

CS1Bh Lecture Note 2

Case Study: Spelling Correction

The divide-and-conquer technique for algorithm design requires to express the solution to one instance of the problem in terms of other ‘smaller’ instances of the same problem. In this lecture, we illustrate this by considering a particular example: spelling correction. We discuss the conditions under which such an approach leads to results, illustrate straightforward implementation using *recursion*, and sketch a dynamic programming approach to making the implementation more efficient in this case.

2.1 Spelling correction

This task occurs in a variety of settings, e.g. in word processors and in some aspects of DNA sequencing. We are given a string that is a correct word w and a specimen string s , and require to find the smallest number of *edit operations* (called the *minimal cost* in the sequel) required to correct s to w . The edit operations allowed are: *delete*, which deletes a single letter in s , and *insert*, which inserts a single letter in s . For example, consider the instance of the problem where the word w is “where” and the specimen s is “ware”. We can correct s by inserting “h” after the “w”, deleting the “a” and inserting an “e” after the inserted “h”. The cost of this correction is 3. The general instance of this problem is where $w = w_1 \dots w_n$ and $s = s_1 \dots s_m$, the w_i and s_j being the individual letters in the strings. The question is well-defined because the smallest number of edit operations is never more than $m + n$ (since we can always just delete all the letters in s and then insert all the letters in w). In solving this problem, there are two different cases to consider:

$s_1 = w_1$: The minimal cost equals the minimal cost of correcting $s_2 \dots s_m$ to $w_2 \dots w_n$ (why?).

$s_1 \neq w_1$: Either we must insert the letter w_1 at the front of s in which case the minimal cost equals the minimal cost of correcting $s_1 \dots s_m$ to $w_2 \dots w_n$ plus one (for the cost of doing the insertion), or we must delete the first letter in s , in which case the minimal cost of the correction equals the minimal cost of correcting $s_2 \dots s_m$ to $w_1 \dots w_n$ plus one (for the cost of deleting the first letter of w).

Notice that in each of the two cases described above, the solution to our typical instance is given in terms of other instances of the spelling correction problem. In

each case, either one or both of the strings being considered gets smaller so, because the total length of the two strings is always decreasing, eventually one or other of the strings being considered will have zero length and we will be done.

However, this indicates that the case analysis above is not quite thorough enough, and we also need to consider two special cases:

s is the empty string: the answer equals the length of w because we have to insert all the letters of w to make s match w .

w is the empty string: the answer equals the length of s because we have to delete all the letters of s to correct s to w .

Problem-solving approaches in which the solution to the typical instance of the problem is expressed in terms of other instances of the problem are usually called *recursive*. When using this way of solving a problem, we need to exercise caution to ensure that the process of expressing the solution to one problem instance in terms of other instances of the same problem cannot continue forever. Eventually we must get to some instance of the problem with a solution not expressed in terms of other instances of the same problem. In the case of the spelling correction problem, we can see that any instance in which either the specimen or the word is the empty string can be solved without reference to more instances of the problem. In all the other cases, one or other, or both, of the strings gets shorter by one, so eventually we will reach an instance in which one of the strings is empty.

2.2 Worst-case run time of the solution

Although this solution to the spelling correction problem is fairly easy to understand, it is not particularly efficient. Lack of efficiency is *not inherent* in recursive solutions to problems but in this case the solution is particularly inefficient.

Consider the case where w consists of n occurrences of the letter “a” and s consists of m occurrences of the “b”. Here, the first letters of s and w will be different in every new instance of the problem arising in solving the original problem instance. This means that, every time, we need to consider two problem instances in which the size of one of the strings has been reduced by one. The result is that the run time is exponential in m and n , and so grows very rapidly with increasing string lengths. Clearly, this is not a good approach for word processors.

2.3 Implementation in Java

Recursive solutions to problems often provide the basis for elegant and efficient implementations. In Java, it is possible to implement recursive solutions fairly directly using recursive methods. A recursive method is one that invokes itself in its definition. A simple implementation of the spelling corrector is given in the class `Spell`. In that class, the method `spell(s, i, j)` implements the solution strategy given above. Notice that on lines 13, 15 and 16 the method invokes itself in its definition. There is no automatic check that such recursive invocations of the method always eventually

reach a non-recursive case, so it is quite possible to construct methods that will repeatedly invoke themselves until no more store is available to keep a record of method invocations. For the method `spell`, we have taken care that this cannot happen.

```

class Spell{ // 1
    private String w; // 2
    public Spell(String s){ // 3
        w = s; // 4
    } // 5
    // 6
    int spell(String s, int i, int j) { // 7
        if (j == w.length()) { // 8
            return (s.length()-i); // 9
        } else if (i == s.length()) { // 10
            return (w.length()-j); // 11
        } else if (s.charAt(i) == w.charAt(j)) { // 12
            return(spell(s,i+1,j+1)); // 13
        } else { // 14
            int ins = spell(s,i,j+1) + 1; // 15
            int del = spell(s,i+1,j) + 1; // 16
            if (ins > del) { // 17
                return(del); // 18
            } else { // 19
                return(ins); // 20
            } // 21
        } // 22
    } // 23
    // 24
    public int spell(String s) { // 25
        return spell(s,0,0); // 26
    } // 27
} // 28
// 29

```

Any object of the class `Spell` has a private `String` attribute `w` that records the correct word that the method `spell` is to check against. This method takes three parameters, the meaning being that `spell(s,i,j)` will return the smallest number of edit operations needed to correct the part-string starting at the `i`th letter of `s` to the part-string starting at the `j`th letter of `w`.

This test program checks its two command-line arguments against each other.

```

class Test {
    public static void main(String[] args) {
        Spell word = new Spell(args[0]);
        System.out.println("Cost: " + word.spell(args[1]));
    }
}

```

For example, the command

```
time java Test ware where
```

checks “ware” against “where”, outputs the result, that the number of edit operations required is 3, and then prints out the time taken.

2.4 Making things more efficient

Our program works fine for the small example, but if we try instead to find how many edits are required to change “abricadobra open says me” to the correct spelling, “abracadabra open sesame”, we would run out of patience after a few minutes of waiting. This is not unexpected, since we have seen that the simple implementation of the spelling corrector has exponential worst-case behaviour. However, we can see from the implementation that if w has n letters and s has m letters, then in correcting s there are only $(m + 1)(n + 1)$ possible different invocations of the `spell` method. This suggests that the simple recursive solution recomputes some values very many times.

If we were to store the values of the different invocations of `spell` in a table the first time each is computed, it should take closer to $O(mn)$ time to compute the spelling correction. For example, when w is “where” and s is “ware”:

s, i	w, j	W	H	E	R	E
	0	0	1	2	3	4
W	0	3	4	3	2	3
A	1	4	3	2	1	2
R	2	3	2	1	0	1
E	3	4	3	2	1	0
	4	5	4	3	2	1

We shall return to this general idea of not recomputing values unnecessarily in the next lecture but one (Note 4).

A new implementation, the `FastSpell` class, which incorporates such a table, is shown on the next page. The table is stored in the private attribute called `memo`. Note that this stores the calculated value *plus one* — this is so that a zero value can denote ‘not computed yet’. Trying out `FastSpell` on the longer example mentioned above gave:

```
> time java FastTest "abricadobra open says me" "abracadabra open sesame"
Cost: 9
```

```
real    0m0.891s
user    0m0.510s
sys     0m0.200s
```

Thus, it finds a solution using nine editing operations in less than a second.

In practice, most spelling correctors only consider very short corrections (e.g. corrections using 2 or 3 edit operations). In this case, we know that m and n must be almost the same and so we do not need to fill in the entire table. In fact, we just need

to consider entries close to the diagonal in the table (why?). For these practical circumstances, it is possible to derive a method that has just $O(n)$ running time and also just $O(n)$ storage space for the table.

```

class FastSpell{
    private String w;
    private int[][] memo; // store 1 + cost
    // -- then zero signifies unitialised

    public FastSpell(String s){
        w = s;
    }

    int spell(String s, int i, int j) {
        int m = memo[i][j];
        if (m > 0) // already computed this
            return (m-1);
        else{
            if (j == w.length()) {
                m = s.length() - i;
            } else if (i == s.length()) {
                m = w.length() - j;
            } else if (s.charAt(i) == w.charAt(j)) {
                m = spell(s,i+1,j+1);
            } else {
                int ins = spell(s,i,j+1);
                int del = spell(s,i+1,j);
                if (ins > del) {
                    m = del + 1;
                } else {
                    m = ins + 1;
                }
            }
            memo[i][j] = m + 1;
            return m;
        }
    }

    public int spell(String s) {
        memo = new int[1 + s.length()][1 + w.length()];
        return spell(s,0,0);
    }
}

```

2.5 Summary

- Some problems have convenient solution techniques that involve expressing the solution to one instance of the problem in terms of other instances. Such solutions are usually called *recursive*.
- Provided we always eventually arrive at instances that do not require us to look at other instances to provide their solution, a recursive solution is well-defined.
- Java allows us to define recursive methods that can mirror recursive solutions.
- In some circumstances, recursive solutions repeatedly recompute some quantities. This can result in significant inefficiencies. One approach to resolve this is to store intermediate results so they can be reused without requiring recomputation. This can lead to significant improvements in the running time of the solution.

*Stuart Anderson and Gordon Brebner.
Javier Esparza, January 23th, 2003.*