# Running time with brute force

When you use a brute force solution, you generally don't expect to change the order of the problem (except in a few cases). You want to reduce the time by a very large constant. Therefore, the normal big-O method of determining order does not go far enough, since it is designed to absorb the constant factor.

We can, however, make crude estimates. Let's assume the number of lines of code that will be executed will be proportional to the time a program takes to run. Generally, a computer nowadays can be expected to execute somewhere about 100 million lines of code per second.

Say your algorithm is $n^2$. If you have input up to 10 000, you would expect it to run in a few seconds, fast enough to pass the judge. 100 000 would take a few minutes to run.

If your algorithm was $2^n$, on the other hand, you would be able to go to 27. Going farther (say to 50) would take years instead of minutes. If your algorithm was n! you could go to 11.

This isn't the most rigorous check, but it is enough to tell you what to expect.

# Branch and bound

Branch and bound speeds up backtracking by checking the state as it branches, and choosing not to branch to dead-end paths.

Example: N Queens

Problem: place n queens on an n by n chess board. Place them so that no two queens can attack each other. Since queens can use in any of the eight compass directions, no two queens can share a row, column, or diagonal.

How do we apply branch and bound? To construct a branch and bound solution, we need to be able to form the solution one step at a time. Since there can be only one queen per row or column (and there must be one queen) we can step through a column at a time.

By recursively going through the columns, our code will look like this:

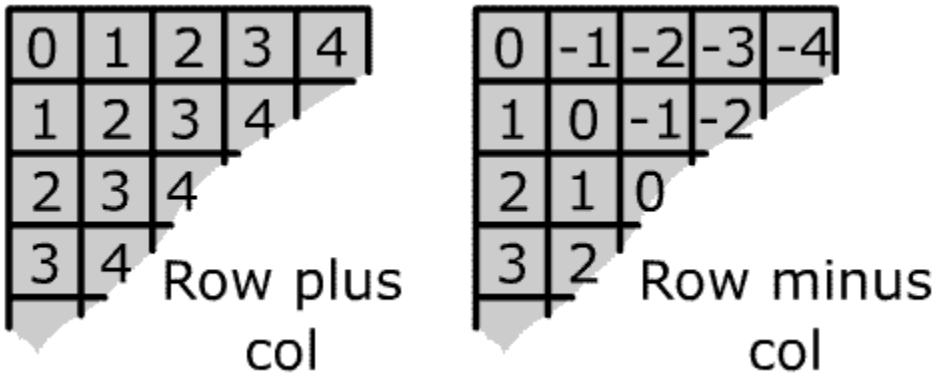
- If all queens have been placed, return success
- If all forward paths(available squares) are ruled out, return false
- Recurse on each potential path until one returns success

This is the general principle for branch and bound – check terminating condition, otherwise continue along any unbounded paths. In this case we will keep track of which paths are bounded by saving which squares are attacked by queens we have placed already.

Rather than constructing a grid, we can contain all the information we need with the following variables:

- row[] - each element of row contains a true/false state telling whether that row is occupied.

- rpc[] -Row Plus Column – this is an easy way of remembering which diagonals are occupied. The next image explains how this works.



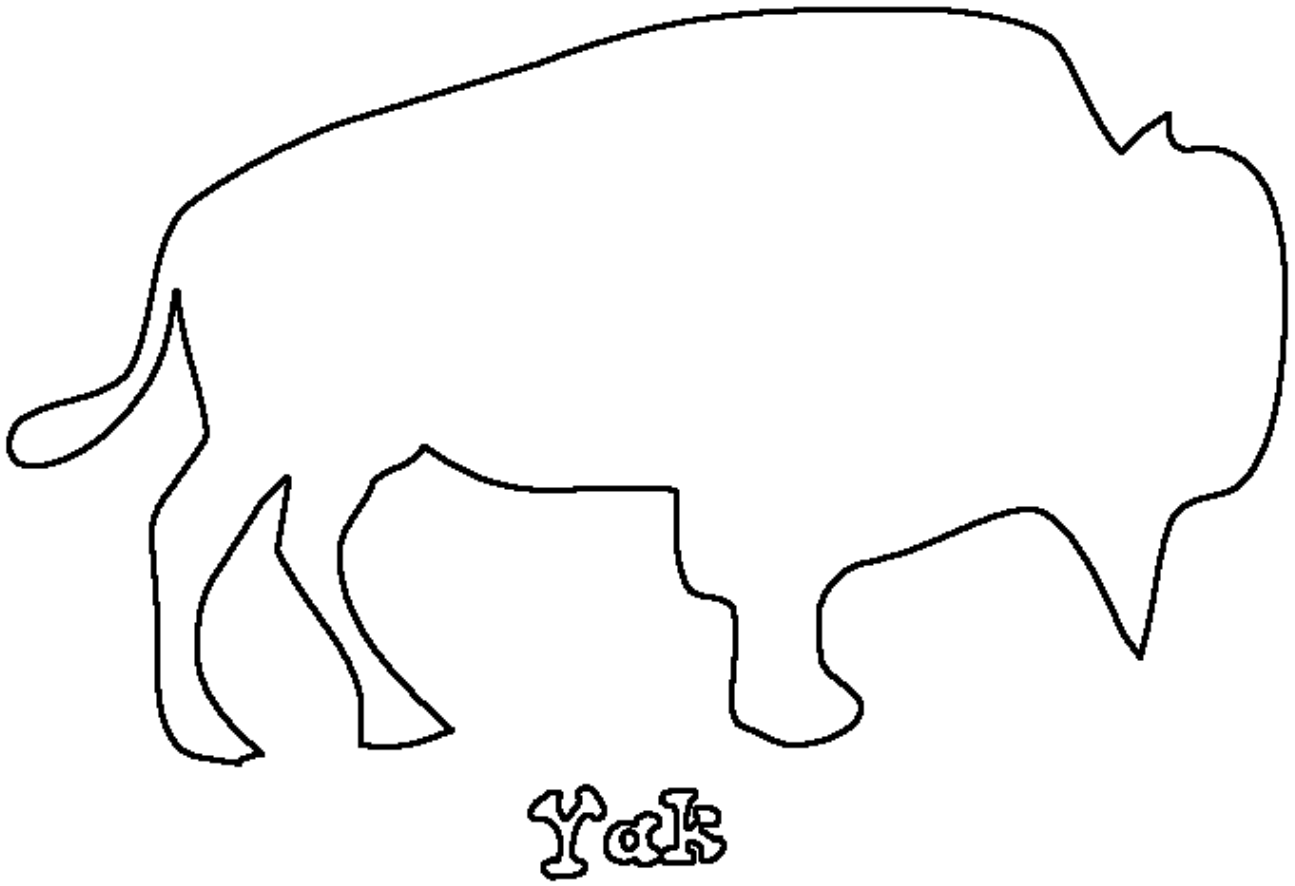

- rmc[] - Row Minus Column – this holds the other diagonal. In the program I use this was done by using element [row-col+n] where n is the number of squares, added to the index to avoid negative elements.

We won't need a column variable, because the code will place only one queen per column.

These elements can be used to tell whether a square should be branched or bounded. If the square in question is still available in all three arrays, then the program should fill those spots and branch branch onwards. Later, that branch will finish. If it returns a success, we have placed all the queens and should also return success. If it brings failure, the function should remove that queen from row, rpc, and rmc and continue to loop through the other possibilities in the same manner. Here is the code for this procedure:

```
int placequeen( int col ){
  if( col == n )
    return 1;
  for( int i=0; i<n; i++ )
    if( row[i] == 0 && rpc[col+i] == 0 && rmc[col-i+n] == 0){
      row[i] = 1;
      rpc[col+i] = 1;
      rmc[col-i+n] = 1;
      queens[col] = i;
      if( placequeen( col + 1 ) == 1 )
          return 1;
      else{
          row[i] = 0;
          rpc[col+i] = 0;
          rmc[col-i+n] = 0;
      }
    }
  return 0;
}
```

Yak coloring page



# Other topics in brute force

## Iterative deepening

Iterative deepening is a way of searching for a solution that limits the search depth. The depth is increased (iteratively!) until a solution is reached. This is useful for searching for solutions in large trees, where the size becomes unmanageable at deeper levels, and where the depth of the solution is unknown. One example of a useful application is searching for chess moves.

To demonstrate, let's consider addition chains. An addition chain is a sequence of positive integers with the property that each integer 1...n can be expressed as a sum of two other integers previous to it in the chain. For example,

( 1, 2, 3, 5, 7 )

is an addition chain for 7. Note that to make two, one was used twice. All addition chains start with one. As well, chains for a certain number are not unique. ( 1, 2, 4, 5, 7 ) and ( 1, 2, 4, 6, 7 ) are two alternate chains for seven (there are even more).

How would we go about finding a shortest chain for a number? Imagine how DFS would fare with backtracking. For large numbers, there may be many paths that lead to the number that are not the shortest path. DFS will not be able to stop after finding the first answer. Furthermore, DFS will not know what the length of the shortest path is, so it will end up checking paths longer than the shorter path. Since each node has a high degree of branching, most of the possibilities backtracking checks will be too long. This is not a good approach.

Now let's try BFS. When we reach a number using BFS we know there is no shorter chain, since we have already tried all of those possibilities. However, BFS has a fatal flaw here: memory. Since the number of connections to each node escalates, the amount of memory required to enqueue all of the nodes under examination will skyrocket.

Now we are in a quandary. DFS uses very little memory, but plunges to the depths of the tree and becomes lost. BFS works great, but chokes on all the data. Iterative deepening can solve this.

Instead of letting DFS go free, we add a parameter to each call. This number is decreased by one each successive call, and when it reaches 0, DFS returns regardless of finding a solution. This harnesses the space efficiency of DFS and the completeness and optimality of BFS.

While using this, we search first for addition chains that are length 1 (easy to do!). Then we check for the ones of length up to 2, then 3, etc until we find one. This path must be the shortest path possible.

The penalty for this seems quite large – we are throwing away many partial solutions. But, this is actually OK, since these require a fractional effort to produce.

Sources:

Wikipedia: http://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search
Picture: http://www.fun-with-pictures.com/yak-coloring-page.html
Last year's notes : )