# CSE 332: Data Structures and Parallelism

## Section 2: Heaps and Asymptotics

## 0. Big-Oh Proofs

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

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

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

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

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

## 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)

```
int x = 0
for (int i = n; i >= 0; i--) {
   if ((i % 3) == 0) {
      break
   }
   else {
      x += n
   }
}
```

(b)

```
int x = 0
for (int i = 0; i < n; i++) {
   for (int j = 0; j < (n * n / 3); j++) {
      x += j
   }
}
```

(c)

```
int x = 0
for (int i = 0; i < n; i++) {
   for (int j = 0; j < i; j++) {
      x += j
   }
}
```

(d)

```
int x = 0
for (int i = 0; i < n; i++) {
   if (n < 100000) {
      for (int j = 0; j < i * i * n; j++) {
         x += 1
      }
   } else {
      x += 1
   }
}
```

(e)

```
int x = 0
for (int i = 0; i < n; i++) {
   if (i % 5 == 0) {
      for (int j = 0; j < n; j++) {
         if (i == j) {
            for (int k = 0; k < n; k++) {
               x += i * j * k
            }
         }
      }
   }
}
```

## 2. Asymptotics Analysis

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

```
int numUnique(String[] values) {
   boolean[] visited = new boolean[values.length]
   for (int i = 0; i < values.length; i++) {
      visited[i] = false
   }
   int out = 0
   for (int i = 0; i < values.length; i++) {
      if (!visited[i]) {
         out += 1
         for (int j = i; j < values.length; j++) {
            if (values[i].equals(values[j])) {
               visited[j] = true
            }
         }
      }
   }
   return out;
}
```

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`

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

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

## 3. Analyzing Data Structures

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

```
PriorityWorkList<Integer> heap = new MinFourHeap<Integer>()
while (list.hasWork()) {
   if (heap.size() >= 25) {
      heap.removeMin()
   }
   heap.add(list.next())
}
```

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()`.

# 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:

```
public static int waddup(int n) {
   if (n > 10000) {
      return n
   } else {
      for (int i = 0; i < n; i++) {
         System.out.println("It's dat boi!")
      }
      return 0
   }
}
```

**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 $\mathcal{O}(n)$.

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

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

**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.

## 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.

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