Announcements (0:00-9:00)

- HCS will be holding a seminar on Vim today at 4 PM in SC 229. Vim is a more advanced text editor.
- Problem set 3 up will be posted tonight.
- By the way, here are the answers to the problem set 2 hacker edition
    - mscott: 1234
    - dhelment: 12345
    - homer: beer
    - sjobs: iPhone
    - bgates: w1nd0ws
    - malan: djmftw!

Making Change: A Greedy Algorithm (9:00-13:00)

- Suppose you go into a convenience store and are owed 44 cents change
- If you're a cashier, what's your algorithm?
- Give back the largest coin less than or equal to 44, repeat
- That is, first, give back a quarter, so you're left with same problem on 19 cents. Next, give back a dime, so you're left with the same problem on 9 cents. Continue until 0 cents remain.
- Is this optimal solution, i.e., does it give back the fewest coins possible? Yes.
- This is a greedy algorithm. In each iteration, we bite off the largest piece of the problem possible.
- But this isn't always the best way to solve a problem.

Choosing Stamps: Exhaustive Search (13:00-18:00)

- Back in the day stamps came in the following denominations: 34, 21, 1
- So now if we want to put 44 cents worth of stamps, ideal combination is 21, 21, 1, 1
- But if we took a greedy approach, we'd be screwed. First we'd put a 34-cent stamp on our envelope, and then the only way to deal with the remaining 10 cents would be with 1's. This would clearly not be optimal.
- Rather than using a greedy algorithm, in this problem we must do an exhaustive search to get the best possible outcome.
- Here's some pseudocode representing this algorithm:

```
public static int howmany(postage p)
{
For all denomination d
      Determine number of stamps required for postage p-d
Return (mnimum # found + 1)
}
```

- That is, first we try to see what would happen if we used a 34-cent stamp, and then we see what would happen if we used a 21-cent stamp, and so on. After looking at all the possibilities, we return an answer.
- To test what would happen if we used a 34-cent stamp, we take the problem of 44 and shrink it down to 44-34 = 10, then call the algorithm again on 10.
- This is called recursion because in each step we break off a piece of the problem that we can do and then call the algorithm again on the rest of the problem.
- Not explicit in this pseudocode is the base case p = 0, which would return 0.
- As we can see from this example, doing an exhaustive search can end up taking a really long time, but in some cases it is the only way to ensure an optimal solution.

The Fibonacci Sequence: Recursion (18:00-31:00)

- We define f(x) as follows:

$$f(x) = \begin{cases} 0, \text{if } x = 0 \\ 1, \text{if } x = 1 \\ f(x-1) + f(x-2), \text{otherwise} \end{cases}$$

- This gives the following sequence: 0 1 1 2 3 5 8 13 …
- We can write a program that generates the nth Fibonacci number using recursion
- Take a look at the main routine of fs1.c:

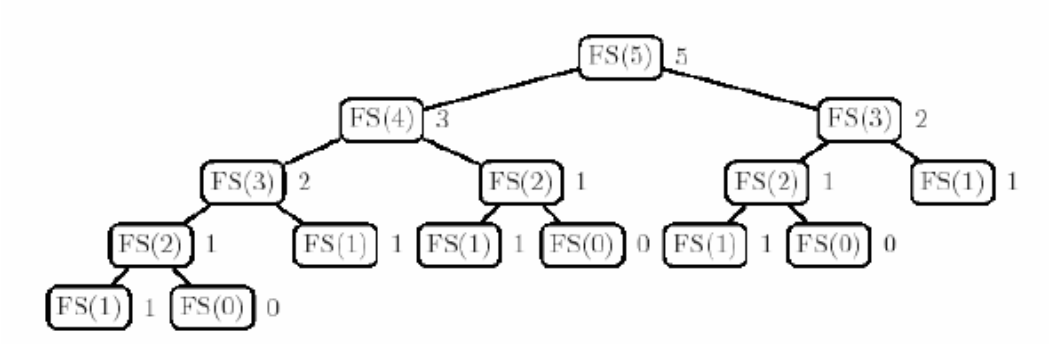
```
int
main(int argc, char * argv[])
{
    int n;

    /* ensure proper usage */
    if (argc != 2)
    {
        printf("Usage: %s n\n", argv[0]);
        return 1;
    }

    /* compute and print n'th number in Fibonacci sequence */
    n = atoi(argv[1]);
    if (n < 0)
        printf("Input must be non-negative.\n");
    else
        printf("fs(%d) = %lld\n", n, fs(n));
}
```

- As a side note, notice we do not need to check to make sure argv[1] is completely numeric as we did in caesar.c
    - This is because atoi() will actually return a 0 if its argument is not numeric.

  - You could figure this out on your own by going to cppreference.com or another documentation website.
- Now let's implement the function fs()
- We can break this down into an if-statement. First, take care of the base cases, and then the recursive definition:

```
long long
fs(int n)
{
    /* compute n'th number */
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return (fs(n-1) + fs(n-2));
}
```

- Notice that functions which are themselves recursive lend themselves extremely well to programs which are recursive.
- Let's compile and test this. We try fs(5), fs(10), fs(15), …, fs(30) and they all seem to work fine, but things start to slow down around 35 or 40
- fs(50) won't even compute after waiting a few minutes
- Why? Even though this algorithm appears to be really elegantly implemented, it's also really inefficient!
- Every time we increment the argument, we double the number of steps we complete. That is, the number of steps grows exponentially.
- This is because we must do many steps multiple times. For instance, in the following diagram, look how many times we compute fs(1) just to compute fs(5):



- If we run fs2.c, we can see how many times fs() was called on each number from 1 to n.
- For example, if we do fs(35), we see that we compute fs(1) 9227465 times.
- But fs(1) = 1 every single time. Why do we need to keep computing it?
- We need to keep computing it because our program has no memory.

- Although recursion make the program very clear and readable, it is not the most efficient solution to this problem

Dynamic Programming (31:00-42:00)

- In our next implementation, we will create an array of size n. This array will serve as a sort of scratch pad in which we will write down all of the Fibonacci numbers as we compute them until we get to n.
- Take a look at the main routine of fs3.c:

```
/* prototype */
long long fs(int);

/* cache of answers */
long long * memo;

/* sentinel value indicating absence of a memoized answer */
const long long SENTINEL = -1;


int
main(int argc, char * argv[])
{

    /* ensure proper usage - not shown */

    /* validate n - not shown */

    /* instantiate memo - not shown */

    /* initialize memo, using sentinel for answers not yet computed */
    memo[0] = 0;
    memo[1] = 1;
    for (i = 2; i <= n; i++)
        memo[i] = SENTINEL;

    /* compute and print n'th number in Fibonacci sequence */

}
```

- Not shown are the steps in which we check the arguments, and dynamically create an array of size n (we will learn how to do this later).
- Next we initialize memo by:
  - putting in the base cases: memo [0] = 0 and memo[1] = 1
  - filling the rest of the array with SENTINEL, a constant defined at the top of the program
- We fill empty slots with the sentinel value as an indication that the value belonging there has not yet been calculated.
- This is more secure than leaving them as 0's or whatever happens to already be there, which could be mistaken for actual Fibonacci values.

- Now when we want to compute fs(n), we check the nth location in the scratch pad. If it's not the sentinel (i.e., it's already been calculated), we simply return whatever is there
- If it is the sentinel (i.e., it has yet to be calculated), we compute it as fs(n-1) + fs(n-2) and stick that in memo[n] for future use.
- Take a look at the implementation of fs():

```
long long
fs(int n)
{
    /* compute n'th number */
    if (memo[n] != SENTINEL)
        return memo[n];
    else
        return (memo[n] = fs(n-1) + fs(n-2));
}
```

- We still seem to have recursion, but since we are remembering values as we compute them we can shortcircuit the process
- That is, we only calculate fs(x) once for any given x. When we need that value later, all we have to do is an array lookup which is very fast.
- Therefore, if we want to find fs(n) we need only calculate fs(x) for all x<n, which is n steps. This implementation is O(n)!
- Indeed, running this version all the way up to fs(6), we find that it is *much* more efficient.

Grading Software (42:00-45:00)

- We grade your code along three axes:
    - Correctness: Does your program do exactly what our pdf tells you to do?
    - Design: Can we read it? Is it efficient?
    - Style: Does it look nice?
- Some people have complained about being graded by the same standards as the experienced programmers in the class. Please do not worry. You need only concern yourself with your own progress and performance.

Debugging Software (45:00-53:00)

- So far, your best strategy for debugging has probably been placing print statements at strategic points in your code
- This is useful for determining how far into the program the computer gets, as well as tracing the values of variables
- However, this strategy does not scale. It takes a lot of time to put in statements and you must constantly recompile to get the benefits.
- A better tool that we will be working with in the coming week is gdb.
- gdb allows you to walk through your program line by line so you can see what's going on

- By stepping through your code, following function calls, and checking the values of variables, you can locate problems with your code
- To run gdb, type at the command line gdb a.out after compiling your program
- Once you have opened gdb, you can run your program by typing run. This will not tell you much. If your program has been seg-faulting, it will continue to do so.
- The way we take advantage of gdb is by inserting break statements. You can insert a break statement at a particular line or at a particular function call.
- To stop your program at, say, main, type `break main`. Then when we run, gdb stops the program when it gets to main
- Another tool we can use is print. At any point while running gdb, you can type, say, `print a` and it will tell you the value of the variable a.
- After gdb has stopped your program at a particular place, you can type `next` to prompt it to execute the next line code.
- The best way to learn about gdb is to use it. Sit down at a terminal and run a program (broken or working) through gdb. Experiment with commands like `break`, `step`, `next`, and `print`, to see how gdb allows you to follow your code.
- A gdb reference card with common commands is available on the website.