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 $n \geq 1$.

By chaining inequalities together, we see that if $n \geq 1$, then $f(n) = 7n \leq 70 * \frac{n}{10} = cg(n)$. This proves the claim, so we conclude that $f(n) \in O(g(n))$

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

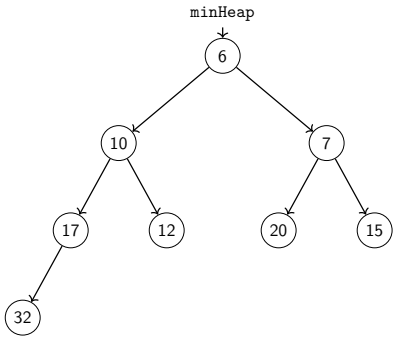
Solution:

Choose $c = 3$, $n_0 = 2$. 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 $n \geq 2$. So, we conclude that $f \in \mathcal{O}(g)$ is true.

1. Look Before You Heap

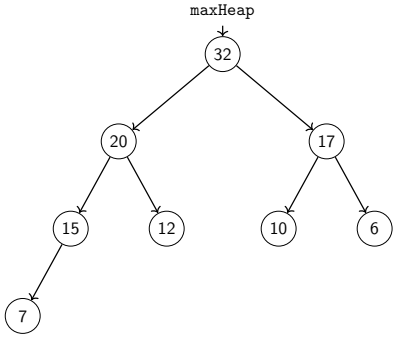
(a) Insert 10, 7, 15, 17, 12, 20, 6, 32 into a *min heap*.

Solution:



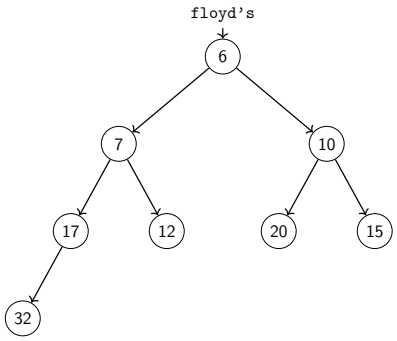
(b) Now, insert the same values into a *max heap*.

Solution:



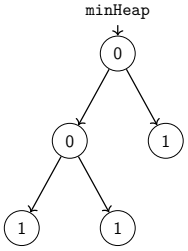
(c) Now, insert 10, 7, 15, 17, 12, 20, 6, 32 into a *min heap*, but use Floyd's `buildHeap` algorithm.

Solution:



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

Solution:



2. $\mathcal{O}$ My God!

Recall the definition of $f \in \Omega(g)$ is as follows:

$$\exists(c, n_0 > 0).\ \forall(n \geq n_0).\ f(n) \geq cg(n)$$

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

Solution:

Choose $c = \frac{1}{100}$ and $n_0 = 1$.
Then, since $n \geq 1$, $4n^2 + n^5 \geq \frac{4n}{500} + \frac{n}{500} = \frac{n}{100}$.

3. Not to Tree

For the following code snippet, find a recurrence for the worst case runtime of the function, and then find a closed form for the recurrence.

Consider the function f:

```
1 f(n) {
2    if (n <= 0) {
3       return 1;
4    }
5    return 2 * f(n - 1) + 1;
6 }
```

- Find a recurrence for $f(n)$.

  Solution:

  $$T(n) = \begin{cases} c_0 & \texttt{if } n <= 0 \\ T(n-1) + c_1 & \texttt{otherwise} \end{cases}$$

- Find a closed form for $f(n)$.

  Solution:

  Unrolling the recurrence, we get $T(n) = \underbrace{c_1 + c_1 + \cdots + c_1}_{n \text{ times}} + c_0 = c_1 n + c_0$.

4. To Tree

Consider the function h:

```
1 h(n) {
2    if (n <= 1) {
3       return 1
4    } else {
5       return h(n/2) + n + 2*h(n/2)
6    }
7 }
```

(a) Find a recurrence $T(n)$ modeling the *worst-case runtime complexity* of $h(n)$.

Solution:

$$T(n) = \begin{cases} c_0 & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + c_1 & \text{otherwise} \end{cases}$$

(b) Find a closed form to your answer for (a).

Solution:

The recursion tree has height $\lg(n)$, each non-leaf level i has has work $c_1 2^i$, and the leaf level has work $c_0 2^{\lg(n)}$. Putting this together, we have:

$$\begin{aligned} \left(\sum_{i=0}^{\lg n-1} c_1 2^i\right) + c_0 2^{\lg(n)} = c_1 \left(\sum_{i=0}^{\lg n-1} 2^i\right) + c_0 n &= c_1 \frac{1 - 2^{\lg n - 1 + 1}}{1 - 2} + c_0 n \\ &= c_1 2^{\lg n} - c_1 + c_0 n \\ &= c_1(n-1) + c_0 n \\ &= (c_0 + c_1)n - c_1 \end{aligned}$$

5. To Tree or Not to Tree

Consider the function f. Find a recurrence modeling the worst-case runtime of this function and then find a Big-Oh bound for this recurrence.

```
1  f(n) {
2     if (n == 0) {
3        return 0
4     }
5     int result = f(n/2)
6     for (int i = 0; i < n; i++) {
7        result *= 4
8     }
9     return result + f(n/2)
10 }
```

(a) Find a recurrence $T(n)$ modeling the worst-case time complexity of $f(n)$.

Solution:

We look at the three separate components (base case, non-recursive work, recursive work). The base case is a constant amount of work, because we only do a return statement. We'll label it c_0. The non-recursive work is a constant amount of work (we'll call it c_1) for the assignments and `if` tests and a constant (we'll call c_2) multiple of n for the loops. The recursive work is $2T\left(\frac{n}{2}\right)$.

Putting these together, we get:

$$T(n) = \begin{cases} c_0 & \text{if } n = 0 \\ 2T\left(\frac{n}{2}\right) + c_2 n + c_1 & \text{otherwise} \end{cases}$$

(b) Find a closed form for $f(n)$

Solution:

The recursion tree has $\lg(n)$ height, each non-leaf node of the tree does $c_2\frac{n}{2^i} + c_1$ work, each leaf node does c_0 work, and each level has 2^i nodes.

So, the total work is $\left(\sum_{i=0}^{\lfloor \lg(n) \rfloor - 1} 2^i(c_2\frac{n}{2^i} + c_1)\right) + c_0 \cdot 2^{\lg n} = \left(\sum_{i=0}^{\lfloor \lg(n) \rfloor - 1} (2^i c_1 + c_2 n)\right) + c_0 n = c_1\frac{1-2^{\lg n}}{1-2} + c_2 n \lg(n) + c_0 n = c_2 n \lg(n) + c_1(n-1) + c_0 n.$

Extra Practice On Following Pages

6. 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: $\sum_{i=0}^{n-1} \sum_{j=0}^{n^2/3-1} 1 = \sum_{i=0}^{n-1} \frac{n^2}{3} = n\left(\frac{n^2}{3}\right) = \frac{n^3}{3}$. 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 $\Theta(n^2)$.

7. 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 $h \in \Theta(n)$.

8. $\mathcal{O}$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 = 1; 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 $\mathcal{O}(n^2)$.

9. Big-Oof Bounds

Consider the function f. Find a recurrence modeling the worst-case runtime of this function and then find a Big-Oh bound for this recurrence.

```
1  f(n) {
2     if (n == 0) {
3        return 0
4     }
5
6     int result = 0
7     for (int i = 0; i < n; i++) {
8        for (int j = 0; j < i; j++) {
9           result += j
10
11       }
12    }
13    return f(n/2) + result + f(n/2)
14 }
```

(a) Find a recurrence $T(n)$ modeling the worst-case time complexity of $f(n)$.

Solution:

We look at the three separate components (base case, non-recursive work, recursive work). The base case is a constant amount of work, because we only do a return statement. We'll label it c_0. The non-recursive work is a constant amount of work (we'll call it c_1) for the assignments and `if` tests and a constant (we'll call c_2) multiple of $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$ for the loops. The recursive work is $2T\left(\frac{n}{2}\right)$.

Putting these together, we get:

$$T(n) = \begin{cases} c_0 & \text{if } n = 0 \\ 2T\left(\frac{n}{2}\right) + c_2\frac{n(n-1)}{2} + c_1 & \text{otherwise} \end{cases}$$

(b) Find a Big-Oh bound for your recurrence.

Solution:

Since we only want a Big-Oh, we can actually leave off lower-order terms when doing our analysis, as they won't affect the runtime bounds; so, we can ignore the constants c_1 and c_2 in our analysis.

Note that $\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \in \mathcal{O}(n^2)$. We can, again, ignore the lower-order term $\left(\frac{n}{2}\right)$ since we only want a Big-Oh bound.

The recursion tree has $\lg(n)$ height, each non-leaf node of the tree does $\left(\frac{n}{2^i}\right)^2$ work, each leaf node does c_0 work, and each level has 2^i nodes.

So, the total work is $\sum_{i=0}^{\lfloor \lg(n) \rfloor - 1} 2^i \left(\frac{n}{2^i}\right)^2 + c_0 \cdot 2^{\lg n} = n^2 \sum_{i=0}^{\lfloor \lg(n) \rfloor - 1} \left(\frac{2^i}{4^i}\right) + c_0 n < n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right) + c_0 n = \frac{n^2}{1 - \frac{1}{2}} + c_0 n.$

This expression is upper-bounded by n^2 so $T \in \mathcal{O}(n^2)$.

10. Odds Not in Your Favor

Consider the function g. Find a recurrence modeling the worst-case runtime of this function, and then find a closed form for the recurrence.

```
1  g(n) {
2     if (n <= 1) {
3        return 1000
4     }
5     if (g(n/3) > 5) {
6        for (int i = 0; i < n; i++) {
7           println("Yay!")
8        }
9        return 5 * g(n/3)
10    }
11    else {
12       for (int i = 0; i < n * n; i++) {
13          println("Yay!")
14       }
15       return 4 * g(n/3)
16    }
17 }
```

(a) Find a recurrence $T(n)$ modeling the worst-case time complexity of $g(n)$.

Solution:

$$T(n) = \begin{cases} c_0 & \text{if } n \leq 1 \\ 2T\left(\frac{n}{3}\right) + c_1 n + c_2 & \text{otherwise} \end{cases}$$

(b) Find a closed form for the above recurrence.

Solution:

The recursion tree has height $\log_3(n)$, each non-leaf level i has work $(\frac{c_1 n}{3^i} + c_2)2^i$, and the leaf level has work $c_0 2^{\log_3(n)}$. Putting this together, we have:

$$\begin{aligned}
\sum_{i=0}^{\log_3(n)-1} \left((\frac{c_1 n}{3^i} + c_2)2^i\right) + c_0 2^{\log_3(n)} &= \sum_{i=0}^{\log_3(n)-1} \left(\frac{c_1 n 2^i}{3^i} + c_2 2^i\right) + c_0 2^{\log_3(n)} \\
&= c_1 n\Big(\sum_{i=0}^{\log_3(n)-1} (\frac{2}{3})^i\Big) + c_2\Big(\sum_{i=0}^{\log_3(n)-1} 2^i\Big) + c_0 2^{\log_3(n)} \\
&= c_1 n \left(\frac{1 - \left(\frac{2}{3}\right)^{\log_3(n)}}{1 - \frac{2}{3}}\right) + c_2\Big(\frac{1 - 2^{log_3(n)}}{1 - 2}\Big) + c_0 2^{\log_3(n)} \quad \text{Finite geometric series} \\
&= 3c_1 n \left(1 - \left(\frac{2}{3}\right)^{\log_3(n)}\right) c_2(2^{log_3 n} - 1) + c_0 2^{\log_3(n)} \\
&= 3c_1 n \left(1 - \frac{n^{\log_3(2)}}{n}\right) + c_2(n^{log_3 2} - 1) + c_0 n^{\log_3(2)} \\
&= 3c_1 n - 3c_1 n^{\log_3(2)} + c_2 n^{log_3(2)} - c_2 + c_0 n^{\log_3(2)} \\
&= 3c_1 n + (c_0 + c_2 - 3c_1) n^{\log_3(2)} - c_2
\end{aligned}$$