distance travelled is minimum?

**Solution:** Starting at median reduces total distance travelled because it is the place which is center to all remaining places.

# Chapter-13

# SYMBOL TABLES

## 13.1 Introduction

Since childhood, we all have used a dictionary, and many of us have a word processor (say, Microsoft Word) which comes with spell checker. The spell checker is also a dictionary but limited. There are many real time examples for dictionaries and few of them are:

- Spelling checker
- The data dictionary found in database management applications
- Symbol tables generated by loaders, assemblers, and compilers
- Routing tables in networking components (DNS lookup)

In computer science, we generally use the term symbol table rather than dictionary, when referring to the ADT.

## 13.2 What are Symbol Tables?

From the above example, we can define the *symbol table* as a data structure that associates a *value* with a *key*. It supports the following operations:

- Search whether a particular name is in the table
- Get the attributes of that name
- Modify the attributes of that name
- Insert a new name and its attributes
- Delete a name and its attributes

There are only three basic operations on symbol tables: searching, inserting, and deleting.

**Example:** DNS lookup. Let us assume that the key in this case is URL and value is an IP address.

- Insert URL with specified IP address
- Given URL, find corresponding IP address

| Key[Website] | Value [IP Address] |
|---|---|
| www.CareerMonks.com | 128.112.136.11 |
| www.AuthorsInn.com | 128.112.128.15 |
| www.AuthInn.com | 130.132.143.21 |
| www.klm.com | 128.103.060.55 |
| www.CareerMonk.com | 209.052.165.60 |

## 13.3 Symbol Table Implementations

Before implementing symbol tables, let us enumerate the possible implementations. Symbol tables can be implemented in many ways and below are some of them.

### Unordered Array Implementation

With this method, just maintaining an array is enough. It needs O($n$) time for searching, insertion and deletion in the worst case.





### Ordered [Sorted] Array Implementation

In this we maintain a sorted array of keys and values.

- Store in sorted order by key
- keys[i] = $i^{th}$ largest key
- values[i] = value associated with $i^{th}$ largest key

Since the elements are sorted and stored in arrays, we can use simple binary search for finding an element. It takes O($logn$) time for searching and O($n$) time for insertion and deletion in the worst case.

### Unordered Linked List Implementation

Just maintaining a linked list with two data values is enough for this method. It needs O($n$) time for searching, insertion and deletion in the worst case.

### Ordered Linked List Implementation

In this method, while inserting the keys, maintain the order of keys in the linked list. Even if the list is sorted, in the worst case it needs O($n$) time for searching, insertion and deletion.

### Binary Search Trees Implementation

Refer *Trees* chapter. Advantages of this method are it does not need much code and fast search [O($logn$) on average].

### Balanced Binary Search Trees Implementation

Refer *Trees* chapter. It is an extension of binary search trees implementation and takes O($logn$) in worst case for search, insert and delete operations.

### Ternary Search Implementation

Refer *String Algorithms* chapter. This is one of the important methods used for implementing dictionaries.

### Hashing Implementation

This method is quite important and refer Hashing chapter for complete discussion.

## 13.4 Comparison of Symbol Table Implementations

Let us consider the following comparison table for all the implementations.

| Implementation | Search | Insert | Delete |
|---|---|---|---|
| Unordered Array | $n$ | $n$ | $n$ |
| Ordered Array (can be implemented with array binary search) | $logn$ | $n$ | $n$ |
| Unordered List | $n$ | $n$ | $n$ |
| Ordered List | $n$ | $n$ | $n$ |
| Binary Search Trees (O($logn$) on average) | $logn$ | $logn$ | $logn$ |
| Balanced Binary Search Trees (O($logn$) in worst case) | $logn$ | $logn$ | $logn$ |
| Ternary Search (only change is in logarithms base) | $logn$ | $logn$ | $logn$ |
| Hashing (O(1) on average) | 1 | 1 | 1 |

**Notes:**

- In the above table, $n$ is the input size.
- Table indicates the possible implementations discussed in this book. But, there could be other implementations.

# Chapter-14

# HASHING

## 14.1 What is Hashing?

Hashing is a technique used for storing and retrieving information as fast as possible. They are used in performing optimal search and is a good technique for implementing symbol tables.

## 14.2 Why Hashing?

In *Trees* chapter we have seen that balanced binary search trees support operations such as *insert*, *delete* and *search* in $O(logn)$ time. In applications if we need these operations in O(1), then hashing provides a way. Remember that worst cast complexity of hashing is still $O(n)$, but it gives O(1) on the average.

## 14.3 HashTable ADT

The common operations on hash table are:

- CreatHashTable: Creates a new hash table
- HashSearch: Searches the key in hash table
- HashInsert: Inserts a new key into hash table
- HashDelete: Deletes a key from hash table
- DeleteHashTable: Deletes the hash table

## 14.4 Understanding Hashing

In simple terms we can treat *array* as a hash table. For understanding the use of hash tables, let us consider the following example: Give an algorithm for printing the first repeated character if there are duplicated elements in it. Before proceeding for the solution, let us think about the possible solutions. The simple and brute force way of solving is: given a string, for each character check whether that character is repeated or not. Time complexity of this approach is $O(n^2)$ with O(1) space complexity.

Now, let us find the better solution for this problem. Since our objective is to find the first repeated character, what if we remember the previous characters in some array?

We know that the number of possible characters are 256 (for simplicity assume *ASCII* characters only). Create an array of size 256 and initialize it with all zeros. For each of the input characters go to the corresponding position and increment its count. Since we are using arrays, it takes constant time for reaching any location. While scanning the input, if we get a character whose counter is already 1 then we can say that the character is the one which is repeating first time.

```
char FirstRepeatedChar ( char *str ) {
        int i, len=strlen(str);
        int count[256]; //additional array
        for(i=0; i<256; ++i)
                count[i] = 0;
        for(i=0; i<len; ++i) {
```

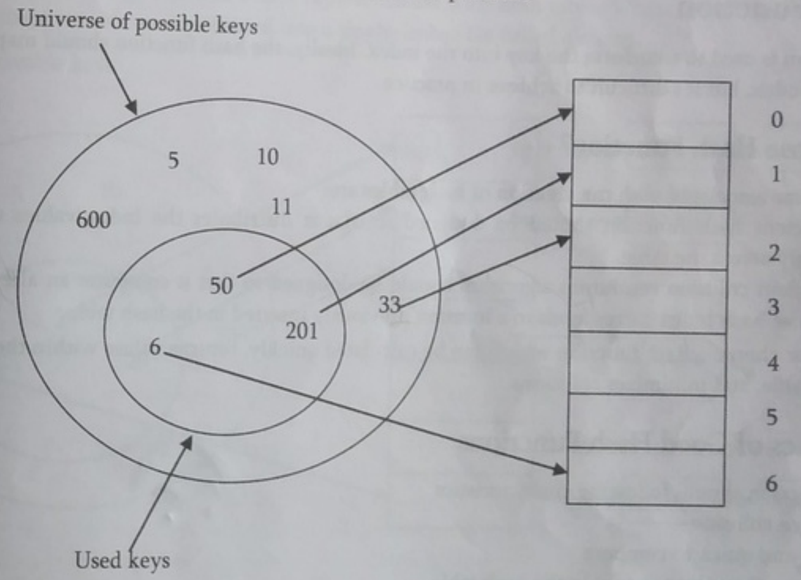




```
                if(count[str[i]]==1) {
                        printf("%c", str[i]);
                        break;
                }
                else    count[str[i]]++;
        }
        if(i==len)
                printf("No Repeated Characters");
        return 0;
}
```

### Why not Arrays?

In the previous problem, we have used an array of size 256 because we know that the number of different possible characters [256] in advance. Now, let us consider the slightly variant of the same problem. Suppose if the given array is having numbers instead of characters then how do we solve the problem?




In this case the set of possible values are infinity (or at least very big). Creating a huge array and storing the counters is practically not possible. That means there are a set of universe of keys and limited memory locations in the memory. If we want to solve this problem somehow we need to map all these possible keys to the possible memory locations.

From the above discussion and diagram it can be seen that, we need a mapping of possible keys to one of the available locations. As a result using simple arrays is not the correct choice for solving the problems whose possible keys are very big. The process of mapping the keys to locations is called *hashing*.

**Note:** For now, do not worry about how the keys are mapped to locations. That depends on the function used for conversions. One such simple function is *key % table size*.

## 14.5 Components in Hashing

There are 4 key components involved in hashing:

1) Hash Table
2) Hash Functions
3) Collisions