Goran Curguz 250914767 gcurguz@uwo.ca

Question 1.

$$\sum_1^n i(i+1)$$

$$= \sum_1^n (i^2 + i)$$
$$= \sum_1^n i^2 + \sum_1^n i$$
$$= n(n+1)(2n+1)/6 + n(n+1)/2$$
$$= (2n^3 + 3n^2 + n)/6 + (n^2 + n)/2$$
$$= (2n^3 + 3n^2 + n + 3n^2 + 3n)/6$$
$$= (2n^3 + 6n^2 + 4n)/6$$
$$= n(n+1)(n+2)/3$$

Question 2.

Theorem. Let T be a complete binary tree of height h. Prove that the number of leaves of the tree is 2^h and the size of the tree (number of nodes in T) is $2^{h+1} - 1$.

Proof:

Induction base: $h = 0$

A complete binary tree of height $h = 0$ will have one node which is the root and also a leaf node. The number of leaves is $1 = 2^h = 2^0 = 1$. The size of the tree is $1 = 2^{h+1} - 1 = 2^1 - 1 = 1$.

Inductive hypothesis: Assume that for complete binary tree of height h, the number of leaves is 2^h and the size of the tree is $2^{h+1} - 1$.

We want to show that for a complete binary tree of height $h + 1$, the number of leaves is 2^{h+1} and the size of the tree is $2^{h+2} - 1$.

By the definition of a complete binary tree, a tree T of height $h + 1$ consists of two complete binary trees, T_1 and T_2, of height h whose roots are connected to a new root. Let L be a function that represents the number of leaves for a tree.

$$L(T) = L(T_1) + L(T_2)$$
$$= 2^h + 2^h \quad \textit{by the induction hypothesis}$$
$$= 2(2^h)$$
$$= 2^{h+1}$$

Let N be a function that represents the size of a tree.

$N(T) = 1 + N(T_1) + N(T_2)$
$= 1 + (2^{h+1} - 1) + (2^{h+1} - 1)$ *by the induction hypothesis*
$= 2(2^{h+1}) - 1$
$= 2^{h+2} - 1$

Question 3.

a. { (1,5), (2,5), (3,4), (3,5), (4,5) }
b. The array (n, n-1, …, 2, 1) has the most inversions. Every pair of i, j will have an inversion. It will have $\sum_1^n i = n(n - 1)/ 2$.
c. The relationship between the runtime of insertion sort and the number of inversions in the input array is the larger the number of inversions in array, the longer insertion sort runtime will be. Suppose that array A contains no inversions, then the inner while loop will never execute since A[i] will always be <= *key*. Now suppose that array A starts with an inversion (m, n) where A[m] > A[n]. When the inner while loop starts executing, A[m] is in A[i], where 1 <= i < m. Each iteration of the while loop results in an elimination of one inversion since it only moves A[i] one position to the right if the element is less than key. For every inversion, the inner while loop must perform more swaps.
d. Count-Inversions(*A, n*)

```
1 if n == 1
2     return 0
3 B = A[0...n/2]
4 C = A[n/2...n]
5 b = Count-Inversions(B, n/2)
6 c = Count-Inversions(C, n/2)
7 d = Merge-Inversions(B,C)
8 return b + c + d
```

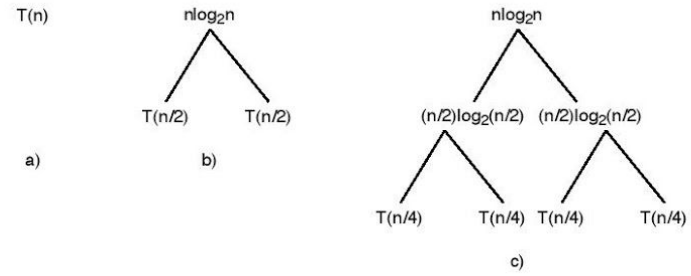
This algorithm computes the number of inversions in an input array A. It is correct because it divides the input array into two sub arrays of half the size recursively and then calls Merge-Inversion to merge the arrays and count the inversions. Merge-Inversion is a modified Merge-Sort where as it goes through each element of both subarray it compares whether $B[i]$ or C[i] is greater. Depending on which is bigger you know that it is inverted with all the elements with index less than i in the other array. Sum all the inversions for every subproblem and you

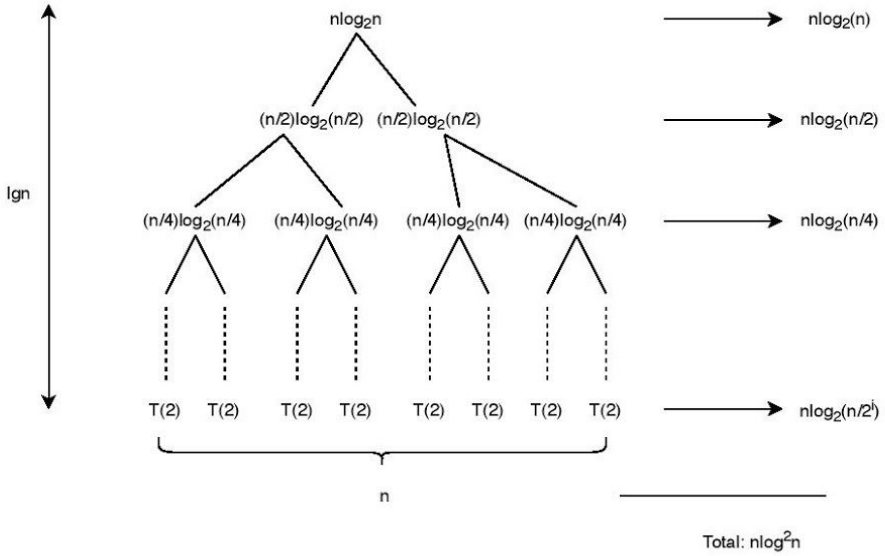
end up with the total inversion count. Lines 5,6 take log(n) time to complete while line 7 takes n time to complete for a total complexity of O($n$log$n$).

Question 4.

O	o	Ω	ω	Θ
yes	yes	no	no	no
tes	yes	no	no	no
no	no	no	no	no
no	no	yes	yes	no
yes	no	yes	no	yes
yes	no	yes	no	yes

Question 5.


T(n)
nlog2n
T(n/2)
T(n/2)
a)
b)
nlog2n
(n/2)log2(n/2)
(n/2)log2(n/2)
T(n/4)
T(n/4)
T(n/4)
T(n/4)
c)


lgn
nlog2n
(n/2)log2(n/2)
(n/2)log2(n/2)
(n/4)log2(n/4)
(n/4)log2(n/4)
(n/4)log2(n/4)
(n/4)log2(n/4)
T(2)
T(2)
T(2)
T(2)
T(2)
T(2)
T(2)
T(2)
n
nlog2(n)
nlog2(n/2)
nlog2(n/4)
nlog2(n/2i)
Total: nlog2n

Since the subproblem size decreases by a factor of 2 each level, we will eventually reach a boundary condition. The subproblem size for a node at depth i is $n/2^i$. The base case is T(2), thus the subproblem size hits $n = 2$ when $n/2^i = 2$, or when $i = \log_2 n$. Thus the tree has $\log_2 n + 1$ levels. The bottom level, at depth $\log_2 n$, has $2^{\log 2(n)} = n$ nodes. The cost per level is $n\log_2 n - ni$. The total cost is guessed as follows,

$$
\begin{aligned}
T(n) &= n\log_2 n + n\log_2(n/2) + \ldots + n\log_2(n/2^{\log 2(n)-1}) + O(n) \\
&= \sum_{i=0}^{\log 2(n)-1} (n/2^i)\, n + O(n) \\
&= \sum_{i=0}^{\log 2(n)-1} (n\log_2 n - ni) + O(n) \\
&= n\sum_{i=0}^{\log 2(n)-1} (\log_2 n - i) + \mathrm{O}(n) \\
&= n\sum_{i=1}^{\log 2(n)} (i) + \mathrm{O}(n) \\
&= n(\log_2 n(\log_2 n + 1)/2) + \mathrm{O}(n) \qquad \text{(A.1)} \\
&= (n(\log_2 n)^2 + \mathrm{n}\log_2 n)/2 + \mathrm{O}(n) \\
&= \mathrm{O}(\mathrm{n}(\log_2 n)^2)
\end{aligned}
$$

Proof:

We want to show that $T(n) \le d(\mathrm{n}(\log_2 n)^2)$ for some constant $d > 0$. Using substitution method.

$$
\begin{aligned}
T(n) &\le 2T(n/2) + n\log_2(n) \\
&\le 2d((n/2)(\log_2(n/2))^2) + n\log_2(n) \\
&= 2d((n/2)(\log_2(n) - \log_2(2))(\log_2(n) - \log_2(2))) + n\log_2(n) \\
&= 2d((n/2)((\log_2 n)^2 - 2\log_2(n) + 1)) + n\log_2(n) \\
&\le d(\mathrm{n}(\log_2 n)^2)
\end{aligned}
$$

Question 6.

b)Fib_B(n, A)

```
1 if A[n] not NULL
2     return A[n]
3 if n ≤ 1
4     A[n] = n
5 else
6     A[n] = Fib_B(n - 1, A) + Fib_B(n - 2, A)
7 return A[n]
```

This algorithm computes the number in the Fibonacci sequence indexed by n. The fibonacci sequence is calculated by adding the previous two terms, starting with 0,1…, n. It is correct because it stores calculated fib values in an array A, using the formula fib(n-1) + fib(n-2) which is how the sequence is defined. The array it builds is the fibonacci sequence with A[n] being the n^{th} element in the fibonacci sequence. All the lines run in constant time except for line 6, which has two recursive calls to n-1 and n-2. Since we store computed values in A, the longest branch in the recursion tree will resolve all the other branches. The size of the longest branch is n thus the complexity is O(n), specifically O(nA(n)), when n is large, where A(n) is the complexity of adding f(n-1) and f(n-2). . The space complexity is the size of the array which is n, therefore O(n).

d) The time to run asn1_a was 1m43.303s while asn1_b was 0m0.022s. Asn1_b computes fib(n) much quicker because it is much more efficient. Asn1_a has complexity $O(2^n)$ while asn1_b has complexity O(n).

e) Asn1_a cannot use int type of 4 bytes to compute F(50) precisely. The largest number that can be represented by 4 bytes is $2^{32} - 1 = 4294967295$ which is less than F(50) = 12586269025. Asn1_b computes F(300) precisely because it uses bigint type of size 8 bytes which can represent a number as large as $2^{64} - 1$ which is greater than F(300).