Algorithms in Systems Engineering
ISE 172

Lecture 7

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - Chapter 4
- References
  - CLRS Chapters 2
  - R. Miller and L. Boxer, *Algorithms: Sequential and Parallel*, 2000, Chapter 2.
  - 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
  - Binary search
  - Euclid's algorithm
    * Divide m by n and let r be the remainder.
    * If $r = 0$, then $\gcd(m, n) = n$.
    * Otherwise, $\gcd(m, n) = \gcd(n, r)$.

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 of a Recursive Algorithm

- There is a natrual connection between induction and recursion.
- Most recursive algorithms can be proven by induction in a very natural way.
- Example: 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?

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
```