CS208/204: Data Structures & Abstractions
Mahidol University International College

Assignment 5:
Performance Characterization

Austin J. Maddison

June 20, 2023

Contents

Task 1: Hello, Definition

P1

To prove that $f(n) \leq c \cdot g(n)$, such that c and n exist in $\mathbf{I}^+$, and $n \geq n_0$, where $f(n) = n$ and $g(n) = n \log n$. We need to find values for c and n_0 such that the inequality holds true.

Let's choose $c = 1$ and $n_0 = 1$.

For $n \geq 1$, we have:

$$f(n) = n$$
$$g(n) = n \log n$$

Substituting these values into the inequality, we get:

$$n \leq n \log n$$

Taking the logarithm of both sides, we have:

$$\log n \leq \log(n \log n)$$

Using the logarithmic property $\log(a \cdot b) = \log a + \log b$, we can rewrite the right side as:

$$\log(n \log n) = \log n + \log(\log n)$$

Now, the inequality becomes:

$$\log n \leq \log n + \log(\log n)$$

Since $\log n$ is a positive value, we can subtract $\log n$ from both sides, resulting in:

$$0 \leq \log(\log n)$$

For $n \geq 1$, the logarithm of the logarithm of n is always a positive value or zero. Therefore, the inequality $0 \leq \log(\log n)$ holds true.

We have shown that for $c = 1$ and $n_0 = 1$, $f(n) = n$ is less than or equal to $c \cdot g(n)$ for all $n \geq n_0$.

Therefore, we can conclude that $f(n) = n$ is $O(n \log n)$.

P2

Consider the proposition $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$ such that $d(n) \cdot e(n)$ is $O(f(n) \cdot g(n))$

To prove that $d(n) \cdot e(n)$ is $O(f(n) \cdot g(n))$ we got to make sure that $f(n) \cdot g(n)$ still valid under the definition of O(). Which means the equality:

$$d(n) \cdot e(n) \leq f(n) \cdot g(n) \text{ needs to be satisfied for } O(f(n) \cdot g(n)) \text{ to be valid.}$$

The equality can be proven because we know the

$$d(n) \text{ is } O(f(n)) \text{ and } e(n) \text{ is } O(g(n))$$

.
Which implies:

$$d(n) \leq f(n) \text{ and } e(n) \leq g(n)$$

Hence the product satisfies the equality

$$d(n) \cdot e(n) \leq f(n) \cdot g(n)$$

thus concludes the O(n) of $d(n) \cdot e(n)$ is $O(f(n) \cdot g(n))$.

P3

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

The following codes snippet's complexity is $O(n^2)$. The for loop inside fnA() has the time complexity of O(n) as it takes n operations to iterate through the list. The time complexity of the fnE() is also O(n) as n increases the amount of operations fnE has to do is linear. Notice that since the call for fnE() is inside fnA()'s for loop multiplies the time complexity making the amount of work needed to by done quadratic.

$$\underbrace{O(n)}_{\text{fnA()}} \cdot \underbrace{O(n)}_{\text{fnE()}} \text{ is } O(n^2)$$

P4

The reason why $h(n) = 16n^2 + 11n^4 + 0.1n^5$ is not n^4 is because $O(n^4)$ does not satisfy the definition of $O()$.
Let $f(n) = n^4$.

$$h(n) \text{ is not } \leq f(n) \text{ for all values}$$

.
We can prove this by evaluating the limits using L'Hopital:

$$\begin{aligned}
&= \lim_{n\to\infty} \frac{h(n)}{f(n)} \\
&= \lim_{n\to\infty} \frac{16n^2 + 11n^4 + 0.1n^5}{n^4} \\
&= \lim_{n\to\infty} \frac{32n + 44n^3 + 0.5n^4}{4n^3} \\
&= \lim_{n\to\infty} \frac{n + n^3 + n^4}{n^3}
\end{aligned}$$

This shows that $h(n)$ contains a polynomail of n with higher order than $f(n)$. This concludes that $h(n)$ is greater than $f(n)$ when n is a big number thus $f(n)$ cannot be an upper-bound to $h(n)$.

Task 2: Poisoned Wine

We can create a scheme to detect what wine bottle has poison by encoding each bottle using the testers. Since we can only have $log_2 n$ testers, this enough bits to represent $1, 2, \ldots, n$ wine bottles. The idea is better illustrated. For example lets have a scenario where there is n=4 (4 wine bottles) which means he get t=$\log_2(2) = 2(2testers)$.

This scenerio looks like the following:

Wine Bottle	T_1	T_2
W_1	0	0
W_2	0	1
W_3	1	0
W_4	1	1

So we have all the testers taste all the wine on the 1st day. By day 31st the wine bottle that has the poison will be encoded by whether the testers laughed or not. Using this scheme, we can detect what bottle has poison and has number of testers that fit the $log_2 n$ requirement.

Task 3: How Long Does This Take

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

programA()

The time complexity of the programA() is $\Theta(\log_2 n)$. This can be proven by translating the for loop into a recurrence relation making sure it has the same iterating and terminating behavior to ensure it shares the same time complexity. We can ignore the variable `prod` as it doesn't have significance in analyzing the time complexity of the function.

$$T(n) = T(\frac{n}{2})$$

Now we can plug and chug to inspect the behavior of the function, specifically the behavior being how fast the function takes terminate relation to how big n is.

$$\begin{aligned} T(16) &= T(\frac{16}{2}) \\ &= T(\frac{8}{2}) \\ &= T(\frac{4}{2}) \\ &= T(\frac{2}{2}) \\ &= \dots \end{aligned}$$

We can see that for every recurrence of the function T(n) input n starts off reducing rapidly then slows down as it gets closer to zero. We can deduce that the time complexity must be some log function.

Furthermore, we might stare the numerators long enough such that we realize they are numbers of base two where their exponent is being reduced by 1 every recurrence.

$$\begin{aligned} T(2^4) &= T(\frac{2^4}{2}) \\ &= T(\frac{2^3}{2}) \\ &= T(\frac{2^2}{2}) \\ &= T(\frac{2}{2}) \\ &= \dots \end{aligned}$$

Thus we can further deduce with confidence that the time complexity must be of $\log_2 n$.
Since we were able to deduce the exact behavior of the function using the recurrence relation it is safe to say that there exist constants that can create a tight upper-bound and lower-bound of programA(). Hence, we can conclude that the time complexity of programA() is $\Theta(\log_2 n)$.

---

programB()

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

Similarly, the time complexity of the programB() is $\Theta(\log_3 n)$. This can be proven by again translating the for loop into a recurrence relation making sure it has the same iterating and terminating behavior to ensure it shares the same time complexity.

$$T(n) = T(\frac{n}{3})$$

Now we can plug and chug to inspect the behavior of the function, specifically the behavior being how fast the function takes terminate relation to how big n is.

$$\begin{aligned} T(81) &= T(\frac{81}{3}) \\ &= T(\frac{27}{3}) \\ &= T(\frac{9}{3}) \\ &= T(\frac{3}{3}) \\ &= \dots \end{aligned}$$

We can see that for every recurrence of the function T(n) input n starts off reducing rapidly then slows down as it gets closer to zero. We can deduce that the time complexity must be some log function.

Again, we might stare the numerators long enough such that we realize they are numbers of base 3 where their exponent is being reduced by 1 every recurrence.

$$\begin{aligned} T(3^4) &= T(\frac{3^4}{3}) \\ &= T(\frac{3^3}{3}) \\ &= T(\frac{3^2}{3}) \\ &= T(\frac{3}{3}) \\ &= \dots \end{aligned}$$

Thus we can further deduce with confidence that the time complexity must be of $\log_3 n$.
Since we were able to deduce the exact behavior of the function using the recurrence relation it is safe to say that there exist constants that have a tight upper-bound and lower-bound. Hence, we can conclude that the time complexity of programB() is $\Theta(\log_3 n)$.

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[2i] + X[2i+1] for i = 0, 1, ..., len(X)/2 - 1
        (3) X = Y
    return X[0]
```

P1

The amount of work being done in steps 1-3 of fsum() in terms of k_1 and k_2 where $k_1, k_2, \in \mathbf{R}_+$ under the assumptions provided is:

$$\underbrace{(\frac{1}{2} \cdot k_1)z}_{\text{Step 1}} + \underbrace{(\frac{z}{2} \cdot k_2)}_{\text{Step 2}} + \underbrace{(k_2)}_{\text{Step 3}}$$

Step 1: We have to allocate `Y[]` which is half the size of array `X`. We know that $k_1 \cdot z$ is the cost of allocating array of size `X`. Hence allocating array of size `Y` is the same as allocating size array of size `X` divided by 2.

$$(\frac{1}{2} \cdot k_1)z$$

Step 2: We have to read, summate, then store `X[i]` and `X[i+1]`in `Y[i]` for $i = 0, 1, 2, \ldots, \frac{z}{2} - 1$. In the assumption its states that this set of operation done for each value that ends up in array `Y` is equal to the cost of k_2. Since we are summating every subsequent element pair `X[]`, `Y[]` will be half the size of `X[]`. Hence, we will have to use the $k_2 \cdot \frac{z}{2}$ times.

$$(\frac{z}{2} \cdot k_2)$$

Step 3: Finally we will have to use 1 more operation k_2 to pass the array made in `Y[]` to `X[]`.

$$k_2$$

P2

To analyze the time complexity of this function I will construct a recurrence relation that has the same behavior as the function `hsum()`. This will help us observe how the length of `X[]` changes as the function iterates. To do this we can create a recurrence relation that halves the input for every iteration of the recurrence.

$$T(n) = T(\frac{n}{2})$$

Let's plug and chug.

$$\begin{aligned} T(32) &= T(\frac{32}{2}) \\ &= T(\frac{16}{2}) \\ &= T(\frac{8}{2}) \\ &= T(\frac{4}{2}) \\ &= T(\frac{2}{2}) \\ &= \dots \end{aligned}$$

From inspection, we can see that the size of `X[]` n, reduces at a rate of $\log_2$. Thus, we can come to the conclusion that the function `hsum()` has the time complexity of $\Theta(n)$,

Task 5: More Running Time Analysis

P1

```
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;
}
```

Time complexity: The best and worst case time complexity of `method1()` is $\Theta(n^2)$. This is because the outer loop runs $n-1$ times and the inner loop runs $n-1$ times. The swap function runs constant time as the amount of operations performed don't change as n changes, so we can ignore it in the calculation of the time complexity. Hence, we can calculate the tight upper-bound and lower-bound of the time complexity as the following:

$$
\begin{aligned}
&= \Theta((n-1)\cdot(n-1)) \\
&= \Theta((n)\cdot(n)) \\
&= \Theta(n^2)
\end{aligned}
$$

P2

```
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;
}
```

Time complexity: The time complexity of `method2()` is $\Theta(n)$. The upper-bound/worst case time complexity is $O(n)$ and the lower-bound/best case time complexity is $\Omega(1)$. Combining the lower-bound and upper-bound time complexity we get $\Theta(n)$.

P3

```
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;
}
```

Time complexity: The best and worst case time complexity of `method3()` is $(n \cdot \log_2 n)$. This is because the most outer loop runs constant time, so it can be ignored. The inner loop runs (n) times because its rate to termination is linear. Lastly, the most inner loop rate to termination shortens exponentially (specifically at a rate of $n/2$ every iteration) thus we can deduce that it runs a time complexity of $\log_2 n$.

Combining the separate time complexities:

$$= (n) \cdot (log_2 n)$$
$$= (n \cdot log_2 n)$$

Since this is the time complexities of the best case and the worst case we can conclude the time complexity for `method3()` is $\Theta(n \cdot log_2 n)$

Task 6: Recursive Code

P1

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

1. Method of Measuring Problem Size:
The problem size is measured using the length of the input array `xs[]`.

2. Recurrence Relation:
Let n = length of the array `xs[]`. The recurrence relation can be written as:

$$T(n) = \underbrace{T(\frac{n}{2})}_{\text{recurrence}} + \underbrace{(\frac{n}{2}) \cdot c_1}_{\text{allocating new array}} + \underbrace{(\frac{n}{2}) \cdot c_2}_{\text{for loop}}; T(1) = O(1)$$

$$T(n) = T(\frac{n}{2}) + O(n); T(1) = O(1)$$

3. Running Time Complexity
According to a table of common recurrences assuming T(0) and T(1) are constant the reccurence relation solves to the following:

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

P2

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

1. Method of Measuring Problem Size:
The problem size is measured using the length of the input array `xs[]`.

2. Recurrence Relation:
Let n = length of the array `xs[]`. The recurrence relation can be written as:

$$T(n) = \underbrace{T(n-1)}_{\text{recurrence}} + \underbrace{(n-1)\cdot c_1}_{\text{copying array}}; T(1) = O(1)$$
$$T(n) = T(n-1) + O(n); T(1) = O(1)$$

3. Running Time Complexity
According to a table of common recurrences assuming T(0) and T(1) are constant the reccurence relation solves to the following:

$$T(n) = T(n-1) + O(n) \rightarrow O(n)$$

P3

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

1. Method of Measuring Problem Size:
The problem size is measured using the length of the input array `xs[]`.

2. Recurrence Relation:
Let n = length of the array `xs[]`. The recurrence relation can be written as assuming that:

$$T(n) = \underbrace{T(\frac{n}{2}) + T(\frac{n}{2})}_{\text{recurrence left + right}} + \underbrace{(\frac{n}{2})\cdot c_1 + (\frac{n}{2})\cdot c_1}_{\text{copying array left + right}}; T(1) = O(1)$$
$$T(n) = 2T(\frac{n}{2}) + O(n); T(1) = O(1)$$

3. Running Time Complexity
According to a table of common recurrences assuming T(0) and T(1) are constant the reccurence relation solves to the following:

$$T(n) = 2T(\frac{n}{2}) + O(n) \rightarrow O(n \cdot log_2 n)$$

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);
    }
}
```

The recurrence relation of the function `printRuler()` is given as the following:

$$g(n) = 2g(n-1) + n; g(0) = 0$$

The output of the recurrence gives the total amount of dashes needed to draw a rule of size n. The following shows how the recurrence can be solved with a closed form function.

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

Let me first list out the outputs of the recurrence relation for when n=0, 1, 2 and 3. This sample will give us an idea on how the output grows relation to n. Additionally, while developing the close form of the recurrence it gives us the opportunity to compare the outputs, so we can strategically tweak variables to get the desired outputs.

$$g(0) = 0$$
$$g(1) = 1$$
$$g(2) = 4$$
$$g(3) = 11$$

(i): Lets first find c. Let us do what is advised and plug in 0 into g(n).

$$\begin{aligned} g(0) &= a \cdot f(0) + b \cdot 0 + c \\ &= a \cdot 0 \quad + b \cdot 0 + c = 0 \end{aligned}$$

(ii): Notice that the first 2 terms of the equation $a \cdot 0 + b \cdot 0$ must equal zero. Hence, $c = 0$ because $g(0) = 0$ shown in the list of outputs. Now we need to find the values of a and b. Lets plugin in 3 into g(n).

$$\begin{aligned} g(3) &= a \cdot f(3) + b \cdot 3 + c \\ &= a \cdot 7 \quad + b \cdot 3 + 0 = 11 \end{aligned}$$

(iii): We can substitute in f(n) to form the closed for g(n).

$$\begin{aligned} g(n) &= 2f(n) - n \\ &= 2(2^n - 1) - n \end{aligned}$$

(iv): Let's prove that the close form is true for all values of n using induction. Let's prove that $g(n) = 2g(n-1) + n = 2(2^n - 1) - n$

Predicate:
Assuming that the recurrence relation is $g(n) = 2g(n-1) + n = 2f(n) - n$.

$$P(n) = \underbrace{2f(n) - n}_{\text{recurrence}} = \underbrace{2(2^n - 1) - n}_{\text{close form}}$$

Base Case:

$$P(0) = 0$$

This base case is true because...

$$P(0) = \underbrace{2f(0) - 0}_{\text{recurrence}} \qquad = \underbrace{2(2^0 - 1) - 0}_{\text{close form}}$$
$$= 0 \qquad = 0 \qquad \rightarrow \text{true}$$

Inductive Step:
Let us assume that $P(k)$ is true for all positive integer values of k. We want to show when $P(k)$ is true then $P(k+1)$ is true also.

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

LHS:

$$= 2f(k+1) - (k+1)$$

Let's substitute the given close form of f(n) thus we get...

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

Hence, this proves that when P(k) is true P(k+1) is also true. Now using mathematical induction we can conclude that P(n) is true for all positive integer values of n which concludes the proof.