CSE 332: Data Structures and Parallelism

Section 2: Heaps and Asymptotics Solutions

0. Big-Oh Proofs

For each of the following, prove that $f \in \mathcal{O}(g)$.

(a) $f(n) = 7n$ $\qquad$ $g(n) = \frac{n}{10}$

Solution:

Recall that $f \in \mathcal{O}(g)$ is true if and only if there exists some constant c and some constant $n_0 > 0$ such that for all $n \geq n_0$, the expression $f(n) \leq c \cdot g(n)$ is true by definition of Big-$\mathcal{O}$.

Now, we choose $c = 70$, $n_0 = 1$. We must now show that $f(n) \leq 70 \cdot g(n)$ is true for all $n \geq 1$.

Note that $c \cdot g(n) = 70g(n) = 70\left(\frac{n}{10}\right) = 7n$. After substituting, we find the inequality $f(n) \leq c \cdot g(n)$ is equivalent to $7n \leq 7n$. We can see this is true for all $n > 1$, so we conclude that $f \in \mathcal{O}(g)$ is true.

(b) $f(n) = 1000$ $\qquad$ $g(n) = 3n^3$

Solution:

We follow the same approach as above.

We choose $c = 1$, $n_0 = 1000$, and so must show that $1000 \leq 1 \cdot 3n^3$ for all $n \geq 1000$.

Now, note that for all $n \geq 1000$ the inequalities $1000 \leq n$, $n \leq n^3$, and $n^3 \leq 3n^3$ are always true.

By chaining the inequalities together, we see that $f(n) = 1000 \leq n \leq n^3 \leq 3n^3 = c \cdot g(n)$ for all $n \geq 1000$ and so conclude that $f \in \mathcal{O}(g)$ is true.

(c) $f(n) = 7n^2 + 3n$ $\qquad$ $g(n) = n^4$

Solution:

We choose $c = 14$, $n_0 = 1$. Then, note that $f(n) = 7n^2 + 3n \leq 7(n^4 + n^4) \leq 14n^4 = c \cdot g(n)$ for all $n \geq 1$. So, we conclude that $f \in \mathcal{O}(g)$ is true.

(As before, we construct and chain inequalities to establish a relationship between f and g).

(d) $f(n) = n + 2n \lg n$ $\qquad$ $g(n) = n \lg n$

Solution:

Choose $c = 3$, $n_0 = 1$. Then, note that $f(n) = n + 2n \lg n \leq n \lg n + 2n \lg n = 3n \lg n = c \cdot g(n)$ for all $n \geq 1$. So, we conclude that $f \in \mathcal{O}(g)$ is true.

1. Is Your Program Running? Better Catch It!

For each of the following, determine the tight $\Theta(\cdot)$ bound for the worst-case runtime in terms of the free variables of the code snippets.

(a)

```
1 int x = 0
2 for (int i = n; i >= 0; i--) {
3    if ((i % 3) == 0) {
4       break
5    }
6    else {
7       x += n
8    }
9 }
```

Solution:

This is $\Theta(1)$ because exactly one of n, $n-1$, or $n-2$ will be divisible by three for all possible values of n. So, the loop runs at most 3 times.

(b)

```
1 int x = 0
2 for (int i = 0; i < n; i++) {
3    for (int j = 0; j < (n * n / 3); j++) {
4       x += j
5    }
6 }
```

Solution:

We can model the worst-case runtime as: $\sum_{i=0}^{n-1} \sum_{j=0}^{n^2/3-1} 1$. This simplifies to: $\sum_{i=0}^{n-1} \sum_{j=0}^{n^2/3-1} 1 = \sum_{i=0}^{n} \frac{n^2}{3} = n\left(\frac{n^2}{3}\right) = \frac{n^3}{3}$. So, the worst-case runtime is $\Theta(n^3)$.

(c)

```
1 int x = 0
2 for (int i = 0; i < n; i++) {
3    for (int j = 0; j < i; j++) {
4       x += j
5    }
6 }
```

Solution:

We can model the worst case runtime as $\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1$ which simplifies to $\sum_{i=0}^{n-1} i = \left(\frac{n(n-1)}{2}\right)$. So, the worst-case runtime is $\Theta(n^2)$

(d)

```
1  int x = 0
2  for (int i = 0; i < n; i++) {
3     if (n < 100000) {
4        for (int j = 0; j < i * i * n; j++) {
5           x += 1
6        }
7     } else {
8        x += 1
9     }
10 }
```

Solution:

Recall that when computing the asymptotic complexity, we only care about the behavior as the input goes to infinity. Once n is large enough, we will only execute the second branch of the if statement, which means the runtime of the code can be modeled as $\sum_{i=0}^{n-1} 1 = n$. So, the worst-case runtime is $\Theta(n)$.

(e)

```
1  int x = 0
2  for (int i = 0; i < n; i++) {
3     if (i % 5 == 0) {
4        for (int j = 0; j < n; j++) {
5           if (i == j) {
6              for (int k = 0; k < n; k++) {
7                 x += i * j * k
8              }
9           }
10       }
11    }
12 }
```

Solution:

We know the runtime of the outer-most loop is $\sum_{i=0}^{n-1} (?)$, where $(?)$ is the (currently unknown) runtime of the middle and inner-most loops. We also know the middle loop by itself has a runtime of $\sum_{j=0}^{n-1} (?)$ and runs only a fifth of the time. Therefore, we can refine our model to $\sum_{i=0}^{n-1} \frac{1}{5} \left(\sum_{j=0}^{n-1} (?) \right)$.

Now, note that the inner-most if statement is true exactly only once per each iteration of the middle loop. So, we can refine our model of the runtime to $\sum_{i=0}^{n-1} \frac{1}{5} \left(\left(\sum_{j=0}^{n-1} 1 \right) + \left(\sum_{k=0}^{n-1} 1 \right) \right)$ which simplifies to $\sum_{i=0}^{n-1} \frac{2n}{5} = \frac{2n^2}{5}$. Therefore, the worst- case asymptotic runtime will be $\Theta(n^2)$.

2. Asymptotics Analysis

Consider the following method which finds the number of unique Strings within a given array of length n.

```
1  int numUnique(String[] values) {
2     boolean[] visited = new boolean[values.length]
3     for (int i = 0; i < values.length; i++) {
4        visited[i] = false
5     }
6     int out = 0
7     for (int i = 0; i < values.length; i++) {
8        if (!visited[i]) {
9           out += 1
10          for (int j = i; j < values.length; j++) {
11             if (values[i].equals(values[j])) {
12                visited[j] = true
13             }
14          }
15       }
16    }
17    return out;
18 }
```

Determine the tight $\mathcal{O}(\cdot)$, $\Omega(\cdot)$, and $\Theta(\cdot)$ bounds of each function below. If there is no $\Theta(\cdot)$ bound, explain why. Start by (1) constructing an equation that models each function then (2) simplifying and finding a closed form.

(a) $f(n) =$ the worst-case runtime of `numUnique`

Solution:

In the worst case, the array will contain entirely unique strings and so must run the inner loop n times.

So, $f(n) = \sum_{i=0}^{n-1} 1 + \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} 1 = n + \frac{n(n+1)}{2}$ which means $f \in \mathcal{O}(n^2)$, $f \in \Omega(n^2)$, and $f \in \Theta(n^2)$.

(b) $g(n)$ = the best-case runtime of `numUnique`

Solution:

In the best case, the array will contain the exact same string repeated n times, causing the inner loop to run only once.

So, $g(n) = \sum_{i=0}^{n-1} 1 + \sum_{i=0}^{n-1} 1 + \sum_{j=0}^{n-1} 1 = 3n$ which means $g \in \mathcal{O}(n)$, $g \in \Omega(n)$, and $g \in \Theta(n)$.

(c) $h(n)$ = the amount of memory used by `numUnique` (the space complexity)

Solution:

`numUnique` will create a boolean array of length n and allocate a few extra variables, which take up a constant and therefore negligible amount of memory.

So, $h(n) = n + k$ (where k is some constant) which means $h \in \mathcal{O}(n)$, $h \in \Omega(n)$, and $h \in \Theta(n)$.

(d) $k(n)$ = the integer `numUnique` will return (the output complexity)

Solution:

Note that `numUnique` can only return an integer in the range $[1, n]$.

If we consider the upper and lower bounds of the possible return values, we can see that $k \in \mathcal{O}(n)$ and $k \in \Omega(1)$. Since the upper and lower bounds differ, we can conclude there is no Big-Θ for k (by definition of Big-Θ).

3. Analyzing Data Structures

(a) Suppose we have a worklist `list` which contains n integers. The following code creates a heap which contains only the 25 largest elements:

```
1    PriorityWorkList<Integer> heap = new MinFourHeap<Integer>()
2    while (list.hasWork()) {
3       if (heap.size() >= 25) {
4          heap.removeMin()
5       }
6       heap.add(list.next())
7    }
```

Determine the tight $\Theta(\cdot)$ bounds for the worst-case runtime complexity and the space complexity of this code snippet. Assume that the given worklist of integers has $\Theta(1)$ runtime for `hasWork()` and `next()`.

Solution:

For the first 25 insertions, the size of `heap` prior to execution of the loop will be less than 25. Therefore, the `if` statement will not be executed. So, we can model the runtime of these insertions as $\sum_{i=1}^{25} \log(i)$.

Then, for all the other insertions, we will have to `removeMin()` from a heap size 25, then add into a heap size 24. We can model the runtime of these insertions as $\sum_{i=26}^{n} \log(25) + \log(24)$.

So, our final summation for the runtime is $\sum_{i=1}^{25} \log(i) + \sum_{i=26}^{n} (\log(25) + \log(24))$.

But now, notice that $\sum_{i=1}^{25} \log(i)$ is just some constant (call it C) and $\log(25) + \log(24)$ is also just some constant (call it D), so our summation can be rewritten as $C + \sum_{i=26}^{n} D = C + (n - 26)D$. This means the runtime complexity is $\Theta(n)$.

Because our heap will always contain at most 25 elements regardless of the initial size of the worklist, we know the space complexity will be $\Theta(1)$.

(b) Suppose we have a worklist `list` which contains t strings, where each string has an average length s. Let k indicate the total number of unique characters in the strings. The following code creates a map containing how frequently any given character appears in all of the strings:

```
1  Map<Character, Integer> frequencies = new HashMap<Character, Integer>()
2  while (list.hasWork()) {
3     String word = list.next()
4     for (int i = 0; i < word.size(); i++) {
5        char c = word.charAt(i)
6        if (!frequencies.containsKey(c)) {
7           frequencies.put(c, 0)
8        }
9        frequencies.put(c, 1 + frequencies.get(c))
10    }
11 }
```

Determine the tight $\Theta(\cdot)$ bounds for the worst-case runtime complexity and space complexity of this snippet of code. Assume the given worklist of strings has $\Theta(\lg(t))$ runtime for `hasWork()` and `next()`.

Solution:

Within each iteration of the outer loop, we call `hasWork()` and `next()` exactly once, and runs a single loop. If we suppose the (currently unknown) runtime of the inner loop takes (?) steps, we can model the runtime of the entire method using the expression $\sum_{i=1}^{t} \left(2\lg(t) + (?)\right)$.

Now, consider the inner loop. The inner loop will on average repeat about s times and perform only $\Theta(1)$ operations – remember that `get()`, `set()`, and `containsKey()` are all $\Theta(1)$ operations for `HashMaps`.

Therefore, we can model the runtime of the inner loop as $\sum_{j=1}^{s} 1$ and model the runtime of the entire method as:

$$\sum_{i=1}^{t} \left(2\lg(t) + \sum_{j=1}^{s} 1 \right) = t(2\lg(t) + s)$$

We conclude the worst-case asymptotic complexity is $\Theta(2t\lg(t) + st)$.

Note that the hashmap will contain exactly one character and integer per each unique character. Since the code will allocate some amount of space directly proportional to k, we conclude the space complexity is $\Theta(k)$.

4. $\mathcal{O}$h Snap!

For each question below, explain what's wrong with the provided answer. The problem might be the reasoning, the conclusion, or both!

(a) Determine the tight $\Theta(\cdot)$ bound for the worst-case runtime of the following piece of code:

```
1  public static int waddup(int n) {
2     if (n > 10000) {
3        return n
4     } else {
5        for (int i = 0; i < n; i++) {
6           System.out.println("It's dat boi!")
7        }
8        return 0
9     }
10 }
```

Bad answer: The runtime of this function is $\mathcal{O}(n)$, because when searching for an upper bound, we always analyze the code branch with the highest runtime. We see the first branch is $\mathcal{O}(1)$, but the second branch is $\mathcal{O}(n)$.

Solution:

The tightest upper bound is $\mathcal{O}(1)$, not $\mathcal{O}(n)$. Picking the code branch with the highest runtime is not necessarily the correct thing to do – instead, we must consider what the runtime is as the input grows towards by infinity.

In this case, we can see the first branch will be executed for when $n > 10000$, so we consider only that branch when computing the asymptotic complexity.

(b) Determine the tight $\Theta(\cdot)$ worst-case runtime of the following piece of code:

```
1  public static void trick(int n) {
2     for (int i = 0; i < Math.pow(2, n); i *= 2) {
3        for (int j = 0; j < n; j++) {
4           System.out.println("(" + i + "," + j + ")")
5        }
6     }
7  }
```

Bad answer: The runtime of this function is $\mathcal{O}(n^2)$, because the outer loop is conditioned on an expression with n and so is the inner loop.

Solution:

While the runtime *is* $\mathcal{O}(n^2)$, the explanation is incorrect. In particular, it glosses over the fact that we are iterating from 0 to $2^n - 1$ in the outer loop.

A more precise explanation should explain that while the outer loop terminates when $i = 2^n$, we are also multiplying i by 2 per each iteration. This means the outer loop does $\lg(2^n)$ iterations, which is just equivalent to n.

The inner loop does $\sum_{j=0}^{n-1} 1 = n$ iterations, so we conclude the overall runtime is $\mathcal{O}(n^2)$.

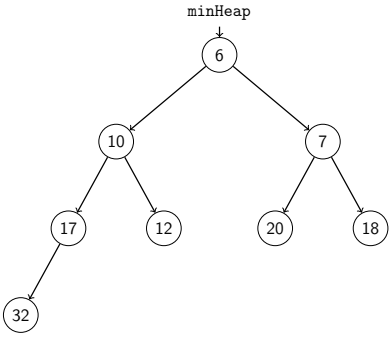
5. Look Before You Heap

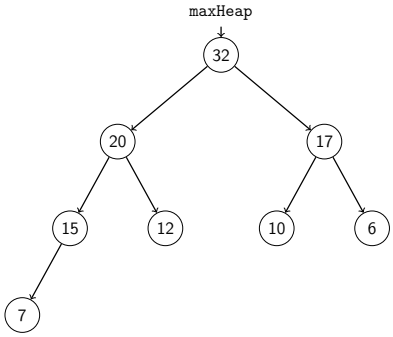
(a) Insert 10, 7, 15, 17, 12, 20, 6, 32 into a *min heap*.

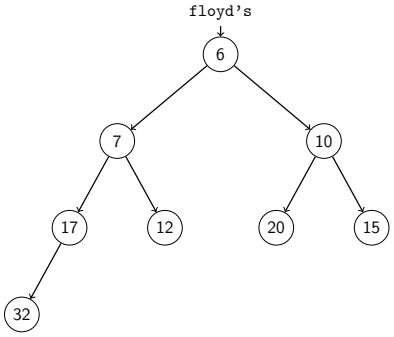
Now, insert the same values into a *max heap*.

Now, insert the same values into a *min heap*, but use Floyd's `buildHeap` algorithm.

Solution:







(b) Insert 1, 0, 1, 1, 0 into a *min heap*.

Solution: