Roll no. 123103005

Que 1: Given n nuts and n bolts such that there is one to one mapping between nuts and bolts for each nut find it’s corresponding bolts

Comparision of nuts with nuts and bolts with bolts is not allowed

Ans:

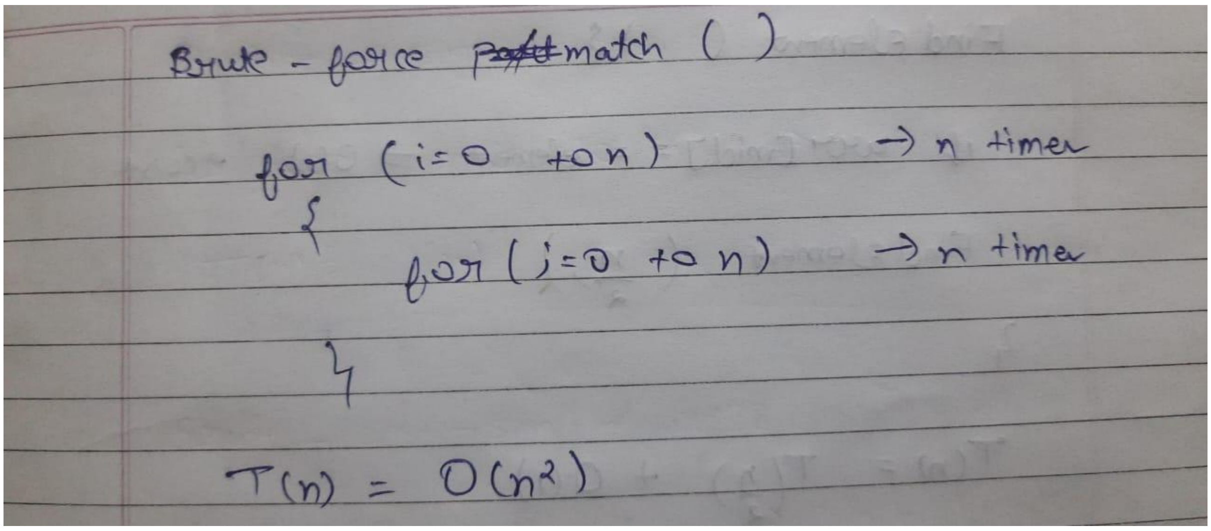
Code:

(i)brute force:

```c
#include <stdio.h>


void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high, int pivot) {
    int i = low;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            swap(&arr[i], &arr[j]);
            i++;
        } else if (arr[j] == pivot) {
            swap(&arr[j], &arr[high]);
            j--;
        }
    }
    swap(&arr[i], &arr[high]);
    return i;
}

void matchPairs(int nuts[], int bolts[], int low, int high) {
    if (low < high) {

        int pivotIndex = partition(nuts, low, high, bolts[high]);


        partition(bolts, low, high, nuts[pivotIndex]);


        matchPairs(nuts, bolts, low, pivotIndex - 1);
        matchPairs(nuts, bolts, pivotIndex + 1, high);
    }
}
```

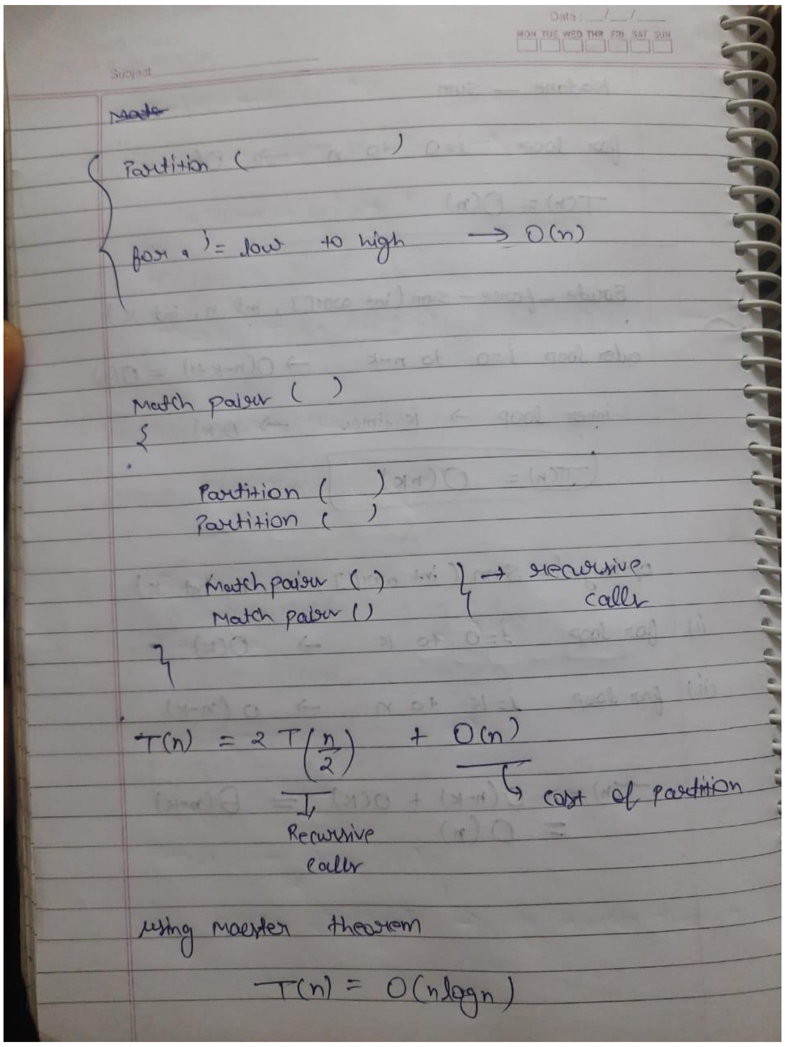
```c
void brute_force_match(int nuts[], int bolts[], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (nuts[i] == bolts[j]) {
                int temp = bolts[i];
                bolts[i] = bolts[j];
                bolts[j] = temp;
                break;
            }
        }
    }
}


void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}


int main() {
    int nuts[] = {4, 2, 5, 1, 3};
    int bolts[] = {2, 4, 3, 5, 1};
    int n = sizeof(nuts) / sizeof(nuts[0]);

    printf("Before matching:\n");
    printf("Nuts:  ");
    printArray(nuts, n);
    printf("Bolts: ");
    printArray(bolts, n);

    matchPairs(nuts, bolts, 0, n - 1);
    // brute_force_match(nuts, bolts,n);

    printf("\nAfter matching:\n");
    printf("Nuts:  ");
    printArray(nuts, n);
    printf("Bolts: ");
    printArray(bolts, n);

    return 0;
}
```

Analysis brute force:



```
Brute - force ~~Pa~~match ( )

    for (i=0 to n)            → n times
    {
        for (j=0 to n)        → n times

    }

T(n) = O(n^2)
```

Analysis Optimized :



~~Made~~

```
{ Partition (          )

  for j = low to high        → O(n)
}

Match pair ( )
{
    Partition (   )
    Partition (   )

    Match pair ( )   }  → recursive
    Match pair ()    }     calls
}
```

$$T(n) = \underbrace{2T\left(\frac{n}{2}\right)}_{\text{Recursive calls}} + \underbrace{O(n)}_{\text{cost of partition}}$$

using master theorem

$$T(n) = O(n\log n)$$

Test cases:

Nuts	Bolts
[4,2,5,1,3]	[1,4,3,5,1]
[1]	[1]
[0,0,0]	[0,0,0]

Que2: gIven an sorted array of n unique integers that have been rotated unknown number of times find the index of element e in the array

Ans:

Code:

```
#include<stdio.h>

int binarysearch(int arr[],int low,int high,int x){
    int mid=(high+low)/2;

    if(arr[mid]==x) return mid;

    else if(x>arr[mid]){
        return binarysearch(arr,mid+1,high,x);
    }
    else if(x<arr[mid]){
        return binarysearch(arr,low,mid-1,x);
    }
}
int findpivot(int arr[],int low,int high){
    if(arr[low]>arr[high] && high-low==1) return high;

    int mid=(low+high)/2;
    if(arr[mid]>arr[low]){
        return findpivot(arr,mid,high);
    }
    else if(arr[mid]<arr[low]){
        return findpivot(arr,low,mid);
    }

}

int findElement(int arr[], int low, int high, int x) {
    if (low > high) return -1;

    int mid = (low + high) / 2;
```

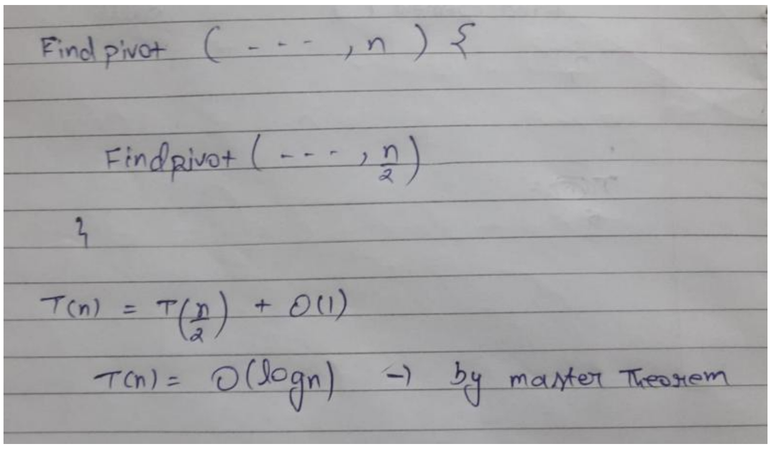
```c
    if (arr[mid] == x) return mid;

    if (arr[low] <= arr[mid]) {
        if (x >= arr[low] && x <= arr[mid]) {
            return findElement(arr, low, mid - 1, x);
        }
        return findElement(arr, mid + 1, high, x);
    }

    if (x >= arr[mid] && x <= arr[high]) {
        return findElement(arr, mid + 1, high, x);
    }

    return findElement(arr, low, mid - 1, x);
}

int brute_search(int arr[],int n,int e){
    for(int i=0;i<n;i++){
        if(arr[i]==e) return i;
    }
}

int main(){
    int arr[]={4,5,6,7,1,2,3};
    int n = sizeof(arr) / sizeof(arr[0]);

    int element;
    printf("enter element to search: ");
    scanf("%d",&element);

    int pivot_index=findpivot(arr,0,n-1);
    //printf("pivot is %d\nindex: %d",arr[pivot_index],pivot_index);

        int index;
        if(element>arr[0]){
            index=binarysearch(arr,0,pivot_index,element);
        }
        else if(arr[0]==element) index=0;
        else index=binarysearch(arr,pivot_index,n-1,element);

        printf("\nindex is %d",index);

}
```

Analysis:

(i) Brute force

T(n)--->O(n)

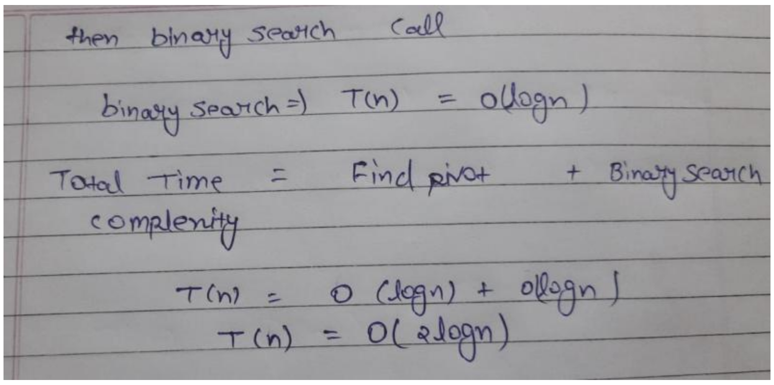
(ii)Optimized : O(2logn)



```
Find pivot ( - - - , n ) {

    Findpivot ( - - - , n/2 )

}
```

$T(n) = T(\frac{n}{2}) + O(1)$

$T(n) = O(\log n)$ → by master Theorem



then binary search call

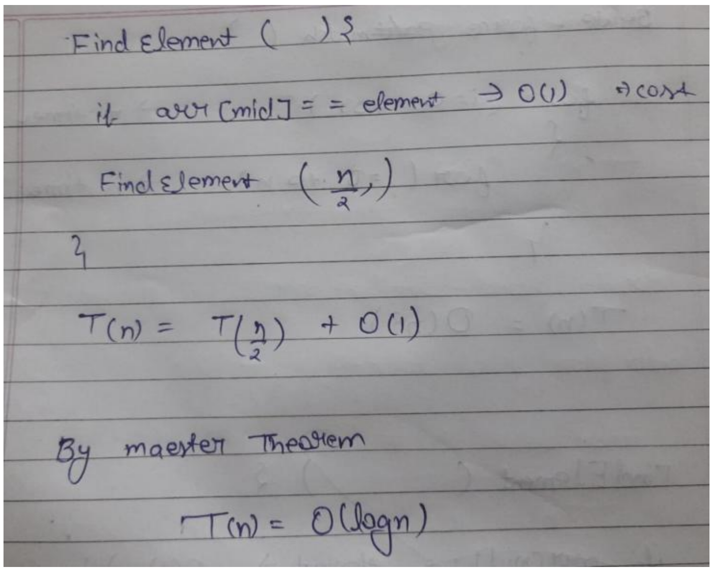
binary search ⇒ $T(n) = O(\log n)$

Total Time complexity = Find pivot + Binary search

$T(n) = O(\log n) + O(\log n)$

$T(n) = O(2\log n)$

(iii)more optimized: O(logn)



```
Find Element ( ) {

    if arr [mid] = = element → O(1) → cost

    Find Element ( n/2 , )

}
```

$T(n) = T(\frac{n}{2}) + O(1)$

By maester Theorem

$T(n) = O(\log n)$

Test cases:

N	A[]	index
12	40,45,12,35,80	2
50	50	0
100	50,100	1
30	10,20,30,40,50	2

Que3: given an array in which each elemnt at which atmost k distace away from it's target position in the sorted array,sort the array in optimized time

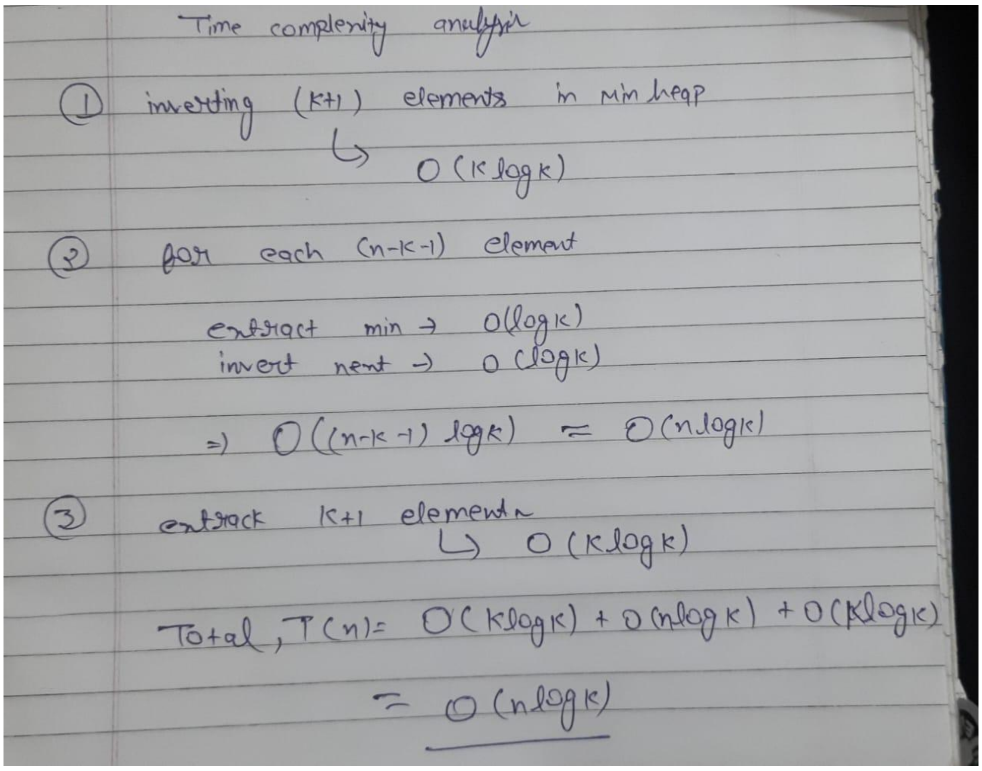
Ans:

Code:

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int* data;
    int size;
} MinHeap;

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void heapify(MinHeap* heap, int index) {
    int smallest = index;
    int left = 2 * index + 1;
    int right = 2 * index + 2;

    if (left < heap->size && heap->data[left] < heap->data[smallest])
        smallest = left;
    if (right < heap->size && heap->data[right] < heap->data[smallest])
        smallest = right;

    if (smallest != index) {
        swap(&heap->data[index], &heap->data[smallest]);
        heapify(heap, smallest);
    }
```

```c
}

int extractMin(MinHeap* heap) {
    if (heap->size == 0) return -1;

    int minValue = heap->data[0];
    heap->data[0] = heap->data[heap->size - 1];
    heap->size--;
    heapify(heap, 0);

    return minValue;
}

void insertHeap(MinHeap* heap, int value, int capacity) {
    if (heap->size == capacity) return;

    int i = heap->size++;
    heap->data[i] = value;

    while (i > 0 && heap->data[i] < heap->data[(i - 1) / 2]) {
        swap(&heap->data[i], &heap->data[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

void sortNearlySortedArray(int arr[], int n, int k) {
    MinHeap heap;
    heap.data = (int*)malloc((k + 1) * sizeof(int));
    heap.size = 0;

    for (int i = 0; i <= k && i < n; i++) {
        insertHeap(&heap, arr[i], k + 1);
    }

    int index = 0;
    for (int i = k + 1; i < n; i++) {
        arr[index++] = extractMin(&heap);
        insertHeap(&heap, arr[i], k + 1);
    }

    while (heap.size > 0) {
        arr[index++] = extractMin(&heap);
    }

    free(heap.data);
}

int main() {
    int arr[] = {6, 5, 3, 2, 8, 10, 9};
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 3;

    sortNearlySortedArray(arr, n, k);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    return 0;
}
```

Analysis:



Time complexity analysis

① inserting (k+1) elements in min heap

↳ $O(k \log k)$

② for each (n-k-1) element

extract min → $O(\log k)$

insert next → $O(\log k)$

$\Rightarrow O((n-k-1)\log k) \simeq O(n \log k)$

③ extract k+1 elements

↳ $O(k \log k)$

Total, $T(n) = O(k \log k) + O(n \log k) + O(k \log k)$

$\simeq \underline{O(n \log k)}$