Assignment #1

Student #: [redacted]

Name: Zaid Albirawi

UWO email: zalbiraw@uwo.ca

1. Find the formula for the summation $\sum_1^n i(i+1)$. Hint $i(i+1) = {(i+1)^3 - i^3 - 1}/{3}$.

Using the hint to solve the reoccurrence equation: $\frac{(i+1)^3-i^3-1}{3}$

$$\sum_1^n i(i+1) = \frac{(1+1)^3-1^3-1}{3} + \frac{(2+1)^3-2^3-1}{3} + \frac{(3+1)^3-3^3-1}{3} + \frac{(4+1)^3-4^3-1}{3} + \cdots + \frac{(n+1)^3-n^3-1}{3}$$

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

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

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

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

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

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

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

Since, $\sum_1^n 1 = n$ then

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

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

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

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

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

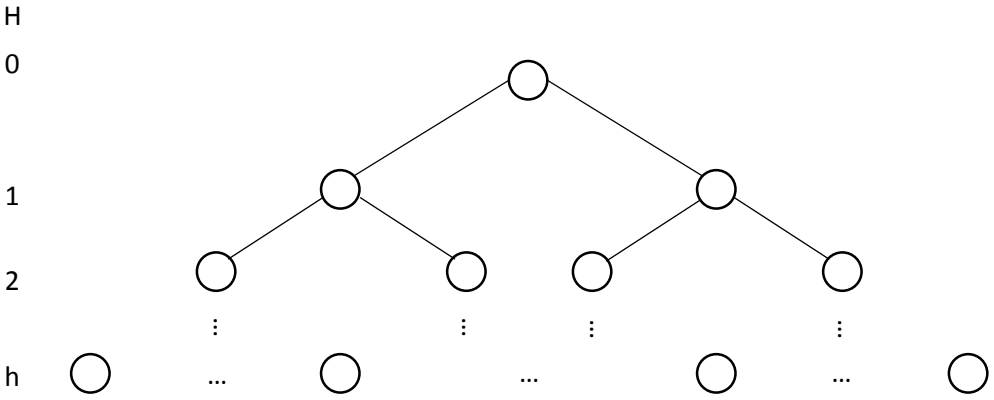
$$\frac{(n^3+2n^2+\cancel{n}+n^2+2n\cancel{+1})\cancel{-n-1}}{3}$$

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

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

Therefore, $\sum_1^n i(i+1) = \frac{n(n+1)(n+2)}{3}$

2. **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 the tree is 2^h and the size of the tree (number of nodes in *T*) is $2^{h+1} - 1$.



Height	#nodes	#leaves
0	1	1
1	3	2
2	7	4
3	15	8
4	31	16
...	...	...
h	2^{h+1}-1	2^h

***Proof** #nodes*:

Basis Case:

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

Inductive Case: $n(T) = n(T_1) + n(T_2) + 1$

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

Therefore, the size of the tree is $2^{h+1} - 1$.

***Proof** #leafs*:

Basis Case: $l(0) = 2^0 = 1$
Inductive Case: $l(T) = l(T_1) + l(T_2)$

$2^h = 2^{(h-1)} + 2^{(h-1)}$
$2^h = (2^{-1})2^h + (2^{-1})2^h$
$2^h = {}^{2^h}/_2 + {}^{2^h}/_2$
$2^h = \cancel{(2)}\,{}^{2^h}/_{\cancel{2}}$
$2^h = 2^h$

Therefore, the number of the leafs in the tree is 2^h.

3. Although merge sort runs in O(n long n) worst-case time and insertion sort runs in O(n^2) worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to coarsen the leaves of the recursion by using insertion sort within merge sort when sub-problems become sufficiently small. Consider a modification to merge sort in which *n/k* sub-lists of length *k* are sorted using insertion sort and then merged using the standard merging mechanism, where *k* is a value to be determined.

    a. Show that insertion sort can sort the *n/k* sub-lists, each of length *k*, in *O (n k)* worst-case time.

    Worst case: The worst case for the insertion sort is when the list that is being sorted is stored in reverse. Therefore, this gives the us *O(n^2)*

    But since we are sorting *n/k* sub-lists, each of length *k*, then the time complexity equation will be, (#sub-lists)*(time complexity of the insertion algorithm's worst case)

    $$\frac{n}{k} * k^2 = \frac{nk^{\cancel{2}}}{\cancel{k}} = nk$$

    Therefore, the time complexity of the insertion sort on *n/k* sub-lists, each of length *k,* is *O(n k)*.

b. Show how to merge the sub-lists in *O (n log (n/k))* worst-case time.

Since the worst case for the merge sort algorithm is *O (n log (n))*, and since this algorithm uses a recursively generated tree to sort the provided list. We can see that the log (n) part of the time-complexity equation represents the height of that tree. Furthermore, the merge sort algorithm creates a tree of height *n* at which every leaf contains the smallest possible number of elements. However, the modified algorithm will create a tree of height *n/k* at which every leaf will contain a sub-list of k length. These sub-lists will be sorted using the insertion algorithm. Lastly, the merge algorithm will start merging all the sub-lists, by comparing the sorted elements of every sub-list pair, to create the final sorted list; this operation will take n time. Therefore, since the recursive tree will have a height of *n/k* and the merge operation will take n time we can conclude that the worst-case time-complexity equation will be *O (n log (n/k)).*

c. Given that the modified algorithm runs in *O(n k + n log(n/k))* worst-case time, what is the largest value of *k* as a function of *n* for which the modified algorithm has the same running time as standard merge sort, in terms of O-notation?

$$O\left(nk + n\log\left(\frac{n}{k}\right)\right)$$
$$O(nk + n\log n - n\log k)$$

We then choose $\log n$ to replace k because $\log n$ if we replace k with a function larger more than $n\log n$ the term nk will have a higher order than $O(n\log n)$. Therefore, $\log n$ is largest value of *k* as a function of *n* for which the modified algorithm has the same running time as standard merge sort, in terms of O-notation.

$$O(n(\log n) + n\log n - n\log(\log n))$$
$$O(n\log n + n\log n - n\log\log n)$$
$$O(2n\log n - n\log\log n)$$
$$O(n\log n)$$

Therefore, the k value for which the modified algorithm has the same running time as standard merge sort is O(n log n.)

d. How should we choose *k* in practice?

The variable k should be the largest value were the insertion sort is faster than the merge sort.

4. 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 tale with "yes" or "no" written in each box.

	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ	Yes	Yes	No	No	No
b.	n^k	c^n	Yes	Yes	No	No	No
c.	$\sqrt{n}$	$n^{\sin n}$	No	No	No	No	No
d.	2^n	$2^{n/2}$	No	No	Yes	Yes	No
e.	$n^{\lg c}$	$c^{\lg n}$	Yes	No	Yes	No	Yes
f.	$\lg(n!)$	$\lg(n^n)$	Yes	No	Yes	No	Yes

5. Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove or disprove each of the following conjectures.

   a. $f(n) = O\big(g(n)\big)$ Implies $g(n) = O\big(f(n)\big)$.

   Proof by counterexample:

   let $f(n) = x, g(n) = x^2$
   $x = O(x^2)$ but $x^2 \neq O(x)$

   Therefore, proven by the counterexample.

   c. $f(n) = O\big(g(n)\big)$ Implies $\lg\big(f(n)\big) = O\big(\lg\big(g(n)\big)\big)$, where $\lg\big(g(n)\big) \geq 1$ and $f(n) \geq 1$ for all sufficiently large n.

   Since $f(n) = O\big(g(n)\big)$, then

   $f(n) \leq c * g(n)$

   $\lg\big(f(n)\big) \leq \lg(\, c * g(n)) = \lg\big(f(n)\big) \leq \cancel{\lg c} + \lg(\, g(n)) = \lg\big(f(n)\big) \leq \lg(\, g(n))$

   Hence, proposition holds.

d. $f(n) = O(g(n))$ Implies $2^{f(n)} = O(2^{g(n)})$

Since $f(n) = O\big(g(n)\big)$, then $f(n) \leq c * g(n)$

$2^{f(n)} \leq 2^{c*g(n)} = \ 2^{f(n)} \leq \frac{2^c}{2} + 2^{g(n)} = 2^{f(n)} \leq 2^{g(n)}$

Therefore, the proposition holds.

f. $f(n) = O(g(n))$ Implies $g(n) = \Omega(f(n))$

If $f(n) = O\big(g(n)\big)$, then $f(n) \leq \frac{c_1}{1} * g(n) \ \equiv f(n) \leq g(n)$

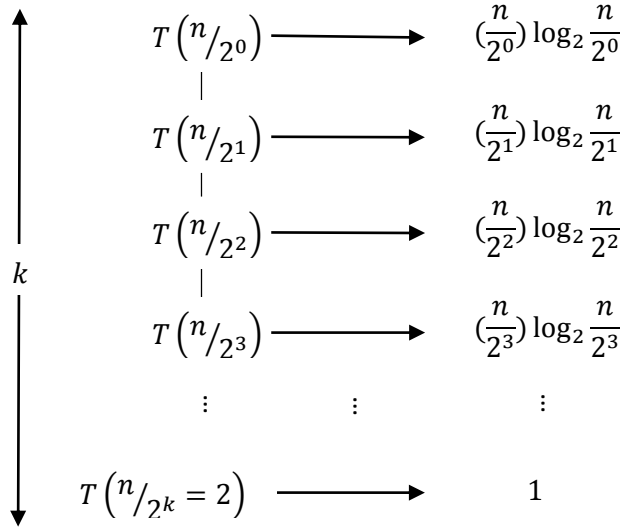
Also, if $g(n) = \Omega(f(n))$, then $g(n) \geq \frac{c_2}{2} * f(n) \equiv g(n) \geq f(n)$

Hence the proposition holds.

6. Suppose that the running time of a recursive program is represented by the following recurrence relation:

$$T(2) = 1$$
$$T(n) \leq T\left({}^{n}/_{2}\right) + n \log_2 n$$

Determine the time complexity of the program using recurrence tree method.



$\frac{n}{2^k} = 2$
$n = 2 * 2^k$
$n = 2^{k+1}$
$\log_2 n = \log_2 2^{k+1}$
$\log_2 n = (k+1)\log_2 2$
$\log_2 n = \mathrm{k} + 1$

Therefore, $\mathrm{k} - 1 = \log_2 n$

$$(\sum_{i=0}^{\log_2 n} \left(\frac{n}{2^i}\right)\log(\frac{n}{2^i})) + 1$$
$$\left(\frac{n}{2^0}\right)\log\left(\frac{n}{2^0}\right) + \left(\frac{n}{2^1}\right)\log\left(\frac{n}{2^1}\right) + \left(\frac{n}{2^2}\right)\log\left(\frac{n}{2^2}\right) + \left(\frac{n}{2^3}\right)\log\left(\frac{n}{2^3}\right) + \cdots + \left(\frac{n}{2^{\log n}}\right)\log\left(\frac{n}{2^{\log n}}\right) + 1$$
$$(n)\log(n) + \left(\frac{n}{2^1}\right)\left(\log n - \log 2^1\right) + \left(\frac{n}{2^2}\right)\left(\log n - \log 2^2\right) + \left(\frac{n}{2^3}\right)(\log n - \log 2^3) + \cdots + \left(\frac{n}{n}\right)\log\left(\frac{n}{n}\right) + 1$$
$$(n)\log(n) + \left(\frac{n}{2^1}\right)(\log n - 1) + \left(\frac{n}{2^2}\right)(\log n - 2) + \left(\frac{n}{2^3}\right)(\log n - 3) + \cdots + (1)\log 1 + 1$$
$$(n)\log(n) + \frac{n}{2^1}\log n - \frac{n}{2^1} + \frac{n}{2^2}\log n - \frac{2n}{2^2} + \log n - \frac{3n}{2^3} + \cdots + 0 + 1$$
$$(n)\log(n) + \frac{n}{2^1}\log n - \frac{n}{2^1} + \frac{n}{2^2}\log n - \frac{2n}{2^2} + \frac{n}{2^3}\log n - \frac{3n}{2^3} + \cdots + 0 + 1$$
$$(n)\log n + \frac{n}{2^1}\log n + \frac{n}{2^2}\log n + \frac{n}{2^3}\log n - \frac{n}{2^1} - \frac{2n}{2^2} - \frac{3n}{2^3} + \cdots + 0 + 1$$
$$n\log n + n\log n\,(\frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + 0) - n(\frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots + 0) + 1$$

Assume that the term $(\frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + 0) \approx 1$ and that the term $(\frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots + 0) \approx 2$. Therefore,

$n\log n + n\log n\,(1) - n(2) + 1$
$2(n\log n) - 2n + 1$
$2n(\log n - 1) + 1$
$2n(\log n - \log 2) + 1$
$2n * \log(\frac{n}{2}) + 1$
Therefore, the time complexity of the program is $O(n\log n)$

7. Consider the Fibonacci numbers:

$$F(n) = F(n-1) + F(n-2), n > 1; F(0) = 0; F(1) = 1$$

d. 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 Fibonacci recursive algorithm is very expensive and requires a significantly larger time to calculate when compared to the algorithm used in section b. The algorithm in section b, has a time complexity of *O(n),* while the recursive Fibonacci algorithm, used in section a has a time complexity of $O(2^n)$. The significant difference between the time complexities of those two algorithms is proven though the test results obtained by the UNIX time facility.

e. Can you use program a) to compute *F(50)*? Briefly explain your answer. Explain why your program in b) computes *F(300)* precisely.

No, program "a" cannot be used to compute the 50^{th} term in the Fibonacci sequence because the 50^{th} term of the sequence is larger than 2^{32}, which is the maximum number that an Integer object is able to store. However, program b is able to compute the 300^{th} term of the sequence precisely, because it uses an object called "bigInt" which is able to store very larger numbers because it treats those numbers as a linked list of integers.