CS 271 (Spring 2013) — Assignment 5
Due: 03/05/2013

From now on until the end of the semester, homeworks are back to the normal individual homeworks. If you want to work in groups, keep the "Gilligan's Island" policy in mind.

(1) Solve the following exercises from the textbook

(a) Section 1.7, Exercises 10, 12, 24, 36, 38

(b) Section 1.8, Exercises 4, 12, 24, 28

(c) Section 2.2, Exercise 22

(d) Section 2.3, Exercises 24, 80

(e) Section 3.2, Exercises 32, 44

(2) [0 points]

Chocolate Problem (1 chocolate bar): Hopefully, you still remember how we were doing big-Θ analysis of running times of programs. Wouldn't it be nice if we could just automate this? We feed a program to some program analyzer, and it spits out the running time in big-Θ notation for us? Unfortunately, it's impossible to write such a program. In fact, even solving the following seemingly simpler task is impossible: Given a program P and an input x, automatically figure out whether P will ever terminate on input x. (This is called the Halting Problem, and you will learn about it in CS270/CS303.) So figuring out the running time in all cases is out of the question.

However, for some simpler programs, we could automate the analysis, and that's what you're asked to do here. Our program gets at most two input variables n, m, and running times are to be expressed in terms of n, m. Only the following constructs will appear in the programs:

(a) Basic statements, simply denoted by `Basic`. Those are any statements that will take time $\Theta(1)$ and don't change any variable values.

(b) Simple `for` loops that only ever run their loop counter from 1 to either m or n, and only incremeent their counters by 1 in each step. (In other words, they are of the form `for (i=1; i <= n; i ++)`, or the same with m in place of n, and with other names for the loop variables. To keep your life simple, you can also assume that no two nested loops will ever use the same loop counter. (In other words, don't worry about overwriting variables for now.)

(c) In addition, there can be blocks grouped with curly braces.

That's it. You are to write a program that takes in such code, and outputs a maximally simplified big-Θ answer of the running time. As an example, for the following code:

```
for (i=1; i <= n; i ++)
{
  Basic;
  for (j=1; j <= n; j ++)
     Basic;
  for (j=1; j <= m; j ++)
     for (k=1; k <= m; k ++)
         Basic;
}
```

you should output $\Theta(n^2 + nm^2)$ (and not, for instance, $\Theta(n + n^2 + nm^2)$). Note that the "maximally simplified" part actually has some non-trivial implications. For instance, suppose that a preliminary analysis shows that the running time is $\Theta(n^2+nm^2+m^4)$. This can be simplified further to $\Theta(n^2+m^4)$, because either $n \leq m^2$, in which case $nm^2 \leq m^4$, or $n > m^2$, in which case $n^2 \geq nm^2$. So you may want to think somewhat carefully about this.

If you're having fun with this problem, you can of course expand it in various directions.

(a) Allow processing more C programs. Part of the difficulty will eventually become parsing. If you want to use this as an occasion to learn how to write a parser, I recommend looking at lex (or flex) and yacc (or bison). It takes a little while, but can be a worthwhile investment, and it's fascinating to learn how parsers work.

The other difficulty is that once you expand your allowed programs enough, you will run into the problem mentioned earlier, that you might have to even figure out whether a program terminates. I think that function calls can be handled, so long as you don't have any recursive calls (direct or indirect). Once you have recursion, all bets are off. Similarly, loops will work so long as you have a loop counter that is only incremented (say, by 1) in the `for`, but never altered by other code. If you allow loop counters to be altered by other code, or allow general `while` loops, again, all bets are off.

(b) A separate extension is to allow more than two variables m, n to measure the running time. I am pretty sure that this will make your task quite a bit more difficult; certainly, it will once you allow arbitrarily many variables. The type of argument we gave above about removing unnecessary terms becomes much more complex there.