CS/COE 1501

cs.pitt.edu/~bill/1501/

Brute-force Search
Brute-force (or exhaustive) search

• Find the solution to a problem by considering all potential solutions and selecting the correct one

• Run-time is bounded by the number of potential solutions
  • $n^3$ potential solutions means cubic run-time
  • $2^n$ potential solutions means exponential run-time
Password cracking

- Brute force password attacks depend on the length of the password, hence the insecurity of short passwords
- We can view the series of guesses we make as a tree
  - Each path from root to leaf is an attempted solution
PIN cracking example
• This tree will enumerate $10^n$ different PINs
  • $n$ is the length of the PIN
  • So for our case $10^4 = 10,000$ different PINs
• Note that this is (for a computer) tiny
  • What would be a long password for a computer?
  • Say 128 bits long
    • $2^n$ different passwords
    • $2^{128} = 340282366920938463463374607431768211456$
      • Assuming a supercomputer can check $33,860,000,000,000$ passwords per second...
      • And we’ll on average find the correct password after guessing half the possibilities...
      • We should be able to crack a 128 bit password on our supercomputer in $1.59 \times 10^{17}$ years using brute force
Back to our PIN cracking example

What if we have background knowledge that the PIN we’re trying to crack doesn’t have more than one 0?
• Removes entire subtrees of our search space exploration

• When we can use it, it makes our algorithm practical for much larger values of $n$

• Does not, however, affect the asymptotic performance of an algorithm
  • Still exponential time requirement for our PIN example
How to enumerate all these possibilities?

- For the PIN example a whole bunch of for loops would do
- In general, exhaustive search trees can be easily traversed via recursion and *backtracking*
  - Recurse until its apparent no solution can be achieved along the current path
  - Undo the path to the point that you can start to move forward again
Place 8 queens on a chessboard such that no queen can take another

- Queens can move horizontally, vertically, and diagonally
- How many ways can you place 8 pieces on a chessboard?
  - $\binom{64}{8}$
  - $= \frac{64!}{8! \cdot (64-8)!}$
  - $= 4,426,165,368$
  - Meaning 35,409,322,944 total queen placements

Do we really need to look through all of these options?
8 queens problem

- Solutions only have one queen per column
  - Still $8^8 = 16,777,216$ possible combinations
- Solutions only have one queen per row
  - Combining these two observations, only $8! = 40,320$
  - Looking quite feasible!
- Finally, prune subtrees with queens on the same diagonal
• Basic idea:
  • Recurse over columns of the board
  • Each recursive call iterates through the rows of the board
  • Check rows/diagonals
    • Are they currently safe?
      • Place a queen in the current row/col
      • If you are at the end of the board, you've found a solution!
      • Otherwise, try recursive call for the next column
    • If they are not currently safe
      • Continue to the next row in the current recursive call
8 Queens

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Another problem: Boggle

- Words at least 3 adjacent letters long must be assembled from a 4x4 grid
- Adjacent letters are horizontally, vertically, or diagonally neighboring
- Any cube in the grid can only be used once per word
Recurring through Boggle letters

- Have 8 different options from each cube
  - From B[i][j]:
    - B[i-1][j-1]
    - B[i-1][j]
    - B[i-1][j+1]
    - B[i][j-1]
    - B[i][j+1]
    - B[i+1][j-1]
    - B[i+1][j]
    - B[i+1][j+1]
  - Naively, the runtime here would be 16!
    - = 20,922,789,888,000
Where do we prune?
Constructing the words over the course of recursion will mean building up and tearing down strings
  • Moving forward adds a new character to the current word string
  • Backtracking removes the most recent character
  • Basically pushing/popping to/from a string stack
• Push/Pop stack operations are generally $\Theta(1)$
  • Unless you need to resize, but that cost can be amortized
• Java Strings, however, are immutable
  • s = new String(“Here is a basic string”);
  • s = s + “ this operation allocates and initializes all over again”;
  • Becomes essentially a $\Theta(n)$ operation
    • Where $n$ is the length of the string
append() and deleteCharAt() can be used to push and pop

• Back to $\Theta(1)$!
  • Still need to account for resizing, though...

• StringBuffer can also be used for this purpose
  • Differences?