Romil V. Shah
20008692

Assignment 1
CS/CPE 600
Prof. Reza Peyrovian
Submission Date: 09/18/2022

REINFORCEMENT Questions

Q1. (No.7)

Order the following list of functions by the big-Oh notation. Group together (for example, by underlining) those functions that are big-Theta of one another.

$6n \log n$	2^{100}	$\log \log n$	$\log^2 n$	$2^{\log n}$
2^{2^n}	$\lceil \sqrt{n} \rceil$	$n^{0.01}$	$1/n$	$4n^{3/2}$
$3n^{0.5}$	$5n$	$\lfloor 2n \log^2 n \rfloor$	2^n	$n \log_4 n$
4^n	n^3	$n^2 \log n$	$4^{\log n}$	$\sqrt{\log n}$

Hint: When in doubt about two functions $f(n)$ and $g(n)$, consider $\log f(n)$ and $\log g(n)$ or $2^{f(n)}$ and $2^{g(n)}$.

Ans. The order from lower to higher functions:

1. $1/n$
2. 2^{100}
3. log log n
4. sqrt(log n)
5. $\log^2 n$
6. $n^{0.01}$
7. sqrt(n), 3 $n^{0.5}$
8. $2^{\log n}$, 5 n
9. n $\log_4 n$, 6 n log n
10. 2 n $\log^2$ n
11. 4 $n^{3/2}$
12. $4^{\log n}$
13. n^2 log n
14. n^3
15. 2^n
16. 4^n
17. 2^{2n}

Q2. (No. 9)

Bill has an algorithm, **find2D**, to find an element x in an $n \times n$ array A. The algorithm **find2D** iterates over the rows of A and calls the algorithm **arrayFind**, of Algorithm 1.3.2, on each one, until x is found or it has searched all rows of A. What is the worst-case running time of **find2D** in terms of n? Is this a linear-time algorithm? Why or why not?

Ans.
The worst-case runtime complexity of FIND2D is $O(n^2)$, as it is a quadratic algorithm instead of a linear-time algorithm.

By examining the worst case where the element x is the very last item in the n × n array to be examined. In this case, find2D calls the algorithm arrayFind n times. arrayFind will then have to search all n elements for each call until the final where x is found.

Therefore, n comparisons are done for each arrayFind call. This means we have n × n operations - $O(n^2)$ running time.

Q3. (No. 22)

Show that n is $o(n \log n)$.

Ans.
We say that n is o(n log n) if for any constant $c > 0$ there is any constant $n_0 >= 0$, such that $n < c * n \log n$ for $n >= n_0$.

So, $1/c < \log n$, we choose $n_0 = 2^{1/c} + 1$ (when log is the base of 2).

Q4. (No. 23)

Show that n^2 is $\omega(n)$.

Ans.
To show that n^2 is w(n), let $c > 0$ be any constant,
there is a constant $n_0 > 0$ such that $n^2 > cn$. So $n > c$.
We can choose $n_0 = c + 1$.

Q5. (No. 24)

Show that $n^3 \log n$ is $\Omega(n^3)$.

Ans.
To prove the expression above, we need to find a constant c > 0 and constant n_0 >= 1, such that $n^3 \log n >= cn^3$.

We can choose c = 1 and n_0 = 2(suppose log is the base of 2).

Q6. (No. 32)

Suppose we have a set of n balls and we choose each one independently with probability $1/n^{1/2}$ to go into a basket. Derive an upper bound on the probability that there are more than $3n^{1/2}$ balls in the basket.

Ans.
Based on Chernoff Bounds,

$\mu = E(X) = n * (1/n^{1/2}) = n^{1/2}$.

Then for δ =2, the upper bound is

$\Pr[X>(1+ \delta)u] < (e^{\delta}/(1+\delta)^{(1+\delta)})^{u} \Rightarrow \Pr(X > 3\mu) < \left[\frac{e^2}{3^3}\right]^{\sqrt{n}}$

CREATIVITY Questions

Q1. (No. 36)

What is the total running time of counting from 1 to n in binary if the time needed to add 1 to the current number i is proportional to the number of bits in the binary expansion of i that must change in going from i to $i+1$?

Ans.
Let t be the time to change every single bit and let k be the total bits.

The total work is:
$t * (n/2^0 + n/2^1 + n/2^2 \ldots + n/2^k) < t * n * 2 \Rightarrow O(n)$

So, the total running time is O(n).

Q2. (No. 39)

Consider the following recurrence equation, defining a function $T(n)$:

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2T(n-1) & \text{otherwise,} \end{cases}$$

Show, by induction, that $T(n) = 2^n$.

Ans.
For T(n):
$T(n) = 2 * T(n-1) = 2 * 2 * T(n-2) = \ldots = 2 * 2^{n-1} * T(0) = 2^n$

Q3. (No. 52)

Show that the summation $\sum_{i=1}^{n} \lceil \log_2 \ i \rceil$ is $O(n \ \log \ n)$.

Ans.
Here, we assume that the base to all log used is 2.

Upper Bound Summation(log i) = log(1) + log(2) + ... + log(n)
= log(n) + log(n) + ... log(n)
= n * log(n)

Lower Bound Summation (log i) = log (1) + ... + log(n/2) + ... + log(n)
= log(n/2) + ... + log (n)
= log(n/2) + ... + log(n/2)
= n/2 * log (n/2)

Hence, we can say that the summation is O(n log(n)).

Q4. (No. 62)

Consider an implementation of the extendable table, but instead of copying the elements of the table into an array of double the size (that is, from n to $2N$) when its capacity is reached, we copy the elements into an array with $\lceil \sqrt{N} \rceil$ additional cells, going from capacity n to $N + \lceil \sqrt{N} \rceil$. Show that performing a sequence of n **add** operations (that is, insertions at the end) runs in $\Theta(n^{3/2})$ time in this case.

Ans.
The size of the array is expanded from N to N + $\lceil N^{1/2} \rceil$

Based on the amortization, each insertion will cost $(N+ N^{1/2})/ N^{1/2} = 1+ N^{1/2}$ cyber dollars ($).

So, total insertion cost is as follow:

$\sum 1+ 1+SQRT(N) = \sum 2+SQRT(N)= 2n + \sum SQRT(N)$ from N=1 to N=n

that we can get it is no more than:

$(2/3)n^{3/2}+ (1/2)n^{1/2}- 1/6$ but no less than

$(2/3)n^{3/2}+ (1/2)n^{1/2}+ 1/3 - (1/2)2^{1/2}$.

So, the total cost of the array operation is $\theta(n^{3/2})$.

APPLICATION Questions

Q1. (No. 70)

Given an array, A, describe an efficient algorithm for reversing A. For example, if $A = [3, 4, 1, 5]$, then its reversal is $A = [5, 1, 4, 3]$. You can only use $O(1)$ memory in addition to that used by A itself. What is the running time of your algorithm?

Ans.

Algorithm reversal (start, end, n, A)

Step1: Initialize
n – number of elements in an array
start ← 0
end ← n-1

Step2: In a loop,
swap (arr[start], arr[end])

Step3: Start ← start + 1
End ← end -1

The Time Complexity will be O(n)

In the above algorithm,
Step 1 will execute in constant time (1) and Step 2 will take n time, swapping will take constant time and the Step 3 again will take the constant time (1).
So, the total running time of the algorithm is O(n).

The running time of this algorithm would be O(n) because it accesses every element in the array the one time.
And no extra space needed, it only needs two variables to record the pointers. So, the space complexity is O(1).

Q2. (No. 77)

Given an integer $k > 0$ and an array, A, of n bits, describe an efficient algorithm for finding the shortest subarray of A that contains k 1's. What is the running time of your method?

Ans.

Input: An array A of n-bits, indexed from 1 to n.

Output: The shortest subarray of A that contains k 1's.

```
Count ← 0
k ← 0   //maximum found so far
for i ← 1 to n do
        if A[i] = 0 then
        count ← 0
        else
        count ← count + 1
        k ← max(k, count)
return k
```

Run Time:

```
Count ← 0                               (1 time)
k ← 0                                   (1 time)
for i ← 1 to n do                       (n times)
        if A[i] = 0 then                (1 time)
        count ← 0                       (1 time)
        else                            (1 time)
        count ← count + 1               (1 time)
        k ← max(k, count)               (1 time)
        return k                        (1 time)
```

Total Run Time = 1 + 1 + n + 1 + 1 + 1 + 1 + 1 + 1

= n + 8

= O(n)

For loop will take n times to run whereas if and else statement will run in constant O(1) time. All other are variables which will take constant time O(1) to run.