Pearploy Chaicharoensin 6381278

Task1: Hello, Definition

---

1) Show, using either definition,that $f(n) = n$ is $O(nlogn)$.

   By definition of Big O: $f(n) \in O(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$

   Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n}{nlogn} = 0 < \infty$

   Then $f(n) = O(nlogn)$.

2) In class,we saw that Big-O multiplies naturally. You will explore this more formally here. Prove the following statement mathematically.

$$d(n) = O(f(n)) \quad --> \quad \lim_{n\to\infty} \frac{d(n)}{f(n)}$$

$$e(n) = O(g(n)) \quad --> \quad \lim_{n\to\infty} \frac{e(n)}{g(n)}$$

$$\text{Then } d(n) \cdot e(n) \quad --> \quad \lim_{n\to\infty} \frac{d(n)}{f(n)} \cdot \lim_{n\to\infty} \frac{e(n)}{g(n)}$$

$$= \lim_{n\to\infty} \frac{d(n)\cdot e(n)}{f(n)\cdot g(n)}$$

$$d(n)\, e(n) = O(f(n) \cdot g(n))$$

3)
```
void fnA(int S[]) {
        int n = S.length; //O(1)
        for (int i=0;i<n;i++) { ////O(n)
                fnE(i, S[i]); //O(1000n)
        }
}
```

The running time of fnA is $O(n^2)$

4) Show that $h(n) = 16n^2 + 11n^4 + 0.1n^5$ is not $O(n^4)$.

   By definition of Big O: $f(n) \in O(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$

   Since $\lim_{n\to\infty} \frac{16n^2+11n^4+0.1n^5}{n^4} = +\infty$

   Then $f(n) \notin O(n^4)$

Pearploy Chaicharoensin 6381278

Task 2: Poisoned Wine

---

1) As we have n bottles of wine, we label number 1 to n on each bottle of wine.
2) Separate the bottles of wine into 2 groups left and right: [1,2,3,...,$\frac{n}{2}$] and [$\frac{n}{2}$ + 1,...,n-1,n]
3) Since the poison is very strong we can drop a tiny bit of each bottle from the left group in the first glass and each from the right group in the second glass.
4) We'll give the first glass of mixed wine to a taster and the second glass of mixed wine to another taster. And wait for the result.
5) When the symptom is occured, we'll look at what group (left or right) that the taster drank the wine from and we'll just ignore the other group completely, since we know that there is no poisoned wine in the other group. We will keep repeating the process 2) to 4) with the group that has the poisoned wine.
6) At the end, after we break down into 2 groups recursively, we will get to have the final bottle with poison. We now can read the number that is labeled on the bottle to know what i-th bottle of wine is poisoned.

This is like a binary search where you break the data into a half and keep breaking them in half recursively to find the one poisoned bottle of wine from n wine bottles and return the n^{th} bottle that has poison by using only O(logn) times. In this case we use logn tasters.

Pearploy Chaicharoensin 6381278

Task 3 : How Long Does This Take?

```
void programA(int n) {
        long prod = 1;  //Θ(1)
        for (int c=n;c>0;c=c/2) // Θ(logn+1)
                prod = prod * c; //Θ(1)
}
```

ProgramA:

For-loop inside	
Iterator number	c
1	n
2	$\frac{n}{2}$
3	$\frac{\frac{n}{2}}{2} = \frac{n}{4}$
:	:
k	$\frac{n}{2^{k-1}}$

If $\frac{n}{2^{k-1}}$ is the last iteration that will happen in the loop and c can be at least 1, since c > 0 .

Then
$$\frac{n}{2^{k-1}} = 1$$
$$n = 2^{k-1}$$
$$log_2 n = log_2 2^{k-1}$$
$$log_2 n = (k-1) log_2 2$$
$$log_2 n = k - 1$$
$$k = log_2 n + 1$$

The total time that the programA takes is Θ(1) + Θ(logn+1) + Θ(1), which is Θ(logn).

```
void programB(int n) {
        long prod = 1;//Θ(1)
        for (int c=1;c<n;c=c*3) //Θ(log(n-4) -1)
                prod = prod * c;//Θ(1)
}
```

ProgramB:

For-loop inside	
Iterator number	c
1	1
2	3
3	9
:	:
k	3^{k-1}

If 3^{k-1} is the last iteration that will happen in the loop and c can at least be n-1, since c < n .

Then $$n - 1 = 3^{k-1}$$

$$log_2(n-1) = log_2(3^{k-1})$$
$$log_2(n-1) = (k-1)log_2(3)$$
$$\frac{log_2(n-1)}{log_2(3)} = k - 1$$
$$log_2(n-1-3) - 1 = k$$
$$k = log_2(n-4) - 1$$

The total time that the programB takes is Θ(1) + Θ(log(n-4)+1) + Θ(1), which is Θ(logn).

Pearploy Chaicharoensin 6381278

Task4: Halving Sum

Part I:

def hsum(X): # assume len(X) is a power of two

    while len(X) > 1:

        (1) allocate Y as an array of length len(X)/2 $//k_1 \cdot \frac{z}{2}$

        (2) fill in Y so that Y[i] = X[2i] + X[2i+1] for i = 0, 1, ..., len(X)/2 − 1 $//k_2 \cdot \frac{z}{2}$

        (3) X = Y $//k_2$

    return X[0]

The work done is $(\frac{k_1+k_2}{2})z + (k_2)$

Part II:

The input size is measured by the variable n.

While-loop		
Iterator number	len(x)	len(x) example
1	n	64
2	$\frac{n}{2}$	32
3	$\frac{n}{4}$	16
:	:	:
k	$\frac{n}{2^{k-1}}$	k= 6, $\frac{64}{2^{6-1}} = 2$

Since the last iteration of the while-loop len(x) > 1, then it can at least be 2.

So $\frac{z}{2^{k-1}} = 2$

$n = 2 \cdot 2^{k-1}$

$n = 2^k$

$\log n = k$

This while-loop takes O(log n) times

```
def hsum(X): # assume len(X) is a power of two
    while len(X) > 1: //O(log n)
        (1) allocate Y as an array of length len(X)/2  //O(1)
        (2) fill in Y so that Y[i] = X[2i] + X[2i+1] for i = 0, 1, ..., len(X)/2 − 1  //O(n/2 - 1) → O(n)
        (3) X = Y//O(1)
    return X[0] //O(1)
```

The total time that the code will take is O(log n)·O(n) + O(1) = O(nlog n) + O(1) = O(n log n).

Pearploy Chaicharoensin 6381278

Task5: More Running Time Analysis

---

(1) Determine the best-case running time and the worst-case running time of method 1 in terms of Θ.

```
static void method1(int[] array) {
        int n = array.length;
        for (int index=0;index<n-1;index++) {
        int marker = helperMethod1(array, index, n - 1);
    swap(array, marker, index);
   }
}
static void swap(int[] array, int i, int j) {
        int temp=array[i];
        array[i]=array[j];
        array[j]=temp;
}
static int helperMethod1(int[] array, int first, int last) {
        int max = array[first];
        int indexOfMax = first;
        for (int i=last;i>first;i--) {
                if (array[i] > max) {
                        max = array[i];
                        indexOfMax = i;
                }
        }
```

```
        return indexOfMax;
    }
```

The worst-case running time is $\Theta(n^2)$, when an array is sorted from small to large.
The best-case running time is $\Theta(n^2)$, when an array is already well-sorted from large to small.

(2) Determine the best-case running time and the worst-case running time of method 2 in terms of Θ.

```
    static boolean method2(int[] array, int key) {
        int n = array.length;
        for (int index=0;index<n;index++) {
            if (array[index] == key) return true;
        }
        return false;
    }
```

The worst-case running time is $\Theta(n)$, when we have to loop through every element either the last element is equal to the key or none of the elements is.
The best-case running time is $\Theta(n)$, when the first element in an array is equal to the key.

(3) Determine the best-case running time and the worst-case running time of method 3 in terms of Θ.

```
static double method3(int[] array) {
        int n = array.length;
        double sum = 0;
        for (int pass=100; pass >= 4; pass--) {
                for (int index=0;index < 2*n;index++) {
                        for (int count=4*n;count>0;count/=2)
                                sum += 1.0*array[index/2]/count;
                        }
                }
                return sum;
}
```

Both of the worst-case running time and the best-case running time take $\Theta(nlogn)$ times, they both have to run through 3 for-loops to get the sum.

Task6: Recursive Code

```
// assume xs.length is a power of 2
    int halvingSum(int[] xs) {
        if (xs.length == 1) return xs[0]; //O(1)
        else {
            int[] ys = new int[xs.length/2];  //O(1)
            for (int i=0;i<ys.length;i++)  //O(n)
                ys[i] = xs[2*i]+xs[2*i+1];  //O(1)
            return halvingSum(ys);  //T(n/2)
        }
    }
```

The input size is measured by the variable n.

The time that halvingSum method takes is T(n) = $T(\frac{n}{2}) + O(n) = O(n)$

```
    int anotherSum(int[] xs) {
        if (xs.length == 1) return xs[0]; //O(1)
        else {
            int[] ys = Arrays.copyOfRange(xs, 1, xs.length); //O(n − 1)
            return xs[0]+anotherSum(ys); //O(1) + T(n − 1)
        }
    }
```

The input size is measured by the variable n.

The time that anotherSum method takes is T(n) = $T(n-1) + O(n) = O(n^2)$

```
int[] prefixSum(int[] xs) {
        if (xs.length == 1) return xs; //O(1)
        else {
                int n = xs.length; //O(1)
                int[] left = Arrays.copyOfRange(xs, 0, n/2); //O(n/2)
                left = prefixSum(left); //T(n/2)
                int[] right = Arrays.copyOfRange(xs, n/2, n); //O(n/2)
                right = prefixSum(right); //T(n/2)
                int[] ps = new int[xs.length]; //O(1)
                int halfSum = left[left.length-1]; //O(1)
                for (int i=0;i<n/2;i++) { ps[i] = left[i]; } //O(n/2)
                for (int i=n/2;i<n;i++) { ps[i] = right[i - n/2] + halfSum; }//O(n/2)
                return ps; //O(1)
        }
}
```

The input size is measured by the variable n.

The time that prefixSum method takes is $T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + O(\frac{n}{2}) + O(1)$

$= 2T(\frac{n}{2}) + O(n) = O(nlogn)$

Task 7: Counting Dashes

```
void printRuler(int n) {
    if (n > 0) {
        printRuler(n-1); // print n dashes
        for (int i=0;i<n;i++) System.out.print('-');
        System.out.println();
        // --------------
        printRuler(n-1);
    }
}
```

Given:

$g(n)=2g(n-1)+n$, with $g(0)=0$ —(1)

$f(n)=2f(n-1)+1$, with $f(0)=0$ —(2)

$g(n) = a \cdot f(n) + b \cdot n + c$ —(3)

Find c:

From $g(n) = a \cdot f(n) + b \cdot n + c$

If n = 0: $0 = a \cdot 0 + b \cdot 0 + c$

$0 = 0 + 0 + c$

$c = 0$

Find a,b:

Substitute $g(n) = a \cdot f(n) + b \cdot n + c$ in $g(n) = 2g(n-1) + n$

Then, $a \cdot f(n) + b \cdot n = 2[a \cdot f(n-1) + b \cdot (n-1)]+n$

$a \cdot f(n) + b \cdot n = 2a \cdot f(n-1) + 2b \cdot (n-1) + n$

$$a \cdot f(n) - 2a \cdot f(n-1) = 2b \cdot (n-1) + n - bn$$

$$a(f(n) - 2f(n-1)) = 2b(n-1) + n - bn$$

From the given $f(n) = 2f(n-1) + 1$, we know that $f(n) - 2f(n-1) = 1$

$$a(1) = 2b(n-1) + n - bn$$

$$a = 2bn - 2b + n - bn$$

$$a = bn - 2b + n$$

$$a - bn + 2b - n = 0$$

$$(-b-1)n + (a+2b) = 0$$

Find a,b:

If $-b-1 = 0$

$b = -1$ —(1)

If $a+2b = 0$ —(2)

Substitute $b = 1$ in (2): $a+2(-1) = 0 \rightarrow a = 2$

Thus, $a= 2$, $b= -1$, and $c=0$

Recall: $g(n)=2g(n-1)+n$, with $g(0)=0$ —(1)

$f(n)=2f(n-1)+1$, with $f(0)=0$ —(2)

Plug in $a= 2$, $b= -1$, $c= 0$ in $g(n) = a \cdot f(n) + b \cdot n + c$

$$= 2 \cdot f(n) - n$$

$$= 2(2^n - 1) - n$$

The closed-form of $g(n) = 2^{n+1} - n - 2$

Let's use induction to verify that the closed form for g(n) actually works

Def $T(n) = 2T(n-1) + n$, $T(0) = 0$ and $g(n) = 2^{n+1} - n - 2$

Prove by induction

Predicate P(i) : T(i) == g(i)

Base Case P(1) : $T(1) = 2(0) + 1 = 1$

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

T(1) == g(1) **TRUE**

Inductive step Assume T(k) = g(k)

WTS T(k+1) = g (k+1)

RHS: $g(k+1) = 2^{k+2} - (k+1) - 2$

$= 2^{k+2} - k - 3$

LHS: $T(k+1) = 2T(k) + k + 1$ // by definition

$= 2g(k) + k + 1$ // by inductive hypothesis

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

$= 2^{k+2} - 2k - 4 + k + 1$

$= 2^{k+2} - k - 3$

Since we can prove that RHS = LSH, then we can conclude that by mathematical induction, the closed-form of g(n) can be written as $2^{n+1} - n - 2$.