Homework 5

Data Struct. and {Abstractions, OOP} (Term I/2024–25)

built on 2024/11/05 at 22:10:49 **due:** *wed nov 13 @ 11:59pm*

This assignment contains only written questions focusing on performance characterization. You must typeset your answer, produce a PDF file called `hw5.pdf`, and upload it to Canvas. No other format will be accepted. To typeset your homework, apart from Microsoft Word, there are LibreOffice and LaTeX, which we recommend. Note that a scan of your handwritten solution will not be accepted. You can find helpful resources about how to typeset your work on the course website.

Collaboration

We interpret collaboration very liberally. You may work with other students. However, each student ***must*** write up and hand in his or her assignment separately. Let us repeat: You need to write your own code. You must not look at or copy someone else's code. You need to write up answers to written problems individually. The fact that you can recreate the solution from memory will be taken as proof that you actually understood it, and you may actually be interviewed about your answers.

Task 1: Hello, Definition (4 points)

This task consists of small problems involving working directly with the definition of Big-O. *(Hint: One very simple solution for the first two problems below involves just taking limits.)*

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

(2) In class, we saw that Big-O multiplies naturally. You will explore this more formally here. Prove the following statement mathematically (i.e. using proof techniques learned in Discrete Math):

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

(3) There is a function `void fnE(int i, int num)` that runs in $1000 \cdot$ `i` steps, regardless of what `num` is. Consider the following code snippet:

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

What's the running time in Big-O of `fnA` as a function of n, which is the length of the array S. You should assume that it takes constant time to determine the length of an array.

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

Task 2: Poisoned Wine (2 points)

You own n bottles of wine, exactly one of which has been poisoned. You don't know which bottle, however. What you know is, if someone drinks just a tiny amount of wine from the poisoned bottle, s/he will start laughing uncontrollably after 30 days. In fact, the poison is so potent that even the faintest drop, diluted over and over, will still cause the symptom.

Design a scheme that determines exactly which one bottle was poisoned. You are allowed 31 days and can expend $O(\log n)$ testers (people). *Explain why your scheme meets the $O(\log n)$-tester requirement.*

(*Hint:* Numbers between 1 and n (inclusive) can be represented using $\lceil \log_2(n+1) \rceil$ bits.)

(*More Hint:* Think hard about the first hint before being spoiled by this hint. If the bottles are labeled 1 through n, how can we find out the i-th bit of the label of the poisoned bottle?)

Task 3: How Long Does This Take? (2 points)

For each of the following functions, determine the running time in terms of Θ in the variable n. **Show your work.** We're more interested in the thought process than the final answer.

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

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

Task 4: Halving Sum (4 points)

Our course staff has a signature process for summing up the values in a sequence (i.e., array). Let X be an input sequence consisting of n floating-point numbers. To make life easy, we'll assume n is a power of two—that is, $n = 2^k$ for some nonnegative integer k. To sum up these numbers, we use the following process, expressed in a Python-like language:

```
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]
```

This task has *two* parts:

Part I: (2 points) Observe that the amount of work done in Steps (1)–(3) is a function of the length of X in that iteration. If $z =$ `X.length` at the start of an iteration, how much work is being done in that iteration as a function z (e.g., $10z^5 + z\log z$ or $k_1 z^2 + k_2 z$ for some $k_1, k_2 \in \mathbb{R}_+$)? Don't use Big-O; answer in terms of k_1 and k_2. Let's make some assumptions here. For some $k_1, k_2 \in \mathbb{R}_+$:

- Allocating an array of length z costs you $k_1 \cdot z$.
- Arithmetic operations, as well as reading a value from an array and writing a value to an array, can be done in k_2 per operation.

Your answer for this part should look like the following:

$$\Big(\quad \cdot \quad \Big) z + \Big(\quad \cdot \quad \Big)$$

Part II: (2 points) Then, you'll analyze the running time of the algorithm (remember to explain how you get the running time you claim). To help you get started, make a table of how `X.length` changes over time if we start with X of length, say, 64. How does this work in general? *(Hint: The geometric sum formula presented in class may come in handy.)*

Task 5: More Running Time Analysis (6 points)

For the most part, we have focused almost exclusively on worst-case running time. In this problem, we are going to pay closer attention to these Java methods and consider their worst-case and best-case behaviors. The *best-case* behavior is the running time on the input that yields the fastest running time. The *worst-case* behavior is the running time on the input that yields the slowest running time.

(1) Determine the *best-case* running time and the *worst-case* running time of `method1` in terms of Θ.

```java
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;
}
```

(2) Determine the *best-case* running time and the *worst-case* running time of `method2` in terms of Θ.

```java
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;
}
```

(3) Determine the *best-case* running time and the *worst-case* running time of `method3` in terms of Θ.

```java
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;
    }
```

Task 6: Recursive Code (6 points)

For each of the following Java functions:

(1) Describe how you will measure the problem size in terms the input parameters. For example, the input size is measured by the variable n, or the input size is measured by the length of array a.

(2) Write a recurrence relation representing its running time. Show how you obtain the recurrence.

(3) Indicate what your recurrence solves to (by looking up the recurrence in our table).

```
// 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);
    }
}

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);
     }
}

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;
    }
}
```

Task 7: Counting Dashes (2 points)

Consider the following program:

```
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);
    }
}
```

We would like to know the total number of dashes printed for a given n. If we are to write a recurrence for that, we will get

$$g(n) = 2g(n-1) + n, \quad \text{with } g(0) = 0,$$

where the additive n term stems from the fact that we print exactly n dashes in that function call.

It may seem hopeless to try to solve this recurrence directly, but you may recall that the number of instructions taken to solve tower of Hanoi on n discs is given by the recurrence

$$f(n) = 2f(n-1) + 1, \quad \text{with } f(0) = 0.$$

As you showed in a previous assignment, $f(n)$ has a closed-form of $f(n) = 2^n - 1$.

The two recurrences are strikingly similar. In this problem, we'll analyze $g(n)$ using our knowledge of $f(n)$. Since $f(n)$ and $g(n)$ have similar recurrences, differing only in an n term, we're going to guess that

$$g(n) = a \cdot f(n) + b \cdot n + c \tag{1}$$

The following steps will guide you through determining the values of a, b, and c—and verifying that our guess indeed works out. In your writeup, **clearly show your work.**

(i) We'll first figure out the value of c. What do you get when plugging in $n = 0$ into equation (1)? It helps to remember that $f(0) = g(0) = 0$. (*Hint:* c should be 0.)

(ii) To figure out the values of a and b, we'll plug in $g(n)$ from equation (1) into the recurrence $g(n) = 2g(n-1) + n$. You should be able write it as

$$\Big(\underbrace{\dots}_{=P}\Big)n + \Big(\underbrace{\dots}_{=Q}\Big) = 0$$

and solve for a and b such that $P = 0$ and $Q = 0$.

Keep in mind: Because $f(n) = 2f(n-1) + 1$, we know that $f(n) - 2f(n-1) = 1$.

(iii) Derive a closed form for $g(n)$.

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