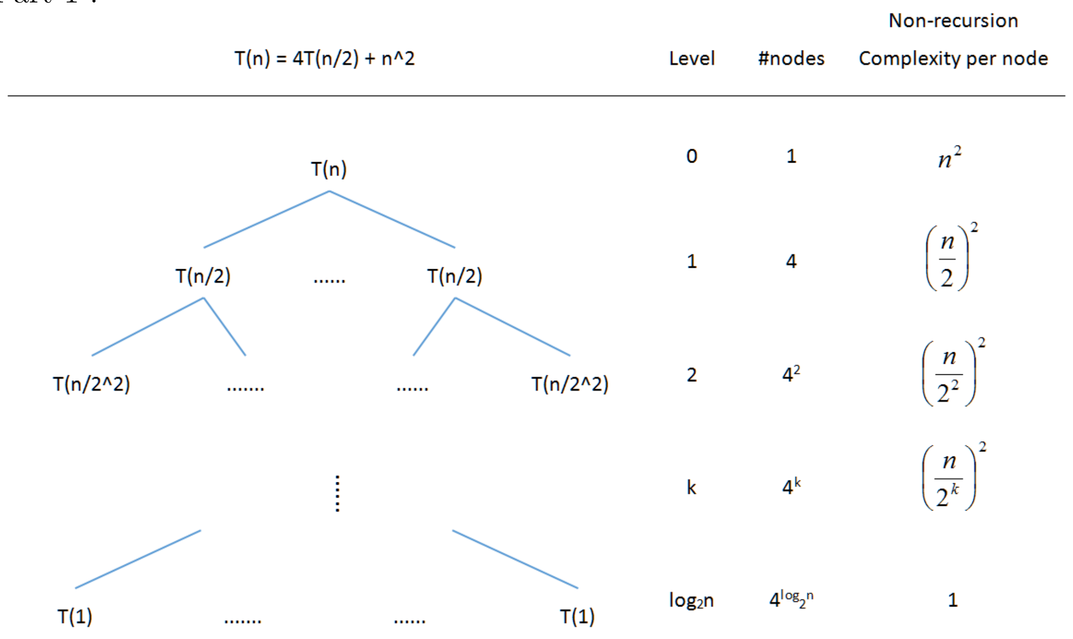
CS 344 Assignment 2

October 22, 2018

Name : Yang Bao
NetID : yb184
Section : 02
The classmate I discussed with : Zitian Qin

Problem 1 :

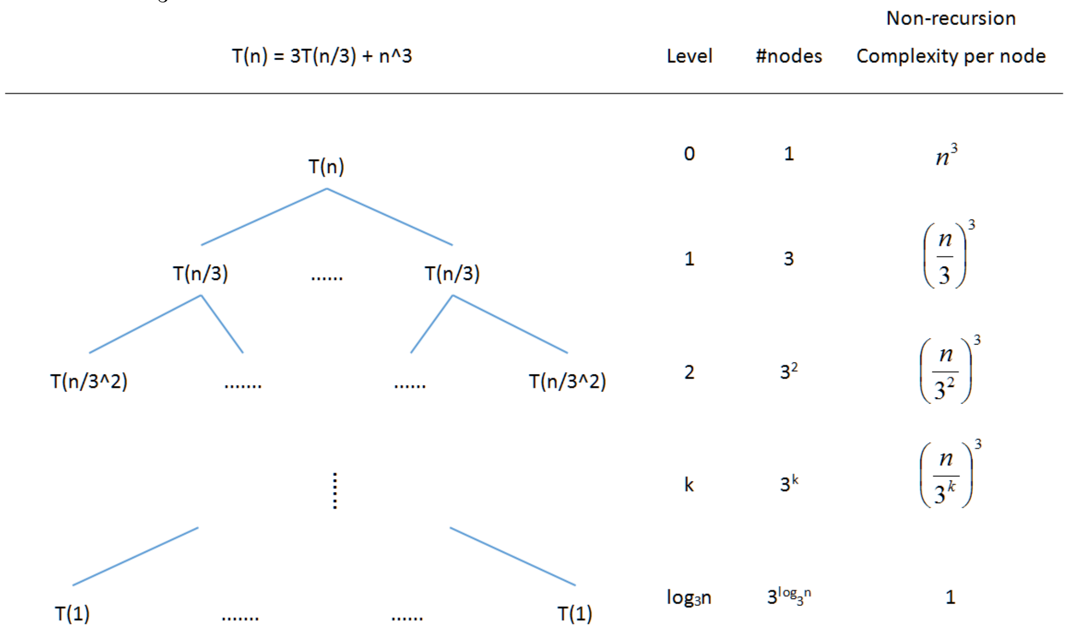
1. Part 1 :



Level	#nodes	Non-recursion Complexity per node
0	1	n^2
1	4	$\left(\frac{n}{2}\right)^2$
2	4^2	$\left(\frac{n}{2^2}\right)^2$
k	4^k	$\left(\frac{n}{2^k}\right)^2$
$\log_2 n$	$4^{\log_2 n}$	1

Running time : $\sum_{k=0}^{log_2 n} 4^k(\frac{n}{2^k})^2 = n^2 \sum_{k=0}^{log_2 n} 1 = O(n^2 logn)$
Prove part :
Let$P(n)$ be the statement that $T(n) \leq 2n^2 logn$. We need to prove that $P(n)$ is true for all $n \geq 2$ by induction.
Base Case : For n = 2, we have $T(2) = 2 \leq 2 \times 2^2 log2$.
Then for the induction part we have $T(\frac{n}{2}) \leq 2(\frac{n}{2})^2 log(\frac{n}{2})$ and $T(n) = 4T(\frac{n}{2}) + n^2$
So we have :
$T(n) = 4T(\frac{n}{2}) + n^2 \leq 2 \cdot 4(\frac{n}{2})^2 log\frac{n}{2} + n^2 = 2n^2 log\frac{n}{2} + n^2 = 2n^2(logn - 1) + n^2 \leq 2n^2 logn$
So $P(n)$ is true. Prove finished.

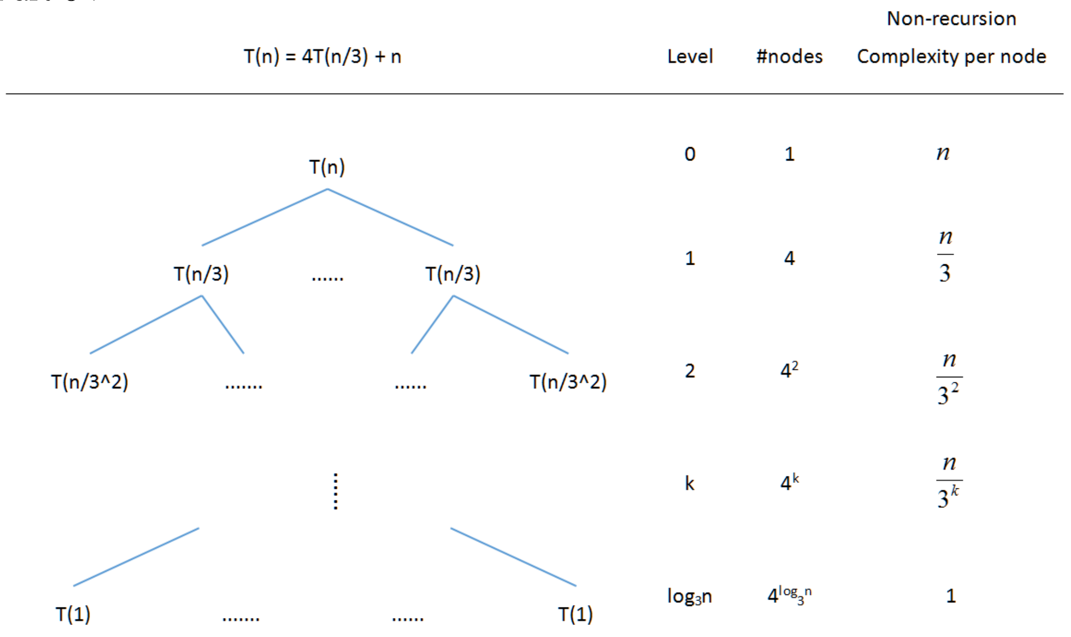
2. Part 2 :
$T(n) = 3T(\frac{n}{3}) + n^3$



Level	#nodes	Non-recursion Complexity per node
0	1	n^3
1	3	$\left(\frac{n}{3}\right)^3$
2	3^2	$\left(\frac{n}{3^2}\right)^3$
k	3^k	$\left(\frac{n}{3^k}\right)^3$
$\log_3 n$	$3^{\log_3 n}$	1

Running time : $\sum_{k=0}^{log3n} 3^k(\frac{n}{3^k})^3 = n^3 \sum_{k=0}^{log3n} (\frac{1}{3})^k = O(n^3)$
Prove part :
Let$P(n)$ be the statement that $T(n) \leq 2n^3$. We need to prove that $P(n)$ is true for all $n \geq 2$ by induction.
Base Case : For n = 2, we have $T(2) = 2 \leq 2 \times 2^3$.
Then for the induction part we have $T(\frac{n}{2}) \leq 2(\frac{n}{2})^3$ and $T(n) = 3T(\frac{n}{3}) + n^3$
So we have :
$T(n) = 3T(\frac{n}{3}) + n^3 \leq 2 \cdot 3(\frac{n}{3})^3 + n^3 = \frac{2}{9}n^3 log + n^3 \leq 2n^3$
So $P(n)$ is true. Prove finished.

3. Part 3 :



Running time : $\sum_{k=0}^{log_3 n} 4^k \frac{n}{3^k} = n\sum_{k=0}^{log_3 n}(\frac{4}{3})^k = O(n^{log_3 4})$

Prove part :

Let$P(n)$ be the statement that $T(n) \leq 6(n^{log_3 4} - n)$. If $P(n)$ is true for all $n \geq 2$, then $T(n) \leq 6(n^{log_3 4} - n) \leq 6 * n^{log_3 4}$, then $T(n) = O(n^{log_3 4})$. So we need to prove that $P(n)$ is true for all $n \geq 2$ by induction.

Base Case : For n = 2, we have $T(2) = 2 \leq 2 \times 6 \cdot (2^{log_3 4} - 2)$.

Then for the induction part we have $T(\frac{n}{3}) \leq 6((\frac{n}{3})^{log_3 4} - \frac{n}{3})$ and $T(n) = 4T(\frac{n}{3}) + n$

So we have :

$T(n) = 4T(\frac{n}{3}) + n \leq 4 \cdot 6((\frac{n}{3})^{log_3 4} - \frac{n}{3}) + n = 6n^{log_3 4} - 8n + n = 6(n^{log_3 4} - n) - n \leq 6(n^{log_3 4} - n)$

So $P(n)$ is true. Prove finished.

Problem 2 :

1. Part 1 :
   Consider the recursive formula $T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$.
   Compared with the algorithm discussed in class, the difference is because we split the array into 9 pieces, so while considering if m is a good partition point, we can only ensure there are $\frac{n}{2} \cdot \frac{5}{9} = \frac{5n}{18}$ elements of A are smaller or larger than m. So the part of $T(\frac{7n}{10})$ will become $T(\frac{13n}{18})$. Also since the array is split into 9 pieces, the part of $T(T(\frac{n}{5})$ will become $T(T(\frac{n}{9})$. And the other steps will not influence the recursive formula.
   So the new recursive formula will be $T(n) \leq T(\frac{n}{9}) + T(\frac{13n}{18}) + O(n)$.

2. Part 2 :
   Assume $T(n) = O(n)$ so we need to prove that $T(n) \leq Cn$ for all $n \geq 2$ is true by induction.
   The average of $\frac{n}{9}$ and $\frac{13n}{18}$ is $\frac{15n}{36}$. So like the instructor used in lecture, we can prove $T'(n) = 2T'(\frac{15n}{36}) + n \leq Cn$ rather than $T(n) \leq Cn$
   Let C = 10 and use Induction prove :
   Base part: for n = 1 , $T'(n) = \frac{13}{9} + 1 \leq 10 \times 1$
   Then for the induction part we have $T'(\frac{15n}{36}) \leq 10(\frac{15n}{36})$ and $T'(n) = 2T'(\frac{15n}{36}) + n$
   So we have :
   $T'(n) = 2T'(\frac{15n}{36}) + n \leq 2 \times 10\frac{15n}{36} + n = \frac{84}{9}n \leq 10n$
   So $T'(n) = O(n)$, so $T(n) = O(n)$.

Problem 3 :

1. Part 1 :
   The pseudocode is :

```
findSpecial(array A){
  sort A by quicksort or mergesort in decreasing order     O(nlogn)
  for A[i] in A :                                           O(n)
    if (A[i] = i)
      output A[i]
  return "no solution"
  }
```

2. Part 2 :
   The pseudocode is :

```
  k= A.length / 2
  findSpecial(array A, int k){
    compute m - the median of array A            O(n) total
    if m > k
        Less = all the elements less than m
        if Less != null
            findSpecial(Less, k)                 T(n/2)
        else
            return "no solution"
    if m < k
        Large = all the elements larger than m
        if Large != null
            findSpecial(Large, k + (Large.length + 1)/2)     T(n/2)
        else
            return "no solution"
    if m = k
        output m
  }
```

   The idea of this algorithm is every time check is the median m (by findMedian algorithm from lecture) equal to the rank k, since there are no duplicates, if $m < k$, all the elements less than m can never be equal to k, so only need to consider the larger half of the array. So does the situation if $m > k$. If $m = k$ then we have found the special number so we can return. If the array becomes *null* then it means we have checked all the elements in array so we can just return "no solution".
   In the algorithm above the recursive part is T(n/2) and other steps are all O(n) so the recursive formula for this algorithm is :

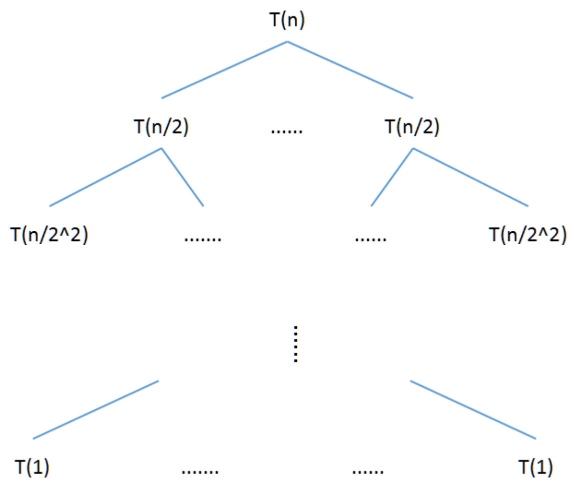
$T(n) \leq T(n/2) + O(n)$
So overall it's a $O(n)$ algorithm.

Problem 4 :
The algorithm for this problem is almost the same with the mergesort. The difference is while the merge part of mergesort, if element in the left array is less then larger than the element in the right array, the count of swapped pairs increases by 1.
Here is the pseudocode :

```
findSwap(array A){
    count = 0
    mergeSort(A,length of A,count)
    output count
}

merge(A, L, R, l, r, count){
    k = 0, i =0, j = 0
    while i < l && j < r
        if (L[i] < R[j])
            A[k++] = L[i++]
        else
            A[k++] = R[j++]
            count++
    put the rest part of L and R into A
}

mergeSort (array A, int n, int count){
    mid = n/2

    for i in 0 to mid
        L[i] = A[i]
    for i in mid to n
        R[i-mid] = A[i]

    mergeSort(L,mid,count)                      T(n/2)
    mergeSort(R,n-mid,count)                    T(n/2)

    merge(A, L, R, mid, n-mid, count)      n
}
```

The idea is use merge sort to sort the whole array and during the merge steps if elements former is bigger than the later elements, then there need a swap and so let count++. So the recursive formula for this algorithm is :
$T(n) = 2T(\frac{n}{2}) + n$
Using the recursive tree :

T(n) = 2T(n/2) + n



Level	#nodes	Non-recursion Complexity per node
0	1	n
1	2	$\frac{n}{2}$
2	2^2	$\frac{n}{2^2}$
k	2^k	$\frac{n}{2^k}$
$\mathrm{Log}_2 n$	$2^{\log_2 n}$	1

$\sum_{k=0}^{log_2 n} 2^k \frac{n}{2^k} = n \sum_{k=0}^{log_2 n} 1 = O(nlogn)$