hw5

Chunya Pattharapinya u6581030

November 12, 2024

1 Task 1: Hello, Definition

1.1 Show, using either definition, that f (n) = n is O(n logn).

By the definition of Big O: $f(n) \in O(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$$

Since

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

we conclude that

$$f(n) = O(n \log n).$$

1.2 Prove the following statement mathematically

Proposition: If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then the product $d(n)e(n)$ is $O(f(n)g(n))$.

$$d(n) = O(f(n)) \Rightarrow \lim_{n\to\infty} \frac{d(n)}{f(n)} < \infty$$

$$e(n) = O(g(n)) \Rightarrow \lim_{n\to\infty} \frac{e(n)}{g(n)} < \infty$$

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

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

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

1.3 What's the running time in Big-O of fnA as a function of n, which is the length of the array S.

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

Therefore, the running time of fnA is O(n^2)

1.4 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

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$$

$h(n) = 16n^2 + 11n^4 + 0.1n^5$ and $g(n) = n^4$.

$$\lim_{n\to\infty} \frac{h(n)}{n^4} = \lim_{n\to\infty} \frac{16n^2 + 11n^4 + 0.1n^5}{n^4}.$$

$$= \lim_{n\to\infty} \left(\frac{16n^2}{n^4} + \frac{11n^4}{n^4} + \frac{0.1n^5}{n^4} \right).$$

$$= \lim_{n\to\infty} \left(\frac{16}{n^2} + 11 + 0.1n \right).$$

$$\lim_{n\to\infty} \frac{h(n)}{n^4} = \infty.$$

Therefore,

$$h(n) \notin O(n^4).$$

2 Task 2: Poisoned Wine

Collaborators: Dominique Bachmann

1. **Label Each Bottle in Binary:**
   Number each of the n bottles from 1 to n and write each number in binary. For example, if $n = 8$, the bottles are labeled from 001 to 111 in binary.

2. **Give Each Tester a Bit to Check:**
   Each bottle number has $\lceil \log_2(n) \rceil$ bits. Each tester checks one position (bit) in the binary labels of the bottles: For $n = 8$, use 3 testers:

   - **Tester 1** checks the rightmost bit.

- **Tester 2** checks the middle bit.
- **Tester 3** checks the leftmost bit.

3. **Testing Procedure:**
Each tester drinks from bottles where their assigned bit is 1 in the binary label.

   - **Tester 1** (rightmost bit): bottles $001, 011, 101, 111$.
   - **Tester 2** (middle bit): bottles $010, 011, 110, 111$.
   - **Tester 3** (leftmost bit): bottles $100, 101, 110, 111$.

4. **Symptoms Appear:**
After 30 days, let's say **Tester 1** and **Tester 3** have symptoms .This gives us the binary number `101` (since **Tester 1** and **Tester 3** have matching number of 1s in `101`). Therefore, bottle number **5** have poison.

   This method requires $O(\log n)$ testers because:

   > Each tester is assigned to check one specific bit position in the binary labels of the bottles. We use exactly $\lceil \log_2(n) \rceil$ testers, which grows in proportion to $\log n$.

   Therefore, we can find the poisoned bottle using only $O(\log n)$ testers.

3 Task 3: How Long Does This Take?

3.1 programA

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

The loop starts with $c = n$ and halves c each iteration: $c = c/2$.
The loop stops when $c < 1$.
After k iterations, $c = \frac{n}{2^k}$.
Solving $\frac{n}{2^k} < 1$ gives $k \approx \log_2(n)$.
Each iteration performs a constant-time operation $O(1)$.

Since the loop runs $\log_2(n)$ times with $O(1)$ work per iteration, the total running time is:

$$\Theta(\log n)$$

3.2 programB

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

The loop starts with $c = 1$ and triples c each iteration: $c = c \times 3$.
The loop will stops when $c \geq n$.
After k iterations, $c = 3^k$.
Solving $3^k \geq n$ gives $k \approx \log_3(n)$.
Each iteration performs a constant-time operation $O(1)$.

Since the loop runs $\log_3(n)$ times with $O(1)$ work per iteration, the total running time is also:

$$\Theta(\log n)$$

4 Task 4: Halving Sum

```
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
        # (2) fill in Y so that Y[i] = X[2*i] + X[2*i+1] for i = 0, 1, ..., len(X)/2  1
        # (3) X = Y
    return X[0]
```

Step 1: $\mathrm{k}_1 \cdot \frac{Z}{2}$
Step 2: $k_2 \cdot \frac{Z}{2}$
Step 3: k_2

Total:

$$\left(\frac{k_1 + k_2}{2}\right) z + k_2$$

4.1 Part II

Iteration Number	**Length of X**	**Length of X**
1	n	64
2	$\frac{n}{2}$	32
3	$\frac{n}{4}$	16
$\vdots$	$\vdots$	$\vdots$
k	$\frac{n}{2^{k-1}}$	For $k = 6$, $\frac{64}{2^{6-1}} = 2$

The number of times the Length After k Iterations:

$$\frac{n}{2^{k-1}}$$

Condition for the Last Iteration The loop stops when $len(X) \leq 2$, so in the final iteration:

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

Solve for k:

$$n = 2^k$$

Taking the logarithm of both sides:

$$k = \log_2 n$$

Thus, the `while` loop runs $O(\log n)$ times.

5 Task 5: More Running Time Analysis

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

```
        max = array[i];
        indexOfMax = i;
        }
    }
    return indexOfMax;
}
```

Answer:
The worst-case running time is $\Theta(n^2)$.
The best-case running time is $\Theta(n)^2$

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

Answer:
The worst-case running time is $\Theta(n)$. When the key is on the last element of the array n length.
The best-case running time is $\Theta(1)$ When the key on the first index

```
    static double method3(int[] array) {
      int n = array.length;
      double sum = 0;
      for (int pass=100; pass >= 4; pass--) { //O(1)
        for (int index=0;index < 2*n;index++) { //O(n)
          for (int count=4*n;count>0;count/=2) // O(log(n)
              sum += 1.0*array[index/2]/count;
        }
     }
     return sum;
}
```

Answer:
Worst Case: $\Theta(nlog(n))$
Best Case: $\Theta(nlog(n))$.

6 Task 6: 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)
    }
}
```

Halving Sum:
$\mathrm{T(n)} = \mathrm{T}\left(\frac{n}{2}\right) + O(n) = O(n)$

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

Another Sum:
$\mathrm{T(n) = T(n - 1) + O(n) = O(n^2)}$

```
int[] prefixSum(int[] xs) {
    if (xs.length == 1) return xs; //O(1)
    else {
         int n = xs.length;
         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];
         int halfSum = left[left.length-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;
         }
    }
```

Prefix Sum:
$\mathrm{T(n)} = 2\mathrm{T}\left(\frac{n}{2}\right) + O(n) = O(nlog(n))$

7 Task 7: Counting Dashes

i. Find c

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

$n = 0$

$0 = a \cdot 0 + b \cdot 0 + c$

$0 = 0 + 0 + c$

$\mathbf{c = 0}$

ii. Find a,b

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

in

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

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

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

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

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

$Because f(n) = 2f(n-1) + 1$

$Therefore, f(n) - 2f(n-1) = 1$

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

$a = 2b \cdot n - 2b + n - b \cdot n$

$a = b \cdot n - 2b + n$

$a - b \cdot n + 2b - n = 0$

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

$b = -1$

$a = 2$

$c = 0$

$Thus,$

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

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

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

$\mathbf{g(n) = 2^{n+1} - n - 2}$.

iv:Use induction to verify that your closed form for g (n) actually works.

Theorem : $\mathbf{g(n) = 2g(n-1) + n}$ is equal to $G(n) = 2^{n+1} - n - 2$.

Proof by Induction:

Base Case: For $n = 0$:

$$LHS : g(0) = 0$$

$$RHS : G(0) = 2^{0+1} - 0 - 2 = 2 - 0 - 2 = 0$$

Since $LHS = RHS$, the base case holds true.

Inductive Step: Assume $g(n) = G(n)$
We want to show that $g(n+1) = G(n+1)$.

Proof:

$$g(n+1) = 2g(n) + (n+1)$$

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

by inductive hypothesis, where $g(n) = G(n)$:

$$LHS : g(n+1) = 2 \cdot (2^{n+1} - n - 2) + (n+1)$$
$$= 2^{n+2} - 2n - 4 + n + 1$$
$$= 2^{n+2} - n - 3$$

Thus,

$$LHS = RHS$$

Since $LHS = RHS$, the inductive step holds.
By mathematical induction, $g(n)$ is equal to $G(n)$.