CIS 575. Introduction to Algorithm Analysis
Material for February 5, 2024

Recurrences, and Why We Need To Solve Them

The topic of this set of notes is also the topic of *Cormen*'s Chapter 4.

1 Recurrences

We have already studied how to analyze the running time of iterative algorithms. We shall now study how to analyze the running time of **recursive** algorithms.

1.1 Recursion on Slightly Smaller Input

Early in this course, we did see a recursive algorithm:

```
InsertionSort(A[1..n])
  if n > 1
    InsertionSort(A[1..n − 1])
    InsertLast(A[1..n])
```

With $T(n)$ the time it takes in the worst case to run that algorithm on input of size n, observe that $T(n)$ (for $n > 1$) will be the sum of

1. the time it takes to check if $n > 1$, which is in $\Theta(1)$
2. the time it takes to do the recursive call, which by definition is $T(n-1)$
3. the time it takes in the worst case to run InsertLast, which is in $\Theta(n)$.

This motivates the below **recurrence** for $T(n)$:

$$T(n) = \Theta(1) + T(n-1) + \Theta(n)$$

which suggests that (with a suitable choice of time unit) it will be the case that

$$T(n) \approx T(n-1) + n$$

which (assuming $T(0) = 0$) suggests that

$$T(n) \approx \sum_{i=1}^{n} i$$

It is now easy to infer $T(n) \in \Theta(n^2)$, which we already knew to be the worst-case running time of the corresponding *iterative* program.

In general, our results about how to approximate sums (such as Theorem 3.28 in *Howell*) will often suffice to also analyze a recursive function, as long as the recursion is “small-step” in the sense that when given parameter (of size) n the function makes at most one recursive call, with parameter (of size) $n-1$.

But there exists other and more powerful kinds of recursion, where the input is divided into chunks that are recursively processed separately, as done by the algorithms based on the *Divide & Conquer* paradigm (to be covered in detail later in this course). In this set of notes, we shall focus on techniques to analyze that kind of recursion.

1.2 Recursion on Chunks of Input

To illustrate the kind of recursion we shall analyze, let us consider the **Merge Sort** algorithm whose workings may be illustrated by an example: given

| 18 | 14 | 12 | 27 | 20 | 28 | 10 | 11 |

the algorithm *recursively* sorts the two halves:

| 12 | 14 | 18 | 27 || 10 | 11 | 20 | 28 |

and then merges the two halves into a *new* array: since $10 < 12$, it puts 10 first; since $11 < 12$, it puts 11 second; since $12 < 20$, it puts 12 third; etc, etc; the end result is

| 10 | 11 | 12 | 14 | 18 | 20 | 27 | 28 |

With Merge a function that given two sorted arrays returns a sorted permutation of their elements, this can be expressed by the algorithm

MergeSort($A[1..n]$)
  if $n > 1$
    $m \leftarrow \lfloor n/2 \rfloor$
    MergeSort($A[1..m]$)
    MergeSort($A[m+1..n]$)
    $B[1..n] \leftarrow$ Merge($A[1..m], A[m+1..n]$)
    copy($B[1..n], A[1..n]$)

Analyzing Merge Sort It is obvious that Merge will run in time and space proportional to the size of its input. As a consequence, we see that MergeSort is *not* “in-place”. Let us now analyze the running time $T(n)$ of MergeSort when applied to an array with n elements:

- it will make two recursive calls; each will take time $T(\frac{n}{2})$
- it will take time in $\Theta(n)$ to run Merge
- it will take time in $\Theta(n)$ to copy B into A.

This motivates a *recurrence* which may be written

$$T(n) \in 2T(\frac{n}{2}) + \Theta(n)$$

This is a very common recurrence; it turns out that the solution is given by $T(n) \in \Theta(n \lg(n))$. In subsequent notes, we shall study how to solve general recurrences.