COP 3502: Introduction

The first question you might be asking is: "How is this class going to be different than C programming, COP 3223?"

In the past, the two courses, COP 3223 and COP 3502 were quite similar here at UCF, since the former was not a prerequisite for the latter. However, we have now had COP 3223 as a prerequisite for COP 3502 for nearly a year now and can safely assume that all students in COP 3502 have taken COP 3223 or an equivalent course.

This indicates that the syllabus for this course can be adapted somewhat. In particular, time will NOT be spent in this course to teach the following:

language basics, including variable declarations, conditional expressions, if statements, loops, functions, and arrays

With respect to the language, the only information that will be reviewed will be pointers, 2D arrays, structures and linked lists. (We will also cover binary trees.)

Otherwise, our goals will be as follows:

1) Improving knowledge of standard data structures and abstract data types.

2) Improving knowledge of standard algorithms used to solve several classical problems.

3) Covering some mathematics that is useful for the analysis of algorithms.

4) Analyzing the efficiency of solutions to problems.

Whereas in COP 3223 we only cared if we found a solution to a problem, in COP 3502, we will also learn standard ways to solve problems and also how to analyze the efficiency of those solutions. Finally, we will simply expand upon our knowledge of our use of the C programming language.

For the rest of the lecture, we will look at two problems and two separate solutions to both problems: one that is straightforward that a COP 3223 student may be able to come up with, and another that is more efficient that a good COP 3502 student may arrive at after some thought. Hopefully this example will illustrate part of the goal of this course.

Problem #1: Finding a value in a sorted array

You are given a sorted array, A, of n integers and a value, x, for which to search. Determine whether or not x is stored in the array A.

Straight-forward Solution

Look in each index of the array, in order. If any of these values are equal to x, return 1 (true). If not, return false. In code it would look like this:

```
int search(int* array, int length, int searchval) {

    int i;
    for (i=0; i<length; i++)
        if (array[i] == searchval)
            return 1;

    return 0;
}
```

This solution could take up to n steps and notice that it doesn't ever utilize the fact that the array is sorted.

Can we do better, if we use this fact?

Faster Solution

Just like the guessing game, let's first look in the middle of the array. If the number we are looking for is LESS than this value, we know we can restrict our search to the left half of the array. Alternatively, we can restrict our search to the right half of the array. In code, we'll indicate our search space with two indexes, representing the low and high indexes within the array where our value might be. The "middle" of our search range will be the average of indexes low and high. Our code would look like this:

```
int binsearch(int* array, int length, int searchval) {

    int low = 0, high = length-1;

    while (low <= high) {

        int mid = (low+high)/2;

        if (searchval < array[mid])
            high = mid-1;
        else if (searchval > array[mid])
            low = mid+1;
        else
            return 1;
    }

    return 0;
}
```

Now, let's analyze how many steps this will take. Notice that if our search space starts at n, that after 1 step, our search space is no bigger than n/2. After 2 steps, the search space is no bigger than n/4. After 3 steps, the search space is no bigger than n/8. In general, after k steps, our search space is no bigger than $n/2^k$. Essentially, the algorithm will stop after our search space is size 1. (Once low equals high, we are guaranteed that our loop won't run again.) Thus, we must solve the following equation for k, the number of steps this algorithm takes:

$$n/2^k = 1$$
$$n = 2^k$$
$$k = \log_2 n$$

Thus, this algorithm, a binary search, takes no more than roughly $\log_2 n$ steps. That means, if we're searching amongst a million items, we'll only make 20 comparisons, at most. This is a HUGE savings over the original algorithm.

Problem #2: Box of 0s and 1s.

You are given an nxn integer array where each row is filled with several 1s followed by all 0s. The goal of the problem is to determine the maximum number of 1s in any row.

First, let's go through a straight-forward solution without using any code:

For each row do the following:

> **Start from the beginning of the row, scanning from left to right, until the first zero is encountered, all the while, keeping track of how many ones have been seen on that row.**
>
> **If the number of ones is greater than the previous maximum seen, then update the maximum number of 1s seen.**

Clearly this will work. Now, let's go through how long this algorithm will take.

Basically, we iterate through each square that contains a 1 in it, as well as the first 0 in each row. If all cells were 0s, we would "visit" n squares total. However, if all the cells were 1s, we would "visit" n^2 squares total. Thus, in the worst case, the number of simple steps this algorithm would take would be approximately n^2, this makes the running time of this algorithm $O(n^2)$. (The meaning of this Big-Oh will be discussed later this semester.)

There seems to be some potential extra work done here. Once we know that one row has 12 1's for example, it seems pointless to start at the beginning of the next row. Why not just start in column 12? If it's a 0, then certainly that row can't be a winner, and if it is a 1, clearly there is no point now to go back and check the previous 11 squares.

This idea leads to the following algorithm:

```
1) Initialize the current row and current column to 0.
2) While the current row is less than n (or before the last row)
        a) While the square at the current row and column is 1,
                Increment the current column.
        b) Increment the current row.
3) The current column index represents the maximum number
of 1s seen.
```

Now, let's trace through a couple examples of this algorithm:

1	1	0	0	0	0
0	0	0	0	0	0
1	1	1	0	0	0
1	1	1	0	0	0
1	1	1	1	1	0
1	1	1	1	0	0

0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0
1	1	1	1	0	0	0	0
1	1	1	1	1	1	0	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1

How many steps will this algorithm take, in terms of n?

Each "step" taken by the algorithm either goes to the right or down in this chart. There are a maximum of n-1 steps to the right and n-1 steps down that could possibly be taken. Thus, the maximum number of "steps" (increments of variables) that can be done during this algorithm is approximately 2n, which makes the running time of this algorithm O(n) in the worst case, an improvement over the previous algorithm shown.

Also, note that the second case shows that we must carefully handle the case where a row has all 1s in it so as to not have any array out of bounds errors or return the wrong answer.

Implementing an Algorithm in C

One of the skills you will have an opportunity to improve upon in this class is write a program in C that implements an algorithm you have learned. While an algorithm specifies the general steps to solve a problem, one needs to pay special attention to certain programming details and know the syntax of a language to properly implement the algorithm.

While there's no set way to create code to implement an algorithm, I will use this as an example to show you some steps you may take in doing so.

Here are some issues to think about:

1) What data structures are going to be used?
2) What functions are going to be used?
3) What run-time errors should we protect against?
4) What atypical cases may we have to deal with?
5) What is an efficient way to execute the steps in the algorithm?

Althugh this was quite a creative exercise, much of what you learn in this class will not be. We have many set algorithms and data structures you will study. Occasionally you will have to come up with new ideas like this one, but mostly, you'll have to apply the data structures and algorithms shown in class fairly directly to solve give problems.