CSC 323 Algorithm Design and Analysis
Module 1 – Analyzing the Efficiency of Algorithms
Instructor: Dr. Natarajan Meghanathan

Sample Questions and Solutions

1) (a) Derive an expression for the average number of key comparisons in a sequential search algorithm, as a function of *the number of elements (n) in the list* and *the probability (p) of finding an integer key*.
(b) Consider a list with 10 integers as keys. The probability of finding an integer key using sequential search on this 10-element list is 0.3. Using the expression derived in (a), compute the average number of key comparisons that would be needed on this 10-element list.

(a)

- If *p* is the probability of finding an element in the list, then (1-*p*) is the probability of not finding an element in the list.
- Also, on an *n*-element list, the probability of finding the search key as the i^{th} element in the list is *p/n* for all values of $1 \leq i \leq n$

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1-p)$$

$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1-p)$$

$$= \frac{p}{n}\frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p).$$

- If p = 1 (the element that we will search for always exists in the list), then $C_{avg}(n) = (n+1)/2$. That is, on average, we visit half of the entries in the list to search for any element in the list.
- If p = 0 (all the time, the element that we will search never exists), then $C_{avg}(n) = n$. That is, we visit all the elements in the list.

(b)
Substituting for n=10 and p=0.3 in the above equation, we get $C_{avg}(n) = (0.3)(11/2) + (10)(0.7) =$ **8.65**.

2) Consider the classical sequential search algorithm (of looking for a key in a list of keys) and a variation of sequential search that scans a list to return *the number of occurrences* of a given search key in the list. Compare the best-case, worst-case and average-case efficiency as well as the overall time complexity of the classical sequential search with that of its variation.

In the classical sequential search (question above), the best-case scenario would be when the search is the first element in the list – thus, requiring only 1 key comparison. At the worst-case, we would have to do *n* key comparisons on an *n*-element list if the search key is the last element in the list or is not at all in the list. The average case # of key comparisons is given by (n+1)/2, when we substitute for p = 1 in the above equation (see Q1-(a)). The overall time complexity is **O(n)**.

With the variation of sequential search, the entire list has to be always searched for (i.e., scanned) to determine the number of occurrences of the search key. This way, the best-, average- and worst-case number of comparisons for sequential are the same. The overall time complexity is **Θ(n).**

3) If $t1(n) \in O(g1(n))$ and $t2(n) \in O(g2(n))$, then prove that $t1(n) + t2(n) \in O(\max\{g1(n), g2(n)\})$

For any four arbitrary real numbers, a_1, b_1, a_2 and b_2 such that $a_1 \leq b_1$ and $a_2 \leq b_2$,
We have $a_1 + a_2 \leq 2 \max\{ b_1, b_2 \}$

Since $t_1(n) \in O(g_1(n))$, then there exists some constant c_1 and non-negative integer n_1 such that

$$t_1(n) \leq c_1 g_1(n) \text{ for all } n \geq n_1$$

Since $t_2(n) \in O(g_2(n))$, then there exists some constant c_2 and non-negative integer n_2 such that

$$t_2(n) \leq c_2 g_2(n) \text{ for all } n \geq n_2$$

Let $c_3 = \max\{ c_1, c_2\}$ and $n_0 = \max\{n_1, n_2\}$

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c3 g_2(n) \\ &= c_3\{ g_1(n) + g_2(n)\} \\ &\leq 2 c_3 \max\{ g_1(n), g_2(n)\} \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{ g_1(n), g_2(n)\}$, with the constants c and n_0 required by the O definition being $2 c_3 = 2 \max\{ c_1, c_2\}$ and $\max\{n_1, n_2\}$ respectively.

4) For each of the following functions, indicate the class $\Theta(g(n))$ the function belongs to. Use the simplest g(n) possible in your answers. Prove your assertions (Hint: Use the Limits approach)

a. $(n^2 + 1)^{10}$ **b.** $\sqrt{10n^2 + 7n + 3}$

c. $2n \lg(n+2)^2 + (n+2)^2 \lg \frac{n}{2}$ **d.** $2^{n+1} + 3^{n-1}$

a)

$$\lim_{n\to\infty} \frac{(n^2+1)^{10}}{n^{20}} = \lim_{n\to\infty} \frac{(n^2+1)^{10}}{(n^2)^{10}} = \lim_{n\to\infty} \left(\frac{n^2+1}{n^2}\right)^{10} == \lim_{n\to\infty} \left(1 + \frac{1}{n^2}\right)^{10} = 1$$

Hence $(n^2 + 1)^{10} \in \Theta(n^{20})$.

b)

Informally, $\sqrt{10n^2 + 7n + 3} \approx \sqrt{10n^2} = \sqrt{10}n \in \Theta(n)$. Formally,

$$\lim_{n\to\infty} \frac{\sqrt{10n^2+7n+3}}{n} = \lim_{n\to\infty} \sqrt{\frac{10n^2+7n+3}{n^2}} = \lim_{n\to\infty} \sqrt{10 + \frac{7}{n} + \frac{3}{n^2}} = \sqrt{10}.$$

Hence $\sqrt{10n^2 + 7n + 3} \in \Theta(n)$.

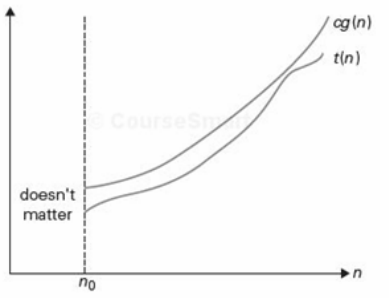
c)

$$2n \lg(n+2)^2 + (n+2)^2 \lg \frac{n}{2} = 2n2 \lg(n+2) + (n+2)^2(\lg n - 1)$$
$$\in \Theta(n \lg n) + \Theta(n^2 \lg n) = \Theta(n^2 \lg n).$$

d)

$$2^{n+1} + 3^{n-1} = 2^n 2 + 3^n \frac{1}{3} \in \Theta(2^n) + \Theta(3^n) = \Theta(3^n)$$

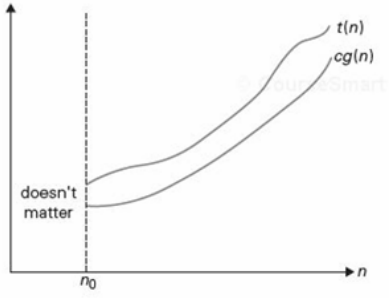
5) Give formal definitions for the Big-Oh (O), Big-Omega (Ω) and Big-Theta (Θ) asymptotic notations. Illustrate the definition using appropriate figures.



$t(n) = O(g(n))$

$t(n) \le c*g(n)$ for all $n \ge n_0$

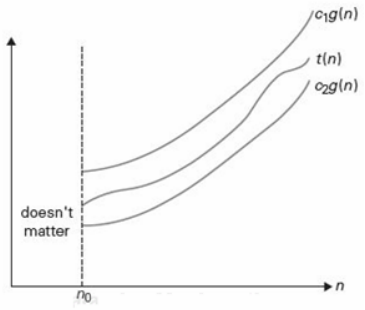
c is a positive constant (> 0)
and n_0 is a non-negative integer



$t(n) = \Omega(g(n))$

$t(n) \ge c*g(n)$ for all $n \ge n_0$

c is a positive constant (> 0)
and n_0 is a non-negative integer



$t(n) = \Theta(g(n))$

$c2*g(n) \le t(n) \le c1*g(n)$ for all $n \ge n_0$

c1 and c2 are positive constants (> 0)
and n_0 is a non-negative integer

6)

Prove that if $t_1(n) = \Omega(g_1(n))$ and $t_2(n) = \Omega(g_2(n))$, then

$t_1(n) + t_2(n) = \Omega(\text{Min}(g_1(n), g_2(n)))$

Proof

$t_1(n) = \Omega(g_1(n)) \Rightarrow t_1(n) \ge c_1 \cdot g_1(n)$ for $n \ge n_1$

$t_2(n) = \Omega(g_2(n)) \Rightarrow t_2(n) \ge c_2 \cdot g_2(n)$ for $n \ge n_2$

$t_1(n) + t_2(n) \ge c_1 \cdot g_1(n) + c_2 \cdot g_2(n)$ for $n \ge \underbrace{\text{Min}(n_1, n_2)}_{n_3}$

Let $c_3 = \text{Min}(c_1, c_2)$

$t_1(n) + t_2(n) \ge c_3 g_1(n) + c_3 \cdot g_2(n)$

$= c_3 [g_1(n) + g_2(n)]$

$g_1(n) + g_2(n) \ge 2 \cdot \text{Min}(g_1(n), g_2(n))$

For example, $4 + 3 \ge 2 * \text{Min}(4, 3) = 6$.

Thus, $t_1(n) + t_2(n) \ge (2c_3) \text{Min}(g_1(n), g_2(n))$ for $n \ge \text{Min}(n_1, n_2)$

($2c_3$: constant)

$t_1(n) + t_2(n) = \Omega(\text{Min}(g_1(n), g_2(n)))$

7)

Find the class $O(g(n))$, $\Theta(g(n))$ and $\Omega(g(n))$ for the following function:

(a) $(n^2+1)^{10}$.

$O(g(n))$: Let $g(n) = n^{21}$

$$\lim_{n\to\infty} \frac{(n^2+1)^{10}}{n^{21}} = \lim_{n\to\infty} \frac{(n^2+1)^{10}}{(n^2)^{10}.n} = \lim_{n\to\infty} \frac{1}{n}\left[\frac{n^2+1}{n^2}\right]^{10} = \lim_{n\to\infty} \frac{1}{n}\left[1+\frac{1}{n^2}\right]^{10}$$

$$= 0 * 1 = 0 \Rightarrow \underline{(n^2+1)^{10} = O(n^{21})}$$

(b) $\Theta(g(n))$: Let $g(n) = n^{20}$

$$\lim_{n\to\infty} \frac{(n^2+1)^{10}}{n^{20}} = \lim_{n\to\infty} \frac{(n^2+1)^{10}}{(n^2)^{10}}$$

$$= \lim_{n\to\infty} \left[\frac{n^2+1}{n^2}\right]^{10} = \lim_{n\to\infty} \left[1+\frac{1}{n^2}\right] = 1$$

$$\Rightarrow \underline{(n^2+1)^{10} = \Theta(n^{20})}$$

(c) $\Omega(g(n))$: Let $g(n) = n^{10}$.

$$\lim_{n\to\infty} \frac{(n^2+1)^{10}}{n^{10}} = \lim_{n\to\infty} \left[\frac{n^2+1}{n}\right]^{10} = \lim_{n\to\infty} \left[n+\frac{1}{n}\right]^{10} = \lim_{n\to\infty} n^{10} = \infty$$

$$(n^2+1)^{10} = \Omega(n^{10})$$

8)

Find the class $O(g(n))$, $\Theta(g(n))$ and $\Omega(g(n))$ for the following functions:

(b) $\sqrt{3n^2+7n+4}$ → $t(n)$

$O(g(n))$: $\sqrt{3n^2+7n+4} \leq \sqrt{n^2} = n$.

Pick $g(n) = n^2 = \sqrt{n^4}$ [degree of $g(n)$ greater than $t(n)$]

$$\lim_{n\to\infty} \frac{\sqrt{3n^2+7n+4}}{\sqrt{n^4}} = \lim_{n\to\infty} \sqrt{\frac{3n^2+7n+4}{n^4}} = \lim_{n\to\infty} \sqrt{\frac{3}{n^2}+\frac{7}{n^3}+\frac{4}{n^4}}$$

$$= 0$$

$$\Rightarrow \sqrt{3n^2+7n+4} = O(n^2)$$

$\Theta(g(n))$:

Pick $g(n)$ to be of the same degree as $t(n) = \sqrt{3n^2+7n+4} \Rightarrow \sqrt{n^2} = n$.

$g(n) = \sqrt{n^2} = n$.

$$\lim_{n\to\infty} \frac{\sqrt{3n^2+7n+4}}{\sqrt{n^2}} = \lim_{n\to\infty} \sqrt{\frac{3n^2+7n+4}{n^2}} = \lim_{n\to\infty} \sqrt{3+\frac{7}{n}+\frac{4}{n^2}}$$

$$= \sqrt{3}$$

$$\sqrt{3n^2+7n+4} = \Theta(n)$$

$\Omega(g(n))$:

Pick $g(n)$ to be of a lower degree than $t(n) = \sqrt{3n^2+7n+4} \geq n$.

So, pick $g(n) = n^{1/2} = \sqrt{n}$

$$\lim_{n\to\infty} \frac{\sqrt{3n^2+7n+4}}{\sqrt{n}} = \lim_{n\to\infty} \sqrt{\frac{3n^2+7n+4}{n}} = \lim_{n\to\infty} \sqrt{3n+7+\frac{4}{n}} = \infty.$$

So, $\sqrt{3n^2+7n+4} = \Omega(\sqrt{n})$

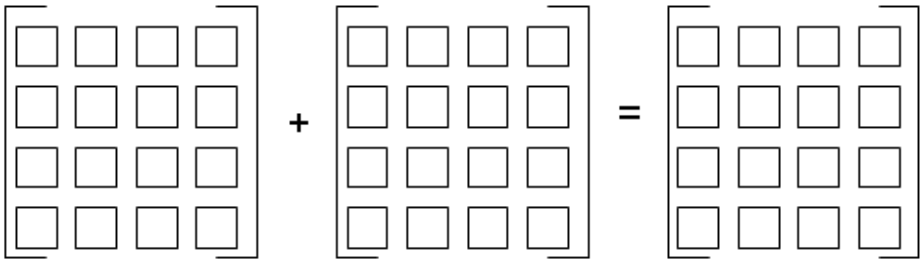
9) Consider the standard definition-based algorithm to add two *n* x *n* matrices.

a. What is its basic operation?

b. How many times the basic operation is performed as a function of the total number of elements in the input matrices?

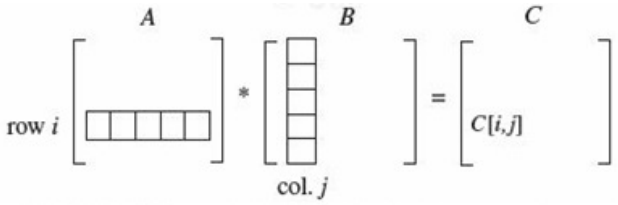
c. Answer (a) and (b) for the standard definition-based algorithm for matrix multiplication.

Matrix Addition



Addition is the basic operation. There are *n* additions per row with each addition operating on 2 integers. On an *n* x *n* matrix with n^2 integers as the input size, there will be n*n = $\Theta(n^2)$ additions.

Matrix Multiplication



$$C[i, j] = A[i, 0]B[0, j] + \cdots + A[i, k]B[k, j] + \cdots + A[i, n-1]B[n-1, j]$$

$$M(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

Overall time complexity: $\mathbf{\Theta(n^3)}$.

10) Compute the following sums:

a. 1 + 3 + 5 + 7 + … + 999

Solution: 1 + 3 + 5 + 7 + … + 999 = [1 + 2 + 3 + 4 + 5 + …. 999] – [2 + 4 + 6 + 8 + … + 998]

$$= \frac{999*1000}{2} - 2[1+2+3+....+499]$$

$$= 999*500 - 2\left[\frac{499*500}{2}\right] = 999*500 - 499*500$$

= 500*(999-499) = 500*500 = 250,000

b. 2+ 4 + 8 + 16 + … + 1024

Solution: 2+ 4 + 8 + 16 + … + 1024 = $2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^{10}$

= $[2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^{10}] - 1$

$$= \left[\sum_{i=0}^{10} 2^i\right] - 1 = \left[2^{11} - 1\right] - 1 = \mathbf{2046}$$

c. $\sum_{i=3}^{n+1} 1 = [(n+1) - 3 + 1] = n+1 - 2 = n-1 = \Theta(n)$

d. $\sum_{i=3}^{n+1} i = 3 + 4 + ... + (n+1) = [1 + 2 + 3 + 4 + ... + (n+1)] - [1 + 2]$

$= \frac{n(n+1)}{2} - 3 = \Theta(n^2) - \Theta(1) = \Theta(n^2)$

e. $\sum_{i=0}^{n-1} i(i+1)$

$= \left[\sum_{i=0}^{n} i(i+1)\right] - n(n+1) = \left[\sum_{i=0}^{n} i^2 + \sum_{i=0}^{n} i\right] - n(n+1)$

$= \Theta(n^3) + \Theta(n^2) - \Theta(n^2) - \Theta(n) = \Theta(n^3)$.

<u>Alternate Way:</u>

$$\sum_{i=0}^{n-1} i(i+1) = \sum_{i=0}^{n-1} i^2 + i = \sum_{i=0}^{n-1} i^2 + \sum_{i=0}^{n-1} i$$

$$= \left[\frac{[n-1][(n-1)+1][2(n-1)+1]}{6}\right] + \left[\frac{[n-1][(n-1)+1]}{2}\right]$$

$$= \left[\frac{[n-1][n][2n-1]}{6}\right] + \left[\frac{[n-1][n]}{2}\right]$$

$= \Theta(n^3) + \Theta(n^2) = \Theta(n^3)$

f. $\sum_{j=1}^{n} 3^{j+1} = 3\sum_{j=1}^{n} 3^j = 3[\sum_{j=0}^{n} 3^j - 1] = 3[\frac{3^{n+1}-1}{3-1} - 1] = \frac{3^{n+2}-9}{2}$

11) Find the order of growth of the following sums. Use the $\Theta(g(n))$ notation with the simples possible g(n):

a)

$$\sum_{i=0}^{n-1} (i^2+1)^2 = \sum_{i=0}^{n-1} (i^4 + 2i^2 + 1) = \sum_{i=0}^{n-1} i^4 + 2\sum_{i=0}^{n-1} i^2 + \sum_{i=0}^{n-1} 1$$

$$\in \Theta(n^5) + \Theta(n^3) + \Theta(n) = \Theta(n^5)$$

b)

$$\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}(i+j) = \sum_{i=0}^{n-1}[\sum_{j=0}^{i-1} i + \sum_{j=0}^{i-1} j] = \sum_{i=0}^{n-1}[i^2 + \frac{(i-1)i)}{2}] = \sum_{i=0}^{n-1}[\frac{3}{2}i^2 - \frac{1}{2}i]$$

$$= \tfrac{3}{2}\sum_{i=0}^{n-1} i^2 - \tfrac{1}{2}\sum_{i=0}^{n-1} i \in \Theta(n^3) - \Theta(n^2) = \Theta(n^3).$$

12) Answer the following questions for the algorithms (pseudo code) given below:

a) What does this algorithm compute?
b) What is its basic operation?
c) How many times is the basic operation executed (best and worst-cases)?
d) What is the overall time complexity of this algorithm?

(i)

```
Algorithm Mystery(n)
//Input: A nonnegative integer n
S ← 0
for i ← 1 to n do
    S ← S + i * i
return S
```

(ii)

```
Algorithm Enigma(A[0..n − 1, 0..n − 1])
//Input: A matrix A[0..n − 1, 0..n − 1] of real numbers
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i, j] ≠ A[j, i]
            return false
return true
```

(i)

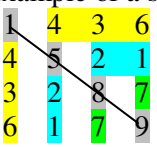
(a) The algorithm computes the sum of the squares of integers from 1 to n, where n is the input
(b) The basic operation is multiplication
(c) The multiplication is executed n times (both best and worst-cases)

Efficiency = $\sum_{i=1}^{n} 1 = n$

(d) $\lim_{n->\infty} \frac{Best-case}{Worst-case} = \lim_{n->\infty} \frac{n}{n} = 1$. Hence, Overall time complexity is **Θ(n).**

(ii)

(a) The algorithm determines whether the input matrix is symmetric (returns true) or not (returns false). Example of a symmetric matrix is:



(b) The basic operation is the comparison of the matrix elements
(c) At the best-case, if the first comparison itself fails, then the algorithm stops returning that the matrix is not symmetric. Depending on the matrix, the algorithm could also stop anywhere, with the number of comparisons ranging from the best-case to the worst-case.

The worst-case number of comparisons is incurred if all the iterations are executed:

$$\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}[(n-1)-(i+1)+1] = \sum_{i=0}^{n-2}(n-1)-i = (n-1)(n-1) - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)\left[\frac{2n-2-n+2}{2}\right] = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

(d) $\lim_{n->\infty} \frac{best-case}{worst-case} = \lim_{n->\infty} \frac{1}{n(n-1)/2} = 0$

The overall time complexity of the algorithm is $\mathbf{O(n^2)}$.

13) Analyze the worst-case run-time complexity of the following algorithm to determine whether or not the elements of an array are unique. Show all your work.

```
ALGORITHM UniqueElements(A[0..n − 1])
    //Determines whether all the elements in a given array are distinct
    //Input: An array A[0..n − 1]
    //Output: Returns "true" if all the elements in A are distinct
    //          and "false" otherwise
    for i ← 0 to n − 2 do
        for j ← i + 1 to n − 1 do
            if A[i] = A[j] return false
    return true
```

Solution:

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2$$

Best-case = 1 comparison.
Hence, overall time complexity is **$O(n^2)$.**

14) Develop a Θ(nlogn) algorithm to determine whether or not the elements of an array are unique. Analyze its overalln-time complexity. Hint: First, pre-sort the array using any Θ(nlogn) algorithm

Solution:

Use a Θ(nlogn) algorithm to sort the *n*-elements of the array. Then, scan the elements of the scanned array from index 0 to n-2 (compare elements at indices 0 and 1, at indices 1 and 2, and etc until elements at indices n-2 and n-1). If in any of these *n*-1 = Θ(n) comparisons, the two elements compared are observed to be the same, then the elements of the array are not unique. The overall run-time complexity is Θ(nlogn) + Θ(n) = Θ(nlogn).

15) Develop an algorithm to compute the largest element in an array of *n* integers. Analyze its overall time complexity.

Solution:

```
ALGORITHM LargestElement(A[0…n-1])
    max = A[0]
    for i ← 1 to n-1 do
      if max < A[i] then
        max = A[i]
      end if
    end for

    return max
```

The basic operation is the element comparison operation that is executed *n*-1 times (as part of the *if* statement inside the *for* loop). The best and worst-case comparisons are *n*-1. Hence, the overall-time complexity is n-1 = **O(n).**

16) Analyze the best, worst and overall time complexity of the following algorithm to determine whether an integer is prime or not. What is the basic operation? Show all your work.

```
Input: Integer n
Output: True (is prime) or False (not prime)

for i = 2 to n -1 do
    if (n mod i = 0) then
        return false
    end if
end for

return true
```

Solution:
"Division" is the basic operation. At the minimum (best-case), we may do just one division and if the number is not prime, we would stop the algorithm (returning false). At the maximum (worst-case), we may do n-2 divisions, to test whether the number n is divisible by 2, 3, ..., n-1.

$$\lim_{n->\infty} \frac{best-case}{worst-case} = \lim_{n->\infty} \frac{1}{n-2} = 0$$

Hence, the overall time complexity is O(n).

17) Solve the following recurrence relations:

a) X(n) = X(n-1) + 5, for n > 1, X(1) = 0

$$\begin{aligned} x(n) &= x(n-1)+5 \\ &= [x(n-2)+5]+5 = x(n-2)+5\cdot 2 \\ &= [x(n-3)+5]+5\cdot 2 = x(n-3)+5\cdot 3 \\ &= \ldots \\ &= x(n-i)+5\cdot i \\ &= \ldots \\ &= x(1)+5\cdot(n-1) = 5(n-1). \end{aligned}$$

= Θ(n)

b) X(n) = 3*X(n-1) for n > 1, X(1) = 4

$$\begin{aligned} x(n) &= 3x(n-1) \\ &= 3[3x(n-2)] = 3^2x(n-2) \\ &= 3^2[3x(n-3)] = 3^3x(n-3) \\ &= \ldots \\ &= 3^ix(n-i) \\ &= \ldots \\ &= 3^{n-1}x(1) = 4\cdot 3^{n-1}. \end{aligned}$$

= $(4/3)3^n$ = $\Theta(3^n)$

c) X(n) = X(n-1) + n for n > 0, X(0) = 0

$$\begin{aligned} x(n) &= x(n-1)+n \\ &= [x(n-2)+(n-1)]+n = x(n-2)+(n-1)+n \\ &= [x(n-3)+(n-2)]+(n-1)+n = x(n-3)+(n-2)+(n-1)+n \\ &= \ldots \\ &= x(n-i)+(n-i+1)+(n-i+2)+\cdots+n \\ &= \ldots \\ &= x(0)+1+2+\cdots+n = \frac{n(n+1)}{2}. \end{aligned}$$

X(n) = Θ(n^2)

d) X(n) = X(n/2) + n, for n > 1, X(1) = 1 [Solve for n = 2^k]

$$\begin{aligned} x(2^k) &= x(2^{k-1})+2^k \\ &= [x(2^{k-2})+2^{k-1}]+2^k = x(2^{k-2})+2^{k-1}+2^k \\ &= [x(2^{k-3})+2^{k-2}]+2^{k-1}+2^k = x(2^{k-3})+2^{k-2}+2^{k-1}+2^k \\ &= \ldots \\ &= x(2^{k-i})+2^{k-i+1}+2^{k-i+2}+\cdots+2^k \\ &= \ldots \\ &= x(2^{k-k})+2^1+2^2+\cdots+2^k = 1+2^1+2^2+\cdots+2^k \\ &= 2^{k+1}-1 = 2\cdot 2^k-1 = 2n-1. \end{aligned}$$

X(n) = Θ(n)

e) X(n) = X(n/3) + 1 for n > 1, X(1) = 1 [Solve for n = 3^k]

$$\begin{aligned} x(3^k) &= x(3^{k-1})+1 \\ &= [x(3^{k-2})+1]+1 = x(3^{k-2})+2 \\ &= [x(3^{k-3})+1]+2 = x(3^{k-3})+3 \\ &= \ldots \\ &= x(3^{k-i})+i \\ &= \ldots \\ &= x(3^{k-k})+k = x(1)+k = 1+\log_3 n \end{aligned}$$

X(n) = Θ(logn)

18)

Consider the following recursive algorithm:

Algorithm $Q(n)$
//Input: A positive integer n
if $n = 1$ **return** 1
else return $Q(n-1) + 2 * n - 1$

Set up a recurrence relation for this function's values and solve it to determine what this algorithm computes.

$$Q(n) = Q(n-1) + 2n - 1 \quad \text{for } n > 1, \quad Q(1) = 1.$$

Computing the first few terms of the sequence yields the following:
$Q(2) = Q(1) + 2 \cdot 2 - 1 = 1 + 2 \cdot 2 - 1 = 4;$
$Q(3) = Q(2) + 2 \cdot 3 - 1 = 4 + 2 \cdot 3 - 1 = 9;$
$Q(4) = Q(3) + 2 \cdot 4 - 1 = 9 + 2 \cdot 4 - 1 = 16.$

Thus, it appears that $Q(n) = n^2$. We'll check this hypothesis by substituting this formula into the recurrence equation and the initial condition. The left hand side yields $Q(n) = n^2$. The right hand side yields

$$Q(n-1) + 2n - 1 = (n-1)^2 + 2n - 1 = n^2.$$

19)

Consider the following algorithm to determine the number of bits needed for the binary representation of a positive integer n. Set up a recurrence relation for the number of times the basic operation is executed and solve for the same.

ALGORITHM *BinRec*(n)
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if $n = 1$ **return** 1
else return *BinRec*($\lfloor n/2 \rfloor$) + 1

Solution:

Addition is the basic operation. # additions: $A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1.$ $A(1) = 0$

Since the recursive calls end when n is equal to 1 and there are no additions made then, the initial condition is

$$A(1) = 0.$$

1	1 bit
2-3	2 bits
4-7	3 bits
8-15	4 bits
16-31	5 bits
32-63	6 bits

Solution Approach: If we use the backward substitution method (as we did in the previous two examples, we will get stuck for values of n that are not powers of 2).
We proceed by setting $n = 2^k$ for $k \geq 0$.

New recurrence relation to solve:

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$
$$A(2^0) = 0.$$

$$
\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1\\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1\\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \cdots\\
&\cdots\\
&= A(2^{k-i}) + i\\
&\cdots\\
&= A(2^{k-k}) + k.
\end{aligned}
$$

$$A(n) = \log_2 n \in \Theta(\log n).$$

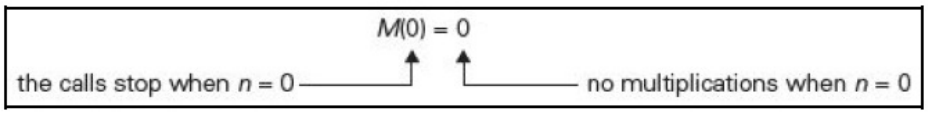
20) Set up and solve a recurrence relation for the number of multiplications to be done while computing F(n) = n! through a recursive algorithm.

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

$$M(n) = \underbrace{M(n-1)}_{\text{to compute } F(n-1)} + \underbrace{1}_{\text{to multiply } F(n-1) \text{ by } n} \quad \text{for } n > 0.$$

$$
\begin{aligned}
&M(n) = M(n-1) + 1 \quad \text{for } n > 0,\\
&M(0) = 0.
\end{aligned}
$$



$$M(n-1) = M(n-2) + 1; \quad M(n-2) = M(n-3)+1$$
$$M(n) = [M(n-2)+1] + 1 = M(n-2) + 2 = [M(n-3)+1+2] = M(n-3) + 3$$
$$= M(n-n) + n = n = \Theta(n)$$

21) Consider the following recursive algorithm for computing the sum of the first *n* cubes: $S(n) = 1^3 + 2^3 + \ldots + n^3$.

Algorithm $S(n)$
//Input: A positive integer n
//Output: The sum of the first n cubes
if $n = 1$ **return** 1
else return $S(n-1) + n * n * n$

a) Set up and solve a recurrence relation for the number of times the algorithm's basic operation is executed.
Multiplication is the basic operation. The recurrence relation is: M(n) = M(n-1) + 2 for n > 1; M(1) = 0
Solving using backward substitution, we get:

$$\begin{aligned} M(n) &= M(n-1)+2 \\ &= [M(n-2)+2]+2 = M(n-2)+2+2 \\ &= [M(n-3)+2]+2+2 = M(n-3)+2+2+2 \\ &= \ldots \\ &= M(n-i)+2i \\ &= \ldots \\ &= M(1)+2(n-1) = 2(n-1). \end{aligned}$$

M(n) = Θ(n)
b) How does this algorithm compare with the straightforward non-recursive algorithm for computing this function?

b. Non-recursive version

Algorithm *NonrecS*(n)
//Computes the sum of the first n cubes nonrecursively
//Input: A positive integer n
//Output: The sum of the first n cubes.
$S \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
    $S \leftarrow S + i * i * i$
return S

The number of multiplications made by this algorithm is given by:

$$\sum_{i=2}^{n} 2 = 2\sum_{i=2}^{n} 1 = 2(n-1)$$