Name: Thaarani Subramaniam

CWID: 20009182

Homework 1

Question 1.6.7

Solution:

The increasing level of big(O) Complexity:

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

Question 1.6.9

Solution:

The worst-case running time of the algorithm is n^2 which is quadratic. Because of arrayFind being called n times and traversing n elements each time until the target(X) is obtained.
The worst case occurs in two situations:

- when the element occurs at the end of the 2D array(A).
- When the element that we are searching for is not found.

In these cases, we must traverse the entire 2D array. so, the time grows quadratically, it is not linear.

Algorithm find2D(A,x)
Input : An element x and an array A which contains n*n elements
Output : The index(i,j) if the element x is present in the array (x = A[i][j]) . In case if the element is not present return -1
i ← 0

```
while i < n do
        j = arrayFind(A[i],x)
        if j != -1 then
                return (i , j)
        else
                j = j+1
return -1
```

Question 1.6.22 - Show that n is O(nlog n).

Solution.

Let's take any constant, c > 0

Let's consider $n_0 = 2^{\frac{1}{c}}$

Taking $\log_2$ on both sides $\log_2 n_0 = \log_2 2^{\frac{1}{c}}$

Simplified as $1/c = \log_2 n_0$, $1 = c\log_2 n_0$

Multiplying n on both sides,

$n1 = cn\log_2 n_0$

Therefore, if $n >= n_0$,

f(n) = n <= nlogn <=cnlogn

Question 1.6.23 - Show that n^2 is ω(n).

Solution.

Let's take any constant, c > 0

If $n_0 = c+1$,

Then for $n >= n_0$

$n^2 >= n * n_0$

$n^2 >= c * n * n_0$

Therefore if $n > = n_0$,

$f(n) = n^2 >= cnn_0$ ($n_0 = c+1$,)

So n^2 is ω(n).

Question 1.6.24 - Show that n^3log n is Ω(n^3).

Solution.

Let's take any constant, c > 0

For $n_0 >= 2$, $f(n) = n^3\log n >= n^3$

Consider c = 1 and n_0 =2(suppose log is the base of 2),

Therefore, if $n >= n_0$,

$f(n) = n^3\log n <= cn^3$ (c = 1)

Therefore n^3log n is Ω(n^3).

Question 1.6.32

Solution.

Let's consider x as the random variable that counts the number of heads

$\mu = E(X) = \sum_{i=1}^{n} p_i = n * \frac{1}{n^{1/2}} = n^{1/2}$

For $\delta = 2$, the upper bound is

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

Question 1.6.36

Solution.

Declaring that a 2p = n integer can be represented by a p bit register. Let k be the number of trailing 0s following the final 1. The register will appear as follows if p = 3. Analyzing the nature of bit changing based on k.

k	Binary	No 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 scenario like this, the number of bits changed when k = 0 is 1.

The total bits changed for n =2^p= (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^pnumbers = (k + 1) 2^{p-k-1} . The number of times k = 2, when p is 3 = 1 = $2^0 = 2^{p-k-1}$

If p = 5, the possible values of k are 0, 1 ,3 and 4, i.e. p – 1.

Therefore total time the bits changed can be given by,

k=0

$N = \sum(k+1)\, 2^{p-k-1}$.

p−1

$N = 1*\ 2\ 2^{p-1} + 2*2\ 2^{p-2}. + 3*\ 2\ 2^{p-3}. + \cdots + p\ *2^{0}$

$N = 2*\ 2^{p-1}+2^{p-2} + 2^{p-3} + 2^{p-2}.\ + \cdots + p$

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

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

N can be writtern as,

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

$2\ p = n,\ p = log2\ n,\ N = 2n - log2\ n - 2$ => O(n)

Question 1.6.39

Solution.

For T(n),

T(n) = 2*T(n-1)

By induction,

$T(0) = 1 = 2^{0}$

$T(1) = 2 * T(0) = 2 = 2^{1}$

$T(2) = 2 * T(1) = 4 = 2^{2}$

T(n-1) = 2 * T(n-2)

Therefore, T(n) = 2*2*T(n-2) = … = $2*2^{n-1}$*T(0) = 2^{n}

Question 1.6.52

Solution.

$$\sum_{i=1}^{n} \log_2 i = \log_2(1) + \log_2(2) + \log_2(3) + \log_2(4) + \log_2(5) + \cdots \log_2(n-1) + \log_2(n) = n\log_2 n$$

Thus,

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

Question 1.6.62

Solution.

The extendable table implementation grows in size from N to $N + \sqrt{N}$.

Based on amortized analysis, each insertion cost, on average, takes $(N+\sqrt{N})/\sqrt{N} = \sqrt{N} + 1$.

Therefore, the total insertion cost for n such add operations will cost,

$$T = \sum_{i=1}^{n} 1 + 1 + \sqrt{\mathrm{i}} = \sum_{i=1}^{n} 2 + \sqrt{\mathrm{i}} \leq 2n + \int_{i=0}^{n} i^{1/2} d_i = 2n + \frac{2}{3} n^{3/2} = n^{3/2}$$

However, there is no closed formula for this summation. This is why, we used integration. But, it is known that the sum is greater than $\frac{2}{3}n^{3/2} + \frac{1}{2}n^{1/2} + \frac{1}{3} - \frac{1}{2} * 2^{\frac{1}{2}}$, and less than $\frac{2}{3}n^{3/2} + \frac{1}{2}n^{1/2} - \frac{1}{6}$. Hence, the total cost of performing n add operations is $\theta(n^{3/2})$.

Question 1.6.70

Solution.

We have to access all the elements in the array at least once to reverse it.

- Take two pointers i and j. one pointer i will be pointing at the beginning of the array and another pointer j will be pointing at the end of the array.
- Swap the elements in i and j, after that increment the i pointer and decrement the j pointer. Repeat the steps until pointer i is lesser than j.

Algorithm reverse (A):

Input : An array A

Output: Reverse of an array A

```
i ← 0
j ← n-1
while i < j do
        t = A[i]
        A[i] = A[j]
        A[j] = t
        i ← i+1
        j ← j-1
```

return A

Since we are going through all the elements in the array the time complexity of the algorithm is O(n) and the space complexity is O(1) as we not using any extra space .Variables i,j and t will take only constant space.

Question 1.6.77

Solution.

We determine the index position of each one in array A and store that value in a different array. Its complexity is O(n).

Then we have to compare the length staring from index position to next k 1's from the index position array and traverses through entire array by updating the length. Its complexity is O(n).

The running time is O(n) because i traverse once as well as j traverse the array once

Algorithm

Shortest Subarray (A): Input: An array, A and an integer k>0.

Output: Shortest subarray in A containing k 1's.

```
x ← 0
for i ← 0 to n-1 do
        if A[i]==1 then
        index[x] = i
        x ← x+1
for j ← 0 to n-1 do
`       ptr ← j+k-1
        len = index[ptr] – index[j]
        if len < minlen
                minlen ← len
                start ← j
return (start, start+k-1)
```