COSC-111 Spring 2014

Lab 12: Fast and Slow Algorithms

Point your browser to (or click on) . . .

`https://app.cs.amherst.edu/sfkaplan/courses/2014/spring/COSC-111/algorithm.py`

. . . and download a copy of the program. Fire up IDLE (version 3!) and take it for a run.

The notation $\Theta(f(n))$ means "proportional to $f(n)$" (as far as you're concerned).

1. This program asks you for a number n. Then it calls **sortem** to sorts n integers using that $\Theta(n^2)$ method I described in class on Wednesday. Then it prints n and c, the number of times the most common instruction (where the counter is in the code) was executed.

   Using the graph paper, make a chart of n vs c, for a handfull of values of n between 1 and 100. Assuming c is proportional to n^2, can you figure out what the constant of proportion is?

   What kind of n is needed for this code to take about 5 seconds to run, if you estimate by counting 1-mississippi, 2-mississippi, etc.?

2. In the main function, I have commented out a call to **perms**, which creates an array of integers and then uses recursion to print all the permutations of the integers. Uncomment the function call (go ahead and comment out the call to **sortem**) and make another chart of n vs c, for values of n between **1 and 10**. (You may want to comment out the print statement in **permu**, too.)

   See what I mean about polynomial $\Theta(n^2)$ vs exponential $\Theta(n!)$ computation times? What kind of n is needed to get this function to run for 5 seconds?

3. I have started to write a function named **matmult** that takes three $n \times n$ matrices called **a**, **b** and **c**. Fill in the code to compute the **matrix product** $c = a \times b$, defined by $c[i][j] = \sum_{k=0}^{n-1} a[i][k] * b[k][j]$. You can do this with three nested loops, for i,j, and k.

   Include a counter that adds up how many times the arithmetic line is performed. The function should return the counter value when it is finished.

   Now make a chart for the computation cost of this $\Theta(n^3)$ function, for values of n betweeen 1 and 100. How does it compare to the other two? What is the constant of proportionality?

4. If you have time, write a function that takes $\Theta(2^n)$ time to compute. You can do this three ways:

- Given n, figure out the value of $m = 2^n$, and then write a loop that counts to m. Note that just calculating the value of m doesn't take $\Theta(2^n)$ time – you have to write the loop so it actually executes code that many times. Include a counter to count the loop iterations, and print it when finished.
- Given n, write a function that recursively calls itself twice, taking the argument n and passing the value $n - 1$ to the other two. (It stops recurring when $n = 0$.) Include a counter that works like this: at $n = 0$ it returns 1 (counting itself). At $n > 0$ it makes the two recursive calls and then returns the sum of counts returned by those two, plus 1 (counting itself).
- Or, find that Towers of Hanoi program from a few labs back – it also takes $\Theta(2^n)$ time. Add a counter to that recursive code as indicated above.

Your recursive function returns a counter c that should be proportional to 2^n. Now make a chart for n vs c for some reasonable values of n. What is the constant of proportion?

When you are finished, use the CS submissions system to submit your program. You can keep the charts.

The takeaway. This lab is about understanding the difference between fast (polynomial time) and slow (exponential time) agorithms and how they can affect computation times. Pay attention to three things:

1. Notice how you can insert counters in code to count how many times the most common statement is executed.
2. Try to build some intution for how code structure corresponds to the growth of these functions. A single loop is typically $\Theta(n)$. Two nested loops are $\Theta(n^2)$ if both loops count all the way to n. The $O(2^n)$ function uses recursion to count the levels, and has 2 recursive calls per level. The $O(n!)$ function uses recursion to count the levels ℓ, and a loop at each level that is proportional to $n - \ell$.
3. The main take-away lesson today: polynomial = good, exponential = bad.