CS 61B
Spring 2021

Homework 2
March 15, 2021

1 Asymptotic Warm Up

Give the tightest asymptotic bound on `foo(n)`.

```
public int foo(int n) {
    if (n == 0) {
        return 0;
    }
    bloop(n);
    return foo(n / 3) + foo(n / 3) + foo(n / 3);
}

public int bloop(int n) {
    for (int i = 0; i < n; i += 1) {
        System.out.println("Ah, loops too");
    }
    return n;
}
```

$O(n)$

Analysing the following calls:

foo(3); -> 4 operations
foo(9); -> 13 operations
foo(27); -> 40 operations
foo(81); -> 121 operations
foo(243); -> 364 operations
foo(729); -> ~1080 operations
foo(2190); -> ~3240 operations

Notice a pattern of ~1.5N recursive operation growth for foo.

Bloop also takes N operations, so total runtime is N^2

2 Asymptotic Potpourri

Note: These are hard problems. If you are stuck on it for a long time, move on to other problems, and post on Ed or come to Office Hours so we can help you.

For the following methods, give the runtime of in Θ notation. Your answer should be a function of N that is as simple as possible with no unnecessary leading constants or lower order terms.

(a) Give the runtime of `mystery1(n)` in Θ notation.

```java
public void mystery1(int n) {
    for (int i = n; i > 0; i = i / 2) {
        for (int j = 0; j < 100000000; j += 2) {
            System.out.println("Hello World");
        }
    }
}
```

i = 1, i > 0, i / 2 -> this is a log n runtime loop, as it halves everytime

j = 0, i < 1000000, j += 2.
this is constant runtime, there is always 5000000 calls.
so log n * 5000000, drop the constant.

Runtime is Theta(log n)

(b) Give the runtime of `mystery2(n)` in Θ notation.

```java
public void mystery2(int n) {
    for (int i = 1; i < n; i += 1) {
        for (int j = 0; j < n; j += 1) {
            i = i * 2;
            j = j * 2;
        }
    }
}
```

The outer loop runs once every time
as i > n inside inner loop every time.
This is Theta(1)

The inner loop doubles j everytime,
which can be thought of as "halving n".
Hence inner loop runs at Theta(log n)

1* log n is still in log terms, so runtime
is Theta(log n)

(c) What sum represents the work done by `mystery3(n)`? No need to simplify the sum, just write out the first few terms and the last term.

Lets analyse n = 8

for (i = 8, i > 0, i = i/2)
i = 8, for (j = 1, j < 64; j *= 2): j = 1, j = 2, j = 4, j = 8, j = 16, j = 32
i = 4, for (j = 1, j < 16; j *= 2): j = 1, j = 2, j = 4, j = 8
i = 2, for (j = 1, j < 4; j *= 2): j = 1, j = 2
i = 1

i loop has a runtime log2(n)

```java
public void mystery3(int n) {
    for (int i = n; i > 0; i = i / 2) {
        for (int j = 1; j < i * i; j *= 2) {
            System.out.println("Hello World");
        }
    }
}
```

Generalising j
for n = 8, 6 j iterations, n = 4, 4 j iterations, n = 2, 2 j iterations
for n = 16, j < 256. j = 1, j=2 …, j = 32, j = 64, j = 128, 8 iterations
for n = 32, j < 1024, j = 1, j = 2, … , j = 128, j = 256, j = 512, 10 iterations

so we are growing by 2 additional iterations everytime we double n.
this is definitely log n, as we double in size, we grow by a constant

ahhh! we can represent this as log2(i^2). As i also runs in log time.
So we can represent this as a summation: sigma(i=n, n/2, n/4){ 2log2(i)}
so the first few terms are 2log2(n), 2log2(n/2)…

(d) Give the runtime of `mystery4(n)` in Θ notation. Assume that the `SLList` constructor, and the `size` and `addFirst` methods take constant time.

```java
public void mystery4(int n) {
    SLList<Integer> list = new SLList<>();
    for (int i = 1; list.size() < n; i += 1) {
        for (int j = 0; j < i; j += 1) {
            list.addFirst(j);
        }
        System.out.print(list.size() + " + ");
    }
}
```

n = 4
i = 1, j=0, j=1, list size = 2
i = 2, j =0, j=1, j=2, list size = 5 BREAK

n=8
i = 3, i=0, j=1,j=2,j=3, list size = 9
we grow by 4, and we have 4 additional operations

n=16
i=4, 5 operations, list size = 14
i=5, 6 operations, list size = 20
we grew by 8, and we have 11 additional operations

n = 32
i = 6, 7 operation, list size = 27
i = 7, 8 operations, list size =35
we grow by 16, and we have 15 additional operations

n=64
i = 8, 9 operations, list size = 44
i = 9, 10 operations, list size = 54
i = 10, 11 operations, list size = 65
we grow by 32 and we had 30 additional operations

Therefore, as n doubles, number of operations as doubles.
Therefore, we are growing by Theta(n)

3 WQU

(a) Draw the Weighted Quick Union object on 0 through 10, that results from the following `connect` calls. Do not use path compression. What is the resulting underlying array? If we connect two sets of equal weight, we will tie-break by making the set whose root has a larger number the parent of the other (the opposite tie-breaking scheme as discussion 6).

[1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

`connect(0, 1);` [1, -2, -1, -1, -1, -1, -1, -1, -1, -1]

`connect(2, 3);` [1, -2, 3, -2, -1, -1, -1, -1, -1, -1]

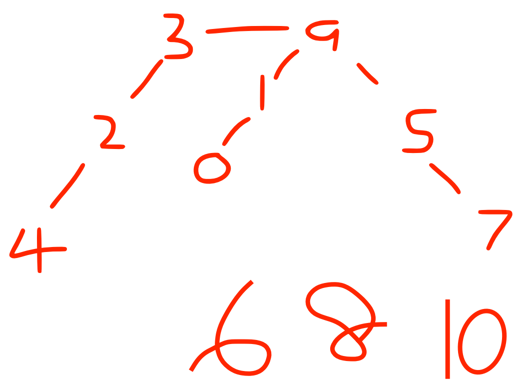
`connect(9, 5);` [1, -2, 3, -2, 9, -1, -1, -1, -1, -2]

`connect(5, 7);` [1, -2, 3, -2, 9, -1, 5, -1, -1, -3]

`connect(7, 1);` [1, 9, 3, -2, 9, -1, 5, 9, -1, -5]

`connect(4, 2);` [1, 9, 3, -3, 9, -1, 5, -1, -1, -5]

`connect(3, 1);` [1,9,3,9,2,9,-1, 5, -1, -8, -1]



(b) Assume that a single node has a height of 0. What are the shortest and tallest heights for a Quick Union object with 16 connected elements? What about for a Weighted Quick Union object?

(c) What are the best and worst runtimes for `connect` and `isConnected` in a Quick Union object with N connected elements? What about in a Weighted Quick Union object?

b. Shortest: both have height 1 (all leafs connected to root.
Tallest: Quick Union is a linked list with height 15.
Weighted is a tree with log2(15) height which is ~ 4

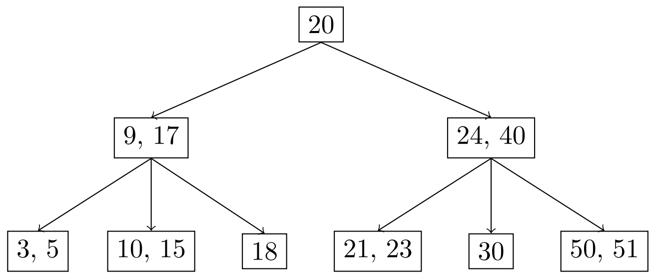
c. Best case runtime for connect and isConneced is 1 in Quick Union and Weighted Quick Union (connect a leaf to the root)

Worst case runtime for connect and isConnected is N for Quick Union (linked list tree).

Worst case runtime for connect and isConnected is logN for Weighted Quick Union

4 Switcheroo

(a) Consider the following 2-3 tree. Convert it to an LLRB, and describe the 6 LLRB operations to balance the tree after inserting the number 11. The LLRB operations are: `rotateRight(x)`, `rotateLeft(x)`, and `colorFlip(x)`.



(b) After inserting 11 and balancing the LLRB, how many nodes are on along the longest path from the root to a leaf.

(c) After inserting 11 and balancing the LLRB, how many red links are on along the longest path from the root to a leaf.

5 Mechanical Hashing

Suppose we insert the following words into an initially empty hash table, in this order: **kerfuffle**, **broom**, **hroom**, **ragamuffin**, **donkey**, **brekky**, **blob**, **zenzizenzizenzic**, and **yap**. Assume that the hash code of a String is just its length (note that this is not actually the hash code for Strings in Java). Use separate chaining to resolve collisions. Assume 4 is the initial size of the hash table's internal array, and double this array's size when the load factor is equal to 1. Illustrate this hash table with a box-and-pointer diagram.

For each index of the final hash table, specify what Strings are stored in it. If it is empty, write "none".