**#1)**

**Base Step:**

If *n* = 3, then clearly, *T*(3) = 9 *T*(3) = 32 = 9. This is trivial because of recurrence.

**Hypothesis Step:**

Now assume *T*(*n*) = *n*2 is true when *n* = 3k for some integer *k* > 0.

**Induction Step:**

Let *k* 🡪 *k* + 1 so that *n* = 3k + 1, then

T(3k+1)

= 6T(3k+1/3) + 1/3\*(3k+1)2

= 6T(3k) + 1/3\*(32k+2)

= 6T(3k) + 32k+1

= 6\*(3k)2 + 32k+1

= 6\*32k + 32k+1

= 2\*32k+1 + 32k+1

= 32k+1 \* (2 + 1)

= 32k+1 \* 3

= 32k+2

= (3k+1)2

**Citations:** 1) <http://cs.boisestate.edu/~jhyeh/teach/cs242_fall04/h1_sol.pdf> Author: Professor Jyh-haw Yeh from Boise State University, Title: cs242\_fall04/h1\_sol, Date: Jan. 17, 2008.

2) <http://answers-by-me.blogspot.com/2010/07/clrs-2e-exercise-23-3.html> Author: Justin Mancinelli, Title: CLRS 2e: Exercise 2.3-3, Date: Tuesday, July 13, 2010.

**#2)**

Binary Search is an algorithm that finds the position in a sorted array by checking the midpoint of the sequence and repeats this procedure with the remaining portion. Since we are taking half the sequence each time we divide *n* by 2. So since it will always be a constant, we have T(n) = T(n/2) + 1

**Base Case:**

We have the base case as *n* = 1 where the list is already sorted so there is no work. Thus we have constant time *O*(1).

**Worst Case:**

When the value is not in list, the algorithm must continue iterating until the span has been made empty. This will take at most log(n) + 1 iterations. Thus, Binary Search on a sorted array has worst case time complexity *ϴ* (log n). Note this is log base 2 of n.

Therefore the recurrence of Binary Search is:

*T*(*n*) = { 1 if *n* = 1,

{ *T*(*n* / 2) + 1 if *n* > 1.

The solution to this recurrence is: *T*(*n*) = *ϴ*(log *n*).

**Citations:** 1) <http://cs.boisestate.edu/~jhyeh/teach/cs242_fall04/h1_sol.pdf>

Author: Professor Jyh-haw Yeh from Boise State University, Title: cs242\_fall04/h1\_sol, Date: Jan. 17, 2008.

2) <http://cs.stackexchange.com/questions/13168/recurrence-for-recursive-insertion-sort>

Author: Aseem Bansal, Title: Recurrence for recursive insertion sort, Date: Jul. 9, 2013.

# 3) <http://stackoverflow.com/questions/18808429/understanding-recurrence-for-running-time> Author: Harrison, Title: [Understanding recurrence for running time](http://stackoverflow.com/questions/18808429/understanding-recurrence-for-running-time), Date: Sep. 15, 2013

# 4) <https://en.wikipedia.org/wiki/Binary_search_algorithm>

# Author: Wikipedia, Title: Binary Search Algorithm, Updated: Jan. 2016.

**#3)**

By doing the matrix multiplication Amxn × Anxo we get Amxo which is a matrix with m rows and o columns.

Let A = Amxn , B = Anxo and C = Amxo so that AxB = C then using Strassen’s algorithm we get:

From this we can make an algorithm which loops over the indices *i* from 1 through *m*, and *j* from 1 through *o*, and *k* from 1 through *n* using nested loops:

--------------------------------

Input: matrices A and B

Let C be a new matrix of the appropriate size

For *i* from 1 to *m*:

For *j* from 1 to *o*:

Let sum = 0

For *k* from 1 through *n*:

Set sum 🡨 sum + Aik + Bkj

Set Cij 🡨 sum

Return C

--------------------------------

Since we iterate through every k for a j value and every j for an i value then we will have ixjxk time complexity to go through all dimensions of the two matrices. Thus the time complexity is *ϴ*(n)\* *ϴ*(m)\* *ϴ*(o) = *ϴ*(n\*m\*o). If both matrices being multiplied were square n by n matrices then the time complexity would be *ϴ*(n3).

**Citations:** 1) Textbook Page 75.

2) <https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm>

Author: Wikipedia, Title: Matrix Multiplication Algorithm, Updated: December 2015

**#4)**

Ordered: slow fast

Justifications:

* First n^n = ω(2^n) and (n + 1)! = ω(2^n) but the limit as n🡪∞ of (n+1)n! / n^n converges to 0 so therefore n^n is faster and (n + 1)! = Ω(n^n).
* Then = Ω((n+1)!) because it has a negative polynomial exponent which hinders its growth so it is e^(-n) \* n^(n + ½) < n!.
* Also, 2lg n  = n because clogba = alogbc . So this is a linear polynomial function which is slower than exponential functions. Thus, n = Ω(e^(-n) \* n^(n + ½)).
* Now lg n^2 = 2 lg n is a poly-logarithmic function which is slower than polynomial function so lg n^2 = Ω(n).
* Lastly, because = 2(1/2)\*lg n = 2lg √n = . This has a decimal exponential which grows slower than a linear polynomial function so < Ω(n) and < Ω(lg n^2).

**Citations:** 1)<http://ocw.mit.edu/resources/res-18-005-highlights-of-calculus-spring-2010/derivatives/growth-rate-and-log-graphs/MITRES18_05S10_Growth_Rate_Log_Graphs.pdf> Author: Gilbert Strang, Title: Highlights of Calculus, MIT OpenCourseWare, Date: Spring 2010.

2) Recitation notes January 15.

**#5)**

**Master Theorem:**

a = 2, b = 2, f(n) = n. So, for all n. So, since then we have case 2, T(n) = ϴ(nlogba lg n) = ϴ(nlog22 lg n) = ϴ(n lg n).

**Secondary Recurrence:**

Let *ni* = *n* and *ni* – 1 = *n*/2. Now assume *T*(1)is the base case and *n0* = 1 and solve for ϴ notation of the function:

*ni* = 2*ni – 1*  🡪 *ni* = α2*i* (corresponds to (E – 2))

Since *n0* = 1, α = 1, and *ni* = 2*i* . Now define *F*(*i*) = *T*(*ni*). Then, the original recurrence:

*T*(*n*) = *T*(*ni*) = 2*T*(*n*/2) + *n* = 2*T*(*ni – 1* ) + *n*

Becomes:

*F*(*i*) = 2*F*(*i* – 1) + n

We have supposed *n* = *ni*, and we derived that *ni* = 2*i*. Therefore, the final recurrence to solve is:

*F*(*i*) = 2*F*(*i* – 1) + (2*i*)

Which is annihilated by (E – 2)2. The corresponding closed formula is (α1 + α2)\*2*i*, which is ϴ(2*i*). Recall that *n* = 2*i*. We can achieve the final ϴ notation by undoing the substitution as follows:

*T*(*n*) = *F*(*i*) = ϴ(2*i*) = ϴ(2log2n lg n) = ϴ(nlog22 lg n) = ϴ(n lg n).

Also, we could solve by substituting n = 2k  into T(n) = 2T(n/2) + n because n/2 will be one lower in the sequence than 2k – 1 . This gives us T(2k) = 2T(2k – 1 ) + 2k. Now, let tk = T(2k) so that we get tk = 2tk-1 + 2k. The annihilator for the homogeneous part tk = 2tk-1 is (E – 2) and the annihilator for the non-homogenous part 2k is (E – 2) so the result annihilator for the whole equation is (E – 2)2. Now un-substitute n = 2k  so that k = lg2(n) to solve the recurrence. Since (E – 2)2 annihilates sequences of type 2kk then plugging in k = lg2(n) will give us 2lg nlg n = n lg n = ϴ(n lg n).

**Citations:** 1) <http://cs.iit.edu/~cs430/scribbling/jan13.pdf> Author: Edward Reingold, Title: in-class scribbling jan13, Date: Jan 13 2016.

2) Textbook page 94

3) <http://cs.iit.edu/~cs430/lecture-notes/jan13.pdf> Author: Edward Reingold, Title: Lecture 2: January 13, Date: Jan 13 2016.