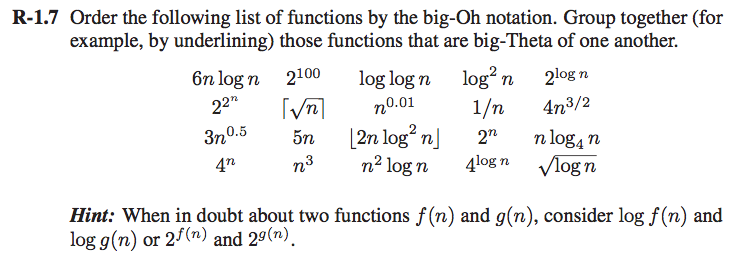
CS 600 Homework 1 | CWID 10430147 | Divyendra Patil | Username: dpatil3  
Date: 08/31/2017



**Solution to R-1.7:**

The following is in increasing order of their big-Oh notation.

1/n

2100

log (logn)

(logn)1/2

log2 n

n0.01

Ceiling of n1/2 = 3n0.5

2logn = 5n

nlog4 n = 6nlogn

[ 2nlog2 n ]

4n3/2

4logn

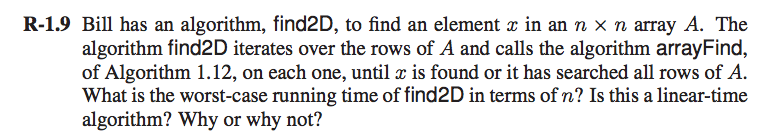
n2 logn

n3

2n

4n

22^n



**Solution to R-1.9:**

If we consider case of the algorithm arrayFind (1.12), it runs from i to n i.e it is called n times since the algorithm has to search for all elements in the array sequentially/linearly ie. O(n) running time.

Similarly, for bill’s algorithm find2D, it will have a total running time **O(n^2)** i.e for each time arrayFind runs for **each row** it will make n operations. (n x n operations). Since it has a running time of O(n^2), **it is not linear but Quadratic time algorithm**.

Eg: Algorithm Find2D(x, A)

**Input: Array A of n x n size and element x to be searched in an array**

**Output : Index i where element is found.**

**for** i 1 **to** n – 1 **do**

**for** j 1 **to** n – 1 **do**

**if** x == A[i][j]

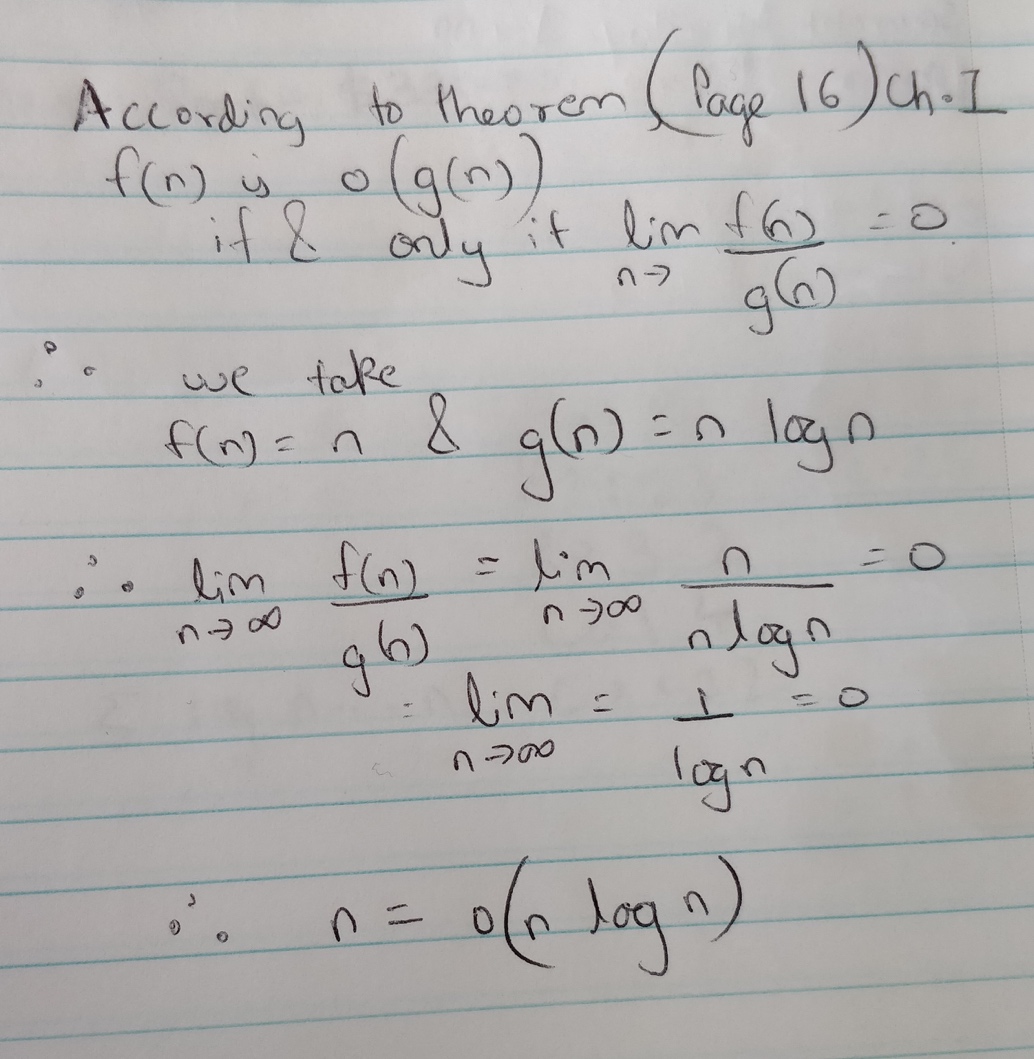
return i

// For each “i”, we will run j times. Therefore, (n x n) = n^2.

/Users/divyendrapatil/Desktop/Screen Shot 2017-09-01 at 11.34.44 AM.png

Solution to R-1.22:

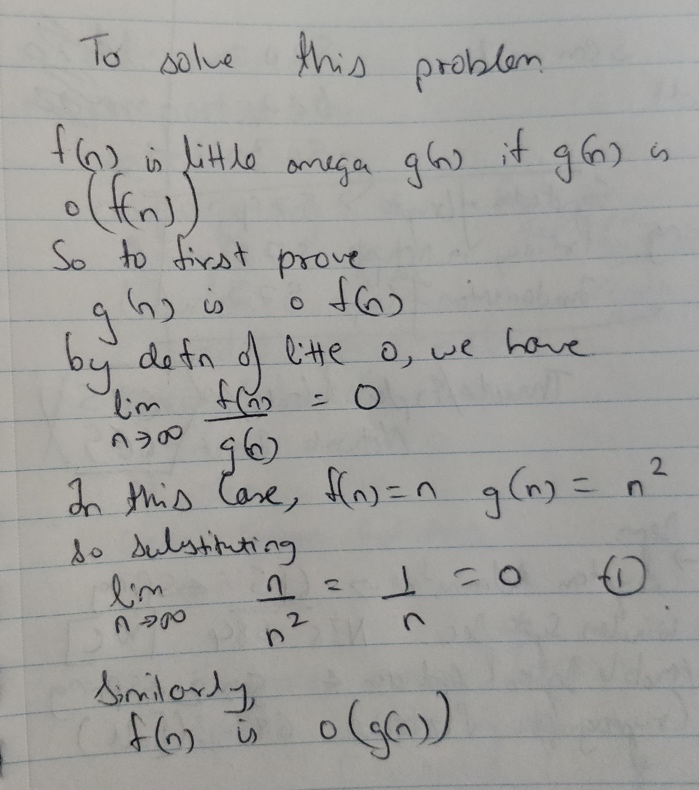
We have to show n = o(n log n)

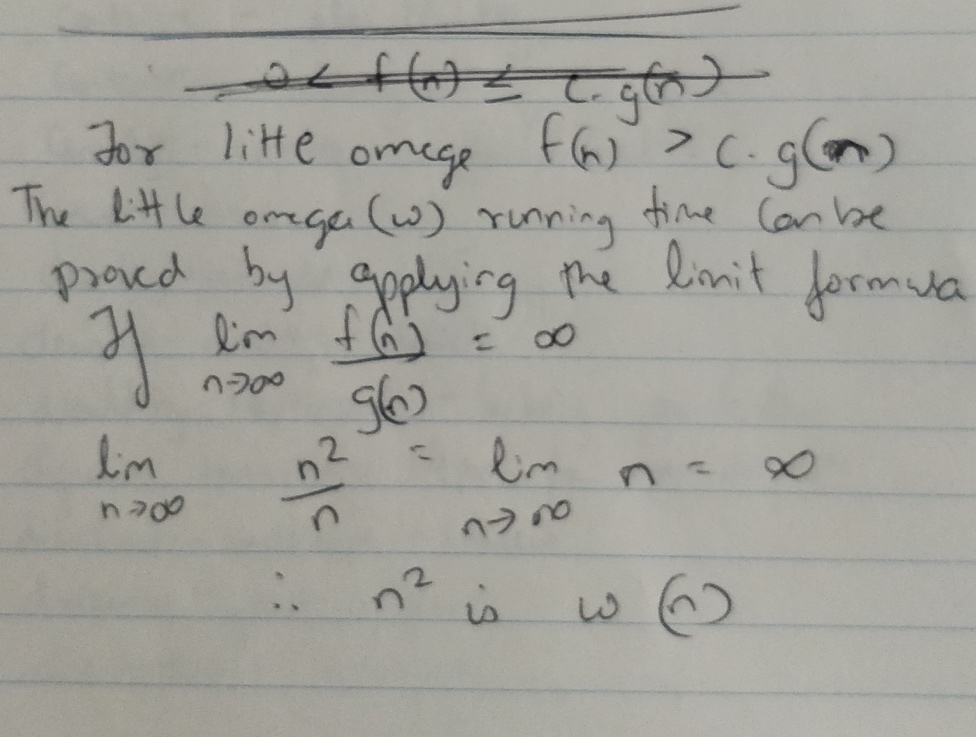
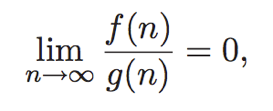


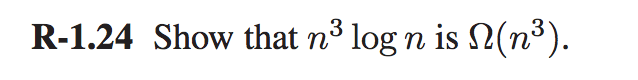
Screen%20Shot%202017-09-01%20at%2012.09.42%20PM.png

Solution to R-1.23:

Solution I

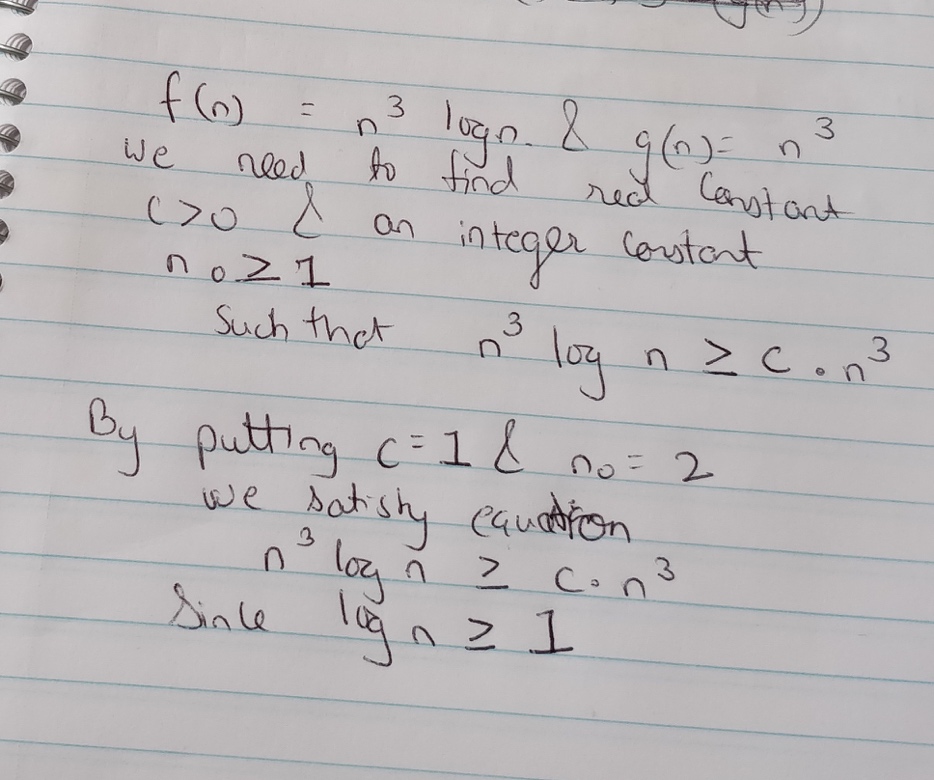


Solution II:  
  
Comments: By definition, f(n) = ω(g(n)), if g(n) is o(f(n)) & by definition of little o we have   
  
(Look at solution 1). So from solution 1, we proved that g(n) is in fact f(n) which eventually proves that f(n) = ω(g(n)).

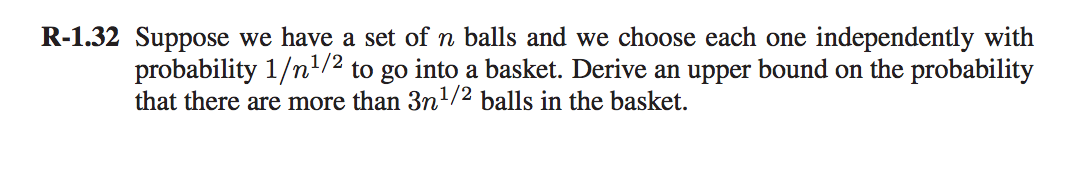


Solution to R-1.24:

We say that f(n) is Ω(g(n)) (pronounced “f(n) is big-Omega of g(n)”) if g(n) is O(f(n)); that is, there is a real constant c > 0 and an integer constant n0 ≥ 1 such that f(n) ≥ cg(n), for n ≥ n0. (Page 14: Concept of Big-Omega and Big-Theta)

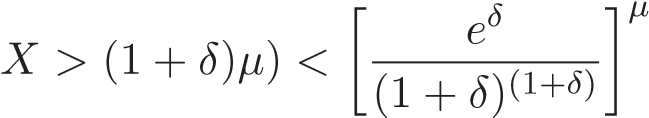


Therefore, we can say that n3 log n is Ω(n3).



Solution to R-1.32:

**According to Chernoff’s, we have the following for *δ >* 0,**

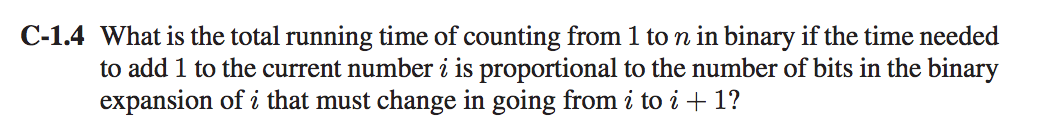


**We have µ as:**

µ = E(X) = X \* P(X) = n \* 1/ √n = √n

**Substituting the value of *δ =* 2 and µ=√n, we have the conclusion as:**

Pr(X >3µ) < [e2/33]√n



Solution to C-1.4:

If N = 3, we write 3 in binary as “11”.   
1 = 01, 2=10, 3=11 (Basic Binary Conversion)

If we add any number eg (1 = 01) to any of these binary numbers, the number of bits get **changed**.

The first bit changes n times, the second bit changes n/2 times, third bit changes n/3 times and so on as the numbers are kept adding.

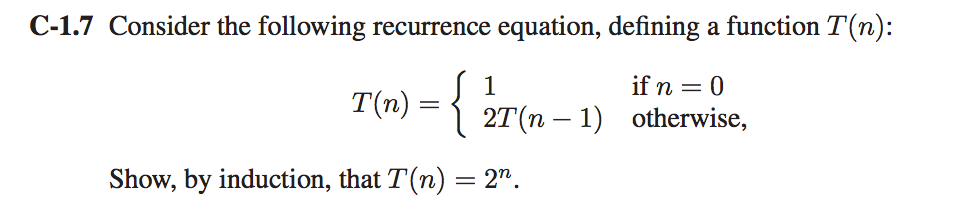
For eg 01+01 = 10 (Number of bits changed is 2)

Now the total running time can be calculated as  
T\* ∑ n/ 2i  = T\* n \* ∑ n/ 2i  < T\*n\*2 = **O(n)**

i=0 to k

or in simple terms: n + n/2 + n/4 and so on   
but we consider now only n + n/2 = (3n)/2.

If we eradicate the constants in this equation what we get is **O(n)** which as above is basically the number of bits that are changing in the sequence.



Solution to C-1.7:

If n=0, T(0) = 1; (Eq 1)

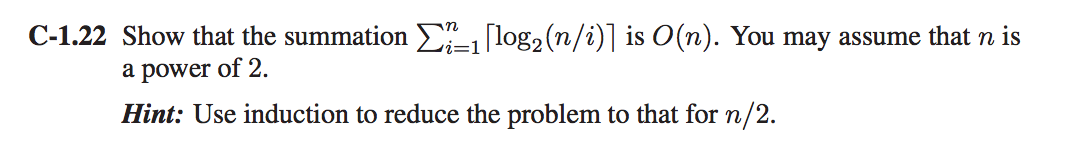
Then calculating on similar basis,

n=1, T(n)=2T(n-1)=2T(1-1)=2T(0)=2 x 1 = 2 [Since T(0)=1 (From Eq (1)] – (Eq 2)

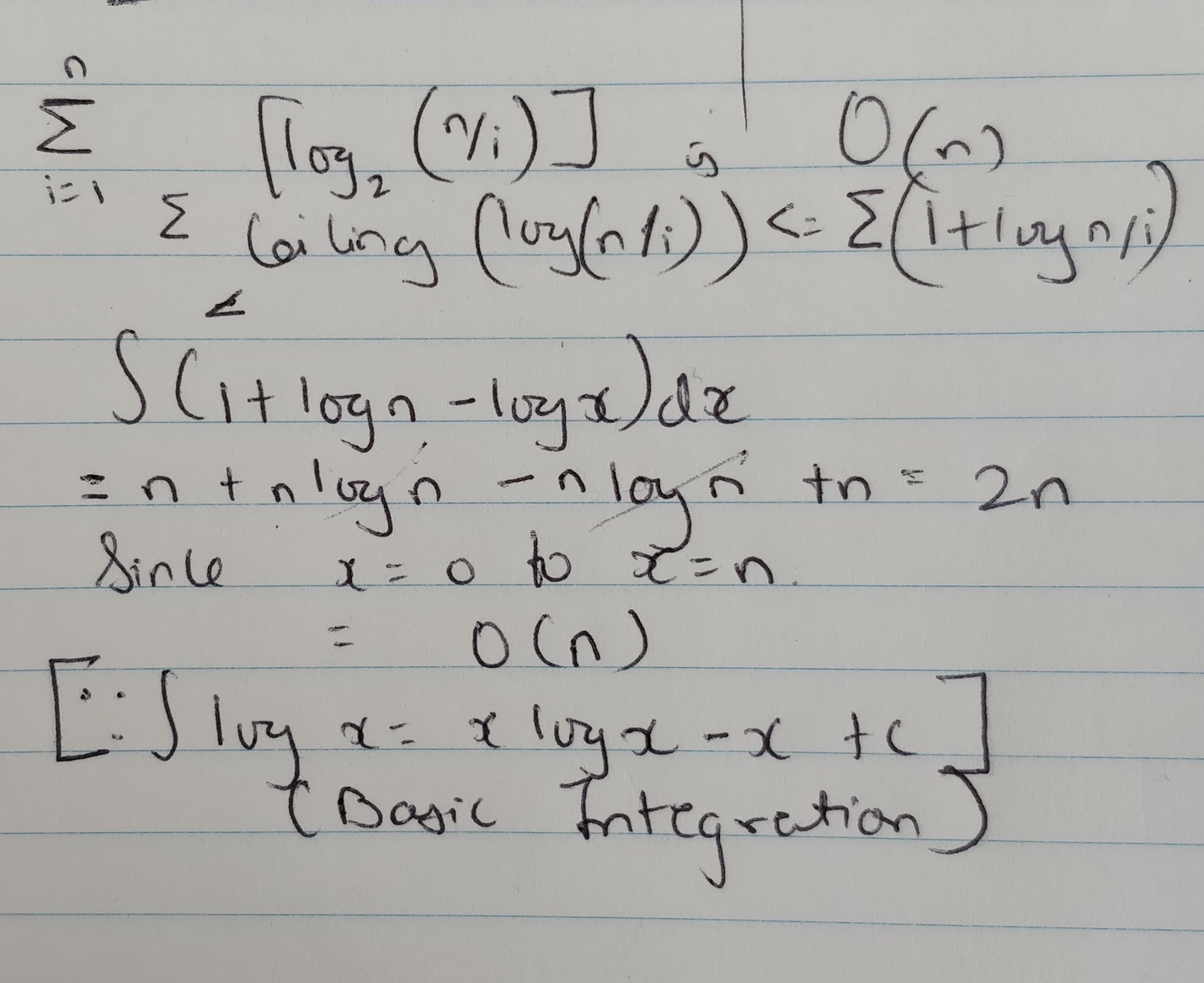
n=2, T(n)=2T(n-1)=2T(2-1)=2T(1)=2 x 2 =4 [From Eq 2]  
n=3, T(3)=8  
n=4, T(4)=16

The values we received 2,4,8,16 which is basically a GM (Geometric Progression).  
The nth term of the progression is given by T(n) = 2n

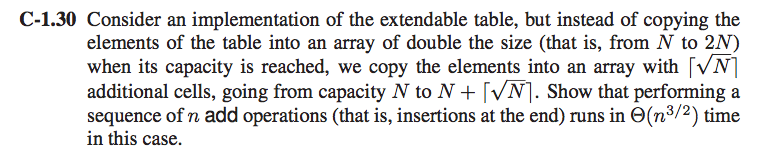
And also 2n i.e here n = 4 so, 24 = 16.  
Hence Proved that for any n term, T(n) = 2^n.



Solution to C-1.22:



f(n) ≤ g(n) and hence is O(n)



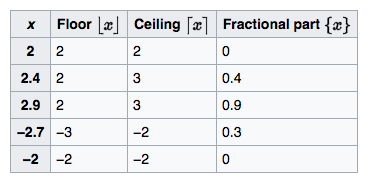
**Solution 1 to C-1.30**:

We use the concept of amortization to solve this problem.

When an overflow occurs when adding a value, the size of the array is to be increased from “n” to “n + ceil ***√***n”.  
  
If we calculate the value of insertions as follows:

1 => 1 + ceil √ 1 = 2 (Overflow) | C = 1  
1 2 => 2 + ceil √ 2 = 4 (Overflow) | C = 2

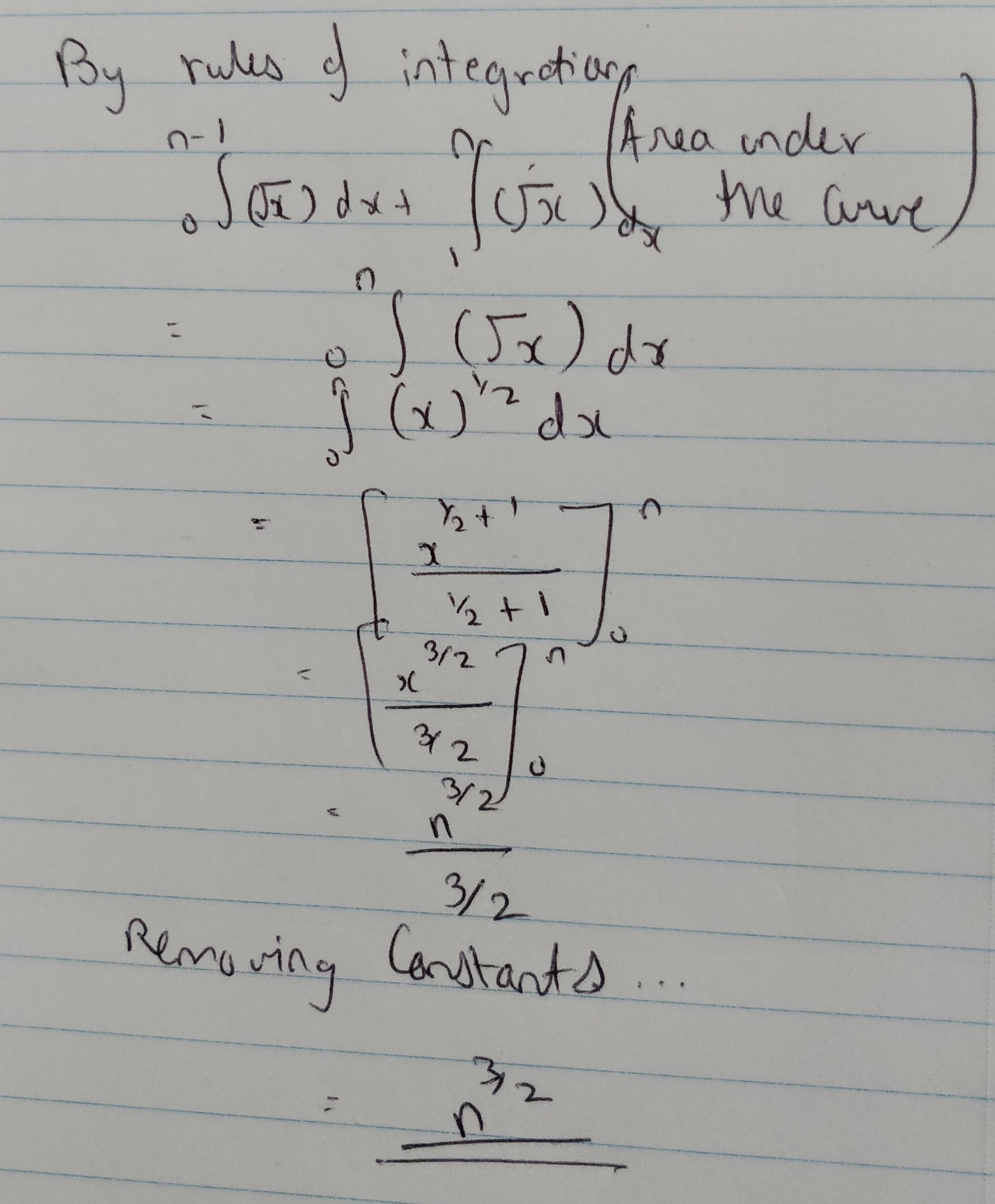
1 2 3 (No overflow) | C = 4

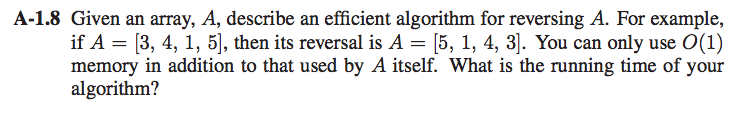
1 2 3 4 => 4 + ceil √ 4 = 6 (Overflow) | C = 1  
(C is number of new elements in the array)  
Similarly the rest will also follow.  
  
  
(Only for reference)

The progression turns out to be of this sort by adding 1 + √ n

∑ 1+ 1+ √ N = ∑ 2+ √ N = 2n + ∑ √ N from N=1 to N=n  
The summation of series 1+2+3+1+5+1+7+1…… is **Θ(*n*3*/*2)**

**Solution 2:**

  
Hence the summation of series is **Θ(n3/2).**



Solution to A-1.8:

**Algorithm:** ArrayReverse(A)

**Input:** An array A of size N.

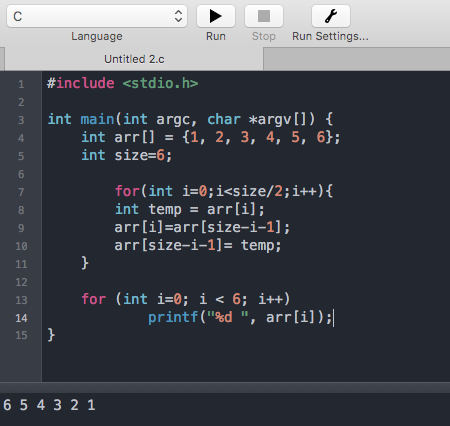
**Output:** Reversed array.

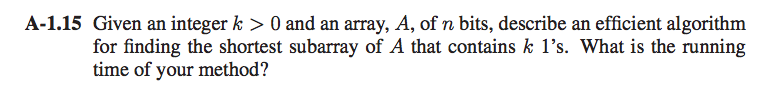
**for** r ← 1 to N/2 **do**

temp ← A[r]  
 A[r] = A[N - r - 1]

A[N - r - 1] = temp

Running time complexity of the above algorithm is **O(N)**

  
(\*Only for Reference)



Solution to A-1.15:

The solution is rather simple by using pointers in the array. If we consider two pointers a & b A[a:b] and **a** & **b** are as close to each other as possible having k 1’s. Each time we increment **b**, we need to increment **a** to keep them as close as possible to maintain **A[a:b]** to keep **k 1’s. We can charge 2n cyber dollars for each operation when we increment either a or b (to pay for operations).** Hence the total running time is O(n).