Algorithm Analysis

Some of the types of problems we've looked at where we'd like to analyze the efficiency of functions or algorithms:

1) Generic Array Algorithms
2) General Recursive Algorithms
3) Sorting:
- **A) Selection Sort**
- **B) Merge Sort**
- **C) Quick Sort**

5) Implementation of Data Structure functions
- **A) Stacks**
- **B) Queues**
- **C) Linked Lists**
- **D) Binary Trees**

In each of these, a few types of operations occur often:

1) A single loop, or set of nested loops
2) A search or some sort, where the search space decreases by a factor of two after each iteration
3) A call or set of calls to the exact same problem with a smaller input
4) You iterate through the "length" of a data structure

A single loop, or set of nested loops

When analyzing any algorithm with this type of structure, here are some of the questions you should answer:

1) How much time does the code inside the inner-most loop take?

2) How many times does the inner-most loop run?

3) Is the answer to question #1 variable upon either loop counter or some value?

4) Is the answer to the question #2 variable upon an outer loop?

In the most straightforward cases, we will have the same amount of work in each loop iteration:

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    array[i][j] += c;
```

In these cases, count the work for each loop inside out. The code inside of the inner-most loop runs in O(1) time, always. This code is run exactly n times, always. Thus, this inner loop always takes O(n) time. Finally, that runs exactly n times, leading to a total run-time of $O(n^2)$.

Now, consider a situation where there isn't the same amount of work in each loop iteration. These types of situations can come in many different types. I'll give examples of two:

```
for (i=0; i<n; i++)
  for (j=i+1; j<n; j++)
    if (array[i]>array[j])
      swap(&array[i], &array[j]);
```

A couple of problems here are that the if statement in the innermost loop may sometimes be entered, but not be entered other times. The nice part though is that whether the if is entered or not entered, the amount of work is constant. So, the amount of work in the inner loop is always O(1). (This is since a comparison takes constant time as does a swap.)

But, now we see that the number of times the inner loop runs, depends on i. Iterating the outer loop, we see that the inner loop first runs n-1 times, then n-2, then n-3, etc.

This is where we must utilize our knowledge of summations to do an accurate analysis.

Of course, there are other situations that fit into this category that don't have a set pattern for the number of times each iteration runs. Consider the following:

```
while (i < j) {

  while (i <= end &&
         values[i] <= values[start])
     i++;
  while (values[j] > values[start])
     j--;

   // Swap out of place values.
   if (i < j)
     swap(values+i, values+j);
 }
```

In this situation, the number of times the inner loops run is variable. In fact, they seem as if they could run up to j-i times. But then the question becomes, how many times can the outer loop run. At first it's not even clear it will ever end. But once you start looking at the meaning of the code, it's clear that each time either i or j will change value so that it must end.

Ultimately, to determine the run-time here, you have to notice that even though we have a double loop structure, the most number of times i and j can be incremented/decremented is the initial value of j-i, and that each increment or decrement occurs in a constant amount of time. Thus, the running time of this loop is O(j-i), where j and i are the original values of those variables at the beginning of the loop.

Analyzing Recursion

Here are some recursive algorithms we looked at:

1) Fibonacci
2) Towers of Hanoi
3) Generating Permutations

One main technique for analyzing the running time of recursive algorithms is to write out a recurrence relation satisfied by T(n), the run-time of the function for an input of size n, (or value n). For Fibonacci, we find that:

T(n) = T(n-1)+T(n-2)+O(1), (assuming that an addition is constant time.)

This is because the work involved in determining F(n) is first determining F(n-1), (which takes T(n-1) time), then determining F(n-2), (which takes T(n-2) time), followed by an addition that takes constant time.

Although we didn't talk about analyzing recurrences of this form in this class, other problems do lead to recurrences that we can solve using the Master Theorem.

However, sometimes, even if we get a recurrence relation we can solve for the run-time of a recursive algorithm, we can sometimes determine the run-time of the algorithm anyway by simply analyzing the total amount of work done by it. Consider the following examples:

Examples of the Analysis of Recursion

In the Towers of Hanoi example, it's clear that a constant amount of work is involved for each "move." (This constant amount of work invoves a recursive function call and a print statement, usually.) Based on our analysis of the game, we found that the total number of moves needed to transfer a tower of n disks was 2^n-1. Since we know how much time each move takes, and how many moves ultimately get done, we can ascertain that the Towers of Hanoi algorithm runs in $O(2^n)$ for an input of n disks.

In minesweeper, the recursive clear is similar to a floodfill that changes the entire color of a bounded region to a different color. If you look at the code for either of these, a single function call could make upto 8 recursive function calls. The analysis here seems disasterous! However, once again, we realize that the total amount of work for a single square is constant. Thus, the running time of either of these is proportional to the number of squares cleared, or the number of pixels who's color is changed.

Sorting Analysis

1) Slow Sorts: Most of these have a nested loop structure that fits into either the first or second example I showed in the first part of these notes. Both turn out to sort n numbers in $O(n^2)$ time. The main key to understanding all of these algorithms is to notice that each iteration of the outer loop accomplishes some task that aids in sorting. (This could by placing the i^{th} smallest element in place, or something to that effect.) Also, almost always the outer loop runs n times while the inner loop runs a variable number of times (either starting at 0 or 1 and counting up, or starting at n or n-1 and counting down.)

2) Merge Sort: The theoretical analysis says that this sort should be the best in performance, but it isn't because in practice you are forced to copy answers from and back to the original array during the Merge function. It's important to understand where the recurrence relation to solve for the running time of the algorithm comes from:

$$T(n) = 2T(n/2) + O(n)$$

The first term on the right hand side comes from two recursive calls to Merge Sort on arrays of size n/2 while the last term comes from the amount of time it takes to Merge two arrays of size n/2 together.

3) Quick Sort: This is usually the best in practice since partition can be implemented easily without extra copying. However, in the worst case (of bad partition splits) this algorithm is $O(n^2)$. This can be avoided a very high percentage of the time by randomly choosing a partition element.

Running Times of Data Structure Functions

Stacks: Push and Pop, *when implemented efficiently,* run in O(1) time. If you use an array implementation, if you dynamically size the array, double the size of the array when an element is added to a full array. This results in maintaining an amortized efficiency of O(1) for Push and Pop operations. (Similarly, one can also halve the size of the array when less than 25% of it is being used.) A linked list implementation may be even easier. You only insert and delete nodes from the front of the list.

Queues: Enqueue and dequeue, *when implemented efficiently,* run in O(1) time. If you use an array implementation, make sure you have all the details straight. You need three parts of a struct: the array, a front index, and the total number of elements in the array. The linked list implementation requires storing a pointer to both the front and back of the list.

Linked Lists: Most functions take O(n) time, where n is the length of the list. These are functions that iterate through the nodes of the list and process each node in some manner (that takes a constant amount of time.) In analyzing this type of code, look for the number of nodes visited, or the number of changes in links.

Binary Trees: Most functions that iterate through all nodes take O(n) time, where n is the number of nodes in the tree. Most functions that go down some random path in the tree take O(h) time, where h is the height of the tree. In AVL Trees, we proved in the worst case that h = log n, approximately.

Choice of Data Structure

1) Choose the easiest data structure that efficiently handles the task at hand. In my experience, this means you get to use lots of arrays.

2) However, don't ignore situations where a more complex data structures could help the efficiency of the problem. One example is a linked list that is part of a hash table.