3340 Assignment 1

Arsalaan Ali

Question 1

Find the Formula for the summation:

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

We know that

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

Write out the equation for n = 3

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

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

$$[\ (n-1)^3 - (n-2)^3 - 1 + (n)^3 - (n-1)^3 - 1 + (n+1)^3 - n^3 - 1\ ] / 3$$

Combine like terms

$$[\ -(n-2)^3 - 1 + -1 + (n+1)^3 - 1\ ] / 3$$

Since c = 3 we know that n-2 = 1

$$[\ -(1)^3 - 1 + -1 + (n+1)^3 - 1\ ] / 3$$

The term -1 shows up (n+1) times

$$[(n+1)^3 - (n+1)]/3$$

Therefore the formula for the summation is:

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

Question 2

Prove that Ln = Fn−1 + Fn+1 using induction

Basis

L1 = F0 + F2

L1 = 0 + 1

L1 = 1

This is true, therefore basis is proved

Induction

Assume for all i<n that $L_i = F_{i-1} + F_{i+1}$

Show that $L_n = F_{n-1} + F_{n+1}$

$L_n = L_{n-1} + L_{n-2}$

$L_n = F_{n-2} + F_n + F_{n-3} + F_{n-1}$

$L_n = (F_n + F_{n-1}) + (F_{n-2} + F_{n-3})$

$L_n = F_{n-1} + F_{n+1}$

Therefore

$L_n = F_{n-1} + F_{n+1}$

Question 3

Insertion Sort Pseudocode

```
def insertion(nums):
    # base case, if array is size of length 1
    if len(nums) == 1:
        return nums

    # get the last element in the array (element to be inserted)
    val = nums[-1]

    # recursive call, get the sorted array of [0 : n-1]
    nums = insertion(nums[0:len(nums)-1])

    # loop through sorted array and insert the current element
    i = 0
    while i < len(nums) and nums[i] < val:
        i += 1
    left = nums[0:i]
    right = nums[i::]

    # return the sorted array
    return left + [val] + right
```

Worst-case recurrence

Base Case

For n = 1, we just return the array with a single element in it, therefore it is 0(1) run time.

Other Cases

For n > 1, we have a recursive call to n-1, and then we have an O(n) loop through the array to insert the value, therefore it is T(n-1) + O(n) run time.

Recurrence Equation:

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

Question 4

a) Sorting n/k sublists in O(nk) worst case time

Insertion sort runs in O(n^2) runtime, so for a sublist of size k, it would be O(k^2)
So sorting n/k sublists at O(n^2) runtime would be:

n/k * O(k^2) = O(nk)

b) Merging in O(n lg(n/k)) worst case time

There are n/k sublists, we have to merge each of these sublists
Since the sublists are already sorted, it will take log(n/k) steps to merge each of them
In each step we are running an O(n) operation to merge them.

Therefore it is O(n lg(n/k)) worst case time.

c) Largest value of k for which runtime is same as mergesort

Our algorithm sorts K sublists with insertion sort: O(nk)
And then merges those n/k sublists: O(n lg(n/k)
Therefore it has O(nk + n lg(n/k)) runtime

Standard merge sort has a runtime of O(n lg(n))
If we want our algorithm to have the same runtime as merge sort we need:

O(nk + n lg(n/k) = O(n lg(n))

K cannot be greater than lg(n), because if it is then the time complexity would be greater than O(n lg(n)) because of nk.

Therefore the possible largest value of k is lg(n)

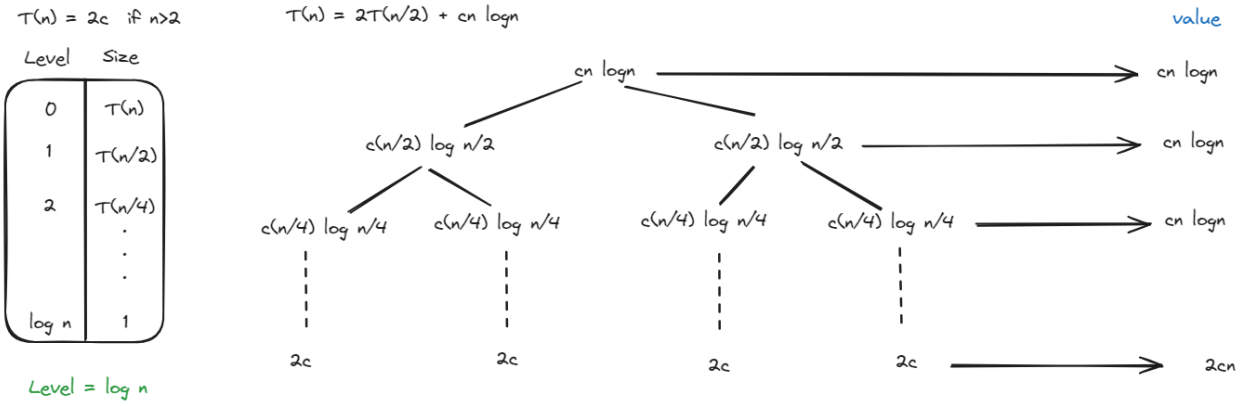
d) How do you choose k in practice
We need a value for k where insertion sort runs faster than merge sort

We can choose different values for k and test with both insertion and merge sort, and find the largest value of k where insertion is faster than merge.

Question 5

A	B	O	o	Ω	ω	⊖
$lg^k n$	n^ϵ	yes	yes	no	no	no
n^k	c^n	yes	yes	no	no	no
$\sqrt{n}$	$n^{\sin n}$	no	no	no	no	no
2^n	$2^{n/2}$	no	no	yes	yes	no
$n^{lg\, c}$	$c^{lg\, n}$	yes	no	yes	no	yes
$lg(n!)$	$lg(n^n)$	yes	no	yes	no	yes

Question 6



T(n) = level * value
= logn * cn logn
= c (logn * n logn) remove constants
= logn * n logn
= n (logn)^2

Question 7

b) Calculating L(500)

```java
import java.util.*;

public class asn1_b {
    public static void main(String[] args) {
        for (int i = 0; i <= 25; i++) {
            System.out.println(FastLucas(i * 20));
        }
    }
    public static BigInteger FastLucas(int n) {
        BigInteger[] memo = new BigInteger[n+2];
        memo[0] = new BigInteger("2");
        memo[1] = new BigInteger("1");
        return FastLucasHelper(n, memo);
    }

    private static BigInteger FastLucasHelper(int n, BigInteger[] memo) {
        if (memo[n] != null) {
            return memo[n];
        }
        BigInteger res = FastLucasHelper(n - 1, memo).add(FastLucasHelper(n - 2, memo));
        memo[n] = res;
        return res;
    }
}

class BigInteger {
    private List<Integer> digits;

    public BigInteger(String number) {
        digits = new ArrayList<>();
        for (int i = number.length() - 1; i >= 0; i--) {
            digits.add(Character.getNumericValue(number.charAt(i)));
        }
    }
    public BigInteger() {
        digits = new ArrayList<>();
    }

    public BigInteger add(BigInteger other) {
        BigInteger result = new BigInteger();
        int size = Math.max(digits.size(), other.digits.size());
        int carry = 0;
        for (int i = 0; i < size; i++) {
            int first = 0;
            int second = 0;
```

```
            if (i < digits.size()) {
                first = digits.get(i);
            }
            if (i < other.digits.size()) {
                second = other.digits.get(i);
            }
            int sum = first + second + carry;
            int remainder = sum % 10;
            carry = sum >= 10 ? 1 : 0;
            result.digits.add(remainder);

        }
        if (carry == 1) {
            result.digits.add(1);
        }
        return result;
    }

    @Override
    public String toString(){
        String result = "";
        for(int i = digits.size()-1; i>=0; i--){
            result+=digits.get(i);
        }
        return result;
    }
}
```

We use an array to implement memoization of the values, this way we never have to recalculate any values, allowing us to have O(n) runtime.

We implement a custom BigInteger class to be able to add and store the large numbers together.

c) Time results:
The first implementation of the Lucas numbers function runs in exponential time, so it takes much longer to calculate much smaller numbers:

1m27s to calculate up to L(50)

The second implementation runs in linear time, so it is much faster to calculate even much bigger numbers

0m0.0217s to calculate up to L(500)

This drastic time difference is indicative of the gigantic difference between exponential and linear time complexity

d) Integer size to store L(500)

An integer type of 4 bytes cannot store a number as large as L(500), therefore if you try to calculate L(500) you will get an integer overflow and result with a number much smaller than the actual answer

I used a custom BigInteger class that I wrote, this class uses an array of integers to represent each of the digits of a number, and has a custom add function that manually adds the integers and handles carrying over.

With this method you can calculate L(500) precisely.