

# Recursive Backtracking I

---

## Recursive Backtracking

So far, all of the recursive algorithms we have seen have shared one very important property: each time a problem was recursively decomposed into a simpler instance of the same problem, only **one** such decomposition was possible; consequently, the algorithm was guaranteed to produce a solution to the original problem. Today we will begin to examine problems with several possible decompositions from one instance of the problem to another. That is, each time we make a recursive call, we will have to make a **choice** as to which decomposition to use. If we choose the wrong one, we will eventually run into a dead end and find ourselves in a state where we are unable to solve the problem and unable to recur any further; when this happens, we will have to **backtrack** to a "choice point" and try another alternative.

If we ever solve the problem, great—we're done. Otherwise, we need to keep exploring all possible paths by making choices and, when they prove to have been wrong, backtracking to the most recent choice point. What's really interesting about backtracking is that we only back up in the recursion as far as we need to go to reach a previously unexplored opportunity. Eventually, more and more of these options will have been explored, and we will backtrack further and further. If we happen to backtrack to our initial position and find we've nothing else to explore, the particular problem at hand is unsolvable.

Continue reading Chapters 7 and 8, and read Sections 1 and 2 of Chapter 9 (and skim the rest of the chapter if you'd like). We'll spend today finishing up Handout 14's examples, and then advance on the recursive backtracking—the subject of Chapter 9.

## Shrinking A Word [courtesy of Julie Zelenski]

Consider the simply stated question: Is it possible to take an English word and remove its letters in some order such that every string along the way is also English?

Sometimes it's possible. For example, we can shrink the word "**smart**" down to the empty string while obeying the restriction that all of the intervening strings are also legitimate words. Check this out:

```

"smart"
"mart"
"art"
"at"
"a"
" "

```

We elect to first remove the **s**, then the **m**, then the **r**, then the **t**, and finally the **a**. Note that every single string in the above diagram is an English word: That means that, for the purposes of the this problem, it's possible to shrink the word "**smart**" down to nothingness.

Not surprisingly, there are some perfectly good English words that can't be shrunk at all: "**zephyr**", "**lymph**", "**rope**", "**father**". A reasonable question to ask at this point: Can we programmatically figure out which words can be shrunk and which ones can't be?

The answer, not surprisingly, is yes, and our answer involves a form of recursion we've not formally seen before.

All prior recursive examples have been fully exhaustive in that every single recursive call that could be made was indeed made and contributed to the overall answer. In this example, we only commit to some of the recursive calls if a previous one failed to provide an answer. Check out the code for our **canShrink** function, which returns **true** if and only if it's possible to shrink the provided word down to the empty string:

```

static bool canShrink(const string& str, const Lexicon& english) {
    if (str.empty()) return true;
    if (!english.contains(str)) return false;

    for (int i = 0; i < str.size(); i++) {
        string subsequence = str.substr(0, i) + str.substr(i + 1);
        if (canShrink(subsequence, english)) {
            return true;
        }
    }

    return false;
}

```

The code includes two base bases—that part isn't new. But look at the **for** loop and how it surrounds the recursive call. Each iteration considers the incoming string with some character removed, and then looks to see if that new string is also a word and can be shrunk down to nothingness. Look at this **for** loop as a series of opportunities to return **true**. If it just so happens to find some path to the empty string sooner than later, it returns

true without making any other **canShrink** calls. Only if every single path turns up nothing do we truly give up and return **false**.

It's pretty clear why **true** and **false** get returned, since the initial call wants a yes or no as to whether or not the provided word can be made to disappear. But what may not be immediately clear is that the recursive calls also rely on those return values to figure out whether or not the path it chose to recursively explore was a good one.

The above version correctly returns **true** or **false**, but it doesn't remember what words lead down to the empty string. This second version uses a **Stack<string>** to take a snapshot of all of the strings that led down the empty string, and it builds this **Stack<string>** as the cascade of return **true** statements prompt the recursion to unwind. Here's the new and improved version:

```
static bool canShrink(const string& str, const Lexicon& english,
Stack<string>& path) {
    if (str.empty()) {
        path.push("");
        return true;
    }

    if (!english.contains(str)) return false;

    for (int i = 0; i < str.size(); i++) {
        string subsequence = str.substr(0, i) + str.substr(i + 1);
        if (canShrink(subsequence, english, path)) {
            path.push(str);
            return true;
        }
    }

    return false;
}
```

If the initial call to **canShrink** returns **true**, then we know that the **Stack<string>** contains the bottom-up accumulation of all of the strings that led to the empty string.

Here's a short program that helps exercise the second version of **canShrink**:

```
static void printShrinkPath(const string& str, Stack<string>& path) {
    cout << endl;
    while (!path.isEmpty()) {
        cout << "\t\" << path.pop() << "\" << endl;
    }
    cout << endl;
}
```

```

int main() {
    Lexicon english("EnglishWords.dat");
    while (true) {
        string word = getLine("Enter a word: ");
        if (word.empty()) break;
        Stack<string> path;
        if (canShrink(word, english, path)) {
            cout << "Yes! Here's the path: " << endl;
            printShrinkPath(word, path);
        } else {
            cout << "That word can't be shrunk down." << endl;
        }
    }
    return 0;
}

```

Just in case you're on the edge of your seat, the longest shrinkable English word is **complecting**:

```

"complecting"
"completing"
"competing"
"compting"
"comping"
"coping"
"oping"
"ping"
"pig"
"pi"
"i"
" "

```

### Periodic Table as Alphabet: Take II

In a previous example, we presented code to list all of those English words that can be spelled out using the symbols of the periodic table and nothing else. Here's a related problem that asks specifically whether or not the provided word can be spelled out using just the periodic table symbols. We'll assume that all of the symbols are 1, 2, or 3 letters long, that the symbols come via a **Set<string>**, and that all of the **Lexicon** methods we're familiar with are case-insensitive.

The idea is to see if the first 1, 2, or 3 letters match some symbol, and if so, to recur on all of the remaining length - 1, length - 2, or length - 3 letters to see if they can also be subdivided into periodic table elements. Here's the solution I came up with:

```

static string elementize(string str) {
    if (!str.empty()) {
        str = toLowerCase(str);    // -> "he"
        str[0] = toupper(str[0]);  // -> "He"
    }

    return str;
}

static bool canSpell(string word, Set<string>& symbols) {
    if (word.empty()) return true;

    int length = word.size();
    for (int i = 1; i <= min(3, length); i++) {
        if (symbols.contains(elementize(word.substr(0, i))) &&
            canSpell(word.substr(i), symbols)) {
            return true;
        }
    }

    return false;
}

```

If we want visual proof the word can be spelled, then we can accumulate the relevant symbols in a **Stack** as the successful search unwinds, and then print the serialization of the **Stack** from the call site.

```

static bool canSpell(string word, Set<string>& symbols, Stack<string>& footprint) {
    if (word.empty()) return true;

    int length = word.size();
    for (int i = 1; i <= min(3, length); i++) {
        string symbol = elementize(word.substr(0, i));
        if (symbols.contains(symbol) &&
            canSpell(word.substr(i), symbols, footprint)) {
            footprint.push(symbol);
            return true;
        }
    }

    return false;
}

```

Here's a **main** function that exercises the second version of **canSpell**, and illustrates how the **footprint** can be drained and printed to standard out, knowing that the last symbol used to spell the word is buried at the bottom, and the first symbol used is at the top:

```

int main() {
    Set<string> symbols;
    addPeriodicTableElements(symbols); // assume this just works as implied
    while (true) {
        string word = getLine("Enter a word: ");
        if (word.empty()) break;
        Stack<string> footprint;
        if (canSpell(word, symbols, footprint)) {
            cout << "That can be spelled as \"";
            while (!footprint.isEmpty()) {
                cout << footprint.pop();
            }
            cout << "\"." << endl;
        } else {
            cout << "That's just not possible." << endl;
        }
    }

    return 0;
}

```

Finally, here's a test run of the above program to illustrate the output is as expected (or at least believable). Note that

```

Enter a word: hen
That can be spelled as "HeN".
Enter a word: foolishness
That can be spelled as "FOOLiSHNEsS".
Enter a word: partial
That can be spelled as "PARTiAl".
Enter a word: hooligan
That can be spelled as "HOOLiGaN".
Enter a word: hooliganism
That can be spelled as "HOOLiGaNiSm".
Enter a word: indefatigable
That's just not possible.
Enter a word: antidisestablishmentarianism
That's just not possible.
Enter a word:

```

### Cubic Decomposition

All perfect cubes—save for a relatively small number of them—can be expressed as a sum of three or more **distinct, smaller** cubes. Just trust me on it as you check out these examples:

$$\begin{aligned}
 6^3 &= 5^3 + 4^3 + 3^3 \\
 13^3 &= 12^3 + 7^3 + 5^3 + 1^3 \\
 69^3 &= 59^3 + 25^3 + 17^3 + 10^3 + 4^3 \\
 1021^3 &= 1014^3 + 231^3 + 177^3 + 157^3
 \end{aligned}$$

Most of them can be expressed as a sum of three, four, or five perfect cubes, but some require more.

We can write a predicate function called **cubicDecompositionExists**, which takes in an **int** we'll call **n** and a reference to an initially empty **Set<int>** called **cubicRoots**, and design it to return **true** if and only if  $n^3$  can be decomposed into a sum of smaller, distinct cubes. When we return **true**, we can make sure the **Set<int>** referenced by **cubicRoots** is populated with a collection of distinct integers that, when cubed and added together, produce  $n^3$ .

- **cubicDecompositionExists(5, cubicRoots)** should return **false**, because  $5^3$  is greater than  $4^3 + 3^3 + 2^3 + 1^3$
- **cubicDecompositionExists(6, cubicRoots)** should return **true**, and after it returns, **cubicRoots** should contain 3, 4, and 5. (As it turns out, this is the only such decomposition.)
- **cubicDecompositionExists(11, cubicRoots)** should return **false**, because there's no cubic decomposition that produces  $5^3$ .
- **cubicDecompositionExists(1021, cubicRoots)** should return **true**, and after it returns, **cubicRoots** might contain 1014, 231, 177, and 157. (I say might, because there are several cubic decompositions of  $1021^3$ , and we'll allow our routine to discover any single one of them.)

And because we all want the most out of the code we write, we'll further require that our implementation identify the decomposition (or at least one of the many decompositions) with the smallest **cubicRoots** set.

```
static bool cubicDecompositionExists(int remaining, int n,
                                   Set<int>& cubicRoots, int numTermsAllowed) {
    if (remaining == 0) return true;
    if (remaining < 0) return false; // not necessary.. huge perf win, though
    if (numTermsAllowed == 0) return false;

    for (int s = n - 1; s > 0; s--) {
        if (cubicDecompositionExists(remaining - s * s * s, s,
                                     cubicRoots, numTermsAllowed - 1)) {
            cubicRoots += s;
            return true;
        }
    }

    return false;
}

static bool cubicDecompositionExists(int n, Set<int>& cubicRoots) {
    for (int maxTerms = 2; maxTerms <= n; maxTerms++) {
        if (cubicDecompositionExists(n * n * n,
                                     cubicRoots, maxTerms))
            return true;
    }

    return false;
}
```