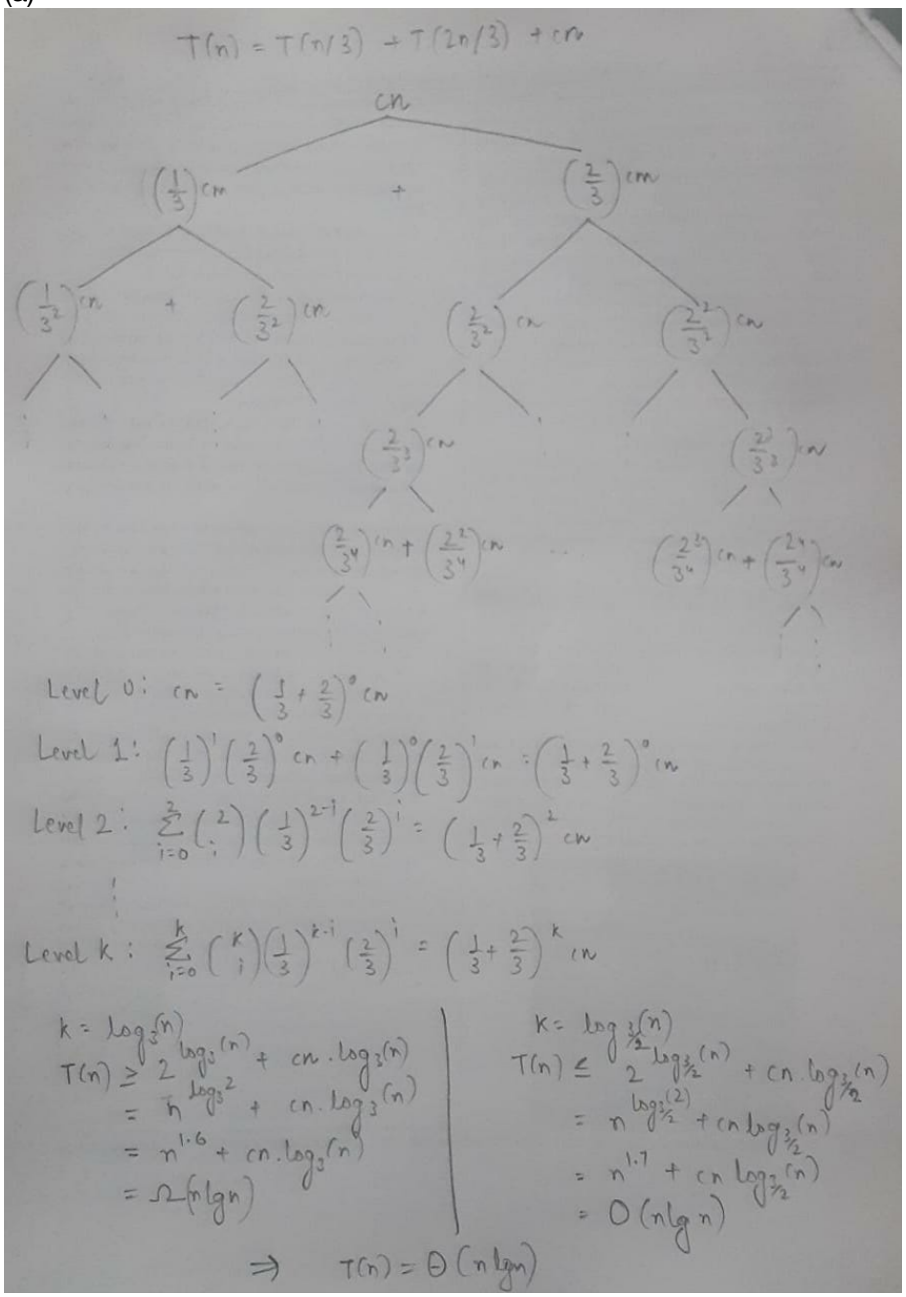
Problem 1)
Step 1

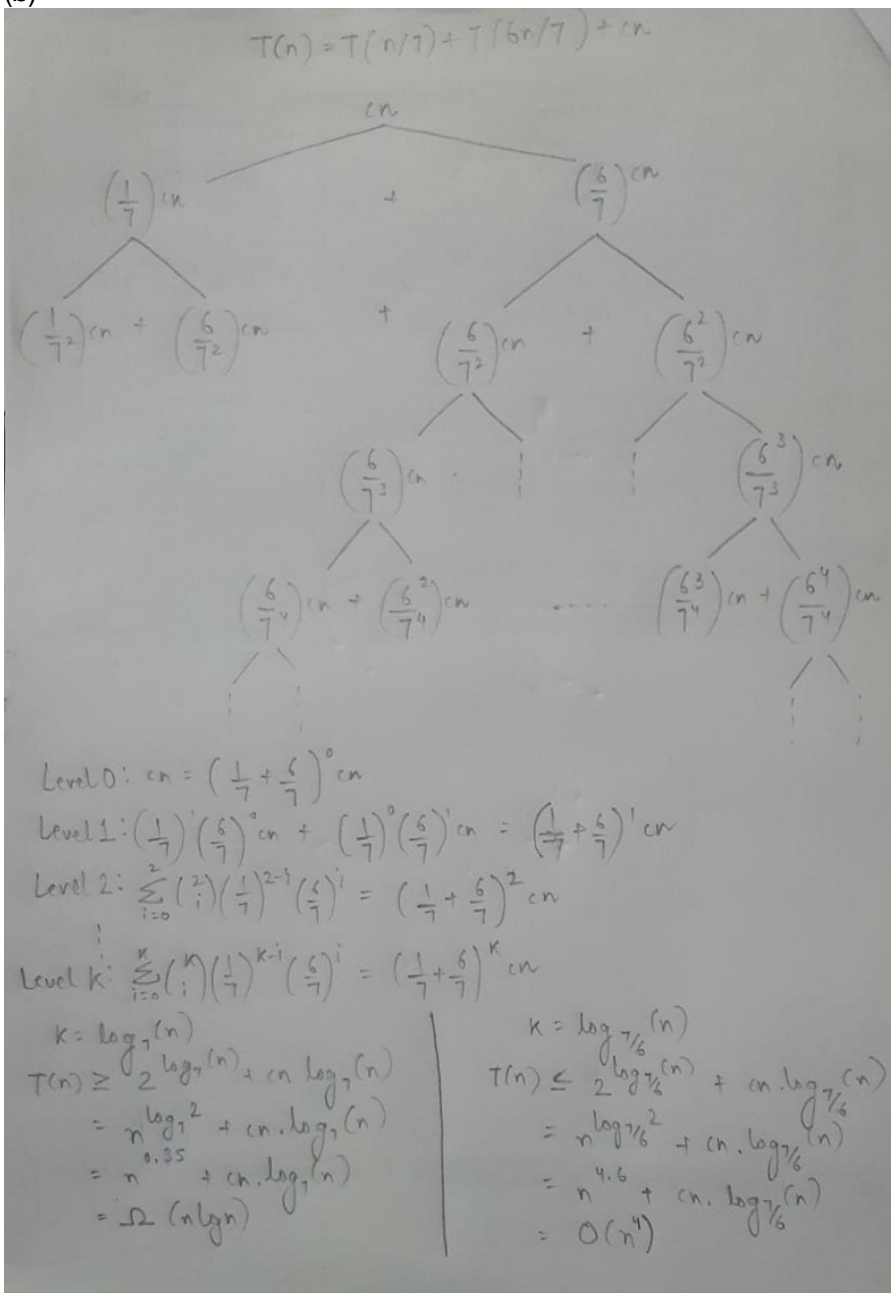
Recurrence relation for the SELECT algorithm when array is split into groups of size 3 is T(n)=T(n/3)+T(2n/3)+O(n)

Step 2

(a)



(b)



Problem 2)

(a)
```
// C++ code for k largest elements in an array
#include <bits/stdc++.h>
using namespace std;

void kLargest(int arr[], int n, int k)
{
   // Sort the given array arr in reverse
   // order.
   sort(arr, arr + n, greater<int>());

   // Print the first kth largest elements
   for (int i = 0; i < k; i++)
       cout << arr[i] << " ";
}
```

```
// driver program
int main()
{
   int arr[] = { 1, 23, 12, 9, 30, 2, 50 };
   int n = sizeof(arr) / sizeof(arr[0]);
   int k = 3;
   kLargest(arr, n, k);
}
```

(b)
```
let rec sort3 (a : 'a list) : 'a list =

  match a with
      [] -> []
    | [x] -> [x]
    | [x; y] -> [(min x y); (max x y)]
    | _ ->
        let n = List.length a in
        let m = (2*n + 2)/3 in
        let res1 = sort3 (take a m) @ (drop a m) in
        let res2 = (take res1 (n-m)) @ sort3 (drop res1 (n-m)) in
          sort3 (take res2 m) @ (drop res2 m)
```

(c) The time complexity of this algorithm depends of the size and structure of the graph. For example, if we start at the top left corner of our example graph, the algorithm will visit only 4 edges.

To compute the time complexity, we can use the number of calls to DFS as an elementary operation: the if statement and the mark operation both run in constant time, and the for loop makes a single call to DFS for each iteration.

- Before running the algorithm, all |V| vertices must be marked as not visited. This takes $\Theta(|V|)$ time.
- Let E' be the set of all edges in the connected component visited by the algorithm. The algorithm makes two calls to DFS for each edge {u, v} in E': one time when the algorithm visits the neighbors of u, and one time when it visits the neighbors of v.

Problem 3)

(a). $T(n) = 4T(n/2) + n^2$

By comparing with $T(n) = aT(n/b) + f(n)$

Here,

$a = 4$

$b = 2$

$f(n) = n^2$

So, $n^{\log_b a} = n^{\log_2 4} = n^2$

$f(n) = \Theta(n^{\log_b a})$

Case 2 implies here.

Thus, $T(n) = \Theta(n^{\log_b a} \log (n))$

$T(n) = \Theta(n^2 \log (n))$

(b). $T(n) = 2T(n/2) + \log n$

By comparing with $T(n) = aT(n/b) + f(n)$

Here,

$a = 2$

b = 2
f(n) = log n

So, $n^{\log_b a} = n^{\log_2 2} = n^1 = n$,
Since f(n) is smaller than $n^{\log_b a}$, case 1 of the master theorem implies that $T(n) = \Theta(n)$.

(c). $T(n) = 0.2T(n/2) + n$
By comparing with $T(n) = aT(n/b) + f(n)$
Here,
a = 0.2
b = 2
f(n) = n

So, $n^{\log_b a} = n^{\log_2 0.2} = n^{-2.322}$, Since f(n) is greater than $n^{\log_b a}$, case 3 of the master theorem asks us to check whether $a f(n/b) \leq cf(n)$ for some $c < 1$ and all n sufficiently large. This is indeed the case, so
$T(n) = \Theta(f(n)) = \Theta(n)$.

(d). $T(n) = 8T(n/2) + 2^n$
By comparing with $T(n) = aT(n/b) + f(n)$
Here,
a = 8
b = 2
$f(n) = 2^n$
So, $n^{\log_b a} = n^{\log_2 8} = n^3$, Since f(n) is greater than $n^{\log_b a}$, case 3 of the master theorem asks us to check whether $a f(n/b) \leq cf(n)$ for some $c < 1$ and all n sufficiently large. This is indeed the case, so
$T(n) = \Theta(f(n)) = \Theta(2^n)$.

(e). $T(n) = 2T(n/2) + n/\log n$

Here,
a = 2
b = 2
$f(n) = n/\log n$

So, $n^{\log_b a} = n^{\log_2 2} = n^1 = n$, f(n) is smaller than $n^{\log_b a}$ but by less than a polynomial factor. Therefore, the master theorem makes no claim about the solution to the recurrence.

Problem 4)
(a)

Assuming that the array A of size n with distinct elements and the positive integer satisfying 1<=k<=n is sorted in increasing order, algorithm with O(n) time complexity to find all the kth smallest elements that is the first smallest, second smallest, and … the kth smallest element is:

For i=1 to k:

Print A[i]

Or if the array is not sorted then, a sorting algorithm with O(n) time complexity must be applied on the array like radix sort or counting sort. Then the loop from 1 to k must be applied on the sorted array to print first k elements since all the elements are distinct and will get the precise result.

Time complexity is O(n) since loop runs from 1 to k and does basic elementary operations which take O(1) time and the sorting loop say counting sort also takes O(n). Hence the time complexity of this algorithm is O(n).

(b)

Assuming that the array A of size n with distinct elements and the positive integers, k and l, satisfying 1<=l<=k<=n is sorted in increasing order, algorithm with O(n) time complexity to find the lth smallest, l+1th smallest, and … the kth smallest element is:

For i=l to k:

Print A[i]

Or if the array is not sorted then, a sorting algorithm with O(n) time complexity must be applied on the array like radix sort or counting sort. Then the loop from l to k must be applied on the sorted array to print first elements from index l to index k, since all the elements are distinct and will get the precise result.

Time complexity is O(n) since loop runs from l to k and does basic elementary operations which take O(1) time and the sorting loop say counting sort also takes O(n). Hence the time complexity of this algorithm is O(n).