Ali Abdullah Ahmad

20031246

Assignment 1

CS 600

Prof. Reza Peyrovian

REINFORCEMENT Questions

1)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.
Ans 1.6.7-
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}

2)Bill has an algorithm, find2D, to find an element x in an n×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 xis 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 1.6.9- In the worst-case scenario, 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. During each call, arrayFind must search through all n elements until x is found.

As a result, n comparisons are performed for each arrayFind call. This leads to n×n operations in total, resulting in an O(n^2) runtime complexity.

The worst-case runtime complexity of FIND2D is O(n^2), as it operates as a quadratic algorithm

3)Show that n is o(nlogn)
Ans 1.6.22
To show that n is $o(n \log n)$, we use the definition of little-o notation. A function $f(n)$ is $o(g(n))$ if:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Here, $f(n) = n$ and $g(n) = n \log n$. Let's compute the limit:

$$\lim_{n \to \infty} \frac{n}{n \log n} = \lim_{n \to \infty} \frac{1}{\log n}.$$

As $n \to \infty$, $\log n \to \infty$, so:

$$\frac{1}{\log n} \to 0.$$

Thus:

$$\lim_{n \to \infty} \frac{n}{n \log n} = 0,$$

which confirms that n is $o(n \log n)$.

4)show that n^2 is $\omega(n)$
Ans1.6.23-

To show that n^2 is $\omega(n)$, we need to use the formal definition of little omega notation, which is:

$$f(n) = \omega(g(n)) \quad \text{if and only if} \quad \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

In this case, we want to show that:

$$n^2 = \omega(n)$$

Step 1: Compute the limit

We start by computing the limit:

$$\lim_{n \to \infty} \frac{n^2}{n}$$

Simplifying the expression:

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

Thus, we need to evaluate:

$$\lim_{n \to \infty} n$$

As n tends to infinity, n also tends to infinity. Therefore, the limit is:

$$\lim_{n \to \infty} n = \infty$$

Step 2: Conclusion

Since the limit is infinity, we have shown that:

$$\frac{n^2}{n} \to \infty \quad \text{as} \quad n \to \infty$$

Therefore, by the definition of little omega notation, we can conclude that:

$$n^2 = \omega(n)$$

This completes the proof.

5)show that $n^3 \log n$ is $\Omega(n^3)$
Ans 1.6.24

To show that $n^3 \log n$ is $\Omega(n^3)$, we apply the formal definition of Big-Omega notation, which states that there exist constants $c > 0$ and n_0 such that:

$$f(n) \geq c \cdot g(n) \quad \text{for all} \quad n > n_0$$

Here, we have:

- $f(n) = n^3 \log n$ - $g(n) = n^3$

We need to show that:

$$n^3 \log n \geq c \cdot n^3$$

Dividing both sides by n^3:

$$\log n \geq c$$

Now, for $c = 1$ and $n_0 = 2$, this inequality holds true because for all $n > 2$, $\log n \geq 1$. Therefore, for all $n > 2$, the inequality $n^3 \log n \geq n^3$ is satisfied.

Hence, we conclude that $n^3 \log n$ is $\Omega(n^3)$ with $c = 1$ and $n_0 = 2$.

This completes the proof.

6)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.
Ans1.6.32

Based on Chernoff Bounds,

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

Then for $\delta = 2, the upper bound is$

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

CREATIVITY Questions

7)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 ithat must change in going from i to i+1?
Ans1.6.36

In binary representation:

- A bit changes when the number i "rolls over" from 1 to 0.
- The least significant bit (LSB) changes with every increment.
- The second least significant bit changes every 2 increments.

- The k-th bit changes every 2^k increments.

The total number of bit changes across all increments from 1 to n can be calculated as:

$$\text{Total Changes} = \sum_{k=0}^{\lfloor \log_2 n \rfloor} \frac{n}{2^k}.$$

Simplifying the summation:

$$n \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{\lfloor \log_2 n \rfloor}} \right).$$

This is a geometric series with a sum approaching:

$$n \cdot 2 = 2n.$$

Thus, the total running time for counting from 1 to n in binary is $O(2n)$, which simplifies to $O(n)$ because constants are ignored in Big-O notation. So, the total running time is O(n).

8)

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

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

Show, by induction, that $T(n) = \frac{n(n+1)}{2}$.

Ans1.6.39
It is given that $T(n) = 1$ when $n = 1$, and otherwise:

$$T(n) = 2 \cdot T(n-1).$$

Let's compute a few values of $T(n)$:

$$T(2) = 2 \cdot T(1) = 2 \cdot 1 = 2^1,$$

$$T(3) = 2 \cdot T(2) = 2 \cdot 2 = 2^2,$$

$$T(4) = 2 \cdot T(3) = 2 \cdot 2^2 = 2^3.$$

By observing the pattern, for $n = k$:

$$T(k) = 2 \cdot T(k-1) = 2 \cdot 2^{k-2} = 2^{k-1}.$$

For $n = k + 1$:

$$T(k+1) = 2 \cdot T(k) = 2 \cdot 2^{k-1} = 2^k.$$

Thus, the formula holds true for all $n \geq 1$.

9) show that $\sum_{i=1}^{n} \log_2 i$ is $O(n \log n)$.

Ans1.6.52
To prove that $\sum_{i=1}^{n} \log_2 i$ is $O(n \log n)$, we proceed as follows:

Definition of Big-O

We know that $f(n)$ is $O(g(n))$ if:

$$f(n) \leq c \cdot g(n),$$

for some constant $c > 0$ and sufficiently large n.

Expanding the Summation

The summation $\sum_{i=1}^{n} \log_2 i$ can be written as:

$$\log_2 1 + \log_2 2 + \log_2 3 + \cdots + \log_2 (n-1) + \log_2 n.$$

This is equivalent to:

$$\sum_{i=1}^{n} \log_2 i = \log_2 (n!).$$

Comparing with $n \log n$

Using the properties of logarithms, we know:

$$n \log n = \log n + \log n + \cdots + \log n \quad \text{(repeated } n \text{ times)}.$$

Thus:

$$\log_2 1 + \log_2 2 + \cdots + \log_2 n \leq c \cdot (\log n + \log n + \cdots + \log n),$$

where $c = 1$.

Final Condition

For $c = 1$ and $n \geq 2$, this inequality holds. Hence:

$$\sum_{i=1}^{n} \log_2 i = O(n \log n).$$

10) 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.

Ans1.6.62
The size of the array is expanded from N to $N + \lceil N^{1/2} \rceil$.

Based on the amortization, each insertion will cost:

$$\frac{N + N^{1/2}}{N^{1/2}} = 1 + N^{1/2} \text{ cyber dollars.}$$

So, total insertion cost is as follow:

1+ 1+SQRT(N) = 2+SQRT(N)= 2n + 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 ($n^{3/2}$).

APPLICATION Questions

11)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
Ans1.6.70
Algorithm reverseArray(A, n):
Input: An array A storing $n \geq 1$ integers
Output: The reverse of an array A

middlePoint $\leftarrow \frac{n}{2}$
for $i \leftarrow 0$ to middlePoint $- 1$ do
temp $\leftarrow A[i]$
$A[i] \leftarrow A[n-i-1]$
$A[n-i-1] \leftarrow$ temp
return A

Steps:

1. We will be using the array data structure to store the n elements. Declare an array A and store the n elements.
2. Set the start variable as 0 and the end variable as $n - 1$.
3. Run a loop.
4. Inside the loop, declare a temp variable and store the value of A[start].
5. Copy the value of A[end] into A[start].
6. Copy the value of `temp` into A[end].
7. Repeat Steps 4 to 6 until start is not equal to end.
8. Return or print the array A.
9. Stop.

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 k1's. What is the running time of your method?
Ans1.6.77

Algorithm: `shortestSubarrayWithKOnes(A, k)`
Input: An array A of n bits, and an integer k ($k > 0$)
Output: The length of the shortest subarray containing exactly k 1's.

Step 1: Declare an array A of size n to store the elements.
Step 2: Declare a variable i and set its value to 0.
Step 3: Find the first occurrence of 1 in the array A from index i to $n - 1$, and store its index in i.
Step 4: Declare a variable j and start scanning from i to $n - 1$ until you find k 1's.
Step 5: Once k 1's are found, calculate the length of the subarray as $j - i + 1$.
Step 6: Discard the leftmost 1 at index i, and continue scanning the array until the next 1 is found.
Step 7: Continue scanning from index j to $n - 1$ until you find k 1's again.
Step 8: If the new subarray length is smaller than the previous one, update the length, i.e., new_length $= j - i + 1$.
Step 9: Return the shortest subarray length.

Time Complexity: $O(n)$
Space Complexity: $O(1)$