CIS 121—Data Structures and Algorithms with Java—Spring 2018

Code Snippets and Recurrences—Monday, January 29/Tuesday, January 30

Learning Goals

- Practice solving recurrences and proving asymptotic bounds
- Strengthen intuition for the *divide and conquer* paradigm
- Review Quicksort

Code Snippets

We can apply our knowledge of Big-Oh 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.

Code Snippet Problems

Problem 1

Problem 1 a

Provide a running time analysis of the following loop. That is, find both Big-Oh 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++;
```

Problem 1 b

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

```
for (int i = 2; i < n; i = i*i)
  for (int j = 1; j < Math.sqrt(i); j = j+j)
    System.out.println("*");
```

Divide and Conquer - Quicksort

Using the divide and conquer paradigm, we can solve problems by dividing them into smaller subproblems, and later merging them together. While Mergesort spends most of its time combining these subproblems after their completion, we can also perform this additional work before we solve the subproblems.

In an instance of Quicksort, we must first decide on a 'pivot'. This could be the element at any location in the input array. The function `Partition` is then invoked. `Partition` accomplishes the following: it places the pivot in the location that it should be in the output and places all elements that are at most the pivot to the left of the pivot and all elements greater than the pivot to its right. Then we recurse on both parts. The pseudocode for Quicksort is as follows.

```
QSort(A[lo..hi])
  if hi <= lo then
    return
  else
    pivotIndex = floor((lo+hi)/2)  (this could have been any location)
    loc = Partition(A, lo, hi, pIndex)
    QSort(A[lo..loc-1])
    QSort(A[loc+1..hi])
```

One possible implementation of the function `Partition` is as follows.

```
Partition(A, lo, hi, pIndex)
  pivot = A[pIndex]
  swap(A, pivotIndex, hi)
  left = lo
  right = hi-1
  while left <= right do
     if (A[left] <= pivot) then
        left = left + 1
     else
        swap(A, left, right)
        right = right - 1
  swap(A, left, hi)
  return left
```

Ideally, we'd like a recurrence describing our Quicksort to look like:

$$T(n) = \begin{cases} 1, & n = 1 \\ 2 * T(n/2) + cn, & n \geq 2 \end{cases}$$

However, certain inputs can cause sub-optimal run time. For example, an instance of the quicksort algorithm in which the pivot is always the first element in the input array performs poorly on an array in descending order of its elements.

The worst case running time of the algorithm is given by

$$T(n) = \begin{cases} 1, & n = 1 \\ T(n-1) + cn, & n \geq 2 \end{cases}$$

Hence the worst case running time of `QSort` is $\Theta(n^2)$.

Recurrences

As you have seen in class, *recurrences* are equations that can help us describe the running time of a recursive algorithm.

You have thus far seen two different ways of solving recurrences:

> **Iteration.** In this method, we expand $T(n)$ fully by substitution and solve for $T(n)$ directly.
>
> **Recursion trees.** In this method, we draw the recursive calls to $T(n)$ in a tree format and count the amount of work done in each level of the tree.

In this lab, we will focus on the *method of iteration.*

Let's first go through some examples before running through problems.

Example: Method of Iteration I

Let's examine the following recurrence $T(n)$.

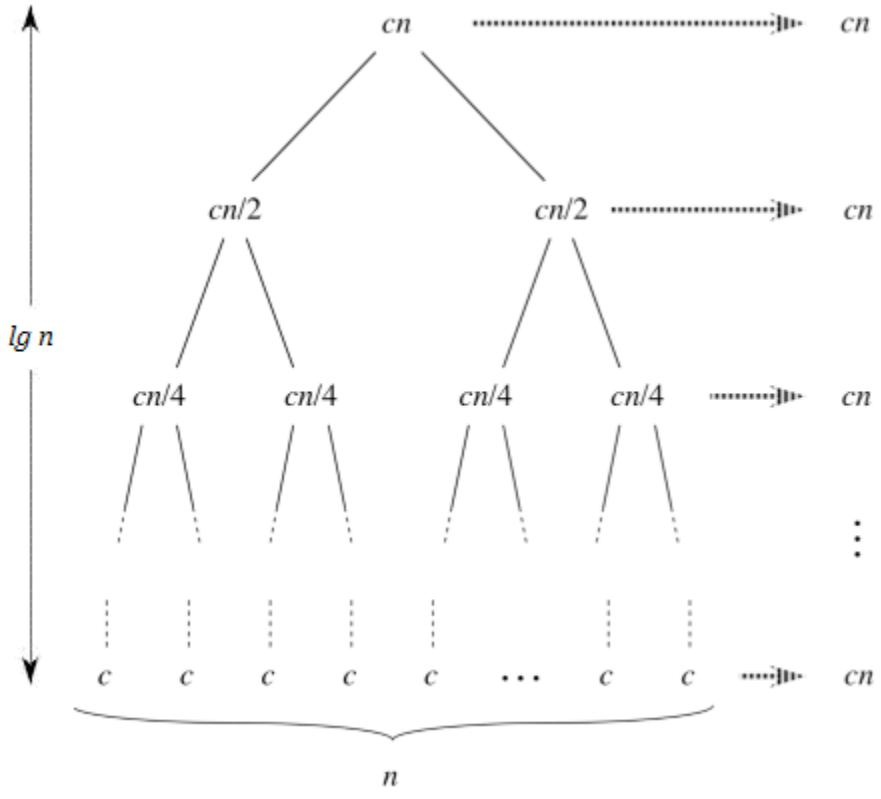
$$T(n) = \begin{cases} T(n-1) + n & n \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

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

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

Example: Method of Recursion Trees

Let's examine the recurrence of merge sort. For those that are unfamiliar with it, the algorithm works by taking an unsorted array, sorting the left and right halves of the array recursively, and then merging the two sorted halves together to end up with the final sorted list. Let $T(n)$ represent the time the algorithm takes for an input of size n. Since the two halves are sorted recursively by the same algorithm, but with inputs that are each half the size of the original, each half should take time $T(\frac{n}{2})$. The merging takes linear time. So we can write $T(n) = 2 \cdot T(\frac{n}{2}) + cn$ for some constant c. The recursion-tree is shown below.



Note that at the very top level (i.e., the end of the algorithm), it costs cn to merge the two sorted halves of the array. But to get there, we needed to solve the two problems of size $\frac{n}{2}$. Each costs $c \cdot \frac{n}{2}$ to solve. Therefore, across that level, the total cost is $c \cdot \frac{n}{2} + c \cdot \frac{n}{2} = cn$. We can continue this all the way until we get to the very bottom of the tree, which are single elements. Note then that every level ends up costing cn. The height of the tree is $\lg n$. Therefore, the total cost is $c \cdot n \lg n$.

To see why the height is $\lg n$, observe that subproblem sizes decrease by a factor of 2 each time we go down one level, we stop when we reach singleton elements. The subproblem size for a node at depth i is $\frac{n}{2^i}$. Thus, the subproblem size hits $n = 1$ when $\frac{n}{2^i} = 1$ or, equivalently, when $i = \lg n$. You can read more examples in CLRS 4.4.

Problems

You may assume in the following cases that n is either some power of 2 or 3. You may also find the following helpful:

Geometric series:

$$\sum_{i=0}^{k} q^i = \frac{q^{k+1} - 1}{q - 1} = \frac{1 - q^{k+1}}{1 - q}$$

Sum of binomial coefficients over upper index:

$$\sum_{j=0}^{n} \binom{j}{m} = \binom{n+1}{m+1}$$

Problem 1

Problem. Solve the following recurrences.

1. $$T(n) = \begin{cases} T(n-1) + \binom{n}{2} & n \geq 2 \\ 1 & \text{otherwise} \end{cases}$$

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

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

4. $$T(n) = \begin{cases} T(n-1) + 2^n & n \geq 1 \\ 1 & \text{otherwise} \end{cases}$$

Problem 2

The CIS160 and CIS121 TAs are having another social gathering. For some reason or another, the 160 TAs particularly enjoy playing the *Tower of Hanoi* and insist that the 121 TAs compete with them to see who can solve the game the fastest. Of course, the clever 121 TAs are very proficient with their running time analysis skills, and know that an optimal strategy is detailed by the following code:

```
function HanoiTower(h, from, to, mid)
    if h ≥ 1 then
        HanoiTower(h − 1, from, mid, to)
        MoveDisk(from, to)                          ▷ O(1) time operation
        HanoiTower(h − 1, mid, to, from)
```

Problem. What is the running time of the above algorithm? Prove it inductively.