Algorithms in Systems Engineering
ISE 172

Lecture 8

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - Chapter 4
- References
  - CLRS Chapters 4
  - R. Miller and L. Boxer, *Algorithms: Sequential and Parallel*, 2000, Chapter 3.
  - R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.

Analyzing Merge Sort

- Suppose the running time of merge sort is given by T.
- We analyze each piece of the algorithm separately.
  - Divide: This operation involves finding the midpoint of the array, which is in $\Theta(1)$.
  - Conquer: We recursively solve two subproblems, each of size $n/2$, which is $2T(n/2)$.
  - Combine: The running time of the merge subroutine is in $\Theta(n)$.
- So T satisfies the following *recurrence*.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- How do we figure out what T is?

Analyzing Recurrences

- In the last slide, we analyzed merge sort using two different methods.
- General methods for analyzing recurrences
  - Telescoping
  - Build a recursion tree.
  - Solve analytically.
  - Make a guess and prove that it's right (usually with induction).
  - Use the *Master Theorem*.
- Note that when we analyze a recurrence, we may not get or need an exact answer.
- We may prove the running time is in $O(f)$ or $\Theta(f)$ for some simpler function f.
- When taking the ratio of two integers, it usually doesn't matter whether we round up or down.

A Few Examples

- This recurrence arises in algorithms that loop through the input to eliminate one item.

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

- This recurrence arises in algorithms that halve the input in one step.

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + 1 & n > 1 \end{cases}$$

- This recurrence arises in algorithms that halve the input in one step, but have to scan through the data at each step.

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + n & n > 1 \end{cases}$$

The Master Theorem

- Most recurrences that we will be interested in are of the form

$$T(n) = \begin{cases} 1 & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

- The Master Theorem tells us how to analyze recurrences of this form.
  - If $f \in O(n^{\log_b a - \varepsilon})$, for some constant $\varepsilon > 0$, then $T \in \Theta(n^{\log_b a})$.
  - If $f \in \Theta(n^{\log_b a})$, then $T \in \Theta(n^{\log_b a} \lg n)$.
  - If $f \in \Omega(n^{\log_b a + \varepsilon})$, for some constant $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and $n > n_0$, then $T \in \Theta(f)$.
- How do we interpret this?

A Few More Examples

- This recurrence arises in algorithms that partition the input in one step, but then make recursive calls on both pieces.

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/2) + 1 & n > 1 \end{cases}$$

- This recurrence arises in algorithms that scan through the data at each step, divide it in half and then make recursive calls on each piece.

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

- We can analyze these using the Master Theorem.

The Call Stack

- The call stack of a program keeps track of the current sequence of function calls.
- When a new function call is made, data for the current one is saved on *the call stack*.
- When a function call returns, it returns to the next function on the top of the stack.
- The *stack depth* is the maximum number of functions on the stack at any one time.
- In a recursive program, the stack depth can be very large.
- This can create memory problems, even for simple recursive programs.
- There is also an overhead associated with each function call.

Iterative Algorithms

- All recursive algorithms have iterative counterparts.
- In the case of linear recursion, the conversion is usually easy.
  - Example: Binary search.
- In the case of a branching recursion, it's not as easy.
  - Example: Merge sort.
- The advantage of the iterative counterpart is that it usually saves memory and the overhead of function calls.
- Generally, the iterative version is much more complex, however.