Student Name: Ashna Mittal

Student Number: 251206758

CS 3340B: Assignment 1

Question 1: $\sum_{1}^{n}$ i(i + 1)

Solution 1: Using $i(i+1)$ = $i(i+1) = \frac{(i+1)^3-i^3-1}{3}$

Substituting i = 1, 2, 3, 4, …n

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

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

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

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

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

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

Using $\sum_{1}^{n} 1 = n$, the substituted equation is-

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

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

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

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

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

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

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

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

Hence the summation formula for $\sum_1^n i(i+1)$ is : $\frac{n(n+1)(n+2)}{3}$

Question 2: To prove : $Ln = Fn - 1 + Fn + 1, n > 0$ where Fi is the Fibonacci number.

Solution 2: The Fibonacci sequence is defined by the recurrence relation $Fn = Fn - 1 + Fn - 2$ with initial conditions $F_0 = 0 \; and \; F_1 = 1$. Lucas numbers L_n are defined by the recurrence relation $L_n = L_{n-1} + L_{n-2}$, with initial terms $L_1 = 1 \; and \; L_2 = 2$. To sum up:

n	0	1	2	3	4	5	6	7	8	…
F_n	0	1	1	2	3	5	8	13	21	…
L_n	2	1	3	4	7	11	18	29	47	…

Using Mathematical Induction, the base case is:

For n=1,

$L_1 = 1$

$F_{n-1} = F_{1-1} = F_0 = 0$

$F_{n+1} = F_{1+1} = F_2 = 1$

Therefore, we get $L_1 = F_{n-1} + F_{n+1}$ as the base case.

Assuming: $L_k = F_{k-1} + F_{k+1}$ for all $1 \le k \le n$

To prove: $L_{k+1} = F_k + F_{k+2}$

Using the definition of Lucas numbers from above, we know that,

$L_{k+1} = L_k + L_{k-1}$

$= (F_{k-1} + F_{k+1}) + (F_{k-2} + F_k)$ (Using inductive hypothesis)

$= (F_{k-1} + F_{k-2}) + (F_{k+1} + F_k)$

$L_{k+1} = F_k + F_{k+2}$ (Using $F_{n+2} = F_{n+1} + F_n$ and $F_n = F_{n-1} + F_{n-2}$)

Therefore, from above, we can conclude that for any n > 0, $L_n = F_{n-1} + F_{n+1}$.
Hence proved.

Question 3: You can also think of insertion sort as a recursive algorithm. In order to sort $A[1{:}n]$, recursively sort the subarray A [1 : n-1] and then insert $A[n]$ into the sorted subarray $A[1{:}n-1]$. Write pseudocode for this recursive version of insertion sort. Give a recurrence for its worst-case running time.

Solution 3: Pseudocode for the recursive version of Insertion sort-

```
Recursive-Insertion-Sort(A, n)
    if n > 1
        Recursive-Insertion-Sort(A, n-1)
        Insert(A, n)
    end if
end Recursive-Insertion-Sort

Insert(A, n)
    key = A[n]
    j = n - 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j - 1
    end while
    A[j + 1] = key
end Insert
```

In the pseudocode above, ‘A’ is the array to be sorted, and ‘n’ is the number of elements in the array to sort. The algorithm first recursively sorts the subarray ‘A[1...n-1]’, and then it inserts the ‘n-th’ element into the sorted subarray.

For the worst-case running time, we can consider the recurrence relation for the insertion sort algorithm. The worst-case scenario for insertion sort is when the array is in reverse order, requiring each insertion to move every element that has already been sorted.
The recurrence for the worst-case running time ‘T(n)’ of the recursive insertion sort is:

$$T(n) = \begin{cases} \Theta(1) \text{ if n } = 1 \\ T(n-1) + \Theta(n)\ if\ n > 1 \end{cases}$$

The ‘T(n-1)’ term represents the time to sort n-1 elements, and ‘Θ(n)’ represents the time to insert the nth element into its proper position in the sorted subarray of n-1 elements. This leads to a quadratic running time since we have to perform an operation that is linear in the number of elements for each element in the array.
To solve this recurrence relation, we can expand it:

$T(n) = T(n-1) + n$

$= (T(n-2) + n - 1) + n$

$= (T(n-3) + n - 2) + n - 1 + n$

$= \ldots$

$= 1 + 2 + \cdots + (n-1) + n$

This sum is the arithmetic series which simplifies to:

$T(n) = n(n+1)/2 = \Theta(n^2)$

Hence, the worst-case running time for the recursive version of insertion sort is $\Theta(n^2)$.

Question 4: Although merge sort runs in $\Theta(nlgn)$ worst-case time and insertion sort runs in $\Theta\ (n^2)$ 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 subproblems become sufficiently small. Consider a medication 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 $\Theta(nk)$ worst-case time.

b. Show how to merge the sub lists in $\Theta\big(nlg(n/k)\big)$ worst-case time.

c. Given that the modified Algorithm runs in $\Theta\big(nk + nlg(n/k)\big)$ 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 Θ -notation?

Solution 4:

a) Insertion sort has a worst-case time complexity of $\Theta(k^2)$ for a single sub list of length k. Since, we have n/k such sub lists, we multiply this time complexity by the number of sub lists to get the total time.

$$\frac{n}{k}\Theta(k^2) = \Theta\left(\frac{n}{k}\,k^2\right) = \Theta(nk).$$

This shows that using insertion sort on n/k sub lists, each of length k, will have a worst-case time complexity of $\Theta(nk)$.

b) Merging two sorted lists of length k, has a linear time complexity, $\Theta(k)$. In the merge sort algorithm, we merge n/k sub lists into $n/2k$ sub lists, then $n/4k$ sub lists, and so on, until we have one sorted list. This is a total of $lg(n/k)$ merge steps since each step halves the number of sub lists. This means that the depth of the merge tree is $lg(n) - lg(k) = lg(n/k)$.

The total work done at each level of merging is $\Theta(n)$ because we're merging two halves of the list, and the number of elements being merged totals to n. Since we have $lg(n/k)$ levels, the total work for merging is:

$$\Theta(n) \cdot \Theta(log(n/k)) = \Theta(nlog(n/k))$$

c) Viewing k as a function of n, as long as $k(n) \in O\big(lg(n)\big)$, it has the same asymptotic. In particular, for any constant choice of k, the asymptotics are the same.

The modified algorithm runs in $\Theta(nk + nlg(n/k))$ worst-case time. To find the largest value of k for which the modified algorithm has the same running time as standard merge sort, we set the modified running time equal to the running time of standard merge sort, which is $\Theta(nlg(n))$

$$O(nk + nlog(n/k)) = O(nlogn)$$

To maintain the equivalence, the nk term needs to be negligible compared to $nlg(n)$, which implies k should be a function that grows slower than $lg(n)$. Since $nlg(\text{k}/n)$ is already $\Theta(nlg(n))$, we can ignore it for this comparison. We can then solve for k in terms of n when nk is on the order of $nlg(n)$:

$$nk = n \log n$$

$$k = \log n$$

Thus, the largest value of k as a function of n for which the modified algorithm has the same running time as the standard merge sort is:

$$k = \log n$$

Question 5: Relative Asymptotic growths

Solution 5:

	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.	$n^{\log c}$	$c^{\log n}$	yes	no	yes	no	yes
f.	$log(n!)$	$log(n^n)$	yes	no	yes	no	yes

Question 6: Determine the time complexity of the program using recurrence tree method (not using master theorem) and then prove your answer.

$$T(2) \leq 2c$$
$$T(n) \leq 2T(n/2) + cnlog_2(n) \quad n > 2$$

Solution 6: The recurrence relation given is:

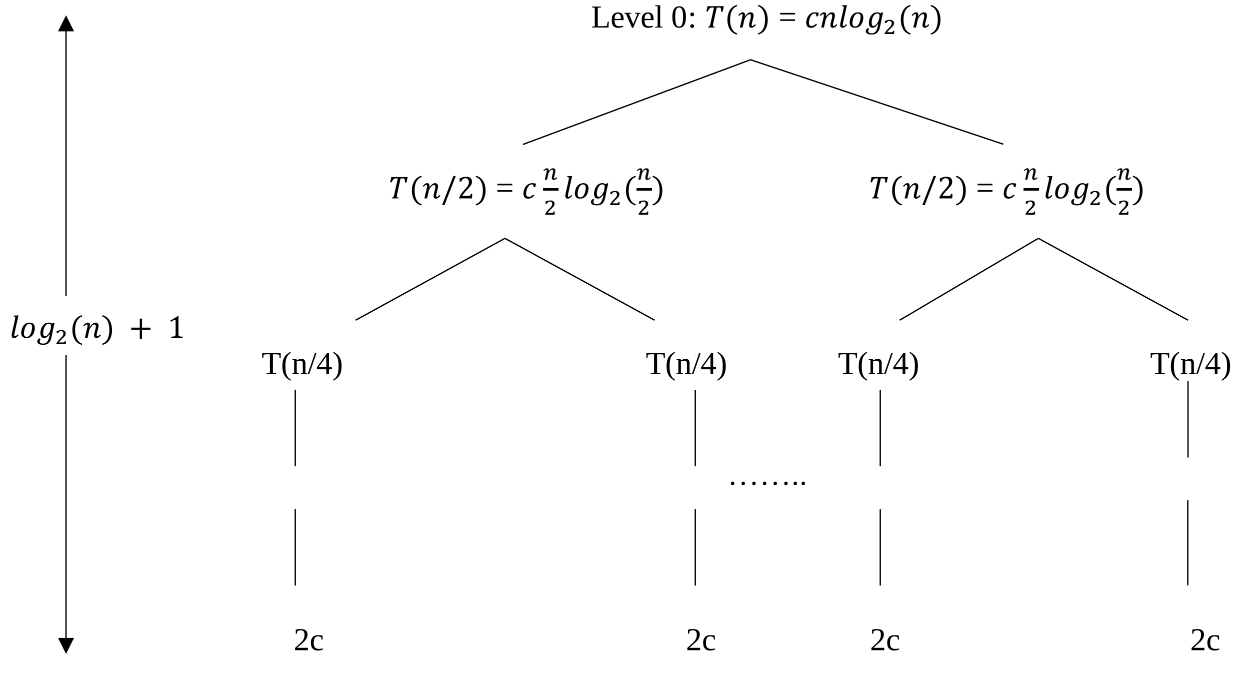
$$T(2) \leq 2c$$
$$T(n) \leq 2T(n/2) + cnlog_2(n) \quad n > 2$$

We will use the recurrence tree method to determine the time complexity. In a recurrence tree, each node represents the cost at a certain level of recursion, and the branches represent recursive calls. Illustrating the first few levels of the tree:

1. At level 0 (the root), we have a single problem of size n costing $cnlog_2(n)$.
2. At level 1, this problem is divided into 2 subproblems of size $\frac{n}{2}$, each costing $c\frac{n}{2}log_2(\frac{n}{2})$.

This division continues, with each level i having 2^isubproblems, each of size $\frac{n}{2^i}$, and each costing $c\frac{n}{2^i}log_2(\frac{n}{2^i})$. The height h of the tree is reached when the size of the subproblems is 2, that is when $\frac{n}{2^h} = 2$. Solving for h gives us $h = log_2(n) - 1$.

Let's calculate the total cost at each level and then sum them up.



Level 0: $T(n) = cnlog_2(n)$
$T(n/2) = c\frac{n}{2}log_2(\frac{n}{2})$
$T(n/2) = c\frac{n}{2}log_2(\frac{n}{2})$
$log_2(n) + 1$
T(n/4)
T(n/4)
T(n/4)
T(n/4)
……..
2c
2c
2c
2c

Level 0 cost: $cnlog_2(n)$

Level 1 cost: 2 . $(c\frac{n}{2}log_2(\frac{n}{2}))$ = $cnlog_2(n) \ - \ cn$

...

Level i cost : 2^i . ($c\frac{n}{2^i}log_2(\frac{n}{2^i})$) = $cnlog_2(n) \ - \ cni$

The total cost is the sum of costs at each level from 0 to h:

$$\sum_{i=0}^{h}(cnlog_2(n) \ - \ cni \ )$$

Now, we need to evaluate this sum:

$$\sum_{i=0}^{h}(cnlog_2(n) \ - \ cni \ ) = cnlog_2(n)(h \ + 1) - cn\sum_{i=0}^{h} i$$

Since h = $log_2(n)$, we have:

$$cnlog_2(n)log_2(n) - cn\sum_{i=0}^{log_2(n)-1} i$$

Now, let's compute the sum $\sum_{i=0}^{log_2(n)-1} i$, which is an arithmetic series:

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

Substituting this back into the total cost expression:

$$cnlog_2(n)log_2(n) - cn\frac{(log_2(n)-1)log_2(n)}{2}$$

Since $log_2(n)log_2(n)$ is the dominant term, we can ignore the lower-order term $-cn\frac{(log_2(n)-1)log_2(n)}{2}$ when discussing asymptotic behavior. Therefore, the time complexity is:

$T(n)$ = O($nlog_2(n)log_2(n)$)

This indicates that the running time grows faster than $nlog_2(n)$ but slower than $n(log_2(n))^2$
It's a superlinear polynomial time complexity.

Question 7: The

Solution 7:

a) Output for Recursive function to compute L_{i*5} is:

```
Lucas(0) = 2
Lucas(5) = 11
Lucas(10) = 123
Lucas(15) = 1364
Lucas(20) = 15127
Lucas(25) = 167761
Lucas(30) = 1860498
Lucas(35) = 20633239
Lucas(40) = 228826127
Lucas(45) = -1757246660
Lucas(50) = -1921017949
```

To compile and run this program, use the following commands in your terminal:

```
javac asn1_a.java
java asn1_a
```

b) Output for Recursive function using matrix multiplication with time complexity $O(n)$ to compute L_{i*20} is:

```
Lucas(0) = 2
Lucas(20) = 15127
Lucas(40) = 228826127
Lucas(60) = 3461452808002
Lucas(80) = 52361396397820127
Lucas(100) = -1139155321138466361
Lucas(120) = -2795939413257253630
Lucas(140) = 5347811994664660839
Lucas(160) = -8271524584511622593
Lucas(180) = -4435149930091013822
Lucas(200) = 8566728179384764591
Lucas(220) = 4955201673824879479
Lucas(240) = -352179712341101566
Lucas(260) = -1268672955607851337
Lucas(280) = -6249783109692392593
Lucas(300) = 362950400237129026
Lucas(320) = -529246468703212673
Lucas(340) = -387354483789832153
Lucas(360) = 7082585619549648130
Lucas(380) = -29558693758691065
Lucas(400) = 6951655735469402015
Lucas(420) = -7162095078750790846
Lucas(440) = 8210776971223662927
Lucas(460) = -8789253581020694569
Lucas(480) = 881586227177639938
Lucas(500) = 7548146805174218327
```

To compile and run this program, use the following commands in your terminal:

```
javac asn1_b.java

java asn1_b
```

Time complexity is $O(nA(n))$ where $A(n)$ is the complexity to add L_{n-1} and L_{n-2}. $A(n)$ is $O(1)$, the $O(n)$ comes from the loop that adds L_{n-1} and L_{n-2} as the loop iterates n times. So, the time complexity is $O(n(1))$ which is $O(n)$.

c) The time needed for asn1_a is:

```
real  0m43.257s
user  0m43.011s
sys    0m0.226s
```

The time needed for asn1_b is:

```
real    0m0.077s
user    0m0.051s
sys     0m0.020s
```

To compile and run this program, use the following commands in the terminal to run the script:

```
./measure_time.sh
```

This will help in running the script and seeing the above outputs. Alternatively, we can also run the following commands to see the output:

```
javac asn1_a.java
javac asn1_b.java
time java asn1_a
time java asn1_b
```

Based on above results, we can conclude the following:

- The direct recursive computation of Lucas numbers (**asn1_a**) will likely show a significant increase in execution time as **n** grows, due to the exponential time complexity of the algorithm.
- The efficient version (**asn1_b**) using matrix exponentiation should show relatively little change in execution time as **n** increases, due to its logarithmic time complexity.
- The matrix exponentiation method is better than direct recursion for computing Lucas numbers at bigger indexes, as can be seen by comparing the real times of the two executions.

d) For part a), the recursive approach to compute \`$L_{50}$\` could lead to integer overflow if a 4-byte \`int\` type is used. In Java, an \`int\` is a 32-bit signed integer, which has a maximum value of 2^{31} - 1 = 2,147,483,647. The Lucas numbers grow exponentially, and while \`$L_{50}$\` itself is within the range of a 4-byte \`int\`, the recursive calls made by the function without any form of memoization or dynamic programming result in many repeated calculations, which can quickly lead to values that exceed this range during the computation before arriving at the final result for \`$L_{50}$\`.

For part b), the program can compute \`$L_{500}$\` precisely even with a 4-byte \`int\` because it uses a more efficient algorithm based on matrix exponentiation. This method computes \`$L_n$\` using a logarithmic number of steps, and it only involves multiplication and addition operations. If we use a data type like \`long\` in Java, which is a 64-bit signed integer with a maximum value of 2^{63} - 1 = 9,223,372,036,854,775,807), we can avoid overflow for \`$L_{500}$\` and compute it precisely. The matrix exponentiation technique does not involve repeated recalculations of the same Lucas numbers, which avoids the combinatorial explosion of recursive calls and the associated risk of overflow.

To conclude:

- The program in part a) may not be able to compute \`$L_{50}$\` using a 4-byte \`int\` due to potential overflow during intermediate calculations.
- The program in part b) can compute \`$L_{500}$\` precisely because it avoids intermediate values that exceed the range of a 64-bit \`long\`, and it performs far fewer operations due to its logarithmic time complexity.