ASSIGNMENT 2 (RITIKA GOYAL) (301401516)

Ques1:

(a)

Input: unsorted array arr[], size of array n, element to be found in array x

Output: index of the element if found in array or -1 otherwise

PseudoCode:

```
RandomSearch(arr, n, x)

  total = 0
  make array named temp_arr of size n and initialise its all elements with -1
  while(total<n)
        bool check = false
        randomIndex = random number from 0 to n-1
        for(i =0;i< total; i++ )
                if randomIndex == temp_array[i] then check = true
        end for
        if check not equal to true then
                if arr[randomIndex] == x then return random index
                end if
                else  temp_array[total] = randomIndex and total++
        end if
  end while
return -1
```

(b)

The probability that the selected random index is equal to x is 1/n.

Since there can be at most n-1 wrong indices and every index picked in iteration j:

The expected value E[n] = $\sum_{i\geq 1} i\ (n-1/n)$ ^i-1 (1/n)

= (1/n) * (1/(1- n-1/n)^2)

= n.

Ques2:

Min-Heapify

Input: Array A of heap size n(0 … n-1 indexes) and index i s.t. binary trees rooted at Left(i) and Right(i) are min-heaps

Output: a min heap rooted at A[i]

```
Min-Heapify(A,i)
        l = Left(i); r = Right(i);
        if l < n and A[l] < A[i] then min =l else min = i;
        if r < n and A[r] < A[min] then min =r;
        if min not equal to i then exchange A[i] with A[min]
          and Min-Heapify(A,min)
```

Heap-Minimum

Input: Array A which is min-heap of size n (0 …. n-1 indexes)

Output: return the minimum element

```
Heap-Minimum(A)
        Return A[0];
```

Heap-Extract-Min

Input: Array A which is min-heap of size n (0 …. n-1 indexes)

Output: returns and removes the minimum element in min-heap

```
Heap-Extract-Min(A)
        if n < 1 then output "error, heap underflow" and stop;
        min = A[0]
        A[0]= A[n-1]
        n = n-1
        Min-Heapify(A,0);
        Return min
```

Heap-Decrease-Key

Input: Array A which is min-heap of size n (0 …. n-1 indexes), index i, key

Output: decreases the value of element at index I and makes it equalt to key and maintain min-heap property.

```
Heap-Decrease-Key(A,i,key)

        if(key > A[i]) the output "error, new key > current key" and stop
        A[i]= key;
        While I > 0 and A[Parent(i)] > A[i] do
                exchange A[i] with A[Parent(i)]; i = Parent(i);
```

Min-Heap-Insert

Input: Array A which is min-heap of size n (0 …. n-1 indexes), key

Output: Insert a new element to min-heap and then maintain the min-heap property.

Min-Heap-Insert(A, key)

```
    n = n+1;
    A[n-1] = +∞;
    Heap-Decrease-Key(A,n-1,key)
```

Ques3:

a) To present a d-ary heap in an array (array has index 0,1, ….., n-1), it will be sufficient to show the index of parent and its child of any node.
The node at index i:
Parent(i) = floor((i-1)/d)
Child(j,i) = di+j , where j is the j^{th} child of node at i.

b) The height of a d-ary heap of n elements in terms if n and d is $O(\log_d n)$.

c) EXTRACT-MAX
Pseudo-code:
Input: d-ary A (0 …. N-1), n is size of array, d number of child of each node
Output: returns and removes the maximum element from d-ary and maintain heap property.

```
EXTRACT-MAX(A,n,d)
max = A[0] ;
A[0] = A[n-1];
n = n-1;
HEAPIFY (A,0,n-1,d);
return max;

HEAPIFY(A,i,n,d)
j = i;
for k =0 to d-1
    if  CHILD(k,i) ≤ n and A[CHILD(k,i)> A[j] then j = CHILD(k,i)
if j is not equal to i then swap A[i] <-> A[j] and HEAPIFY (A,j,n,d)
```

Running time of algorithm is constant plus work done by HEAPIFY. In HEAPIFY, the i^{th} node and each of its children are compare to fond the maximum value for all the nodes and each depth, d loops are done, so running time of HEAPIFY is $O(d \log_d n)$.

Running time = $O(1) + O(d \log_d n)$
$= O(d \log_d n)$

d) INSERT

<u>Pseudo-code:</u>

```
INSERT(A,key,n)
n = n+1;
A[n] = key;
for i = n to i>1 and A[Parent(i)] < A[i]
        exchange A[i] with A[PARENT(i)] and i = PARENT(i);
```

Running time: After inserting the element at last, the INSERT maintains the heap property by looping at each step and comparing the increased node to its parent and swap to maintain heap property. So the running time of INSERT is $O(\log_d n)$.

e) INCREASE-KEY

<u>Pseudo-code:</u>

```
INCREASE-KEY(A,key,i)
if key < A[i] then error "new key is smaller than current key"
  else   A[i] = key
while i>1 and A[PARET(i)] < A[i]
      exchange A[i] with A[PARENT(i)] and i = PARENT(i);
```

Running Time: After increasing the value at index i, INCREASE-KEY maintains the hap property by looping at each step and comparing the increased node to its parent and swap to maintain heap property. The insertion of key takes constant time. So the running time of INCREASE-KEY is $O(\log_d n)$.

Ques4:

In the original analysis of randomized quick sort, there are log(n) levels to the recursion tree and then the total steps will be log(n/k) which makes the expected running time O(n log(n/k)). Since the quick sort is called on entire array. The insertion sort will take the shifting of at most k elements giving O(nk) as when top level of quick sort is returned, the insertion sort is called on entire array this gives the running time of O(nk + n log(n/k)).

The k should be selected so that the running time of the algorithm reduces and is minimum.

$n \log n \geq nk + n \log n/k$

$\log n \geq k + \log n - \log k$

$\log k \geq k$ (not possible)

adding the constant factors to above equation. Let the added constants be c and d.

$c\, n \log n \geq d\, nk + c\, n \log(n/k)$

$c \log n \geq dk + c \log n - c \log k$

$\log k \geq (d/c)\, k$

In practice, the larger values of n should be tried with different values of k. I have chosen 10 in the code given below.

<u>Code in C++:</u>

```
#include <iostream>
```

```
#include <ctime>
#include <cmath>
using namespace std;
//Randomized quick sort algo discussed in class
void Swap(int &a, int &b) {
  int temp = a;
  a = b;
  b = temp;
}
int Partition(int arr[], int left, int right)
{
        int pivot = arr[right];
        int i = left-1;
        for(int j = left ;j<right;j++)
        {
                if(arr[j]<= pivot)
                {
                        i = i+1;
                        Swap(arr[i],arr[j]);
                }
        }
        Swap(arr[i+1],arr[right]);

        return i+1;
}
int Randomized_Partition(int arr[],int left,int right)
{
        srand(time(NULL));
        int randomIndex = rand()%(right-left+1) + left;

        Swap(arr[right],arr[randomIndex]);

        return Partition(arr,left,right);
}
void Randomized_Quicksort(int arr[], int left, int right)
{
        int partition;
        if(left < right)
        {
                partition = Randomized_Partition(arr,left,right);
                Randomized_Quicksort(arr, left, partition -1);
                Randomized_Quicksort(arr, partition+1, right);
        }
}
```

```
//Randomized quick sort and insertion sort

void insertion_sort(int arr[] , int left, int right)
{
        int key ;
        for(int i= left +1; i <right; i++)
        {
                key = arr[i];
                int j;
                for( j = i-1; j>=0 && key < arr[j];j--)
                {
                        arr[j+1] = arr[j];
                }
                arr[j+1]=  key;
        }
}

void Randomized_Quicksort_InsertionSort(int arr[],int left, int right,int k)
{
        if(left>=right)
                return;
        int partition;
        if(right-left > k)
        {
                partition = Randomized_Partition(arr,left,right);
                Randomized_Quicksort(arr, left, partition -1);
                Randomized_Quicksort(arr, partition+1, right);
        }
        else
        {
                insertion_sort(arr,left,right+1);
        }
}



int main()
{
        srand(time(NULL));
        int n_values[] = {0,1,2,3,4,5};

        for(int i =0 ; i< 6 ; i++)
        {
```

```
                    int size = pow(2,n_values[i]) * 1000;
                    int* arr1 = new int[size];
                    int* arr2 = new int[size];

                    for (int j =0; j<size; j++)
                    {
                              int temp =  rand()%200 + (-100);
                              arr1[j] = temp;
                              arr2[j] = temp;
                    }

                    cout<<"FOR N = "<<size<<" : "<<endl;

                    clock_t random_quick = clock();
                    Randomized_Quicksort(arr1,0,size-1);
                    random_quick = clock() - random_quick;
                    cout<<"TIME FOR RANDOMIZED QUICK SORT = "<<(float) random_quick /
CLOCKS_PER_SEC<<" seconds"<<endl;

                    int k =10;

                    clock_t quick_insertion = clock();
                    Randomized_Quicksort_InsertionSort(arr2,0,size-1,k);
                    quick_insertion = clock() - quick_insertion;
                    cout<<"TIME FOR RANDOMIZED QUICK SORT-INSERTION SORT = "<<(float)
quick_insertion / CLOCKS_PER_SEC<<" seconds"<<endl;

                    cout<<endl;
                    delete [] arr1;
                    delete [] arr2;
          }


          return 0;
}
```

(recorded observations, time in seconds)

N	1000	2000	4000	8000	16000	32000
Quick-sort	0.002649 s	0.006022 s	0.007907 s	0.012855 s	0.051382 s	0.090624 s
Quick-Insertion sort	0.002599 s	0.004648 s	0.006653 s	0.012414 s	0.028375 s	0.065489 s

Ques5:

Stable: insertion sort, merge sort

Unstable: heap-sort, quicksort

Any algorithm can be made stable by mapping the array to an array of pairs, where the first element in each pair is the original element and the second is its index.

Example array [9,8,7,6,5,4] becomes [(9,0), (8,1), (7,2), (6,3), (5,4), (4,5)] and then it is made sure that the ordered pair (A[i], i) < (A[j], j) if A[i]<A[j] or A[i] =A[j] and i<j. The array is sorted in lexicographic order and the algorithm is guaranteed to be stable. The running time will be asymptotically unchanged, but it takes additional $\Theta(n)$ space and hence this doubles the space requirement.

Ques6:

a) The algorithm that runs in O(n) time and is stable:

<u>Pseudo-code:</u>

```
Sort(A,n)
Create and initialise the array C of length n (0 … n-1) with all elements equal to -1
k =0;
for i = 0 to n-1
        if A[i] ==  0 then C[k] = A[i] and k++;
for i = 0 to n-1
        if A[i] == 1 then C[k] = A[i] and k++;
for i=0 to n-1
        A[i] = C[i]
```

In the above algorithm, first a temporary array is made and it is sorted by maintaining the property of stable sort and then every element of the temporary array is copied into original array. Since there are 3 for loops and it only make 3 passes through the array, its running time is O(n).

b) The algorithm that runs in O(n) and is in place.

<u>Pseudo-code:</u>

```
Sort(A,n)
j =0;
for i = 0 to n-1
        if A[i] == 0 then exchange A[i] with A[j] and j++;
```

In the above algorithm, only 1 pass is made through array and thus it has linear time complexity that is O(n) and it does not require any extra space complexity, thus it is in place.

c) The algorithm that is stable and is in place:

Pseudo-code:

```
Sort(A,n)
for i =0 to n-1
        for j =0 to n-i-1
                if A[j] > A[j+1] then exchange A[j] with A[j+1]
```

In the above algorithm, the space complexity remains the same as no additional space is required and it maintains the property of stable algorithm. This is a bubble sort algorithm.

Ques7:

a) Let the number of x_i smaller than x_k be m. Since the wight of each x_i is same that is 1/n, we have: Weight of first m elements less than x_k = $\sum_{xi<xk} w\, i = m/n$

Weight of n-m-1 elements greater than x_k = $\sum_{xi>xk} wi = (n-m)/n$

The only value of m that satisfies the condition of weighted median for m/n <1/2 and (n-m-1)/2 ≤ ½ is when m = ceiling(n/2) - 1 which is equal to median as it has equal number of x_i's which are larger and smaller than it.

b) The elements are first sorted in increasing order of their weights. Then the scan through the array is made and wights are added are stored in a variable, the first x_i whose cumulative weight is more than ½ is the weighted median. For sorting, merge sort or heap sort can be used as there worst case running time is n log n and then the scanning is done to at most n elements which gives the worst case to be : O(n log n) + O(n)

= O(n log n)

Pseudo-code:

```
Weighed-Median(A)
SORT(A) /* sort A using heap sort or merge sort*/
total_weight = 0;
index = 0;
while total_weight < 1/2
        total_weight = total_weight + $w_{index}$
        if total_weight < ½ then
                index = index+1
 return $x_{index}$
```

c) The weighted median in O(n) first partitions the elements around the actual median of the elements x_k. Then the weight of two halves are calculated, if both are strictly less than ½ then x_k is the weighted median. If not, then the algorithm is called recursively as the weighted median would be in half with total weight exceeding more than ½ and the search continues within the half that weighs more than ½. To get this approach, SELECT is modified.

Pseudo-code:

```
Weighted-Median(A)
if n=1 then return $x_1$
else if n =2 then
        if $w_1 \geq w_2$ then return $x_1$
        else return $x_2$
else
        $x_k$ = Median of X= { $x_1$, $x_2$, ……., $x_n$}
        partition the set around $x_k$
        $W_L = \sum_{xi<xk} wi$
        $W_R = \sum_{xi>xk} wi$
        if $W_L < ½$ and $W_R < ½$
                then return $x_k$
        else if $W_L > ½$
                then $w_k = w_k + W_R$
                        X’ = {x Ɛ X: $x_i \leq x_k$ }
                        return Weighted-Median(X’)
        else
                $w_k = w_k + W_L$
                X’ = { x Ɛ X: $x_i \geq x_k$ }
                return Weighted-Median(X’)
```

Worst case running time of Weighted-Median = O(n/2) + O(n)
= O(n)