Backtracking

Backtracking is a technique used to solve problems with a large search space, that systematically tries and eliminates possibilities.

A standard example of backtracking would be going through a maze. At some point in a maze, you might have two options of which direction to go:

```
          Junction
     ---------------------- portion A
                 |
                 |
                 |
          portion B
```

One strategy would be to try going through portion A of the maze. If you get stuck before you find your way out, then you *"backtrack"* to the junction. At this point in time you know that portion A will NOT lead you out of the maze, so you then start searching in portion B.

Clearly, at a single junction you could have even more than two choices. The backtracking strategy says to try each choice, one after the other, if you ever get stuck, *"backtrack"* to the junction and try the next choice. If you try all choices and never found a way out, then there IS no solution to the maze.

Eight Queens Problem

The problem is specified as follows:

Find an arrangement of eight queens on a single chess board such that no two queens are attacking one another.

In chess, queens can move all the way down any row, column or diagonal (so long as no pieces are in the way).

Due to the first two restrictions, it's clear that each row and column of the board will have exactly one queen.

The backtracking strategy is as follows:

1) Place a queen on the first available square in row 1.
2) Move onto the next row, placing a queen on the first available square there (that doesn't conflict with the previously placed queens).
3) Continue in this fashion until either (a) you have solved the problem, or (b) you get stuck. When you get stuck, remove the queens that got you there, until you get to a row where there is another valid square to try.

When we carry out backtracking, an easy way to visualize what is going on is a tree that shows all the different possibilities that have been tried.

Consider the following page with a visual representation of solving the 4 Queens problem (placing 4 queens on a 4x4 board where no two attack one another).

```
                                          _ _ _ _
                                          _ _ _ _
                                          _ _ _ _
                                          _ _ _ _
                          /                                                  \
                 Q _ _ _                                                   _ _ _ _
                  _ _ _ _                                                  Q_ _ _
                  _ _ _ _                                                   _ _ _ _
                  _ _ _ _                                                   _ _ _ _
              /               |           \                                  /
     Q _ _ _             Q _ _ _         STUCK                     _ _ _ _
      _ _ _ _             _ _ _ _        (NO MORE                  Q_ _ _
      _ Q_ _              _ _ _ _          OPTIONS                 _ _ _ _
      _ _ _ _             _ Q _ _          ON COL 2)               _ Q_ _
      /                     /                                          |
 STUCK(1)            Q _ _ _                                       _ _ Q _
                      _ _ Q_                                       Q_ _ _
                      _ _ _ _                                      _ _ _ _
                      _ Q_ _                                       _ Q_ _
                           |
                     STUCK(2)                                          |
                                                                   _ _ Q _
                                                                   Q_ _  _
                                                                   _ _ _ Q
                                                                   _ Q_ _
                                                                   DONE!(3)
```

The neat thing about coding up backtracking, is that it can be done recursively, without having to do all the bookkeeping at once.

Instead, the stack or recursive calls does most of the bookkeeping (ie, keeping track of which queens we've placed, and which combinations we've tried so far, etc.)

Here is some code that is at the heart of the eight queens solution:

```
void solveItRec(int perm[], int location, struct onesquare
usedList[]) {

  int i;

  if (location == SIZE) {
     printSol(perm);
  }

  for (i=0; i<SIZE; i++) {

     if (usedList[i].selected == 0) {

        if (!conflict(perm, location, usedList[i].row)) {

             perm[location] = usedList[i].row;
             usedList[i].selected = 1;
             solveItRec(perm, location+1, usedList);
             usedList[i].selected = 0;
          }
       }
    }
}
```

Sudoku and Backtracking

Another common puzzle that can be solved by backtracking is a Sudoku puzzle. The basic idea behind the solution is as follows:

1) Scan the board to look for an empty square that could take on the fewest possible values based on the simple game constraints.
2) If you find a square that can only be one possible value, fill it in with that one value and continue the algorithm.
3) If no such square exists, place one of the possible numbers for that square in the number and repeat the process.
4) If you ever get stuck, erase the last number placed and see if there are other possible choices for that slot and try those next.

Mazes and Backtracking

A final example of something that can be solved using backtracking is a maze. From your start point, you will iterate through each possible starting move. From there, you recursively move forward. If you ever get stuck, the recursion takes you back to where you were, and you try the next possible move.

In dealing with a maze, to make sure you don't try too many possibilities, one should mark which locations in the maze have been visited already so that no location in the maze gets visited twice. (If a place has already been visited, there is no point in trying to reach the end of the maze from there again.

Tentaizu

In this problem, you are given a 7 x 7 board where there are exactly 10 hidden bombs. On the board, some squares have numbers on them, representing the number of adjacent bombs to that square, just like the numbers in the game, Minesweeper. The numbers are given in such a way that there is only one arrangements of 10 bombs that is consistent with the numbers given. The goal of the problem is to determine where all of the 10 bombs go.

Regular brute force would try up to $\binom{49}{10}$ combinations of placements of stars. This is far too many to run in a reasonable amount of time.

But…this is where backtracking comes in. Consider placing bombs in squares one by one, from top to bottom, going left to right in each row. A vast majority of possible combinations can be eliminated very early on due to inconsistencies!!!

Thus, our idea is as follows:

1) **Step through the board, in row major order (row by row from top to bottom, and from left to right in each row).**
2) **For each square, try placing a bomb in it and recursively see if this solution works.**
3) **If it doesn't, then try skipping the square and see if that solution works.**

Thus, our recursive function needs two critical pieces of information:

1) **Which square I am considering**
2) **The number of bombs already placed.**

The key to back-tracking is effectively cutting off “doomed” partial solutions as early as possible. In our recursion, this corresponds to base cases in the beginning – typically if statements to handle “easy” cases.

For our problem, it’s very easy to test consistency issues just with the number of bombs already placed and our current position.

Then, we can test more detailed consistency issues.

Finally, we move onto our recursive case(s).

For some squares, we can’t place a bomb (these are squares that already have a number or that are adjacent to a 0). For these squares, we make one recursive call only, corresponding to not placing a bomb in that squre.

For other squares we try two things:

1) **Place the bomb and see what the recursion returns.**
2) **If that didn’t work, undo that bomb and run the recursion again.**

In the code, the recursive function solveRec is structured very similar to the description given above.

Hexagram

In this problem, you are given a star design with 12 circles to fill with numbers in such a way that each of the six lines of four numbers adds up to the same value. You are given 12 distinct positive integers and asked to count the number of valid solutions. Two solutions are distinct if any corresponding circle contains two different values in the two solutions.

A regular brute force algorithm would check 12! different placements of numbers. Unfortunately, this is too many to run fast enough. Once again though, we see that if we try to build a partial solution, many of the possible placements are impossible if we use some of the given information.

First off, notice that the sum of each line can be calculated since each circle is part of exactly two lines. Thus, twice the sum of the twelve numbers is equal to the sum of each of the six of the lines. Thus, our desired line sum is twice the sum of the twelve numbers, divided by six, or more simply, the sum of the numbers divided by three.

From here, we just fill in the numbers in some specified order, trying each unused number in each square, and cutting out of our search if the filled in information is inconsistent. Consider the following trace by hand for the input case.

More Divisors

Warning: This is a difficult problem, so don't feel bad if you don't understand its solution. I wanted to include it because the solution uses backtracking and a fact taught in COT 3100, which all the students in this class have taken. Also, it highlights the use of a HashMap and shows how much smaller a search space can get utilizing just a few constraints.

The goal of the problem is very simply stated:

For any integer n, consider all positive integers less than or equal to it. Of all of those integers, count the number of divisors each one has. Find the number that has the maximum number of divisors. If multiple numbers have this same maximum number of divisors, just report the minimum of those numbers. You are asked to solve the problem for any n ≤ 10^{16}.

For example, 6 has four factors. The next integer that has more factors is 12, which has six factors. Thus, if the input was 6, 7, 8, 9, 10 or 11, our answer should be 6.

Notice that solving this by straight brute force (looping through all possible values less than or equal to n), is absolutely not a possibility, since 10^{16} is much, much larger than 10^8, which is roughly the most number of simple operations we can run in a couple seconds.

First, let's review the formula for calculating the number of factors of a number, given its prime factorization:

If the prime factorization of n = $\prod_{p_i \in Primes} p_i^{a_i}$, then the number of divisors of n is $\prod_{p_i \in Primes}(a_i + 1)$.

The key observation that is so helpful here is that the primes in the prime factorization don't affect the number of factors, only the exponents to those primes do. Thus, a list containing every positive exponent is all that is needed to calculate the number of divisors of an integer. Remember our goal is to minimize n, for any number of factors. Thus, given any list of exponents, it makes sense to raise 2 to the largest of these, 3 to the next largest, etc. since it's very obvious that $2^5 3^4 < 2^4 3^5$. (Basically, by lowering 2's exponent by 1 and increasing 3's exponent by one, I am dividing by 2 and multiplying by a larger number, 3, which will ultimately yield a larger number, one that is irrelevant to this problem.)

Thus, our key observation is that we only care about exponent sequences in reverse sorted order. Each unique sequence maps to a single integer n that matters for the sake of this problem. We can simply store all pairs number of factors with the minimal integer that has that number of factors by iterating through all possible exponent sequences.

One question hasn't been answered yet: How can backtracking limit our search of exponent sequences. If 60 is my largest exponent and I could have a sequence of 14 or so terms, that search space seems WAY TOO BIG!!!

Answer: A vast majority of sequences correspond to numbers bigger than 10^{16}. As we build a sequence, we can calculate how big the corresponding number is. We can stop building a sequence as soon as adding 1 to one of its exponents makes it too big!

So, quite literally, this is what we do. We store an array list with the "current sequence" under investigation, and a HashMap with our best factor, number pairs, updating it as necessary.