Solution Set

CIS 1210—Data Structures and Algorithms—Fall 2024

Recurrence Relations & Code Snippets—Tuesday, September 10 / Wednesday, September 11

Readings

- Lecture Notes Chapter 6: Analyzing Runtime of Code Snippets
- Lecture Notes Chapter 7: Recurrence Relations

Review: Recurrences

Recurrences are equations that can help us describe the running time of recursive algorithms, denoted as $T(n)$, in terms of the runtime of strictly smaller inputs. There are a few different methods from lecture so far that we can use to solve recurrences:

Expansion (Iteration): We compute $T(n)$ using repeated substitution, expanding $T(n)$ fully and/or until we notice a pattern that has a formula (geometric series, consecutive integers, etc.) that we can use to simplify the algebra and obtain an overall bound for the recurrence.

Induction: We can "guess" what the asymptotic bound on $T(n)$ is and prove that it holds true via induction (often strong induction) on n.

Recursion Trees: To confirm our bound, we can draw the recursive calls to $T(n)$ in a tree format, count the amount of work done in each level of the tree, and sum this up across levels to compute/estimate $T(n)$.

Review: Code Snippets

We can also apply our knowledge of Big-O and summations to find the running time of code snippets, which typically contain nested iteration. Besides recursion, nested iteration is another big area where the runtime of an algorithm can be bottlenecked. There are a few different methods to analyze the running time of code snippets:

Superset/Subset Method: To find a Big-O bound, since we are upper bounding, we can take *supersets* of the values that the incremented variables take on in the code snippet and analyze the runtime on these supersets. To find a Big-Ω bound, since we are lower bounding, we can take *subsets* of the values that variables take on in the code snippet and analyze the runtime on these subsets. Note that this method is typically most useful when the conditions in the for loops increment the variables through addition (rather than when the variables are multiplied by a constant, for example).

Table Method: We can use a table to explore how the values of the variables change as a function of the number of iterations. Thus, this method could be a good option when the conditions in the for loops look confusing (e.g. square roots, multiplying, etc.). After using the table method, we typically follow-up with the summation method to analyze runtime.

Summation (Direct Equality) Method: We can convert nested for-loops into nested summations and then evaluate the nested summations to find our bounds. Usually this entails finding the number of iterations each loop runs and the amount of work done inside of the nested loops.

Problems

Problem 1

Assume n is a power of 2.

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + n^2 & n > 1 \\ 1 & \text{otherwise} \end{cases}$$

Solution

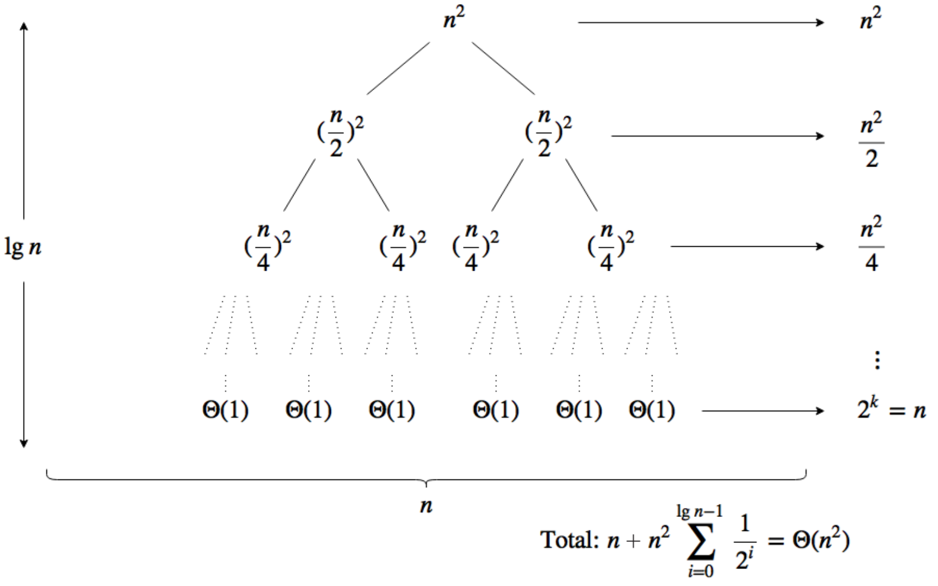
Since we can assume that n is some power of 2, observe that $n = 2^k$ implies that $k = \lg n$. Using the method of iteration/expansion, we expand $T(n)$ as follows:

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n^2 \\
&= 2\left[2T\left(\frac{n}{2^2}\right) + \left(\frac{n}{2}\right)^2\right] + n^2 && = 2^2T\left(\frac{n}{2^2}\right) + \frac{n^2}{2} + n^2 \\
&= 2\left[2\left[2T\left(\frac{n}{2^3}\right) + \left(\frac{n}{2^2}\right)^2\right] + \left(\frac{n}{2}\right)^2\right] + n^2 && = 2^3T\left(\frac{n}{2^3}\right) + \frac{n^2}{2^2} + \frac{n^2}{2} + n^2 \\
&\vdots \\
&= 2^kT\left(\frac{n}{2^k}\right) + n^2\sum_{i=0}^{k-1}\left(\frac{1}{2}\right)^i \\
&= 2^kT\left(\frac{n}{2^k}\right) + n^2 \cdot \frac{1-(\frac{1}{2})^k}{1-\frac{1}{2}} && \text{(geometric series, } c = \tfrac{1}{2} \text{ and } n = k-1\text{)}
\end{aligned}$$

The recursion bottoms out when $n/2^k = 1$, so when $k = \lg n$. Thus, we get

$$\begin{aligned}
T(n) &= 2^{\lg n} \cdot T(1) + 2n^2 \cdot \left(1 - \frac{1}{n}\right) && \text{(substituting } k = \lg n\text{)} \\
&= n + 2n^2 - 2n \\
&= \Theta(n^2)
\end{aligned}$$

We can also expand $T(n)$ using a recursion tree to confirm the Theta bound we just found:



On the right side of the diagram, we can see the total amount of work done at each level of the tree. On the left side of the diagram, we can see how many levels the tree has or how long it takes for the recursion to "bottom out." From the recursion tree, we can visualize our final sum without all the initial algebraic manipulation! Our total adds up the cost of the leaves with the costs of all the remaining levels, giving us the same answer as the expansion method.

Problem 2

Provide a running time analysis of the following loop. That is, find both Big-O and Big-Ω:

```
for(int i = 0; i < n; i++)
  for (int j = i; j <= n; j++)
    for (int k = i; k <= j; k++)
      sum++;
```

Solution

We can try to analyze the running time using an exact summation while also leveraging some Big-O notation. We know that the innermost loop runs in at most $(j - i + 1)$ time for fixed some i, j. Hence, the body of the middle loop runs at most $c(j - i + 1)$ time. Since $0 \leq i < n$ and $i \leq j \leq n$,we can express the running time of the code snippet as

$$\sum_{i=0}^{n-1} \sum_{j=i}^{n} c(j - i + 1)$$

Note that this summation is difficult to compute by hand. However, since the conditions in the nested for loops just deal with **incrementing variables with addition,** we can find a Big-O and Big-Ω bound for the code snippet by finding the running time on a superset and subset of values, respectively.

First, we find Big-O. Since Big-O upper bounds the running time of an algorithm, we consider supersets of values for i, j, and k and upper bound the running time of the code on these supersets. We are able to do this because an upper bound on the running time of these supersets is also going to an upper bound on the running time of the code snippet! Specifically, we choose supersets where $0 \leq i \leq n$, $0 \leq j \leq n$, $0 \leq k \leq n$. Since we have three loops, each loop runs at most $n + 1$ times, and we perform constant work inside of the nested loops, we can say that the entire code snippet runs in $O((n + 1)^3) = O(n^3)$ time.

Next, we find Big-Ω. Since Big-Ω lower bounds the running time of an algorithm, we consider subsets of values for i, j, and k and lower bound the running time of the code on these subsets. We are able to do this because a lower bound on the running time of these subsets is also going to be a lower bound on the running time of the code snippet! Specifically, we choose subsets where $0 \leq i \leq n/4$ and $3n/4 \leq j \leq n$. Since the inner loop runs from when $k = i$ until when $k \leq j$, we know that for each $n^2/16$ possible combinations of these values for i and j, the innermost loop runs at least (or a minimum of) $3n/4 - n/4 = n/2$ times, since $3n/4$ is the minimum value of j we bound on and $n/4$ is the maximum value of i we bound on. Hence, since we also perform constant work inside of the loops, the running time is $\Omega\left(\frac{n^2}{16} \cdot \frac{n}{2}\right)$, or equivalently, $\Omega(n^3)$.

The subset method for finding the lower bound can be a bit tricky, so we have provided the diagrams below for visualization. The first diagram shows the correct subsets for i, j, while the second one shows an example of an incorrect subset. In both diagrams, the top row represents the values that i takes on, starting at 0, then 1, then moving onto representative values such as $\frac{n}{4}$, $\frac{n}{2}$, $\frac{3n}{4}$, and $n-1$. The columns represent the values that j takes on. The gray and dark green colored cells mark the range that j goes through for each value of i.

The green rectangle in the first diagram represents the subset for i and j we chose above. Specifically, when $0 \leq i \leq \frac{n}{4}$ and $\frac{3n}{4} \leq j \leq n$, the rectangle is formed by the intersection of these two regions. The reason the first subset is valid is because all of the values in the green rectangle are part of the original values, i.e. there's no part of the green rectangle where there are no values. Conversely, in the second diagram, we chose the subsets $0 \leq i \leq \frac{n}{2}$ and $\frac{n}{4} \leq j \leq \frac{n}{2}$, which is not valid. As you can see on the top right of the green triangle, there's part of it that has no values (shaded a lighter green), meaning that our current ”subset” is not proper.

Value of i

	$\mathbf{0}$	$\mathbf{1}$	$\cdots$	$\frac{n}{4}$	$\cdots$	$\frac{n}{2}$	$\cdots$	$\frac{3n}{4}$	$\cdots$	$n-1$
	0									
	1	1								
	$\vdots$	$\vdots$								
	$\frac{n}{4}$	$\frac{n}{4}$	$\cdots$	$\frac{n}{4}$						
Value of j	$\vdots$	$\vdots$		$\vdots$						
	$\frac{n}{2}$	$\frac{n}{2}$	$\cdots$	$\frac{n}{2}$	$\cdots$	$\frac{n}{2}$				
	$\vdots$	$\vdots$		$\vdots$		$\vdots$				
	$\frac{3n}{4}$	$\frac{3n}{4}$	$\cdots$	$\frac{3n}{4}$	$\cdots$	$\frac{3n}{4}$	$\cdots$	$\frac{3n}{4}$		
	$\vdots$	$\vdots$		$\vdots$		$\vdots$		$\vdots$		
	n	n	$\cdots$	n	$\cdots$	n	$\cdots$	n	$\cdots$	n

Value of i

	$\mathbf{0}$	$\mathbf{1}$	$\cdots$	$\frac{n}{4}$	$\cdots$	$\frac{n}{2}-1$	$\frac{n}{2}$	$\cdots$	$\frac{3n}{4}$	$\cdots$	$n-1$
	0										
	1	1									
	$\vdots$	$\vdots$									
	$\frac{n}{4}$	$\frac{n}{4}$	$\cdots$	$\frac{n}{4}$							
	$\vdots$	$\vdots$		$\vdots$							
Value of j	$\frac{n}{2}-1$	$\frac{n}{2}-1$	$\cdots$	$\frac{n}{2}-1$	$\cdots$	$\frac{n}{2}-1$					
	$\frac{n}{2}$	$\frac{n}{2}$	$\cdots$	$\frac{n}{2}$	$\cdots$	$\frac{n}{2}$	$\frac{n}{2}$				
	$\vdots$	$\vdots$		$\vdots$		$\vdots$	$\vdots$				
	$\frac{3n}{4}$	$\frac{3n}{4}$	$\cdots$	$\frac{3n}{4}$	$\cdots$	$\frac{3n}{4}$	$\frac{3n}{4}$	$\cdots$	$\frac{3n}{4}$		
	$\vdots$	$\vdots$		$\vdots$		$\vdots$	$\vdots$		$\vdots$		
	n	n	$\cdots$	n	$\cdots$	n	n	$\cdots$	n	$\cdots$	n

Note: There are usually many different supersets/subsets that you can take to get achieve a Big-O/Big-Ω bound. However, it is important to double check that your subsets and supersets are valid. In this example, since the value of j depends on i, the values of j we take as a “subset” should actually be a subset for each value of $0 \leq i \leq n/4$ we bound on for us to have a “valid” subset of the original values j can take on. That is, when $i = 0$, we see that $0 \leq j \leq n$; when $i = 1$, we see that $1 \leq j \leq n$; and then when $i = n/4$, $n/4 \leq j \leq n$. In each case, we can see that $3n/4 \leq j \leq n$ will always be a subset of the actual values j takes on given a value of i in our subset. The choice of $\frac{1}{4}n$ and $\frac{3}{4}n$ as the upper bound for i and lower bound for j respe ctively is somewhat arbitrary. As long as the two intervals are disjoint, the two inner loops will yield a term that is easily computable. But we cannot be too arbitrary, as the intervals should grow in proportion to n in size. Additionally, observe that some code snippets may not have a matching Big-O/Big-Ω bound; while this code snippet does, depending on what superset/subset we take, we may not find a Θ-bound.

Problem 3

Provide a running time analysis of the following loop:

```
for (int i = 4; i < n; i = i*i)
  for (int j = 2; j < Math.sqrt(i); j = j+j)
    System.out.println(‘‘*”);
```

Solution

At first glance, because of the conditions in the nested for loops, it is difficult to see how the values i and j change as the code runs. Hence, we will analyze the running time of this code snippet by first exploring how i and j change with the table method, and then by using what we found with a table to derive a summation for the running time.

Let x and y be the numbers of iterations that the outer loop and inner loop run, respectively. To get a better understanding of how the values of i and j are changing as the code runs, we construct the following table:

Iteration	Value of i	-	Iteration	Value of j
1	$2^{2^1} = 4$	-	1	$2 = 2^1$
2	$2^{2^2} = 16$	-	2	$4 = 2^2$
3	$2^{2^3} = 256$	-	3	$8 = 2^3$
$\vdots$	$\vdots$	-	$\vdots$	$\vdots$
x	2^{2^x}	-	y	2^y

Now, we represent the running time as a summation. To do this, we need to solve for the number of iterations the outer loop runs, the number of iterations the inner loop runs, and how much work is performed inside the nested (outer and inner) loop. Inside the nested loop, we print a character, so we perform constant work. Thus, we can set up our initial summation (with placeholders for the number of iterations of the outer and inner loop) as follows:

$$\sum_{x=1}^{?} \sum_{y=1}^{?} 1$$

First, we solve for x, the number of iterations the outer loop runs, in terms of n. From the code snippet, the outer loop runs as long as $i < n$. Referencing our table, we can see that for the xth iteration, the value of i is 2^{2^x}, so the number of iterations of the outer loop is:

$$2^{2^x} \approx n \Rightarrow x \approx \lg \lg n$$

Then, we solve for y, the number of iterations the inner loop runs. From the code snippet, the inner loop runs as long as $j < \sqrt{i}$, so we will first solve for y in terms of x. Referencing our table, we can see that for the yth iteration, the value of j is 2^y, so the number of iterations of the inner loop is:

$$2^y \approx \sqrt{i} \Rightarrow y \approx \lg \sqrt{i}$$

Plugging in the upper bounds for x and y into our summation gives us

$$\sum_{x=1}^{\lg \lg n} \sum_{y=1}^{\lg \sqrt{i}} 1$$

Note that i is still on the top of the inner summation. Our goal is to simplify the summation into a running time for the code snippet, so to do this, we want to represent i in terms of x, the current variable on the outer loop/summation. Referencing our table, the value of i at the xth iteration is 2^{2^x}, so plugging this in gives us

$$i = 2^{2^x} \Rightarrow \sum_{x=1}^{\lg\lg n} \sum_{y=1}^{\lg\sqrt{2^{2^x}}} 1$$

Simplifying the inner summation and applying log properties,

$$\sum_{x=1}^{\lg\lg n} \left(\lg\sqrt{2^{2^x}}\right) = \sum_{x=1}^{\lg\lg n} \frac{1}{2}\left(\lg 2^{2^x}\right) = \sum_{x=1}^{\lg\lg n} \frac{1}{2}2^x = \Theta\left(\sum_{x=1}^{\lg\lg n} 2^x\right)$$

Applying the formula $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ to solve $\sum 2^x$,

$$T(n) = \Theta(2^{\lg\lg n}) = \Theta(\lg n)$$

Note: In many cases, you can approximate the upper limits of the loops, as it often will not affect the Big-O runtime. If you want to be precise, you should use ceilings and floors to get the proper runtime. Sometimes, you can also ignore constants, and when you go through the algebra of a problem for the first time, it is okay to ignore them. However, after you finish, you should look back at your solution and see if an extra constant could have mattered. For example, $f(n) = 2n$ and $g(n) = 3n$ are asymptotically equivalent, but $f'(n) = 2^n$ and $g'(n) = 3^n$ are not.