Solution Set

CIS 121—Data Structures and Algorithms—Fall 2020

Recurrence Relations, Code Snippets—Monday, September 21 / Tuesday, September 22

Readings

- Lecture Notes Chapter 6: Analyzing Runtime of Code Snippets
- Lecture Notes Chapter 7: Divide & Conquer and Recurrence Relations

Problems: Recurrences

Problem 1 [Solve the Following Recurrences]

Problem 1 a

$$T(n) = \begin{cases} T(n-1) + n & n \geq 1 \\ 1 & \text{otherwise} \end{cases}$$

Solution.

Using the method of iteration, we expand $T(n)$ as follows:

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= [T(n-2) + (n-1)] + n \\ &= [[T(n-3) + (n-2)] + (n-1)] + n \\ &\vdots \\ &= T(0) + 1 + 2 + ... + (n-1) + n \\ &= T(0) + \sum_{i=1}^{n} i \\ &= 1 + \frac{n(n+1)}{2} \\ &= \Theta(n^2) \end{aligned}$$

Problem 1 b

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.

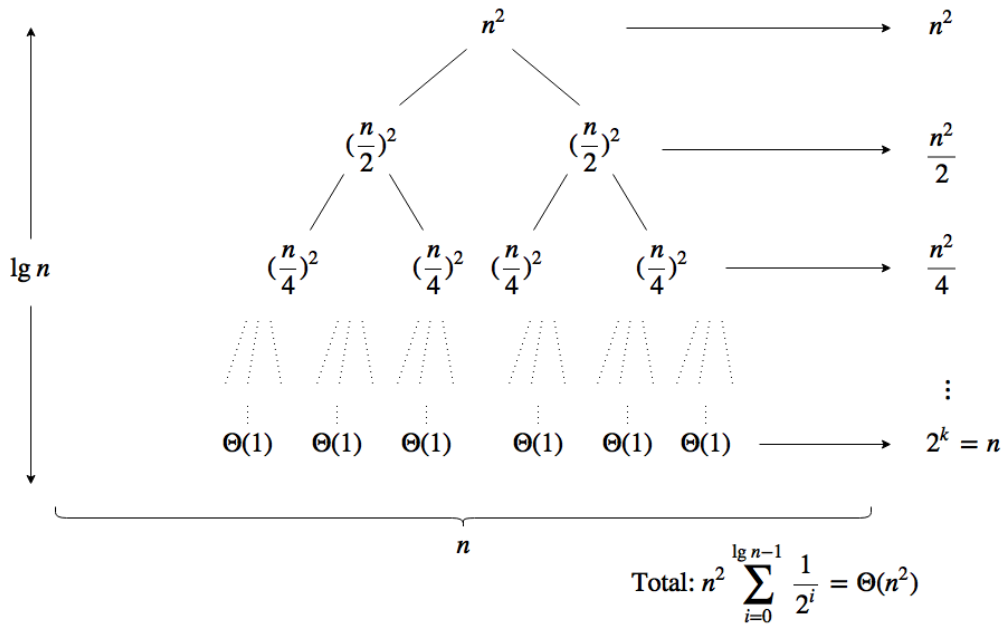
We first solve by iteration. First, we may assume that n is some power of 2 such that $n = 2^k \implies k = \lg n$.

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n^2 \\
&= 2\left[2T\left(\frac{n}{2^2}\right) + \left(\frac{n}{2}\right)^2\right] + n^2 \\
&= 2\left[2\left[2T\left(\frac{n}{2^3}\right) + \left(\frac{n}{2^2}\right)^2\right] + \left(\frac{n}{2}\right)^2\right] + n^2 \\
&\vdots \\
&= 2^k T\left(\frac{n}{2^k}\right) + n^2 \sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^i
\end{aligned}$$

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

$$\begin{aligned}
T(n) &= 2^k T\left(\frac{n}{2^k}\right) + n^2 \sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^i \\
&= 2^k T\left(\frac{n}{2^k}\right) + n^2 \frac{1 - (\frac{1}{2})^k}{1 - \frac{1}{2}} && \text{(geometric series)} \\
&= 2^{\lg n} T(1) + 2n^2\left[1 - \frac{1}{n}\right] && \text{(substituting } k = \lg n\text{)} \\
&= n + 2n^2 - 2n \\
&= \Theta(n^2)
\end{aligned}$$

Now, let's consider a different approach—we'll expand the recurrence using a recursion tree:



On the right side, we can see the total amount of work done at each level of the tree. Our sum becomes immediately apparent without all the initial algebraic soup! Note that the final summation does not include the work done at the leaves, n, but this does not change the Theta bound.

Code Snippets

We can apply our knowledge of Big-O and summations to find the run time of a snippet of our code. Besides recursion, nested iteration is where our code's efficiency will be bottlenecked. We should consider the loop as a summation, and use our knowledge to simplify it from there. Try starting from the innermost loop with fixed bounds and working outwards.

Problems (Code Snippets)

Problem 2

Problem 2 a

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.

Let x and y be the numbers of iterations that the outer and inner loops run. To get a better sense of how 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

We can represent the run time as a summation:

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

First, solve for the number of iterations of the outer loop (x) (**in terms of** n):

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

Then, solve for the number of iterations of the inner loop (y) (**in terms of** i):

$$2^y \approx \sqrt{i} \Rightarrow y \approx \lg \sqrt{i} \tag{3}$$

Plug in the upper bound of x, y into the recurrence:

$$\sum_{x=1}^{\lg \lg n} \sum_{y=1}^{\lg \sqrt{i}} 1 \tag{4}$$

Note that i is still on the top of the inner summation, now we want to represent i in terms of x, which is the current variable on the outer loop:

$$i = 2^{2^x} \Rightarrow \sum_{x=1}^{\lg \lg n} \sum_{y=1}^{\lg \sqrt{2^{2^x}}} 1 \tag{5}$$

Simplify the inner summation:

$$\sum_{x=1}^{\lg\lg n}(\lg\sqrt{2^{2^x}}) = \sum_{x=1}^{\lg\lg n}\frac{1}{2}(\lg 2^{2^x}) = \sum_{x=1}^{\lg\lg n}\frac{1}{2}\dot{2^x} = \Theta(\sum_{x=1}^{\lg\lg n}2^x) \tag{6}$$

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

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

Note: In many cases, you can approximate the upper limits of the loops as it won't affect the Big-O runtime. If you want to be precise, you'd have to 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's not wrong to ignore them, but after you finish you must 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.

Problem 2 b

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

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

Solution.

Observe that each of these loops runs at most $n+1$ times. Since the three nested loops each run at most $n+1$ times, we can say that the entire block of code runs in $O(n^3)$

To find a lower bound on the running time, we consider smaller subsets of values for i, j and lower-bound the running time for the algorithm on these subsets. (A lower bound there would also be a lower bound for the original code's running time!) Consider the values of i such that $0 \le i \le n/4$ and values of j such that $3n/4 \le j \le n$. For each of the $n^2/16$ possible combinations of these values of i and j, the innermost loop runs at least $n/2$ times. Therefore, the running time is at least $\Omega\left(\frac{n^2}{16}\cdot\frac{n}{2}\right)$, or equivalently, $\Omega(n^3)$. $\square$

Alternate Solution for Big-O.

One could solve this using exact sums, but we will leverage some Big-O notation. We know that the innermost loop runs in at most $(j-i+1)$ time for fixed i, j (see other solution). Therefore the body of the middle loop runs at most $c(j-i+1)$ times. Therefore, we can express the running time of the code shown as

$$\sum_{i=0}^{n}\sum_{j=i}^{n}c(j-i+1) = O(n^3) \qquad \square$$

Note: This summation is difficult to compute by hand. Therefore, we need to have other ways of solving these problems (as shown above).

Additional Practice Problems

Problem 3

Assume n is a power of 3.

$$T(n) = \begin{cases} 2T(\frac{n}{3}) + n & n > 1 \\ 1 & \text{otherwise} \end{cases}$$

Solution.

We solve this by iteration. First, we may assume that n is some power of 3 such that $n = 3^k \implies k = \log_3 n$.

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{3}\right) + n \\ &= 2\left(2T\left(\frac{n}{3^2}\right) + \frac{n}{3}\right) + n \\ &= 2\left(2\left(2T\left(\frac{n}{3^3}\right) + \frac{n}{3^2}\right) + \frac{n}{3}\right) + n \\ &\vdots \\ &= 2^k T(\frac{n}{3^k}) + n\sum_{i=0}^{k-1}\left(\frac{2}{3}\right)^i \end{aligned}$$

The recursion bottoms out when $n/3^k = 1$, i.e., $k = \log_3 n$. Thus, we get

$$\begin{aligned} 2^{\log_3 n}T(1) + n\sum_{i=0}^{\log_3 n-1}\left(\frac{2}{3}\right)^i &= n^{\log_3 2} + n\left(\frac{1-\left(\frac{2}{3}\right)^{\log_3 n}}{1-\frac{2}{3}}\right) \\ &= n^{\log_3 2} + 3n\left(1 - \frac{2^{\log_3 n}}{3^{\log_3 n}}\right) \\ &= n^{\log_3 2} + 3n - 3n\left(\frac{2^{\log_3 n}}{n}\right) \\ &= 3n - 2n^{\log_3 2} \\ &= \Theta(n) \end{aligned}$$

Problem 4

You are given the following algorithm for `Bubble-Sort`:

Algorithm 1 Bubble Sort

```
function BUBBLE-SORT(A, n)
    for i ← 0 to n − 2 do
        for j ← 0 to n − i − 2 do
            if A[j] > A[j + 1] then
                swap(A[j], A[j + 1])
            end if
        end for
    end for
end function
```

Given some sequence $\langle a_1, a_2, \ldots, a_n\rangle$ in A, we say an inversion has occurred if $a_j < a_i$ for some $i < j$. At each iteration, `Bubble-Sort` checks the array A for an inversion and performs a swap if it finds one. How many swaps does `Bubble-Sort` perform in the *worst-case* and in the *average-case*?

Solution.

It should be fairly obvious that the worst-case scenario for bubble-sort occurs when A contains elements in reverse-sorted order. Let I denote the number of inversions. In this situation, the total number of inversions is the exact number of possible pairs of elements, as each swap removes exactly one inversion:

$$I_{worst} = \binom{n}{2}$$

In the average-case scenario, we determine the expected number of inversions in a random array of n elements. Specifically, we consider a random permutation of n distinct elements, $\langle a_1, a_2, \ldots, a_n \rangle$.

Let X_{ij} denote an indicator R.V. such that,

$$X_{ij} = \begin{cases} 1 & \text{if } a_i, a_j \text{ inverted} \\ 0 & \text{otherwise} \end{cases}$$

Thus, we have

$$\begin{aligned} I_{average} &= \sum_{i,j \,:\, i<j} X_{ij} \\ \mathrm{E}[I] &= \sum_{i,j \,:\, i<j} \mathrm{E}[X_{ij}] \\ &= \sum_{i,j \,:\, i<j} \Pr[X_{ij} = 1] \\ &= \frac{\binom{n}{2}}{2} \\ &= \frac{n(n-1)}{4} \end{aligned}$$

In the above, we let $\Pr[X_{ij} = 1] = \frac{1}{2}$ because in a random permutation (uniform probability), the probability of $a_i < a_j$ is the same as $a_i > a_j$. □