Computational Optimization
ISE407

Lecture 7

Dr. Ted Ralphs

Readings for Today's Lecture

- Miller and Boxer, Chapters 2 and 3.
- Aho, Hopcroft, and Ullman, Sections 2.5–2.9.
- R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.

Recursion

- A recursive function is one that calls itself.
- There are two basic types of recursive functions.
  - A *linear recursion* calls itself once.
  - A *branching recursion* calls itself two or more times.
- Examples of linear recursion
  - Euclid's algorithm
  - Binary search

Properties of Recursive Algorithms

- Generally speaking, recursive algorithms should have the following two properties to be guarantee well-defined termination.
  - They should solve an explicit base case.
  - Each recursive call should be made with a smaller input size.
- All recursive algorithms have an associated *tree* that can be used to diagram the function calls.
- Execution of the program essentially requires *traversal* of the tree.
- By adding up the number of steps at each *node* of the tree, we can compute the running time.
- We will revisit trees later in the course.

Divide, Conquer, and Combine

- Many recursive algorithms arise from employment of a *divide-and-conquer* approach.
- This means breaking a larger problem into pieces that can be solved independently.
- The solutions to the various pieces may then have to recombined in some way.
- More accurately, these are *divide, conquer, and combine* algorithms.
- Such algorithms have natural implementations using branching recursions.
- Example: Merge sort
  - Divide the list in half.
  - Sort each half (recursively).
  - Merge the two halves together.
- The running time depends on how we do the merging.

Implementing Merge Sort

- Here is the subroutine for implementing a basic merge sort.
- To sort an entire array the call would be `MergeSort(array, 0, length)`.

```
MergeSort(list, beg, end)
    if beg < end:
        mid = (beg + end)/2
        MergeSort(list, beg, mid)
        MergeSort(list, mid + 1, end - mid)
        Merge(list, beg, mid, end)
```

Implementing Merge

- There are many ways to implement the merge, but here is one simple one.
- Note that this involves copying over the elements of the array.

```
Merge(list, beg, end, mid)
    temp1 = list[beg:mid + 1]
    temp2 = list[mid + 1:end]
    i, j = 0, 0
    for k in range(end - beg)
        if i == mid - beg:
             list[k] = temp1[i]; i+=1
             continue
        if j == end - mid:
             list[k] = temp2[j]; j+=1
             continue
        if temp1[i] < temp2[j]:
             list[k] = temp1[i]; i+=1
        else:
             list[k] = temp2[j]; j+=1
```

Proving Correctness

- As we mentioned earlier, there is a natural connection between induction and recursion.
- Most recursive algorithms can be proven by induction in a very natural way.
- Merge Sort
  - Assuming the merge is done correctly, correctness of the main subroutine is "obvious."
  - It can be shown formally by induction.
  - To show the merge works correctly, we can use a *loop invariant*.
  - What is the loop invariant in the merge subroutine?

Aside: Some Simple Optimization

- Handling small arrays
- Eliminating copying (reduce memory requirements)
- Using sentinels

```
Merge(list, beg, end, mid)
   temp1 = list[beg:mid + 1]
   temp2 = list[mid + 1:end]
   temp1[mid - beg +1] = MAXINT
   temp2[end - mid]    = MAXINT
   i, j = 0, 0
   for k in range(end - beg)
      if temp1[i] < temp2[j]:
          list[k] = temp1[i]; i+=1
      else:
          list[k] = temp2[j]; j+=1
```

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.

Recursion and Complexity

- Many algorithms can be expressed very naturally using recursion (whether it should be used in implementation is another question).
- Example: SAT Problem
  - Recursion is a natural way to express the naive enumeration algorithm for solving the SAT Problem.
  - We reduce the original problem of size n to two subproblems of size $n-1$ by setting x_1 to either TRUE or FALSE.
  - By recursively solving these two subproblems, we solve the original problem.
- Recursion can also be used as a way of building up classes of functions.
- Computability theory (also called recursion theory) is a theory related to complexity theory in which recursion is a central concept.

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 stack* (recall the stack memory we discussed earlier).
- When a function call returns, it returns to the next function on the *call stack*.
- The *stack depth* is the number of function calls on the stack at any one time and is limited essentially by the availability of stack memory.
- In a recursive program, the stack depth can get 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.
- Essentially, the iterative version must manually replicate the call stack data structure.
- In the case of linear recursion, this is 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.
- The recursive version is usually much easier to implement, but only because it exploits the automated data structures provided by the compiler.
- These data structures can be costly.