CSE 332: Data Structures and Parallelism

Section 2: Heaps and Asymptotics

Definition of Big-Oh:

Suppose $f: \mathbb{N} \to \mathbb{R}$, $g: \mathbb{N} \to \mathbb{R}$ are two functions,

$$f(n) \in \mathcal{O}(g(n)) \equiv \exists_{c \in \mathbb{R}_{>0},\, n_0 \in \mathbb{N}} \forall_{n \in \mathbb{N} \geq n_0} f(n) \leq c \cdot g(n)$$

Definition of Big-Omega:

Suppose $f: \mathbb{N} \to \mathbb{R}$, $g: \mathbb{N} \to \mathbb{R}$ are two functions,

$$f(n) \in \Omega(g(n)) \equiv \exists_{c \in \mathbb{R}_{>0},\, n_0 \in \mathbb{N}} \forall_{n \in \mathbb{N} \geq n_0} f(n) \geq c \cdot g(n)$$

Definition of Big-Theta:

Suppose $f: \mathbb{N} \to \mathbb{R}$, $g: \mathbb{N} \to \mathbb{R}$ are two functions,

$$f(n) \in \Theta(g(n))$$

$$\equiv f(n) \in \mathcal{O}(g(n)) \wedge f(n) \in \Omega(g(n))$$

$$\equiv \exists_{c_0 \in \mathbb{R}_{>0},\, n_0 \in \mathbb{N}} \forall_{n \in \mathbb{N} \geq n_0} f(n) \leq c_0 \cdot g(n) \wedge \exists_{c_1 \in \mathbb{R}_{>0},\, n_1 \in \mathbb{N}} \forall_{n \in \mathbb{N} \geq n_1} f(n) \geq c_1 \cdot g(n)$$

0. Big-Oh Proofs

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

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

We are trying to prove that $7n \in \mathcal{O}(\frac{n}{10})$.
Let $n \in \mathbb{N} \geq n_0$ be arbitrary and let $c = 70$ and $n_0 = 1$.
We start with a true statement:
$7n \leq 7n$
$7n \leq 70 \cdot \frac{n}{10}$
$7n \leq c \cdot \frac{n}{10}$
As such, by the definition of Big-Oh, $7n \in \mathcal{O}(\frac{n}{10})$

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

We are trying to prove that $1000 \in \mathcal{O}(3n^3)$.
Let $n \in \mathbb{N} \geq n_0$ be arbitrary and let $c = 1$ and $n_0 = 1000$.
We start with a true statement:
$1000 \leq 1 \cdot n$
$1000 \leq 1 \cdot n^2$
$1000 \leq 1 \cdot n^3$
$1000 \leq 1 \cdot 3n^3$
$1000 \leq c \cdot 3n^3$
As such, by the definition of Big-Oh, $1000 \in \mathcal{O}(3n^3)$

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

We are trying to prove that $7n^2 + 3n \in \mathcal{O}(n^4)$.
Let $n \in \mathbb{N} \geq n_0$ be arbitrary and let $c = 10$ and $n_0 = 1$.
We start with a true statement:

$$7n^2 + 3n \leq 7n^4 + 3n^4 \ (b.c.\ n^4 > n^2\ and\ n^4 > n\ when\ n \geq 1)$$
$$\leq 10 \cdot n^4$$
$$\leq c * g(n)$$

As such, by the definition of Big-Oh, $7n^2 + 3n \in \mathcal{O}(n^4)$

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

We are trying to prove that $n + 2n\lg n \in \mathcal{O}(n\lg n)$.
Let $n \in \mathbb{N} \geq n_0$ be arbitrary and let $c = 3$ and $n_0 = 2$.
We start with a true statement:

$$n + 2n\lg n \leq n + 2n\lg n$$
$$\leq n\lg n + 2n\lg n \text{ (b.c } n\lg n \geq \text{n for } n \geq 2\text{)}$$
$$\leq 3 \cdot n\lg n$$
$$\leq c * g(n)$$

As such, by the definition of Big-Oh, $n + 2n\lg n \in \mathcal{O}(n\lg n)$

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
    }
}
```

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

b)

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

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

c)

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

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

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
    }
}
```

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)

```
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
                }
            }
        }
    }
}
```

We know the runtime of the outermost loop is $\sum_{i=0}^{n-1} ?$, where ? is the (currently unknown) runtime of the middle and innermost 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 $\sum_{i=0}^{n-1} \frac{1}{5} (\sum_{j=0}^{n-1} ?)$.

Now, note that the innermost if statement is true exactly only once per ea iteration of the middle loop. So, we can refine our model of the runtime tc $\sum_{i=0}^{n-1} \frac{1}{5} (\sum_{j=0}^{n-1} 1 + \sum_{k=0}^{n-1} 1)$ 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.

```
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`

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 $f(n) \in \mathcal{O}(n^2)$, $f(n) \in \Omega(n^2)$, and $f(n) \in \Theta(n^2)$.

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

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 $g(n) \in \mathcal{O}(n)$, $g(n) \in \Omega(n)$, and $g(n) \in \Theta(n)$.

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

`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 $h(n) \in \mathcal{O}(n)$, $h(n) \in \Omega(n)$, and $h(n) \in \Theta(n)$.

3. Oh 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 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 $\mathcal{O}(n)$.

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 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 $\Theta(\cdot)$ bound 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.

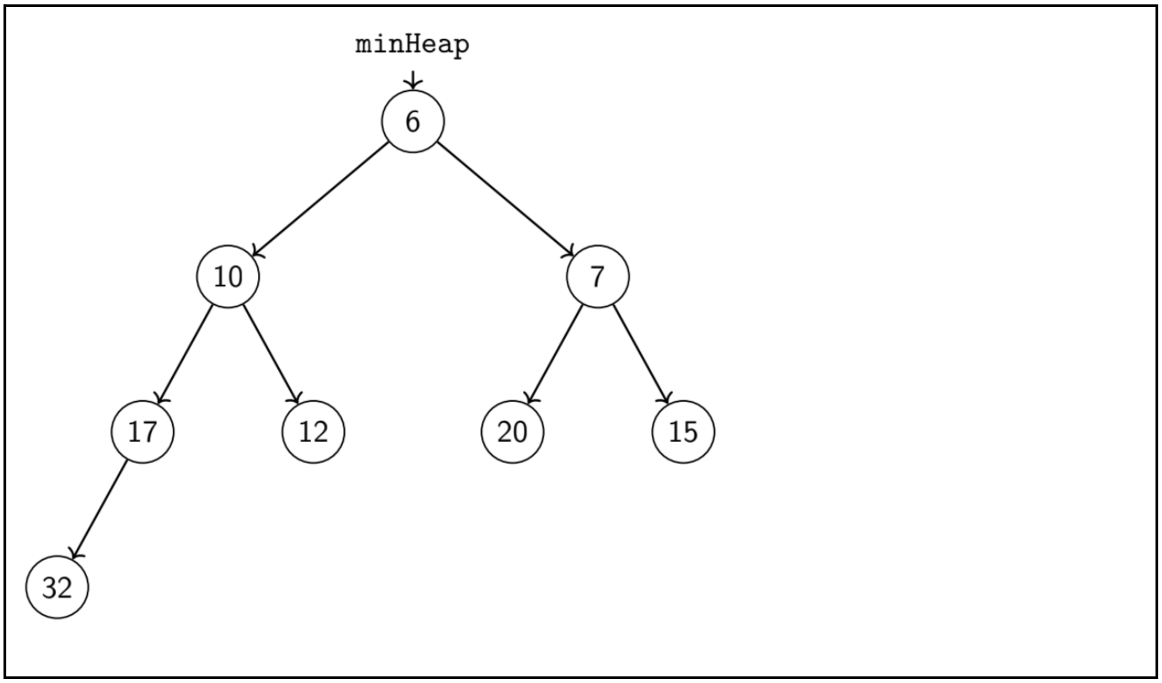
While the runtime is $\mathcal{O}(n^2)$, the explanation is incorrect. In particular, it glosses over the fact that we are iterating from 0 to $2^n - 1$ in the outer loop.

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.

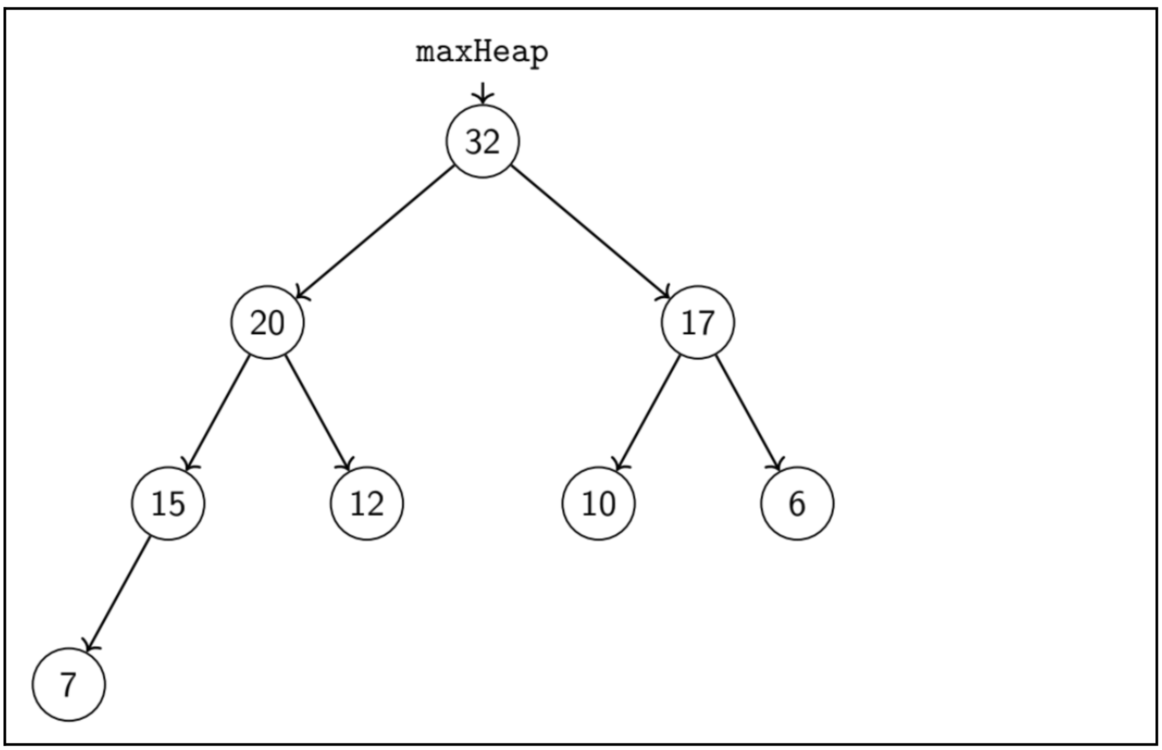
The inner loop does $\sum_{j=0}^{n-1} 1 = n$ iterations, so we conclude the overall runtime is $\mathcal{O}(n^2)$.

4. Look Before You Heap

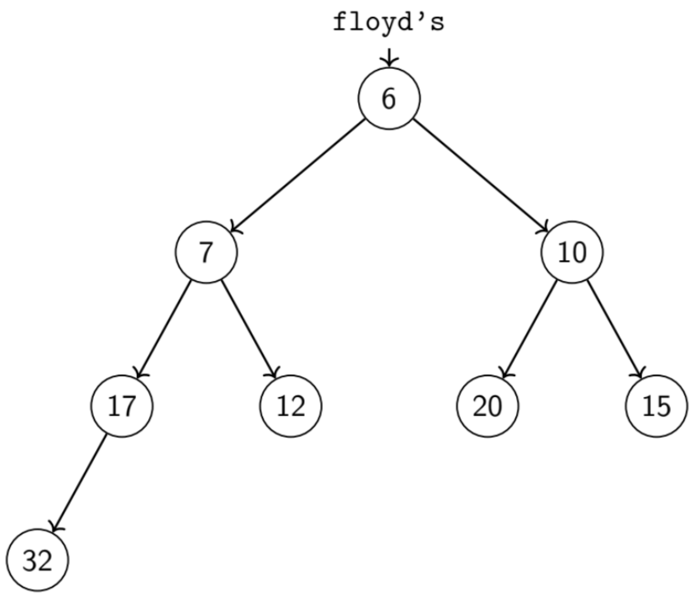
a) Insert 10, 7, 15, 17, 12, 20, 6, 32 into a *min* heap.



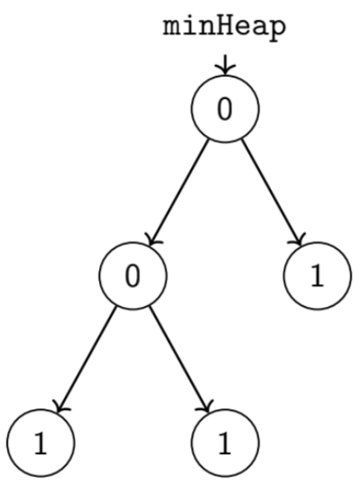
b) Now, insert the same values into a *max* heap.



c) Now, insert 10, 7, 15, 17, 12, 20, 6, 32 into a *min* heap, but use Floyd's `buildHeap` algorithm.



d) Insert 1, 0, 1, 1, 0 into a *min* heap.



5. $\mathcal{O}$ My Gosh!

Prove that $4n^2 + n^5 \in \Omega(n)$. Use the definition of Big-Omega above.

Scratch Work:

Prove that $4n^2 + n^5 \in \Omega(n)$.

Same as $\exists_{c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}} \forall_{n \in \mathbb{N} \geq n_0} 4n^2 + n^5 \geq c \cdot n$.

We want to find a c and n_0.

From here, we can do a technique called "demotion" where we observe that:

$n^2 \geq n$ when $n \geq 1$

$n^5 \geq n$ when $n \geq 1$

From this, we can "demote" the LHS:

$4n^2 + n^5 \geq 4n + n$

$= 5n$

We can observe here that this matches the formatting of the RHS where $c = 5$.

Hence, we found the values c and n_0 (from before where we observe the promotion):

$c = 5, n_0 = 1$

Now that we have this scratch work, we can work on our solution proof. Right now, this scratch work has backwards reasoning (bad!), so we need to reverse the order so we DO NOT start with the claim we wanted to prove ($4n^2 + n^5$) and DO NOT end with a true statement (n).

On to the solution!

Solution:

We are trying to prove that $4n^2 + n^5 \in \Omega(n)$.

Let $n \in \mathbb{N} \geq n_0$ be arbitrary and let $c = 5$ and $n_0 = 1$.

We start with a true statement:

$4n^2 + n^5 \geq 4n + n$

$4n^2 + n^5 \geq 5n$

$4n^2 + n^5 \geq c \cdot n$

As such, by the definition of Big-Omega, $4n^2 + n^5 \in \Omega(n)$.