Algorithms/Backtracking

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, A

Backtracking is a general algorithmic technique that considers searching every possible combination in order to solve an optimization problem. Backtracking is also known as **depth-first search** or **branch and bound**. By inserting more knowledge of the problem, the search tree can be pruned to avoid considering cases that don't look promising. While backtracking is useful for hard problems to which we do not know more efficient solutions, it is a poor solution for the everyday problems that other techniques are much better at solving.

However, dynamic programming and greedy algorithms can be thought of as optimizations to backtracking, so the general technique behind backtracking is useful for understanding these more advanced concepts. Learning and understanding backtracking techniques first provides a good stepping stone to these more advanced techniques because you won't have to learn several new concepts all at once.

> Backtracking Methodology
>
> 1. View picking a solution as a sequence of **choices**
> 2. For each choice, consider every **option** recursively
> 3. Return the best solution found

This methodology is generic enough that it can be applied to most problems. However, even when taking care to improve a backtracking algorithm, it will probably still take exponential time rather than polynomial time. Additionally, exact time analysis of backtracking algorithms can be extremely difficult: instead, simpler upperbounds that may not be tight are given.

Longest Common Subsequence (exhaustive version)

The LCS problem is similar to what the Unix "diff" program does. The diff command in Unix takes two text files, *A* and *B*, as input and outputs the differences line-by-line from *A* and *B*. For example, diff can show you that lines missing from *A* have been added to *B*, and lines present in *A* have been removed from *B*. The goal is to get a list of additions and removals that could be used to transform *A* to *B*. An overly conservative solution to the problem would say that all lines from *A* were removed, and that all lines from *B* were added. While this would solve the problem in a crude sense, we are concerned with the minimal number of additions and removals to achieve a correct transformation. Consider how you may implement a solution to this problem yourself.

The LCS problem, instead of dealing with lines in text files, is concerned with finding common items between two different arrays. For example,

```
let a := array {"The", "great", "square", "has", "no", "corners"}
let b := array {"The", "great", "image", "has", "no", "form"}
```

We want to find the longest subsequence possible of items that are found in both *a* and *b* in the same order. The LCS of *a* and *b* is

> "The", "great", "has", "no"

Now consider two more sequences:

```
let c := array {1, 2, 4, 8, 16, 32}
let d := array {1, 2, 3, 32, 8}
```

Here, there are two longest common subsequences of *c* and *d*:

> 1, 2, 32; and
> 1, 2, 8

Note that

> 1, 2, 32, 8

is *not* a common subsequence, because it is only a valid subsequence of *d* and not *c* (because *c* has 8 before the 32). Thus, we can conclude that for some cases, solutions to the LCS problem are not unique. If we had more information about the sequences available we might prefer one subsequence to another: for example, if the sequences were lines of text in computer programs, we might choose the subsequences that would keep function definitions or paired comment delimiters intact (instead of choosing delimiters that were not paired in the syntax).

On the top level, our problem is to implement the following function

```
// lcs -- returns the Longest common subsequence of a and b
function lcs(array a, array b): array
```

which takes in two arrays as input and outputs the subsequence array.

How do you solve this problem? You could start by noticing that if the two sequences start with the same word, then the longest common subsequence always contains that word. You can automatically put that word on your list, and you would have just reduced the problem to finding the longest common subset of the rest of the two lists. Thus, the problem was made smaller, which is good because it shows progress was made.

But if the two lists do not begin with the same word, then one, or both, of the first element in *a* or the first element in *b* do not belong in the longest common subsequence. But yet, one of them might be. How do you determine which one, if any, to add?

The solution can be thought in terms of the back tracking methodology: Try it both ways and see! Either way, the two sub-problems are manipulating smaller lists, so you know that the recursion will eventually terminate. Whichever trial results in the longer common subsequence is the winner.

Instead of "throwing it away" by deleting the item from the array we use array slices. For example, the slice

$a[1,..,5]$

represents the elements

$\{a[1], a[2], a[3], a[4], a[5]\}$

of the array as an array itself. If your language doesn't support slices you'll have to pass beginning and/or ending indices along with the full array. Here, the slices are only of the form

$a[1,..]$

which, when using 0 as the index to the first element in the array, results in an array slice that doesn't have the 0th element. (Thus, a non-sliced version of this algorithm would only need to pass the beginning valid index around instead, and that value would have to be subtracted from the complete array's length to get the pseudo-slice's length.)

```
// lcs -- returns the longest common subsequence of a and b
function lcs(array a, array b): array
  if a.length == 0 OR b.length == 0:
    // if we're at the end of either list, then the lcs is empty

    return new array {}
  else-if a[0] == b[0]:
    // if the start element is the same in both, then it is on the lcs,
    // so we just recurse on the remainder of both lists.

    return append(new array {a[0]}, lcs(a[1,..], b[1,..]))
  else
    // we don't know which list we should discard from. Try both ways,
    // pick whichever is better.

    let discard_a := lcs(a[1,..], b)
    let discard_b := lcs(a, b[1,..])

    if discard_a.length > discard_b.length:
      let result := discard_a
    else
      let result := discard_b
    fi
    return result
  fi
end
```

Shortest Path Problem (exhaustive version)

To be improved as Dijkstra's algorithm in a later section.

Largest Independent Set

Bounding Searches

If you've already found something "better" and you're on a branch that will never be as good as the one you already saw, you can terminate that branch early. (Example to use: sum of numbers beginning with 1 2, and then each number following is a sum of any of the numbers plus the last number. Show performance improvements.)

Constrained 3-Coloring

This problem doesn't have immediate self-similarity, so the problem first needs to be generalized. Methodology: If there's no self-similarity, try to generalize the problem until it has it.

Traveling Salesperson Problem

Here, backtracking is one of the best solutions known.

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, A