CIS 1210—Data Structures and Algorithms—Spring 2025

Recurrence Relations & Code Snippets—Tuesday, February 4 / Wednesday, February 5

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.

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

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++;
```

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(``*");
```