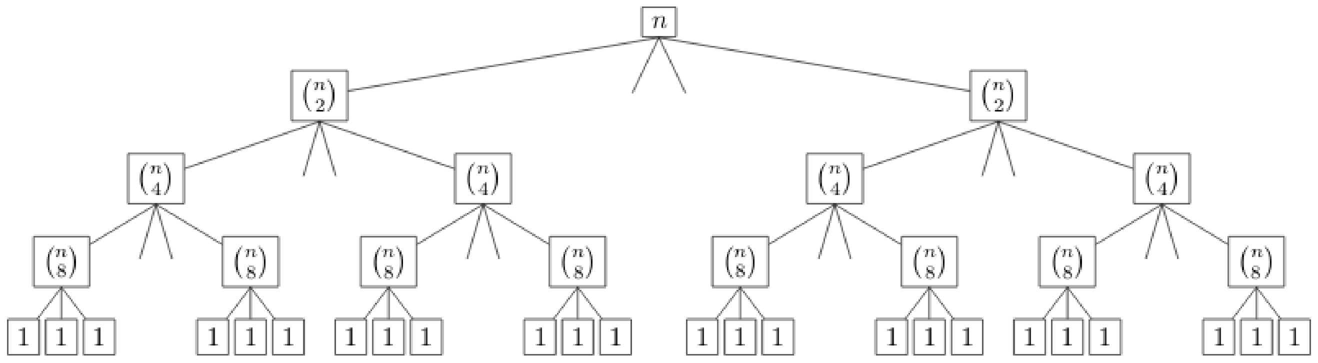
CSCI 602 Algorithm Design & Analysis Assignment 3 10 pts

First name: Michele Last name: Fattoruso Zid: Z1840898

a) Draw the recursion tree for T(n) = 4T(n/2) + n and give a tight asymptotic bound on its solution.



Analyzing the computational time of each level, we see that :

Level	Computational Cost
0	$4(n/2)$
1	$4(\frac{n}{2})=2n$
2	$16(\frac{n}{4})=4n$
...	...
k	$2^{2k}(\frac{n}{2^k})=2^k n$

considering that the problem gets divided every time by half, we know that the height of the tree will be $\log_2 n$.
So, the computational cost of the tree will be equal to:

$$\sum_{i=0}^{\log_2 n} 2^i n$$

this is equal to

$$n\sum_{i=0}^{\log_2 n} 2^i$$

This is an infinite series, and we know that the series is equal to $2^0+2^1+2^2+...+2^k=2^{k+1}-1$
We can so write T(n) in the following way:

$$\begin{aligned}T(N)&=n\sum_{i=0}^{\log_2 n} 2^i\\ &=n\cdot 2^{\log_2 n+1}-1\\ &=n\cdot(2\cdot 2^{\log_2 n})\end{aligned}$$

$$\begin{aligned} T(N) &= n\cdot(2\cdot 2^{\log_2 n}-1) \\ &= n\cdot(2\cdot n^{\log_2 2}-1) \\ &= n\cdot(2n-1)=2n^2-2 \end{aligned}$$

From this so we know that

$$T(n)=\theta(n^2)$$

b) Verify your bound by induction

$$T(n)=\begin{cases} 1 & if\ n=1 \\ 4T(\frac{n}{2})+n & otherwise \end{cases}$$

verify:

$$T(n)=4T(\frac{n}{2})+n$$

$$=4(4T(\frac{n}{4})+\frac{n}{2})+n=16T(\frac{n}{4})+3n$$

$$=16(4T(\frac{n}{8})+\frac{n}{4})+3n=64T(\frac{n}{8})+7n$$

$$=64(4T(\frac{n}{16})+\frac{n}{8})+7n=256T(\frac{n}{16})+41n$$

From the current recurrence, we can deduct that T(n) is in the form of:

$$2^{(2\cdot k)}\cdot(\frac{n}{2^k})+cn$$

considering that we divide at every step, the problem in half, we will reach T(1) after $\log_2 n$ steps. So we have that:

$$k=\log_2 n$$

We can then rewrite the T(n) formula as:

$$\begin{aligned} T(n) &= 2^{2\cdot\log_2 n}\cdot\frac{n}{2^{\log_2 n}}+c\cdot n \\ &= n^{2\cdot\log_2 2}\cdot\frac{n}{n^{\log_2 2}}+c\cdot n \\ &= n^2\cdot\frac{n}{n}\ +\ c\cdot n \\ &= n^2+c\cdot n \end{aligned}$$

We can conclude so, that

$$T(n)=\theta(n^2)$$

3) Use the master's method to solve above recurrence.

$$T(n)=4T(\frac{n}{2})+n$$

in the current recurrence, $f(n)=n$, $a=4$ and $b=2$.
To apply the master theorem, we need to compare $f(n)$ with $n^{\log_b a}$ and compare the polynomial of the two functions.

$$f(n) = n$$
$$n^{\log_b a} = n^{\log_2 4} = n^2$$

Since $n^{\log_b a}$ grows polynomialy faster than $f(n)$
$$T(n) = \theta(n^{\log_b a}) = \theta(n^2)$$

2) Use the master method to resolve $T(n)=aT(\frac{n}{2})+cn^3$ **for three cases a=7,8,9 and a constant c > 0**

a) in the current recurrence, $f(n)=n^3$, $a=7$ and $b=2$.
$$f(n)=n^3$$
$$n^{\log_b a} = n^{\lfloor \log_2 7 \rfloor} = n^2$$
since $f(n)$ grows polynomialy faster than $n^{\log_b a}$, then
$$T(n) = \theta(f(n)) = \theta(n^3)$$

b) in the current recurrence, $f(n)=n^3$, $a=8$ and $b=2$.
$$f(n)=n^3$$
$$n^{\log_b a} = n^{\lfloor \log_2 8 \rfloor} = n^3$$
since $f(n)$ grows at the same speed of $n^{\log_b a}$, then
$$T(n) = \theta(n^{\log_b a} logn) = \theta(n^3 logn)$$

c) in the current recurrence, $f(n)=n^3$, $a=9$ and $b=2$.
$$f(n)=n^3$$
$$n^{\log_b a} = n^{\lfloor \log_2 9 \rfloor} = n^3$$
since $f(n)$ grows at the same speed of $n^{\log_b a}$, then
$$T(n) = \theta(n^{\log_b a} logn) = \theta(n^3 logn)$$

3.You are given an unsorted array of n integers.

1. **What is the time cost to find the maximum or minimum?**

To find the maximum and minimum in the array, we can loop over all the elements of the array, comparing the current value with two stored variables containing the minumim and maximum value

```
Procedure FIND_MIN_MAX(A)
    min ← A[1]
    max ← A[1]
    for i=1 to A.length()
        if A[i] > max
            max ← A[i]
        end if
        if A[i] < min
            min ← A[i]
        end if
    end for
```

end procedure

The time cost of this algorithm is $T(n)=\theta(n)$, as it doesn't matter where the elements are located, the algorithm will always check all the elements in the array, even if the minimum and maximum of the array are in the first and second position of the array.

2. **Assume the time cost to find each of the maximum and minimum is T(n). A straightforward way to find both of the maximum and minimum costs 2T(n). Please give a better approach using divide-and-conquer. Describe the recurrence and solve it. You can assume n is power of 2.**

An approach to the problem, using the Divide and Conquer technique, could be to sort the array with a sorting algorithm like merge sort, and then pick the first and last element from the array as minimum and maximum.

The recurrence would be

$$T(n)=\begin{cases} 1 & if\ n=1 \\ 4T(\frac{n}{2})+n & otherwise \end{cases}$$

verify:

$$T(n)=2T(\frac{n}{2})+n$$
$$=2(2T(\frac{n}{4})+\frac{n}{2})+n=4T(\frac{n}{4})+2n$$
$$=4(2T(\frac{n}{8})+\frac{n}{4})+2n=8T(\frac{n}{8})+3n$$

From this recurrence, we can see that the recurrence is in the form of:

$$T(n)=2^k(\frac{n}{2^k})+kn$$

To reach the step where the array has 1 element we know that every time we divide the problem in half, so we will reach T(1) after $\log_2 n$ steps. At that point we will have:

$$T(n)=2^{\log_2 n}T(1)+n\cdot\log_2 n$$
$$=n+n\cdot nlog_2 n$$

We can so conclude that the time complexity of this algorithm is

$$T(n)=\theta(n\cdot\log_2 n)$$

4. Banks often record transactions on an account in order of the times of the transactions, but many people like to receive their bank statement with checks listed in order by check number. People usually write checks in order by check number,and merchants usually cash them with reasonable dispatch. The problem of converting time-of-transaction ordering to check-number ordering is therefore the problem of sorting almost-sorted input. Argue that insertion sort would tend to beat quick sort on this problem.

For quick sort, we know that the average execution time is $nlog_2 n$ .Said so, we know that the worst case for quick sort is n^2 .

The worst case scenario happens when the chosen pivot so divide the array is either the maximum or the minimum of the array we are trying to sort. The problem comes from the fact that the divide step of quick-sort will divide the problem in two sub-problems of size 0 and n-1, creating a totally unbalanced recursion tree.
If we consider our problem as an almost sorted array, quick sort will most probably perform line in it's worst case scenario.

$$T(n)=\mathrm{O}(n^2)$$

In the case of Insertion sort, the algorithm won't perform any swap in the array, if an element is already in the sorted position. The only operation to be executed will be a comparison with the previous element, that for simplicity we define as constant time $T(1)$.
The only swap operations that insertion sort will perform, is when an element is not in the right ordered position.
Knowing that the checks are cashed by the merchants really fast, we can suppose that each check number, even if cashed not in the right order, won't be too much far from it's correct sorted position in the array.
Said so, insertion sort will perform mostly constant time operations for it's already sorted elements, and a maximum of k comparison and swaps for each unsorted element, where k is the distance of the element from it's sorted position.

In this case, Insertion sort will have a time complexity equal to:

$$T(n)=\mathrm{O}(kn)$$

In conclusion we say that being insertion sort time complexity linear for the input size of this problem, it will be faster than quick sort.

5.Let A[1..n] be an array of n distinct numbers. If i < j and A[i] > A[j], then the pair (i, j) is called an inversion.

1. **List the inversions in the array <2, 3, 8, 6, 1>.**

(2,1) (3,1)

2. **What array with elements from the set {1, 2, .., n} has the most number of inversions? How many does it have?**

The most number of inversions are in a descending sorted array <n,n-1,n-2,… 1>

To calculate how many inversions we have, we have to consider that:

- for the first element in the array, we have n-1 inversions
- for the second element in the array, we have n-2 inversions
- …
- for the last element in the array we have n-n inversions

This can be expressed mathematically as:

$\sum_{i=1}^{n} n-i$ which is the sum of all the number from n-1 to 0. which is equal to the sum of the numbers from zero to n-1.

We know that this is equivalent to:

$$\frac{n\cdot(n-1)}{2}=\frac{n^2}{2}-\frac{n}{2}$$

so the number of inversions in the array would be equal to

$$\frac{n^2}{2}-\frac{n}{2}$$