***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.

6*n* log *n* 2100 log log *n* log2 *n* 2log *n*

22*n \_*

*√*

*n\_ n*0*.*01 1*/n* 4*n*3*/*2

3*n*0*.*5 5*n* 2*n* log2 *n\_* 2*n n* log4 *n*

4*n n*3 *n*2 log *n* 4log *n*

*√*

log *n*

*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*).

**Answer:**

* 1/*n*
* 2100
* log log *n*
* √log *n*
* log2*n*
* *n*0.01
* ┌√n┐, 3 *n*0.5
* 2log*n*, 5 *n*
* n log4n, 6 n log n
* |\_2 *n* log2 *n\_|*
* 4 *n*3/2
* 4log*n*
* *n*2 log *n*
* *n*3
* 2*n*
* 4*n*
* 22*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?

**Answer:** The worst case running time of the algorithm find2D is O (n2).

Suppose there is an element X as the last value in n\* n array. Here find2D would call the algorithm arrayFind n number of times and it would search the elements until X is found in the array. So, n number of comparisons are executed for every time arrayFind runs. Therefor the running time for this algorithm is n \* n which is O (n2) running time. Here the size of N of A is n2 which shows it is O(N) time algorithm. This algorithm hence is a linear time algorithm. As it is running time is same as the linear function of input size.

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

**Answer:**

F(n) = o g (n)

F(n)/ g(n) 0 for n ∞

F(n) /g(n) = n/ n log n =0 for n ∞

Therefor n = o (n log n).

Or

Via the little o definition, there exist a constant c>0, where n0 >0 | n < c n log n.

The log n is > 1/c for n ≥ n0 is true if n >21/c. So, n0 = ceiling of (1 + 21/c).

***R-1.23*** Show that *n*2 is *ω*(*n*).

**Answer:** F(n) = *ω* g (n)

F(n)/ g(n) 0 for n ∞

F(n) /g(n) = n2/ n

Therefor *n*2 is *ω*(*n*).

***R-1.24*** Show that *n*3 log *n* is Ω(*n*3).

**Answer:**

According to the definition of big-Omega, the real constant is supposed to be searched c>0 and integer constant n0 ≥ 1 such that *n*3 log *n* is Ω(*n*3).

So, when c =1 & n0 =2log n ≥ 1 in the range.

***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 3*n*1*/*2 balls in the basket.

**Answer:**

P =1 / n1/2

here binomial distribution is used as number of trails is n and it is independent.

So, let the probability of getting selection in the basket be p.

P (X =x) = (nx) px qn-x

P (X =x) = (nx) (1 / n1/2)x ( 1 – (1 / n)1/2) n-x

Using Chernoff bound E(X) = n \* (1 / n(1/2)) = n1/2

P(Xi =1) = 1/ n1/2

The upper bound on the probability is more than 3 n1/2 balls in the basket

Is P(X≥3 n1/2) =P ((X≥ n1/2(1+2))

= e-(2n\*\*(1/2)ln2) /2

***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?

**Answer:**

Suppose n = 16, then when 1 to 16 is counter binary is as follow:

0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 0110 1110 1111 10000

As shown above the right bit changes as we count to 16 times, the second bit changes every 4 times, the third bit changes 4 times and the last bit changes twice. We can assume n is the power of 2 and hence 2x =n so if it is 0 to n. the first bit changes n times. The second changes n/2 and so on.

Let’s call work required to change a bit a W

W \* ∑i=0 to x n/2i = w \* n \* ∑ n/2i < 2 \* n \* 2 = O(n)

***C-1.7*** Consider the following recurrence equation, defining a function *T*(*n*):

*T*(*n*) =

\_

1 if *n* = 0

2*T*(*n −* 1) otherwise,

Show, by induction, that *T*(*n*) = 2*n*.

**Answer:**

Here n = 1, then T(1) = 2, T(0) =2 is true.

So, assume true for n-1

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

***C-1.22*** Show that the summation

\_*n*

*i*=1

*\_*log2(*n/i*)*\_* is *O*(*n*). You may assume that *n* is

a power of 2.

*Hint:* Use induction to reduce the problem to that for *n/*2.

**Answer:**

∑ni=1 [log2(n/i)]

Let i =1

* [log2 (n/i)]
* log2 n

Let i = 2

* [log2 (n/i)]
* log2 n/2

..

Let i = 4

* [log2 (n/i)]
* log2 n/22

……..

Let i = k

* [log2 (n/i)]
* log2 n/2k-1

Let i = n

* [log2 (n/i)]
* log2 n/n
* log2 1

Now, suppose 2K =n then

Log 2K =log n

K= log n

Adding all the values:

[ log n/20 + log n/21 + log n/22 +…. + log n/k+ log 1]

* log [ n/20 \* n/21 \* n/22  \*…. \* 1]
* log2(2n)
* nlog22=n
* 0(n)

***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 *\_*

*√*

*N\_*

additional cells, going from capacity *N* to *N* + *\_*

*√*

*N\_*. Show that performing a

sequence of *n* add operations (that is, insertions at the end) runs in Θ(*n*3*/*2) time

in this case.

**Answer:**

The time taken for copying a single value from a table to the array takes O(1) time and hence the time taken to copy N elements is O(N). If the length of the array is extended to √N instead of 2N, then total time into an array = O(N+N3/2).

The time it takes to add N elements into an array N + √N = O (∑i=Ni=1(N +√Ni))

=O(N+√N)

= O(N+N3/2).

The worst-case time complexity and the average case time complexity is O(N+N3/2) and O(N3/2) respectively

**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?

**Answer**: Let the length of an array be n and duplicate the values of A[0] : A[n-1] to X[n-1] to X[0]

Execute this loop: - for (i=0; i<n; i++)  
X[n-1-i] = A[i];

The running time of this algorithm will be 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?

**Answer:**

The efficient algorithm for finding the shortest subarray of A that contains k 1’s, the algorithm is as follows:

* Let I be the first index of the occurrence of 1 in an array.
* Find inside the array the value K 1’s.
* The left side index I is removed and scanning continues until next 1 is found.
* The right side is extended so that 1’s = k
* Update the length when the window is smaller.
* Time Complexity of this algorithm is n time traveled so, it is O(n).