CS-600-A Homework 1 - Solution

R-1.7

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.

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

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 $2^{g(n)}$.

Ordering the functions in the increasing level of their big-Oh complexity:

$1/n \;<\; 2^{100} \;<\; \log\log n \;<\; \sqrt{\log n} \;<\; \log^2 n \;<\; n^{0.01} < \underline{\lceil\sqrt{n}\rceil \;<\; 3n^{0.5}} \;<\; \underline{2^{\log_2 n} \;<\; 5n} \;<\; \underline{n\log_4 n \;<\; 6n\log n} \;<\; \lfloor 2n\log^2 n\rfloor < 4n^{3/2} \;<\; 4^{\log n} \;<\; n^2\log n \;<\; n^3 \;<\; 2^n \;<\; 4^n \;<\; 2^{2^n}$

R-1.9

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 1.12, 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?

The worst case running time for algorithm `find2D` will be $O(n^2)$ where the element to be searched will be stored at [*n*, *n*]. It is not a linear-time algorithm, instead it is a quadratic algorithm. Here the `arrayFind` as described in the text has a complexity of $O(n)$ and this algorithm is invoked for *n* times, making its complexity quadratic time.

Algorithm find2D may be written as follows:

```
Algorithm find2D (x, A):
Input: An element x and an n x n-element array, A.
Output: The index (i, j) such that x = A[i, j] or −1 if no element of A is equal to x.
i ← 0
while i<n do
        j = arrayFind  (x,A[i])
        if j != -1 then
                return (i, j)
else
        j = j + 1
return -1;
```

R-1.22
Show that *n* is *o(n log n)*.

Let $c > 0$ be any constant.
If we take $n_0 = 2^{1/c}$, $1/c = \log_2 n_0$, $1 = c \log n_0$, $n = cn \log n_0$.
Thus, if $n >= n_0$,

$f(n) = n <= cn < cn \log n$.
Thus, $f(n)$ is $o(n \log n)$.

R-1.23
Show that n^2 is $\omega(n)$.

Let $c > 0$ be any constant.
Thus, if $n >= n_0$, $n^2 >= nn_0$, $n^2 >= cn_0n$

$f(n) = n^2 >= cn_0n$, for $n_0 = c + 1$
Thus, $f(n)$ is $\omega(n)$.

R-1.24
Show that n^3 *log n* is $\Omega(n^3)$.

Let $c > 0$ be any constant.
For $n_0 >= 2$, $n^3 \log n >= n^3$.

Thus, if $n >= n_0$, $f(n) = n^3 \log n <= cn^3$, say c=1
Thus, $f(n)$ is $\Omega(n^3)$.

R-1.32
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.

Let X be the random variable that counts the number of heads.
$\mu = E(X) = n * n^{\frac{1}{2}} = n^{\frac{1}{2}}$

By Chernoff bounds, for $\delta = 2$, upper bound is
$$Pr(X \geq (1+\delta)\mu) = P\left(X \geq 3n^{\frac{1}{2}}\right) < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu} = \left[\frac{e^2}{3^3}\right]^{\sqrt{n}}$$

C-1.4

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?

Saying that we have an p bit register which represents $2^p = n$ numbers. Let k represent the number of trailing 0's after the last 1. Studying the nature of changing bits in terms of k.

If $p = 3$, the register will look like following:

k	**binary**	**# of bits changed**
not defined	000	-
0	001	1
1	010	2
0	011	1
2	100	3
0	101	1
1	110	2
0	111	1

So, the number of times $k = 0$, when p is $3 = 4 = 2^2 = 2^{p-k-1}$. In each such scenario, the number of bits changed when $k = 0$ is 1. So the total bits changed for $n = 2^p$ numbers $= (k + 1)2^{p-k-1}$.

Similarly, number of times $k = 1$, when p is $3 = 2 = 2^1 = 2^{p-k-1}$. In each such scenario, the number of bits changed when $k = 1$ is 2. So the total bits changed for $n = 2^p$ numbers $= (k + 1)2^{p-k-1}$. The number of times $k = 2$, when p is $3 = 1 = 2^0 = 2^{p-k-1}$. In each such scenario, the number of bits changed when $k = 2$ is 3. So the total bits changed for $n = 2^p$ numbers $= (k + 1)2^{p-k-1}$.

If $p = 3$, the possible values of k are 0, 1 and 2, i.e. $p – 1$.

Thus, total time the bits changed can be given by,

$$N = \sum_{k=0}^{p-1} (k+1)2^{p-k-1}$$

$$N = 1.2^{p-1} + 2.2^{p-2} + 3.2^{p-3} + 4.2^{p-4} + \cdots + p.2^0$$
$$N = 2^{p-1} + 2^{p-1} + (2+1)2^{p-3} + 2^2.2^{p-4} + \cdots + p$$
$$N = 2.2^{p-1} + 2^{p-2} + 2^{p-3} + 2^{p-2} + \cdots + p$$
$$N = 2^p + 2^{p-1} + 2^{p-3} + \cdots + p$$

As, $2^{p+1} = 2^p + 2^{p-1} + 2^{p-2} + 2^{p-3} + \cdots + 2^1 + 2^0$

We can write N as,

$$N = 2^{p+1} - p - 2$$

Since $2^p = n, p = log_2 n$,
$N = 2n - log_2 n - 2$ => O(n)

C-1.7
Consider the following recurrence equation, defining a function
$$\boldsymbol{T(n)} = \begin{cases} \mathbf{1}, & \boldsymbol{if\ n = 0} \\ \boldsymbol{2T(n-1)}, & \boldsymbol{otherwise} \end{cases}$$
Show, by induction, that $\boldsymbol{T(n)} = \mathbf{2}^{\boldsymbol{n}}$.

For $T(n)$,
$T(n) = 2.T(n-1)$

By Induction, $T(n-1) = 2.T(n-2)$
Thus, $T(n) = 2.2.T(n-2) = \cdots = 2.2^{n-1}.T(0) = 2^n$.

C-1.22
Show that the summation $\sum_{i=1}^{n} \lceil \boldsymbol{log_2(n/i)} \rceil$ is O($\boldsymbol{n}$). You may assume that *n* is a power of 2. Hint: Use induction to reduce the problem to that for *n*/2.

$$\begin{aligned}
\sum_{i=1}^{n} \left\lceil log_2\left(\frac{n}{i}\right) \right\rceil &= \sum_{i=1}^{n} \lceil log_2 n \rceil - \sum_{i=1}^{n} \lceil log_2 i \rceil && , by\ log\ a - log\ b = log\ a/b \\
&= n\ log\ n - \int_{i=1}^{n} log\ i\ \ di && , by\ \ log\ a + log\ b = log\ ab \\
&= n\ log\ n - (\ n \log n - n + c) && , using \int v\ du = uv - \int v\ du \\
&= n\ log\ n - n \log n + n - c && , by\ Stirling's\ approx, c => O(\log n) \\
&= n - c => O(n)
\end{aligned}$$

C-1.30
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 2*N*) when its capacity is reached, we copy the elements into an array with $\lceil \sqrt{\boldsymbol{N}} \rceil$ additional cells, going from capacity *N* to *N* + $\lceil \sqrt{\boldsymbol{N}} \rceil$. Show that performing a sequence of *n* add operations (that is, insertions at the end) runs in Θ($\boldsymbol{n^{3/2}}$) time in this case.

The extendable table implementation here grows form size *N* to $N + \sqrt{N}$.
According to Amortized analysis, each insertion cost on average will take $\frac{N+\sqrt{N}}{\sqrt{N}} = \sqrt{N} + 1$.
Thus, total insertion cost for such *n* add operations will cost,
$$T = \sum_{i=1}^{n} 1 + 1 + \sqrt{i} = \sum_{i=1}^{n} 2 + \sqrt{i} \leq 2n + \int_{i=1}^{n} i^{\frac{1}{2}} di = 2n + \frac{2}{3} n^{\frac{3}{2}} - \frac{2}{3} = O(n^{\frac{3}{2}})$$

Note that there is no closed formula for this summation that is why we used integration, but it is known that the sum
is greater than $\frac{2}{3}n^{\frac{3}{2}}+\frac{1}{2}n^{\frac{1}{2}}+\frac{1}{3}-\frac{1}{2}2^{\frac{1}{2}}$, and less than $\frac{2}{3}n^{\frac{3}{2}}+\frac{1}{2}n^{\frac{1}{2}}-\frac{1}{6}$.

So, the total cost of performing n add operations is $\theta(n^{\frac{3}{2}})$.

So, the total cost of performing n add operations is $\theta(n^{\frac{3}{2}})$.

A-1.8
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?

The most efficient way requires a linear time since all the element in the array are going to be accessed once.

```
Algorithm reverse (A):
Input: An n-element array, A.
Output: Reverse of array A.
i ← 0
j ← n-1
while i<j do
        temp = A[i]
        A[i] = A[j]
        A[j] = temp
        i ← i + 1
        j ← j - 1
return A;
```

Since this program requires only an additional space for storing 3 values, *i, j* and *temp* => O(1). We have an efficient algorithm with complexity O(*n*).

A-1.15
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?

Assuming we have an algorithm countOne(A, i, j) which return the number of 1's in array A from index i to j.

There is a single loop here which traverses through entire array using two different pointers and thus it is a linear time algorithm with complexity O(*n*).

The shortestSubarray algorithm can be written as following:

Algorithm shortestSubarray (A):
Input: An n-element array, A.
Output: The shortest subarray of A that contains *k* 1’s.

```
i ← 1
j ← 2
while j < n do
        if (countOne(A, i, j) = k) then
                localMinLen = j - i + 1
        j ← j + 1
        if (A[j] = 1) then
                do i ← i + 1
                while A[i] != 1
                globalMinLen = min( j - i + 1, localMinLen)
return globalMinLen;
```