Jay Kalyanbhai Savani
20009207

ASSIGNMENT 1

Reinforcement

EXERCISE 1.6.7

(a) 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 $2^{g(n)}$.

Feedback?

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}

EXERCISE 1.6.9

(a) 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: Since the Find2D approach is a quadratic one rather than a linear one, its worst-case runtime complexity is O(n2). By analyzing the worst scenario, in which element x is the very last element to be inspected in the n*n array. Find2D in this situation repeatedly uses arrayFind. Then, until the last element where x is located, arrayFind will have to search through all n items for every call. For each arrayFind call, n comparisons are made. Because of this, our running time is O(n2) for n*n operations.

EXERCISE 1.6.22

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

Ans: If there is a constant n_0 >= 0 for any constant c > 0 and n >= n_0, then we say that n is o(n log n), and n < c * n log n for n >= n_0.

Therefore, using the formula 1/c log n, we pick $n_0 = 2^{1/c} + 1$. (When the log is the base of 2).

EXERCISE 1.6.23

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

Feedback?

Ans: Let c > 0 be any constant; there is a constant n_0 > 0 such that $n^2 > cn$. This one will imply that n^2 is w(n). So, n > c. Yet another approach is $n_0 = c + 1$.

EXERCISE 1.6.24

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

Feedback?

Ans: Finding a constant c > 0 and a constant n0 >= 1 that causes n^3 log n to exceed cn^3 will allow us to demonstrate the expression above.

The options are c = 1 and n_0 = 2. (Assume log is the base of 2).

EXERCISE | 1.6.32

(a) 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.

Feedback?

Ans: Based on Chernoff Bounds,

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

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

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

Creativity

EXERCISE 1.6.36

(a) 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 $i+1$?

Feedback?

Ans: t=time to change every single bit

let k=total bits.

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

The total running time is O(n).

EXERCISE 1.6.39

(a) 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 $T(n) = 2^n$.

Feedback?

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

EXERCISE 1.6.52

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

Feedback?

Ans: 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)

summation is O(n log(n)).

EXERCISE 1.6.62

(a) 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.

Feedback?

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 ($).

The total insertion cost:

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

that we can get is no more than:
$(2/3)n^{3/2} + (1/2)n^{1/2} - 1/6$

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

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

Application

EXERCISE | 1.6.70

(a) 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?

Feedback?

Ans: Algorithm reversal (start, end, n, A)

Initialize an array with n elements by starting at 0 and ending at n-1 in step one.

The Second Step is Swap in a loop (arr[start], arr[end])

The third step is Start Start + 1 End End -1.

There will be a time complexity O(n)

The preceding method will run through Step 1 in constant time (1), Step 2 in constant time, swapping in constant time, and Step 3 again in constant time (1). Consequently, the algorithm's overall running time is O(n). Because it only accesses each element of the array once, this algorithm would execute in O(n) time. It simply requires two variables to store the pointers, therefore there is no need for additional storage. The space complexity is therefore O(1).

EXERCISE 1.6.77

(a) 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?

Feedback?

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 others are variables that will take constant time O(1) to run.