7/20/2020 - Backtracking

Monday, July 6, 2020 11:27 AM

Backtracking is "smart" brute force. Where we still "try everything", but we skip trying over partial solutions that are "doomed to fail"

Will show you pre-coded work:

1. Eight Queens
2. Digit Divisibility (past Foundation Exam question)

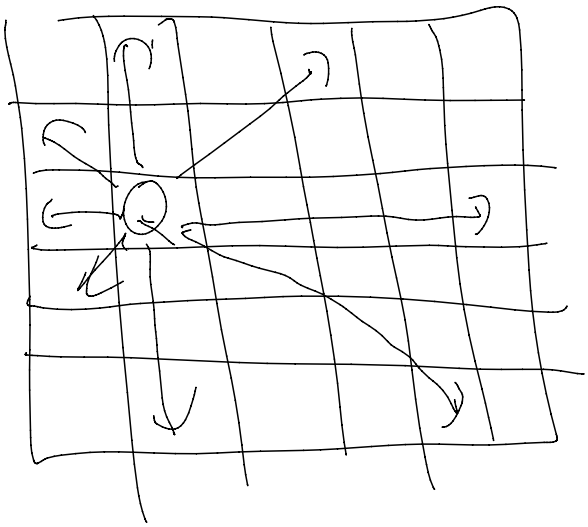
Will code live:

1. Sudoku Solver

Eight Queens

General Version is the n Queens problem.
Figure out how to place 8 queens on an n by n chessboard so that no two queens are attacking one another.

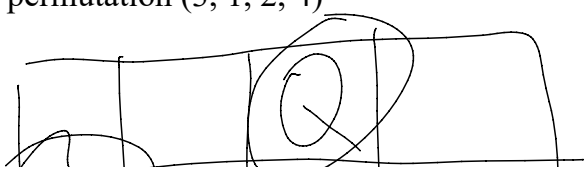
Queens can go as far as they want on their row, column or diagonal(s).

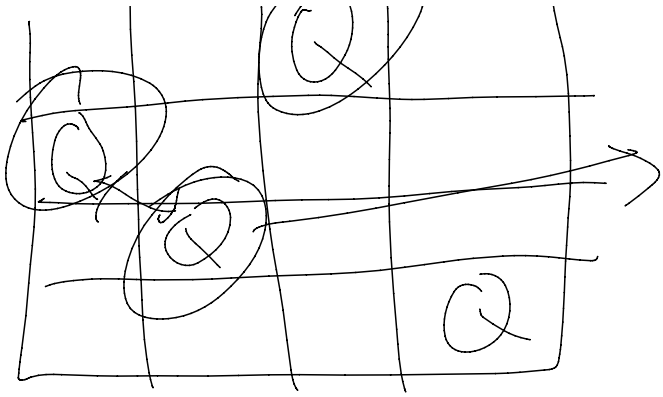


At most we can have one queen per row. And also at most one queen per column. So what we can do is place our queens one by one, on row 1, then row 2, etc. And each queen must be placed on a unique column, otherwise there would be two queens on the same column. So, all the possible ways we would try would be permutations of 1, 2, 3, …, n

1234

Consider n = 4, permutation (3, 1, 2, 4)





Each permutation corresponds to a potential placement of queens.
Brute force solution: Try each permutation. Place the queens. See if any pair attacks each other.

Run time $O(n! * n^2)$

Can we do better?

With real backtracking, it's hard to pin down an exact run time…

No point in placing down the third queen in row 3, column 2, because that is a path that is doomed to fail. There is no reason to continue with any permutation that starts 3, 1, 2…

Imagine instead that n = 10, and we have placed down the permutation 3,1,2. There will be 7! permutations (7 = 10 - 3) that are all not going to be valid solutions because the second and third queens attack each other.

Idea behind backtracking: Well, skip all of these that are doomed to fail, Specifically, as we are building all the permutations, if a prefix of a permutation CAN NOT lead to a valid solution, SKIP IT!!!

Normal permutation code:

```
for (int i=0; i<n; i++) {
   if (!used[i]) {
      perm[k] = i;
      used[i] = 1;
      // do recursion
      used[i] = 0;
  }
}
```

Backtracking idea is to change permutation code as follows:

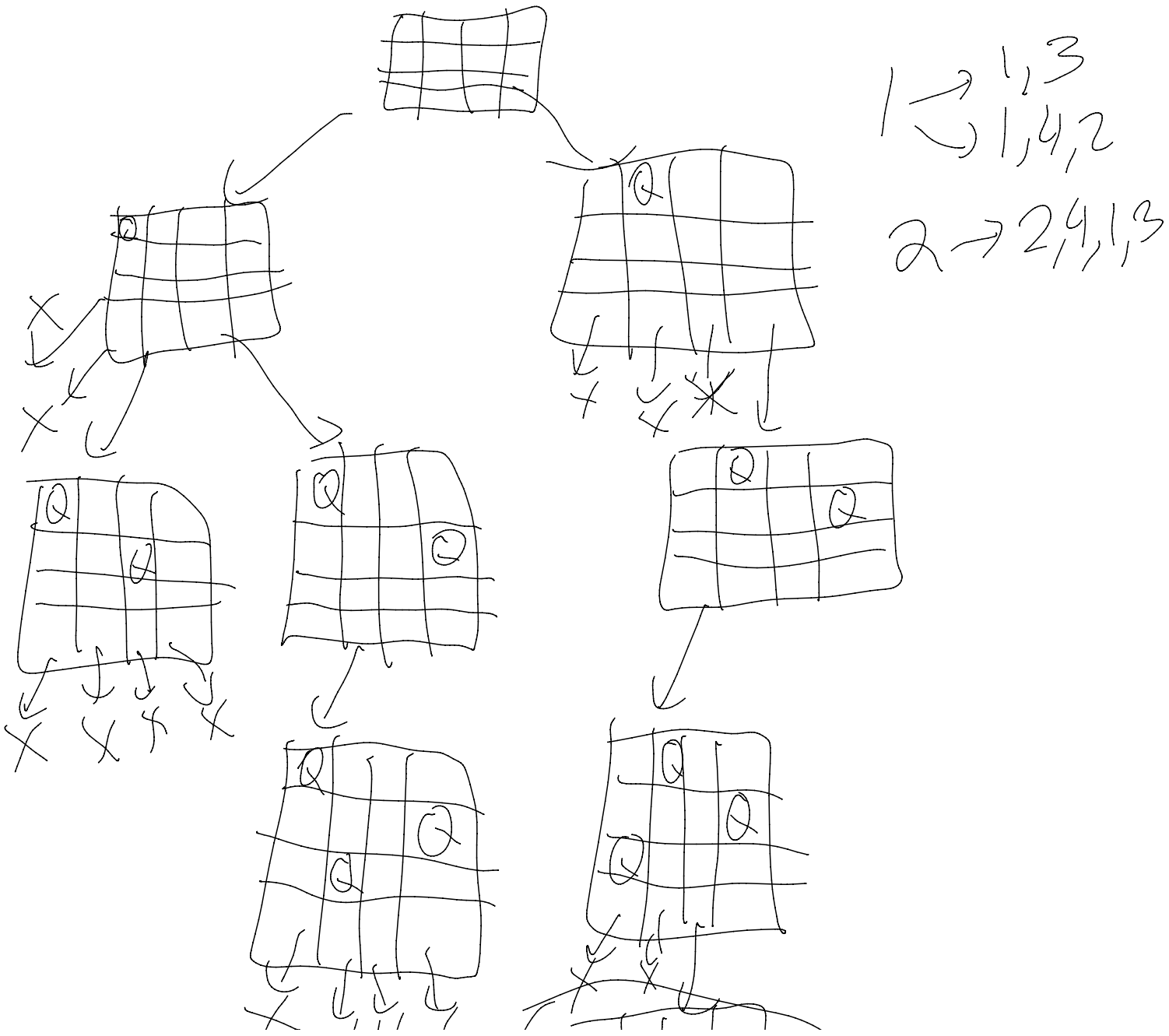
```
for (int i=0; i<n; i++) {
   if (!used[i] && SOME VIABILITY CONDITION) {
      perm[k] = i;
      used[i] = 1;
      // do recursion
```

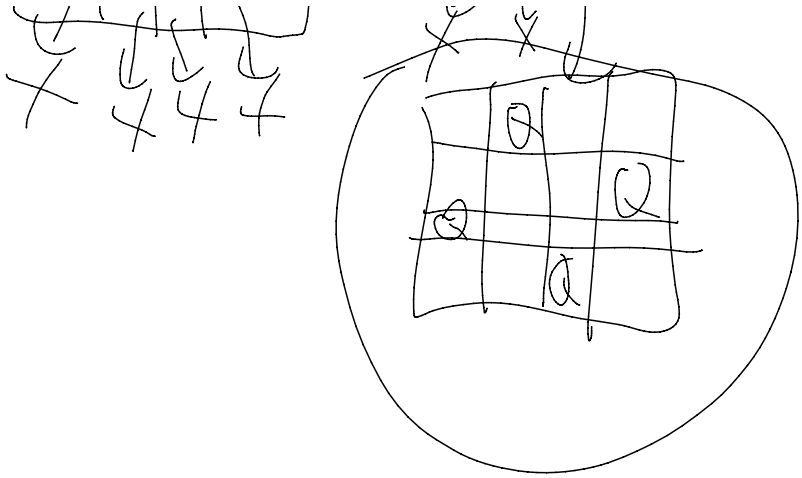
```
        used[i] = 0;
    }
}
```

Or we can code it like this:

```
for (int i=0; i<n; i++) {
    if (used[i] || DOOMED TO FAIL) continue;
    perm[k] = i;
    used[i] = 1;
    // do recursion
    used[i] = 0;
}
```

Show n queens backtrack on n = 4:





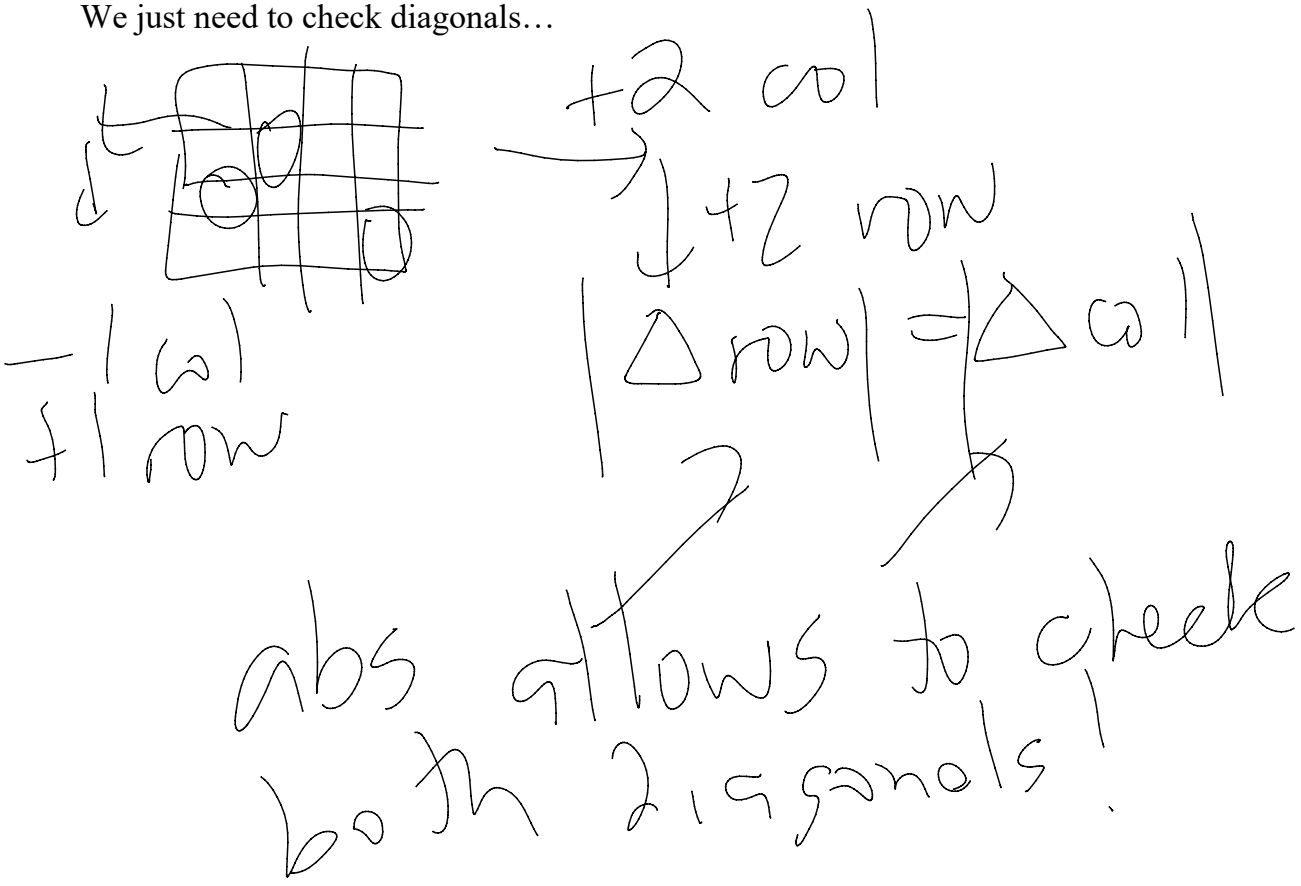
How do I express "doomed to fail"?

We'll just store a permutation of 0 to n-1, as we are building it:

Consider perm	2	0	1

This means my queens are at the ordered pairs (0, 2), (0, 1), (1, 2).

We just need to check diagonals…



Consider perm	2	0	1

This means my queens are at the ordered pairs (0, 2), (0, 1), (1, 2).

So to check a single pair perm[i] and perm[j], we want the absolute value in the change of rows, which is abs(i - j), and also the absolute value in the change of columns, which is abs(perm[i]-perm[j]).

If these two things are equal, then the two corresponding queens attack each other.

So, if I am thinking about placing a queen on row I, I must check it will all other queens before me, in rows 0, through i-1.

```
int conflict(int perm, int curRow, int curCol) {

   for (int i=0; i<curRow; i++) {

        // Check queen on row I, column perm[i], with potential queen
        // at (curRow, curCol).
        if (abs(curRow-i) == abs(perm[i]-curCol))
            return 1;
   }

   // If we get here, no conflict.
   return 0;
}
```

Digit Divisibility

We call an integer k-digit divisible if for each prefix (length 1, length 2, etc. The prefix is divisible by its length.)

For example,

54325, is 5-digit divisible because

5 is divisible by 1
54 is divisible by 2
543 is divisible by 3
5432 is divisible by 4
54325 is divisible by 5

Goal: to print out ALL k-digit divisible integers up to k=18.

long long goes up to 8×10^{18}.

Full brute force would take a really long time, trying each integer up to 10^{18}. But, a vast majority of these are doomed to fail…

1, the second digit can't be 1, 3, 5, 7 or 9

12, the third digit can ONLY be 0, 3, 6 or 9

We are coding the odometer, but skipping digits that would not make that prefix divisible by the proper value.

We need to be able to "build" our value, and add onto it.

5432 stored in a long long, I need to to see of all the 10 digits, which ones, when added to the end, will create a value divisible by 5.

long long cur = 5432.

```
for (int next=0; next<10; next++) {
   int newvalue = 10*cur + next;
   if (newvalue%(k+1) == 0) { // slot k means using k+1 digits
      // do recursion in here…

   }

}
```

Function must take in current number and k, and a stopping point.

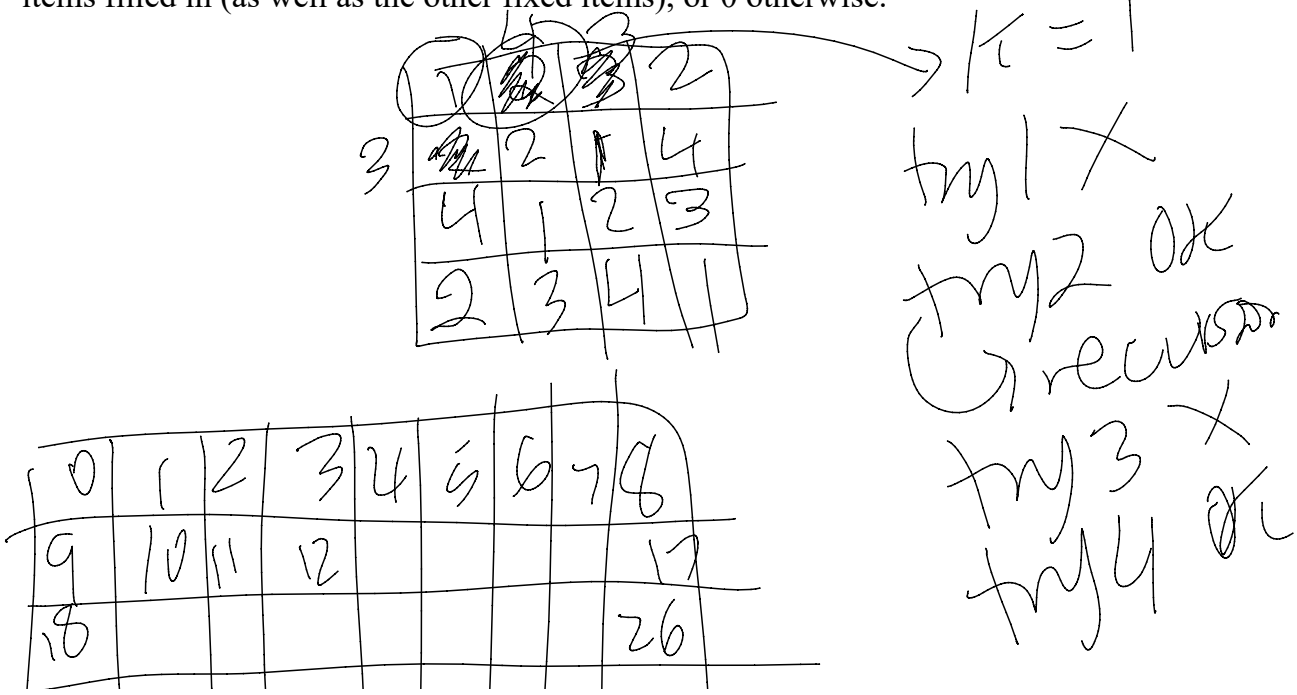
Sudoku Solver

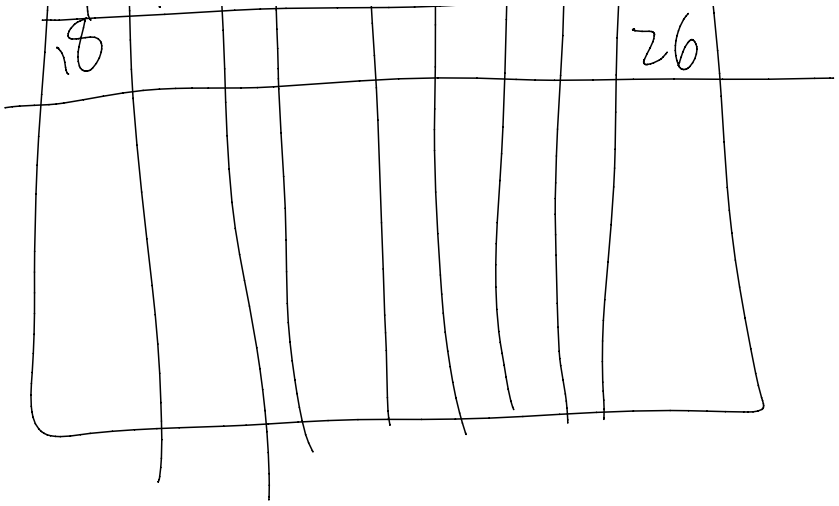
We are filling in a board of 81 squares…I like coding with a 1D array, but then we can print it as two d.

Goal: fill in an array like perm (call it board), of size 81, with the appropriate values 1 through 9…

int solve(int* board, int k)

This code will return 1 if there is a solution to board with the first k items filled in (as well as the other fixed items), or 0 otherwise.





→ +1 screen

↓ +9 next row