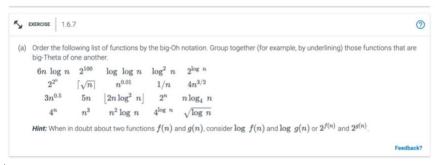
### Jay Kalyanbhai Savani 20009207

### **ASSIGNMENT 1**

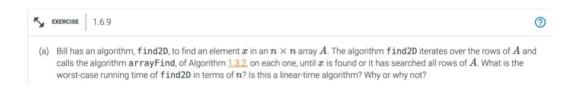
### Reinforcement



### 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)  $n \log_4 n$ ,  $6 n \log n$
- $10)2 \text{ n } \log^2 \text{n}$
- $11)4 n^{3/2}$
- $12)4^{\log n}$
- $13)n^2 \log n$
- $14)n^3$
- $15)2^{n}$
- 16)4<sup>n</sup>
- $17)2^{2n}$



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.



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

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

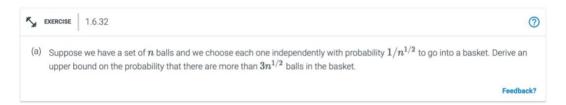


Ans: Let c > 0 be any constant; there is a constant  $n_0 > 0$  such that  $n^2 > cn$ . This one will imply that  $n^2$  is w(n). So, n > c. Yet another approach is  $n_0 = c + 1$ .



Ans: Finding a constant c > 0 and a constant  $n_0 >= 1$  that causes  $n_0^3 \log n$  to exceed  $cn_0^3$  will allow us to demonstrate the expression above.

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



Ans: Based on Chernoff Bounds,

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

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

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

# Creativity



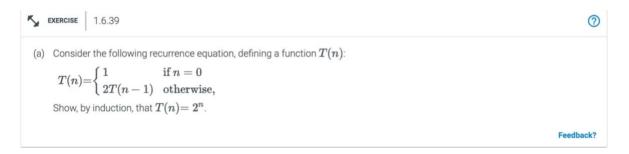
Ans: t=time to change every single bit

let k=total bits.

The total work is:

$$t*(n/2^0 + n/2^1 + n/2^2... + n/2^k) < t*n*2 => O(n)$$

The total running time is O(n).



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



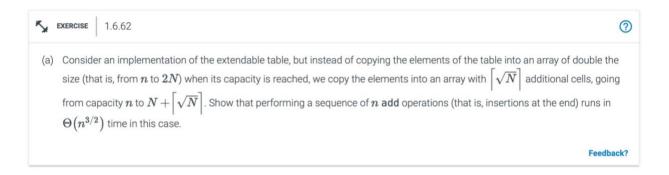
Ans: Assume that the base to all log used is 2.

## **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  $O(n \log(n))$ .



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:

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

that we can get is no more than:

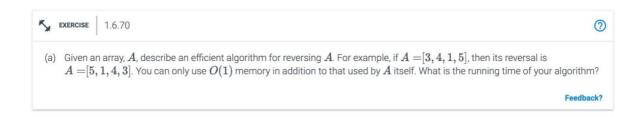
$$(2/3)n^{3/2} + (1/2)n^{1/2} - 1/6$$

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**



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

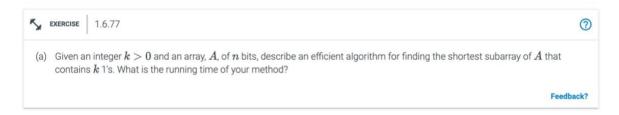
Initialize an array with n elements by starting at 0 and ending at n-1 in step one.

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).



### Ans:

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

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

```
\begin{aligned} & \text{Count} \leftarrow 0 \\ & \text{$k \leftarrow 0$} \quad \text{//maximum found so far} \\ & \text{for } i \leftarrow 1 \text{ to n do} \\ & & \text{if } A[i] = 0 \text{ then} \\ & & \text{count} \leftarrow 0 \\ & & \text{else} \\ & & \text{count} \leftarrow \text{count} + 1 \\ & & \text{$k \leftarrow \text{max}(k, \text{count})$} \end{aligned}
```

### Run Time:

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

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.