Thinking Recursively
Part V
A Little Word Puzzle
“What nine-letter word can be reduced to a single-letter word one letter at a time by removing letters, leaving it a legal word at each step?”
One Solution

STARTLING
One Solution

STARTING
One Solution

STARING
One Solution

STRING
One Solution

STING
One Solution

S I N G
One Solution

S I N
One Solution
One Solution
Shrinkable Words

• Let's define a *shrinkable word* as a word that can be reduced down to one letter by removing one character at a time, leaving a word at each step.

• **Base Cases:**
  • A string that is not a word is not a shrinkable word.
  • Any single-letter word is shrinkable (A, I, and O).

• **Recursive Step:**
  • A multi-letter word is shrinkable if you can remove a letter to form a shrinkable word.
  • A multi-letter word is not shrinkable if no matter what letter you remove, it’s not shrinkable.
Finding a Good Shrink

CART

ART

CRT

CAT

CAR

RT

AT

AR

RT

CT

CR

AT

CT

CA

AR

CR

CA

TR

TA

AR

TR

TC

RC

TA

TC

CAC

RA

RC

AC
Recursive Backtracking

- This code is an example of **recursive backtracking**.
- At each step, we try one of many possible options.
- If *any* option succeeds, that's great! We're done.
- If *none* of the options succeed, then this particular problem can't be solved.
bool isShrinkable(const string& word, const Lexicon& english) {
    if (!english.contains(word)) return false;
    if (word.length() == 1) return true;

    for (int i = 0; i < word.length(); i++) {
        string shrunken = word.substr(0, i) + word.substr(i + 1);
        if (isShrinkable(shrunken, english)) {
            return true;
        }
    }

    return false;
}
bool isShrinkable(const string& word, const Lexicon& english) {
    if (!english.contains(word)) return false;
    if (word.length() == 1) return true;

    for (int i = 0; i < word.length(); i++) {
        string shrunken = word.substr(0, i) + word.substr(i + 1);
        if (isShrinkable(shrunken, english)) {
            return true;
        }
    }
    return false;
}


```cpp
bool isShrinkable(const string& word, const Lexicon& english) {
    if (!english.contains(word)) return false;
    if (word.length() == 1) return true;

    for (int i = 0; i < word.length(); i++) {
        string shrunken = word.substr(0, i) + word.substr(i + 1);
        return isShrinkable(shrunken, english); // △ Bad Idea △
    }

    return false;
}
```
bool isShrinkable(const string& word, const Lexicon& english) {
    if (!english.contains(word)) return false;
    if (word.length() == 1) return true;

    for (int i = 0; i < word.length(); i++) {
        string shrunken = word.substr(0, i) + word.substr(i + 1);
        return isShrinkable(shrunken, english); // 🔴 Bad Idea 🔴
    }
    return false;
}
Tenacity is a Virtue
When backtracking recursively, *don’t give up if your first try fails!*

Hold out hope that something else will work out. It very well might!
Recursive Backtracking

if (problem is sufficiently simple) {
  return whether or not the problem is solvable
} else {
  for (each choice) {
    try out that choice
    if (that choice leads to success) {
      return success;
    }
  }
  return failure;
}

Note that if the recursive call succeeds, then we return success. If it doesn’t succeed, that doesn’t mean we’ve failed – it just means we need to try out the next option.
Extracting a Solution

• We now have a list of words that allegedly are shrinkable, but we don't actually know how to shrink them!

• Can the function tell us how to shrink the word?
Output Parameters

• An *output parameter* (or *outparam*) is a parameter to a function that stores the result of that function.

• Caller passes the parameter by reference, function overwrites the value.

• Often used with recursive backtracking:
  • The return value says whether a solution exists.
  • If one does, it’s loaded into the outparameter.
Generating the Answer

CART

ART
  RT
  AT
  AR
CRT
  RT
  CT
  CR
CAT
  AT
  CT
  CA
CAR
  AR
  CR
  CA
Generating the Answer
Generating the Answer

CART

ART

CRT

CAT

CAR

RT AT AR

RT CT CR

AT CT CA

AR CR CA
Generating the Answer

CART

ART

CRT

CAT

CAR

RT

AT

AR

RT

CT

CR

AT

CT

CA

AR

CR

CA

T

R

T

A

R

A

T

R

T

C

R

C

T

A

T

C

A

C

R

A

R

C

A

C
Generating the Answer

CART

ART

CRT

CAT

CAR
Generating the Answer

CART

ART

RT

AT

AR

CRT

RT

CT

CR

CAT

AT

CT

CA

CAR

AR

CR

CA
Generating the Answer

CART

ART

RT AT AR RT CT CR

CRT

AT CT CA

CAT

AR CR CA

CAR
Generating the Answer
Generating the Answer

CART

ART

CRT

CAT

CAR

RT

AT

AR

RT

CT

CR

AT

CT

CA

AR

CR

CA
Generating the Answer
Generating the Answer
Generating the Answer

CART

ART

CRT

CAT

CAR
Generating the Answer

CART

ART

CRT

CAT

CAR

RT AT AR

RT CT CR

T A R A T R T C R C

T A T C A C R A R C A C
Generating the Answer

CART

ART

CRT

CAT

CAR
Generating the Answer
Generating the Answer
Generating the Answer
Generating the Answer
Generating the Answer

Question to ponder: How would you update the function so that it generates the sequence in reverse order?
Dense Crosswords
aahs
abet
heme
stem
Generating Dense Crosswords
Generating Dense Crosswords
Generating Dense Crosswords

A A H E D
A A H E D
A A H E D
Generating Dense Crosswords
Generating Dense Crosswords
Generating Dense Crosswords

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>A</th>
<th>H</th>
<th>E</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>H</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>H</td>
<td>E</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>H</td>
<td>E</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>H</td>
<td>E</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>H</td>
<td>E</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>H</td>
<td>E</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>
Generating Dense Crosswords
Generating Dense Crosswords

<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
<th>H</th>
<th>E</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>H</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>H</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>H</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>L</td>
<td>I</td>
</tr>
</tbody>
</table>
Generating Dense Crosswords
Generating Dense Crosswords
Generating Dense Crosswords
Generating Dense Crosswords

- **Idea:** Solve the problem “is there a way to extend this partial crossword into a full one?”

- **Base Case:**
  - If the crossword is already filled in, then we just check whether it’s legal.

- **Recursive Step:**
  - For each possible word that can go in the current row, try extending the crossword with that word.
  - If the remainder can be extended to a full crossword, we’re done!
  - If no matter what word we put in that row, we can’t complete the crossword, there’s no way to extend what we have.
Generating Dense Crosswords
Generating Dense Crosswords
Generating Dense Crosswords
Generating Dense Crosswords

A A H E D
A A H E D
A A H E D
Generating Dense Crosswords
Generating Dense Crosswords
Generating Dense Crosswords
Generating Dense Crosswords
Generating Dense Crosswords
Generating Dense Crosswords

A A H E D
A B A C A
A A H E D
Generating Dense Crosswords
Generating Dense Crosswords
Generating Dense Crosswords

A A H E D
A B A C A
A A L I I
Generating Dense Crosswords
Generating Dense Crosswords
Generating Dense Crosswords

• **Idea:** Solve the problem “is there a way to extend this partial crossword into a full one?”

• **Base Case:**
  • If the crossword is already filled in, then we just check whether it’s legal.
  • If any column contains a string that isn’t a prefix of any English word, report a failure without checking anything else.

• **Recursive Step:**
  • For each possible word that can go in the current row, try extending the crossword with that word.
  • If the remainder can be extended to a full crossword, we’re done!
  • If no matter what word we put in that row, we can’t complete the crossword, there’s no way to extend what we have.
Closing Thoughts on Recursion
You now know how to use recursion to view problems from a different perspective that can lead to short and elegant solutions.
You’ve seen how to use recursion to enumerate all objects of some type, which you can use to find the optimal solution to a problem.
You’ve seen how to use recursive backtracking to determine whether something is possible and, if so to find some way to do it.
You’ve seen that optimizing code is more about changing strategy than writing less code.
Next Time

- Algorithmic Analysis
  - How do we formally analyze the complexity of a piece of code?