FIND THE FORMULA FOR THE SUMMATION $\sum_1^n i(i+1)$.

$$\sum_{i=1}^{n} i(i+1) = \sum_{i=1}^{n} \frac{(i+1)^3 - i^3 - 1}{3}$$

$$= \frac{(1+1)^3 - 1^3 - 1}{3} + \frac{(2+1)^3 - 2^3 - 1}{3} + \frac{(3+1)^3 - 3^3 - 1}{3} + \frac{(4+1)^3 - 4^3 - 1}{3} + \cdots + \frac{(n+1)^3 - n^3 - 1}{3}$$

$$= \frac{1}{3}((2)^3 - 1^3 - 1 + (3)^3 - 2^3 - 1 + (4)^3 - 3^3 - 1 + (5)^3 - 4^3 - 1 + \cdots + (n+1)^3 - n^3 - 1)$$

$$= \frac{1}{3}\big(((2)^3 + (3)^3 + (4)^3 + (5)^3 + \cdots + (n+1)^3) - (1^3 + 2^3 + 3^3 + 4^3 + \cdots + n^3) - n\big)$$

$$= \frac{1}{3}\left(\sum_{i=1}^{n+1} i^3 - 1 - \sum_{i=1}^{n} i^3 - n\right)$$

$$= \frac{1}{3}\left(\sum_{i=1}^{n} i^3 - 1 + (n+1)^3 - \sum_{i=1}^{n} i^3 - n\right)$$

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

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

$$= \frac{n^3 + 3n^2 + 2n}{3}$$

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

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

A complete binary tree is defined inductively as follows. A complete binary tree of height 0 consists of 1 node which is the root. A complete binary tree of height $h+1$ consists of two complete binary trees of height h whose roots are connected to a new root. Let T be a complete binary tree of height h.

PROVE THAT THE NUMBER OF LEAVES OF A COMPLETE BINARY TREE IS 2^h AND THE SIZE OF THE TREE IS $2^{h+1}-1$.

Let $N(h)$ be the function that represents the number of nodes in the binary tree in terms of the height, h of a tree, T.

Let $L(h)$ be the function that represents the number of leaves in the binary tree in terms of the height, h of a tree, T.

(In the examples below, h_L and h_R refer to the height of the left and right subtrees of T respectively.)

$$N(0) = 1 \qquad L(0) = 1$$

$$N(h) = N(h_L) + N(h_R) + 1 \qquad L(h) = L(h_L) + L(h_R)$$

Proof by induction:

Number of Nodes

$N(h) = 2^{h+1} - 1$

Base Case:

$N(0) = 2^{0+1} - 1$
$= 1$

Inductive Step

$$\begin{aligned} N(h) &= N(h-1) + N(h-1) + 1 \\ 2^{h+1} - 1 &= (2^{h-1+1} - 1) + (2^{h-1+1} - 1) + 1 \\ &= (2^h - 1) + (2^h - 1) + 1 \\ &= 2^h + 2^h - 1 \\ 2^{h+1} - 1 &= 2^{h+1} - 1 \end{aligned}$$

∴The number of nodes in the tree can be represented as 2^{h+1}-1.■

Number of Leaves

$L(h) = 2^h$

Base Case:

$L(0) = 2^{0+1} - 1$
$= 1$

Inductive Step

$$\begin{aligned} L(h) &= L(h-1) + L(h-1) \\ 2^h &= 2^{h-1} + 2^{h-1} \\ &= 2 \cdot 2^{h-1} \\ 2^h &= 2^h \end{aligned}$$

∴The number of leaves in the tree can be represented as 2^h.■

QUESTION 2-4 (PP. 41) IN THE TEXT BOOK.

a. List the five inversions of the array $\langle 2,3,8,6,1 \rangle$.

In terms of i and j: $\{(0,4), (1,4), (2,3), (2,4), (3,4)\}$
In terms of A[i] and A[j]: $\{(2,1), (3,1), (8,6), (8,1), (6,1)\}$

b. What array with elements from the set $\{1,2,\ldots,n\}$ has the most inversions? How many does it have?

Such an array would be arranged in descending order starting with n: $\langle n, n-1, n-2, \ldots, 3, 2, 1 \rangle$.
It would have $\binom{n}{2}$ inversions.

c. What is the relationship between running time of insertion sort and the number of inversion in the input array? Justify your answer.

The number of inversions relative to the number of elements can be expressed as $I(n) = \binom{n}{2}$, where n is the number of elements, indicating that they are directly related. As each inversion represents a misplaced element, there are $I(n)$ insertions that must be made, and $I(n)$ comparisons that must be performed. Since the algorithm will need to scan the entire list at the end, the time complexity is of $T(n) = \Omega(n + 2I(n))$. This is perfectly consistent with the worst-case time complexity of $O(n^2)$, and the best case time complexity of $O(n)$.

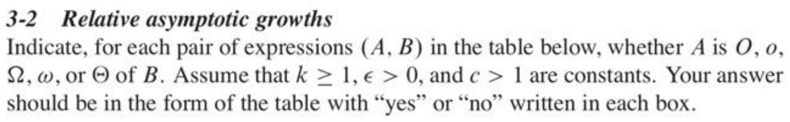
d. Give an algorithm that determines the number of inversions in any permutation of n elements in $\Theta(n \log n)$ worst-case time.

```
Algorithm countInversions(A,start,end)
In: Integer Array A, indexes for the start and end of the count
Out: The number of inversions present in the range given
if start >= end
        return 0
else
{
        middle = (start + end) / 2
        count = countInversions(A,start,middle)                 // Count the left half
        count = count + countInversions(A,middle,end)           // Count the right half
        count = count + inversionCounter (A,start,middle,end)

        return count
}
```

(Continued on next page)

```
Algorithm inversionCounter(A,start,middle,end)
In: Integer Array A, indexes for the start, middle, and end of the count
Out: a sorted list and the number of inversions
n = middle - start + 1
m = end - middle

let L[1...n + 1] and R[1...m + 1] be new arrays

for i = 1 to n
        L[i] = A[start + i -1]

for j = 1 to m
        R[j] = A[middle + j]

L[n + 1] = INF
R[m + 1] = INF

i = 1
j = 1
count = 0

for k = p to r
{
        if L[i] <= R[j]
        {
                A[k] = L[i]
                i = i + 1
        }

        else
        {
                A[k] = R[j]
                j = j + 1
                count = count + n - i  // If an element from R is selected before one from
                                L, it is smaller then each of the remaining elements in L,
                                representing n - i inversions
        }
}

return  count
```

READ THE TEXT BOOK FOR THE DEFINITION OF o AND ω AND ANSWER QUESTION 3-2 (PP. 61) IN THE TEXT BOOK.



3-2 Relative asymptotic growths

Indicate, for each pair of expressions (A, B) in the table below, whether A is O, o, Ω, ω, or Θ of B. Assume that $k \geq 1$, $\epsilon > 0$, and $c > 1$ are constants. Your answer should be in the form of the table with "yes" or "no" written in each box.

A	B	O	o	Ω	ω	Θ
lg^k n	n^ϵ	Yes	Yes	No	No	No
n^k	c^n	Yes	Yes	No	No	No
n^(1/2)	n^sin n	No	No	No	No	No
2^n	2^(n/2)	No	No	Yes	Yes	No
n^lgc	c^lgc	Yes	No	Yes	No	Yes
lg(n!)	lg(n^n)	Yes	No	Yes	No	Yes

QUESTION 4-2 (PP. 107) IN THE TEXT BOOK.

1. An array is passed by pointer. $Time = \Theta(1)$
2. An array is passed by copying. $Time = \Theta(N)$, where N is the size of the array.
3. An array is passed by copying only the subrange that might be accessed by the called procedure. $Time = \Theta(q - p + 1)$, if the subarray $A[p \ldots q]$ is passed.

a. Give recurrences and upper bounds for the above cases for BINARY SEARCH.
With binary search, there is one sub problem of size $\frac{n}{2}$ per recursion. Binary search has the recurrence
$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$

1. $a = 1, b = 2, f(n) = c = \Theta(1)$

By master theorem:

$$T(n) = T\left(\frac{n}{2}\right) + c$$

$$\Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 1}\right) = \Theta(1)$$

$$f(n) = \Theta(1)$$

$$\therefore T(n) = \Theta\left(n^{\log_2 1} \log n\right) = \Theta(\log n)$$

2. $a = 1, b = 2, f(n) = cN = \Theta(N)$

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + cN \\ &= T\left(\frac{n}{4}\right) + 2cN \\ &= T\left(\frac{n}{8}\right) + 3\,cN \ldots \\ &= \Theta(1) + \sum_{i=0}^{\log_2 n - 1} cN \end{aligned}$$

$$T(n) = \log_2(n-1) \cdot cN + 1 = \Theta(n \log n)$$

$$\therefore T(n) = \Theta(n \log n)$$

3. $a = 1, b = 2, f(n) = cn$

By master theorem:

$$T(n) = T\left(\frac{n}{2}\right) + cn$$

$$n^{\log_2 1} \leq cn$$

$$n^{\log_2 1 + 1} = n = \Theta(n)$$

$$f(n) = \Theta(n)$$

$$\therefore T(n) = \Theta(f(n)) = \Theta(n)$$

b. Give recurrences and upper bounds for the above cases for MERGE-SORT.

Merge-sort has two problems of size $\frac{n}{2}$ per recursion. Merge-sort has the recurrence $T(n) = 2T\left(\frac{n}{2}\right) + cn$

1. $a = 2, b = 2, f(n) = c_1 n + c_2 = \Theta(n)$

By master theorem:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

$$\Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 2}\right) = \Theta(n)$$

$$f(n) = \Theta(n)$$

$$\therefore T(n) = \Theta\left(n^{\log_2 2} \log n\right) = \Theta(n \log n)$$

2. $a = 2, b = 2, f(n) = cn + 2N$

$$T(n) = 2T\left(\frac{n}{2}\right) + cn + 2N$$

$$= 4T\left(\frac{n}{4}\right) + cn + \frac{2cn}{2} + 4N$$

$$= 8T\left(\frac{n}{8}\right) + cn + \frac{2cn}{2} + \frac{4cn}{4} + 8N \ldots$$

$$T(n) = cn + \sum_{i=1}^{\log_2 n - 1} \left(2^i N + icn\right)$$

$$T(n) = cn + N \sum_{i=1}^{\log_2 n - 1} 2^i + cn \sum_{i=1}^{\log_2 n - 1} i$$

$$= cn + N\left(2^{\log_2 n} - 1\right) + \frac{(\log_2 n - 1)(\log_2 n - 1 + 1)}{2}$$

$$= cn + Nn - N + (\log_2 n)^2 - \log_2 n$$

$$\therefore T(n) = \Theta(Nn) = \Theta(n^2)$$

3. $a = 2, b = 2, f(n) = cn = \Theta(n)$

By master theorem:

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

$$\Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 2}\right) = \Theta(n)$$

$$f(n) = \Theta(n)$$

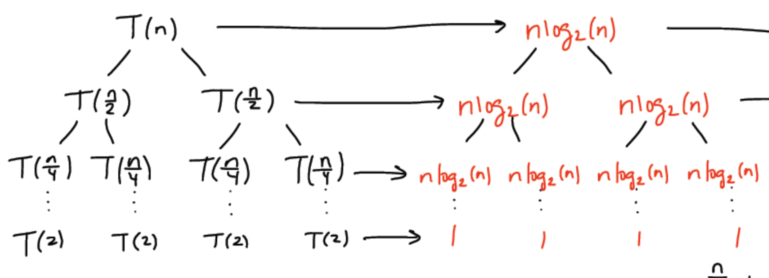
$$\therefore T(n) = \Theta\left(n^{\log_2 2} \log n\right) = \Theta(n \log n)$$

SUPPOSE THAT THE RUNNING TIME OF A RECURSIVE PROGRAM IS REPRESENTED BY THE FOLLOWING RECURRENCE RELATION:

$$T(2) = 1$$

$$T(n) = 2 \cdot T(n/2) + n\,log_2(n)$$

DETERMINE THE TIME COMPLEXITY OF THE PROGRAM USING RECURRENCE TREE METHOD.



Level	T(n)	Σ
0	n	$n\log_2(n)$
1	$\frac{n}{2}$	$2(n\log_2(n))$
2	$\frac{n}{2^2}$	$2^2(n\log_2(n))$
⋮	⋮	⋮
K	$\frac{n}{2^K}$	$2^K(n\log_2(n))$
$\log_2 n$	$\frac{n}{2^{\log_2 n}}$	$2^{\log_2 n}(n\log_2(n))$
$= \log_2 n$	1	$n^2\log_2(n)$

$$\frac{n}{2^K} = 1$$

$$K = \log_2 n$$

$$T(n) = n^2\log_2 n + (n\log_2 n)\sum_{i=0}^{\log_2 n - 1} 2^i$$

$$= n^2\log_2 n + (n\log_2 n)(2^{\log_2 n} - 1)$$

$$= n^2\log_2 n + n^2\log_2 n - n\log_2 n$$

$$\therefore T(n) = \Theta(n^2)$$

$$T(n) = \sum_{i=0}^{\log_2 n} n\log_2\left(\frac{n}{2^i}\right)$$

$$= n\sum_{i=0}^{\log_2 n}(\log_2 n - \log_2 2^i)$$

$$= n\sum_{i=0}^{\log_2 n}(\log_2 n - i)$$

$$= n\sum_{i=0}^{\log_2 n}\log_2 n - n\sum_{i=0}^{\log_2 n} i$$

$$= n(\log_2 n)(\log_2 n) - n\frac{\log_2 n\,(\log_2 n + 1)}{2}$$

$$= n(\log_2 n)^2 - \frac{n(\log_2 n)^2 - n\log_2 n}{2}$$

$$= \frac{n(\log_2 n)^2 - n\log_2 n}{2}$$

$$\therefore T(n) = \Theta(n(\log_2 n)^2)$$

WRITE A RECURSIVE FUNCTION TO COMPUTE F(N) USING THE ABOVE DEFINITION DIRECTLY.

See assignment folder.

WRITE A RECURSIVE FUNCTION/PROCEDURE TO COMPUTE $F(n)$ WITH TIME COMPLEXITY $O(n)$

See assignment folder.

WRITE A RECURSIVE FUNCTION/PROCEDURE TO COMPUTE $F(n)$ WITH TIME COMPLEXITY $O(log(n))$

See assignment folder.

USE THE UNIX TIME FACILITY (/USR/BIN/TIME) TO TRACK THE TIME NEEDED TO COMPUTE FOR EACH RUN AND FOR EACH ALGORITHM. COMPARE THE RESULTS AND STATE YOUR CONCLUSION IN TWO OR THREE SENTENCES.

The UNIX time facility measured that the program in a) was by far the slowest, calculating the first 35 values (printing the first 8) in nearly 0.11 seconds. Next, was the program in c), which calculated the first 300 values in 0.26 seconds, due to the fact that matrix multiplication takes a bit of time. Finally, the fastest was the program in b), as it was able to calculate the first 300 values in 0.12 seconds, as many of its values were retrieved rather than recalculated.

CAN YOU USE YOUR PROGRAM IN A) TO COMPUTE $F(50)$? BRIEFLY EXPLAIN YOUR ANSWER. EXPLAIN WHY YOUR PROGRAM IN B) COMPUTES $F(300)$ PRECISELY?

Using the program in a to compute F(50) will not work as F(50) exceeds the maximum number capable of being stored in an integer. However, using programs 2 or 3, this can be done easily. The reason why these programs can compute such large (seemingly infinite) Fibonacci numbers is because they use the self-defined BigInt, which represents numbers as strings, thus, so long as there is enough memory, there is no limit to the length of the number that can be computed.