

**Solution**: Order the functions in Big- Oh notation

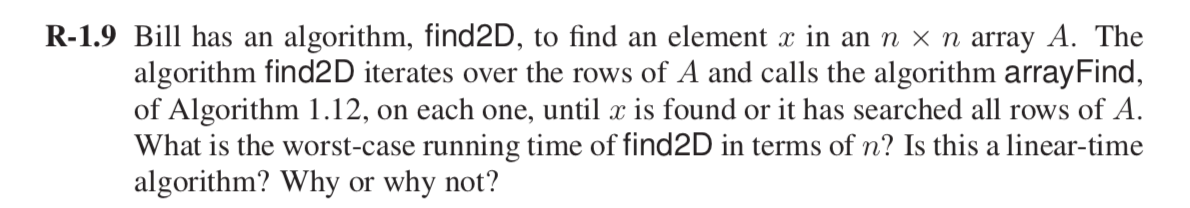
1. 1/n (Takes less than O(1))
2. 2100 (O(1))
3. log log n (O(log n ))
4. (O(log n))
5. log2 n (O(log n))
6. (O(n))
7. (O(n))
8. (O(n))
9. (O(n))
10. (O(n))
11. n log4 n (O (n log n))
12. 6 n log n (O (n log n))
13. 2n log2 n (O(n log n))
14. (O(n1.5))
15. (O (n2))
16. log n (O (n2))
17. (O (n3))
18. (O (2n))
19. (O (22n))
20. 22∧n (O (2n))

Functions that are big-Theta of one another

and

and

n log4 n and 6 n log n



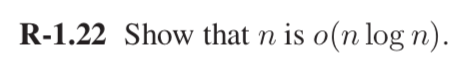
**Solution**

Find2D(x, A), arrayFind(x, A)

*Input:* An element x and an n-element array, A.

For this scenario Find2D algorithm will call the arrayFind algorithm *n* times and arrayFind will have to search all the elements until x is found every time the was made. This process will run n times for each arrayFind. So, both the process in together will have n \* n operations that will result to O(n2) running time of Find2D.

No, this is not a linear time algorithm, it is quadratic. To be a linear time algorithm, the running time would be proportional to the input size i.e. n \* n.



**Solution:**

By using limit of the fraction format:

= 0

For any c, and for sufficiently large n, f(n) < c. g(n)

Thus *n* is *o(n log n).*



**Solution:**

By using limit of the fraction format:

=

For any c > 0, and for sufficiently large n, f(n) > c. g(n)

Thus *n2* is *ω(n).*



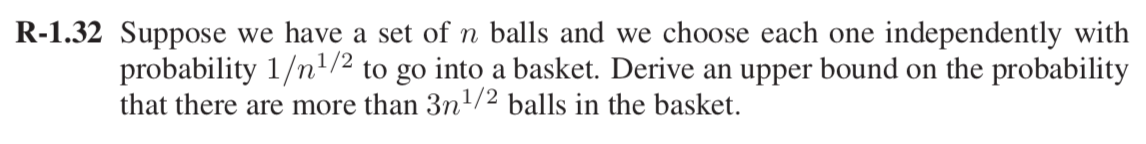
**Solution:**

By using limit of the fraction format:

=

Then intuitively f(n) > g(n),

Therefore, f(n) = Ω g(n)



**Solution:** Using Markov Inequality

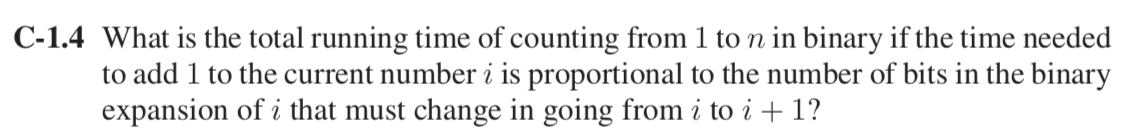
Let X be the number balls go into the basket.

P[Xi =1] = 1/sqrt n

E(X) =

P[X >= 3 ] <= 1/3

This shows that there are more than 3 balls in the basket.



**Solution:** For example (Used stack exchange for the reference )

Binary representation of the numbers for:

i=0 0000

i=1, 0001

i=2, 0010

i=3, 0011

i=4, 0100

i=5, 0101

i=6, 0110

i=7, 0111

i=8, 1000

Let T(n) be the running time of counting 1 to n in binary.

So according to above pattern, taking from Least Significant Bit(LSB):

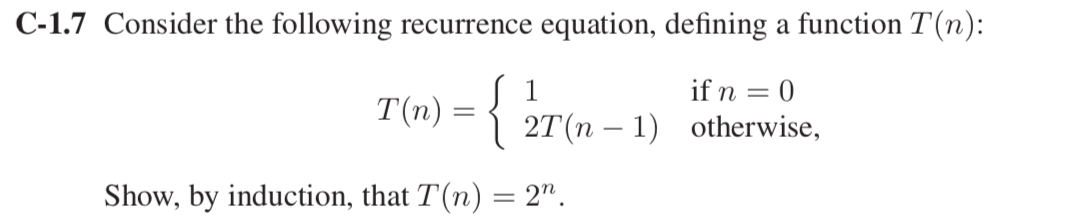
* 20 i.e. ones bit changes with every i to i+1, so it switching 0 to 1 and again to 0 in every *n* times.
* 21 i.e. second bit from left changes for every i to i+2, so it switching 0 to 1 in every *n/2* times.
* 22 i.e. third bit from left changes for every i to i+4, so it switching 0 to 1 in every *n/4* times. And so on.

So total running time T(n) =

=

= n (If n ∞, = 1)

Running time T(n) = O(n)



**Solution:**

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

To prove by induction method

Let T(n) = 2n

Put n= 0

T(0) = 20 = 1 (Condition holds true)

Now check for other values:

Suppose T(n) satisfies the recurrence T(n) = 2k T(n-k) + k for any positive integer *k.*

Check the above recurrence relation putting k=1

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

Now check for k-1

T(n) = 2k-1 T(n-(k-1)) + (k-1) +1 (By Induction Hypothesis)

= 2k-1 (2T(n-k) +1) + k (recurrence for T(n-(k-1)))

= 2k T(n-k) + k

Now putting the value of *k* in terms of *n* will give the following equation:

T(n) = 2n T(n-n) + n

T(n) = 2n T(0) + n

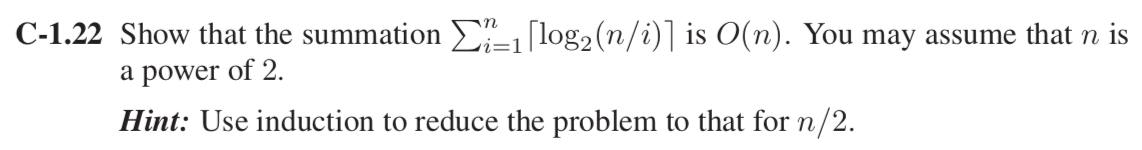
Given the value of T(0) = 1 in the question

T(n) = 2n + n

For higher values of 2n neglecting *n*

Thus T(n) = 2n

Hence Proved.



**Solution:**

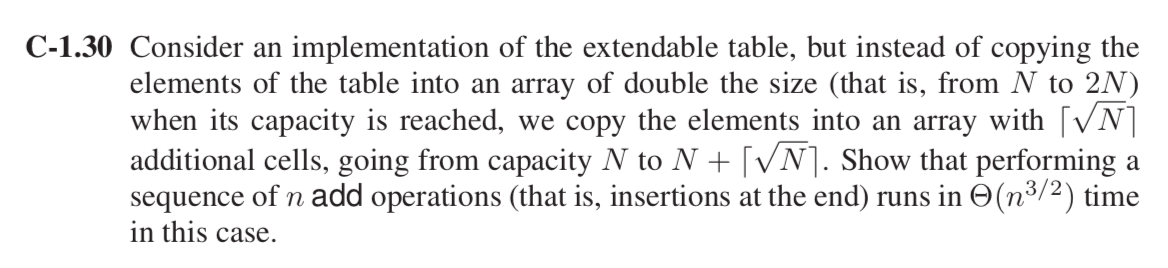
=

=

(Using Stirling’s Approximation)

=

= O(n)



**Solution:** Capacity of the array extended from N to N+

Average insertion cost for 1 insertion =

= 1 +

Total insertion cost for n operations=

= n +

= n + + c

= + c

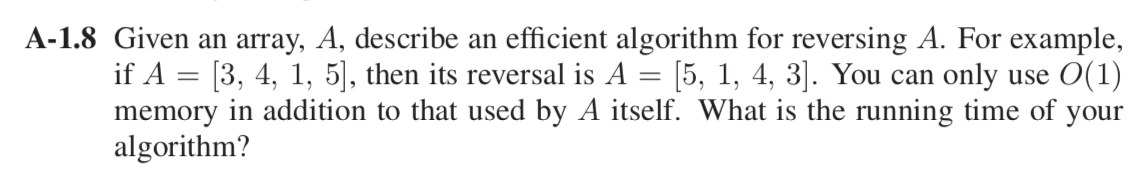
Now, f(n) = (for large values of n)

Big - Theta

f(n) = Θg(n) for constant c’> 0 and c’’> 0 and an integer constant n0 ≥ 1 such that

c’ g(n) ≤ f(n) ≤ c’’ g(n), for n ≥ n0.

This shows that insertion of **n** *add* operation runs in Θ()



**Solution**

Algorithm reversal (start, end, n, A)

Step1: Initialize

n – number of elements in an array

start 0

end n-1

Step2: In a loop,

swap (arr[start], arr[end])

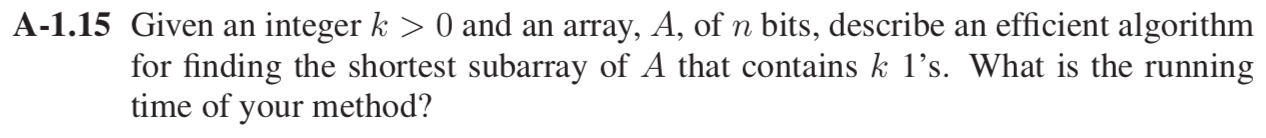
Step3: Start start + 1

End end -1

Time complexity: O(n)

The above algorithm step1 will execute in constant time (1) and the second step will take *n* time , swapping will take constant time and the Step3 again will take the constant time (1).

So, the total running time of the algorithm is O(n).



**Solution:** Algorithm shortestSubArray(k, n, A)

***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

Explanation: This algorithm calculates the partial number of 1’s of every possible subarray. For every subarray k(total number of 1’s come together), it compares that k to a running maximum and if the new value is greater than the old, it updates the maximum to the new value. In the end k will consists of shortest subarray for k 1’s.

**Running 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 running 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 other are variables which will take constant time O(1) to run.