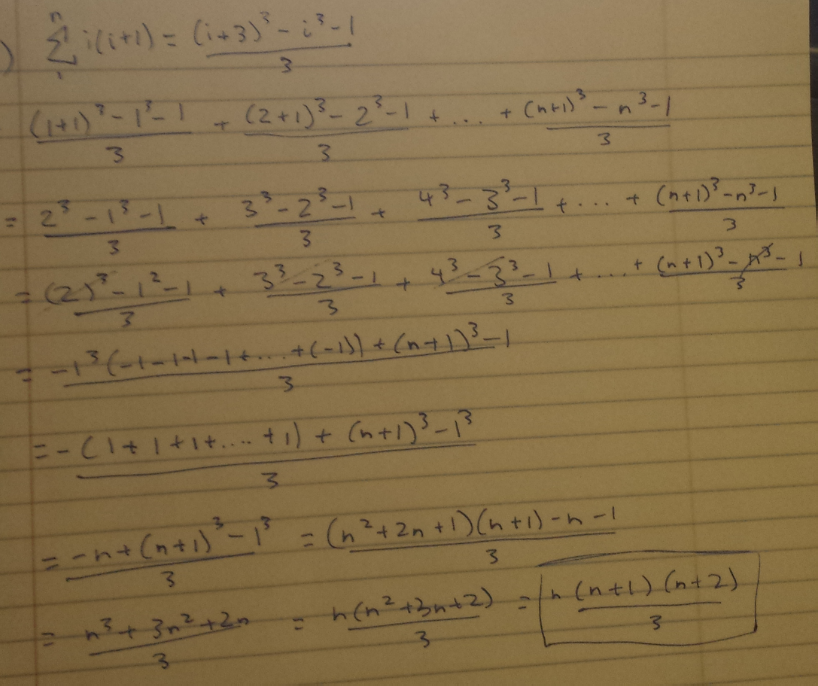
**Yaqzan Ali – yali6**

**Cs3340 Assignment 1**

**1)**

****

**2)**

Based on this table, we can see the correlation between height and leaves of a binary tree:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| h | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| leaves | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 156 |

We can see that, for all h>0, the leaves of a binary tree is double that of the previous size.

*Proof:*

**Induction Base:** For **h=0**,

20 = 1, indeed a binary tree of height 0 has 1 leaf (the root).

**Induction Hypothesis:** Assume a binary tree of height h has 2h leaves.

Then we must show that a binary tree of height h+1, has 2h+1 leaves.

Indeed, 2h+1 = **2(2h)**

By the induction hypothesis, 2h is the number of leaves of a binary tree at height h (the previous size). Therefore, we can conclude that 2h is the height of a binary tree.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Based on this table, we can see the correlation between height and size of a binary tree:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| h | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| size | 1 | 3 | 7 | 15 | 31 | 63 | 127 | 155 | 311 |

We can see that, for all h>0, the size of a binary tree is the size of 2 times the previous size, plus 1.

*Proof:*

**Induction Base:** For **h=0,**

2(0+1) -1 = 21 – 1 = 2 – 1 = **1**

Indeed a binary tree of height 0 has size 1.

**Induction Hypothesis:** Assume 2(h+1) – 1 is the size of a binary tree of height h.

Indeed 2((h+1)+1) – 1 = 2(2(h+1) – 1) + (21 – 1) = 2(2(h+1) – 1) + 1

By induction hypothesis, 2(h+1) – 1 is the size of a binary tree of height h, and the next height is indeed twice that, plus 1. Therefore, we can conclude that 2(h+1) – 1 is the size of a binary tree at height h.

**3) a)**

Array of [1 . . . n] of n elements

Split into (n/k) groups of k length

Goal: A[1] is the smallest, A[2] is the second smallest and so on, for each group

Assume A[1…(k-1)] are sorted. Then to insert A[k], we need at most (k-1) comparisons and k data movements.

This implies T(k) ≤ c · k2 which means T(k) = O(k2)

We would then need to repeat this (n/k) times, for each group. This would mean the time complexity would be O(k2 (n/k)) = **O(nk)**

**b)**

The total number of merge-sort recursions in a tree is log(n) + 1, where n is the number of leaves, or the input size. Proof: Assume as an inductive hypothesis that the number of levels of a recursion tree with 2i leaves is log 2i + 1 = i + 1 (since for any value of i, we have that log 2i = i ). Because we are assuming that the input size is a power of 2, the next input size to consider is 2i + 1. A tree with n = 2i+1 leaves has

one more level than a tree with 2i leaves, and so the total number of levels is (i + 1) + 1 = log 2i+1 + 1.

Since we are dealing with subgroups of size (n/k), instead of leaves, we can say that the number of recursions in a tree is log(n/k) + 1. Since each level of recursion has cn iterations, we do cn(log(n/k)+1) = cnlog(n/k) + cn. Removing constant terms and lower order functions, we get **O(c log(n/k)).**

**c)**

Running time for mergesort is O(n Log(n)).

In order for O(nk + nlog(n/k)) to have the same runtime, the highest value k can be is **1.**

Plugging in 1, we get (n(1) + nlog(n/1)) = (n + nlog(n)). Removing the lower order terms, we get O(nlogn)

**d)**

In practice, we would find the largest k possible

**4)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | Big - O | Little - o | Omega | w | O |
| Logk n | ne | Yes | yes | no | no | no |
| nk | cn | yes | yes | no | no | no |
| Root(n) | nsinn | No | No | No | No | no |
| 2n | 2n/2 | no | no | yes | yes | no |
| nlogc | clogn | yes | no | yes | no | yes |
| Log(n!) | Log(nn) | yes | yes | no | no | no |

**5) a)**

This is **incorrect.**

Assume F(n) = n and G(n) = n2

Then N = O(n2), but n2 ≠ O(n)

**c)**

This is **correct.**

If we take the log of both sides, then

Log(f(n) ≤ log(c\*g(n))

Log(f(n) ≤ log(c) + log(g(n)) for all n ≥ n0

If c’ = log(c) +1, then log(f(n)) ≤ log(c)log(g(n)) + log(g(n)) = **c’log(g(n))** for all n ≥ n0

Therefore, log(f(n)) = c’log(g(n)) = **O(log(g(n)))**

**d)**

This is **incorrect.**

Assume f(n) = 2n and g(n) = n

2n = O(n), however

22n ≠ 2n and 4n ≠ 2n

Therefore, 2f(n) ≠ O(2g(n))

**f)**

**This is correct.**

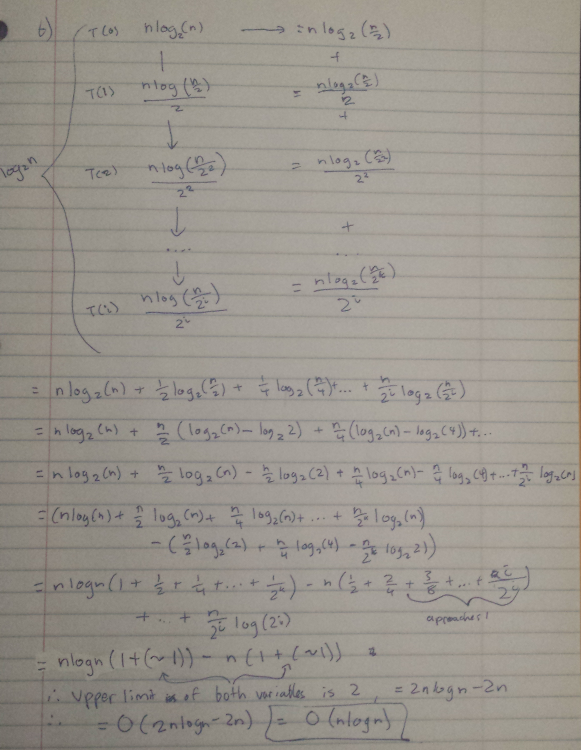
If f(n) = O(g(n), then

0 ≤ f(n) ≤ c\* g(n) for some c > 0 and all n> n0

0 ≤ (1/c)f(n) ≤ g(n)

G(n) = Ω (d(n))

6)



**7)**

**d)**

Part A took 90.436344 milliseconds and part B took 35.8758 milliseconds. Although part B computed 10 times more numbers than part A, it still managed to finish in a third of the time. This is because the time complexity of part a (O(2^n)) is much slower than part b (O(n)).

**e)**

My program in a) cannot compute the value of F(50), because that digit exceeds the maximum value that the primitive integer can hold. The highest Fibonacci number an integer can hold is the 48th value. Even if the integer class was capable of holding more than that, it would still take a long time to compute because of the time complexity O(2^n).

My Program in b) uses an iterative approach, which reduces the time complexity to O(n). Since the primitive integer value cannot hold more than the 48th Fibonacci number, I had to make my own Big Integer Data class.