Brute Force and Backtracking

Most of the problems in computer science are not solvable in polynomial time (at least as far as we know). This is true both in the theoretical sense and in practice. When faced with one of these problems, there are few options: give up, try brute force or settle for an approximate solution. In this section, we will focus on the second option. There are a surprising number of difficult problems that have brute force algorithms that are able to solve fairly large problem instances.

Estimating the Running Time

First of all, we need a way of estimating the running time of a given algorithm. The most obvious tool is the big-O notation. But what does it mean if an algorithm takes $O(n^2)$ time? How big can n be before the algorithm becomes impractical? Obviously, that depends on your CPU speed and type, your RAM, the operating system and lots of other complicated details. That was the main point behind the big-O notation – to avoid thinking about these details! This is why the big-O notation has a hidden constant that is meant to absorb the delays caused by hardware and software differences.

Actually, these differences are not that great, especially if we are talking about brute force algorithms that are exponential in time. $2 \cdot \log(n)$ seconds is very different from $2000 \cdot \log(n)$ seconds for a lot of interesting values of n. However, $2 \cdot 2^n$ seconds is about the same as $1000 \cdot 2^n$ seconds when $n = 100$ in the sense that both are longer than our lifetime. So if we have a $O(2^n)$ algorithm and want to run it on a problem of size 100, it doesn't matter what kind of computer we are using – it will not work.

We can go further and note that processor speeds have reached a plateau. Moore's law that promises exponential growth of CPU speeds no longer holds. Intel has announced that they have abandoned plans of creating a 4GHz CPU for a while and will instead focus on finding ways to put several processors on one chip. Apple has put off the release of the G5 Powerbook because they are having troubles with CPU cooling and power consumption. Processors are not likely to become any faster in the near future.

With this in mind, let me make a few "bold" statements. If you have a a program that requires 1 million operations, it will run in less than a second. Something with 100 million operations will run in a few seconds. With 1 billion operations, prepare to wait for several minutes. By "operations", I mean evaluating a simple expression, reading or storing a variable, etc. Think of one simple line in your program as one operation. This is basically true for all modern computers – if you want your program to run in a few seconds, you better make sure it is not executing more than a few hundred million lines of code (if you imagine unrolling all of the loops and functions calls so that there are no jumps anywhere in the code).

This simple rule is very vague, and I am sure that there are lots of examples when it does not hold, but it's close to the truth and it is very useful. For instance, we can use it to answer the original question, "How big can n be if we plan to run a simple $O(n^2)$ algorithm on it?" The answer is, "About 10000 if you want it to terminate in a few seconds." This is a crude estimate, but it gives a concrete number to work with. The true number might be somewhere between 5000 and 20000 - you can try it on your computer by writing a simple loop. What if an algorithm requires 2^n operations? In this case, the answer is, "27, give or take 2."

The purpose of the rule is to give a crude, but concrete estimate of the running time in places where

the big-O notation does not give enough information. Armed with this little tool, let's look at a few techniques for applying brute force to one classic problem.

Backtracking with DFS

By far the most useful technique in problem solving is "divide and conquer". Backtracking is one of its simplest applications. We want to find a solution to a problem instance. The solution is complicated and consists of many small parts. So we will try all possibilities for picking the first part. For each of these, we will try all possibilities of picking the second part. And so on until we finally build a complete solution. Or perhaps we are looking for the best solution. Then we will generate all solutions part by part and remember the best solution we find.

Here is a very famous problem. How many ways are there to place 8 queens on a chess board so that no two queens attack each other? A chess board is an 8-by-8 grid. Queens can be placed in the cells of the grid. Two queens are under mutual attack if they share a row, a column or a diagonal. Here is one possible solution:

X							
						X	
				X			
							X
	X						
			X				
					X		
		X					

A solution consists of 8 parts – the individual positions of the 8 queens. This is our basis for divide-and-conquer. Select a location for queen #1 (64 possibilities). For each of those, select a location for queen #2 (63 possibilities), etc. When we have all 8 queens placed, check that no two share a row, a column or a diagonal. If they do not, then we have a valid configuration – count it. At the end, return the number of valid configurations found.

This works, but it's ridiculously slow. There are 64*63*...*57 configurations to check, and it takes about 8*8 operations to check each one, for a total of over 10^{16} operations. There is no hope that this program will finish running in any kind of reasonable time. But at least it works; let's see how we can speed it up.

We know that no two queens can share a column, and there are 8 of them, so each column has exactly one queen on it. This gives us a different way to divide up a solution. A solution is 8 numbers in the range from 0 to 7 giving the rows on which the 8 queens reside, reading from left to right. We can represent the solution above by the 8 numbers (0,4,7,5,2,6,1,3).

Let's run the same algorithm, but this time, we will have only 8 possibilities for queen 1, 7 possibilities for queen 2, etc. After we have placed all 8 queens, we still have to check that no two share a diagonal. That still requires about 8*8 operations, but now we have only 8*7*...*1 configurations to check. Here is what an implementation might look like.

Example 1:

```
int row[8], used[8];
int queens( int i ) {
    if( i == 8 ) {
        // Placed all 8 queens. Verify diagonals.
        for( int j = 0; j < 8; j++ )
        for( int k = j + 1; k < 8; k++ ) {
            if( row[j] + j == row[k] + k ) return 0;
            if( row[j] - j == row[k] - k ) return 0;
        }
        return 1;
    }

    int total = 0;
    for( row[i] = 0; row[i] < 8; row[i]++ )
    {
        // ensure we are not sharing a row
        if( used[row[i]] ) continue;

        // place the queen and recurse on the rest
        used[row[i]] = true;
        total += queens( i + 1 );

        // mark the row as unused again
        used[row[i]] = false;
    }
    return total;
}
```

To use the function queens(), first clear used[] to be all false and then call queens(0). row[i] is the row in which queen in column i resides. used[r] is true if some queen we have already placed uses row r. Once i reaches 8, we verify that no two queens share a diagonal and return 1. queens(0) finishes in about 30 milliseconds and returns 92. That's quite an improvement.

You may have noticed that we are essentially generating all the 8! permutations with this function. The exact number of operations required is about 20 million (8!*8*36) - there are 8! recursive calls, each one requiring 8 iterations of the loop, and 36 operations to check for a valid configuration. This can also be done without any recursion, simply using next_permutation().

Note how similar this implementation is to DFS. If we consider each partial configuration (where only the first i queens have been assigned rows) a vertex, then we a doing a depth-first search on a certain configuration graph. It is useful to think of backtracking as graph search or DFS because we will be able to apply some of the ideas from graph theory here. For instance, think about what it would mean to use BFS in this case instead of DFS. Which way is more efficient? We will come back to this idea later.

It may seem like a waste to explore 8! (40320) configurations just to find the 92 that we are looking for. If we increase the size of the board to 10x10, the program will require about 5 seconds. For a 13x13 board, I waited for half and hour for it to finish and gave up. In the next section we will see one very useful technique that will allow us to improve the algorithm so that it works on larger board sizes. For more info on this classic problem, see
http://www.research.att.com/cgi-bin/access.cgi/as/njas/sequences/eisA.cgi?Anum=A000170