Dynamic Programming

We have seen several examples of dynamic programming (Floyd, Warshall, Bellman-Held-Karp).

An example to illustrate:

- Top-down versus bottom-up solutions
- Memoisation

We take as our example the following word break problem:

Given a string of characters s, can s be split into words occurring in a dictionary?

For example, s = ‘*windown*’ can be split as ‘*win down*’ (or ‘*wind own*’).

Looking at all possible splits would take too long, as there are exponentially many.

We start with a ‘top-down’ recursive solution:

```
procedure wb1(s):
        if len(s) == 0:
                return true
        else:
                for i = 0 to len(s) - 1:
                        if indict(s[i:]):
                                if wb1(s[: i]):
                                        return true
        return false
```

Here indict checks if a string is a word in the dictionary. We shall use the number of indict lookups as a measure of running time. We use a Python-like slice notation:

- $s[i:j]$ is string s from index i to index $j-1$
- $s[:i]$ is string s from start index 0 to index $i-1$
- $s[i:]$ is string s from index i to the end index $len(s)-1$

The recurrence relation for the worst case on strings of length n is:

$$\begin{aligned} W_1(0) &= 0 \\ W_1(n) &= n + W_1(0) + \cdots + W_1(n-1) \quad (n \geq 1) \end{aligned}$$

This has solution:

$$W_1(n) = 2^n - 1$$

This is exponential.

It is inefficient to compute $wb1(s[:i])$ repeatedly.

The recursion tree for a word of length four:

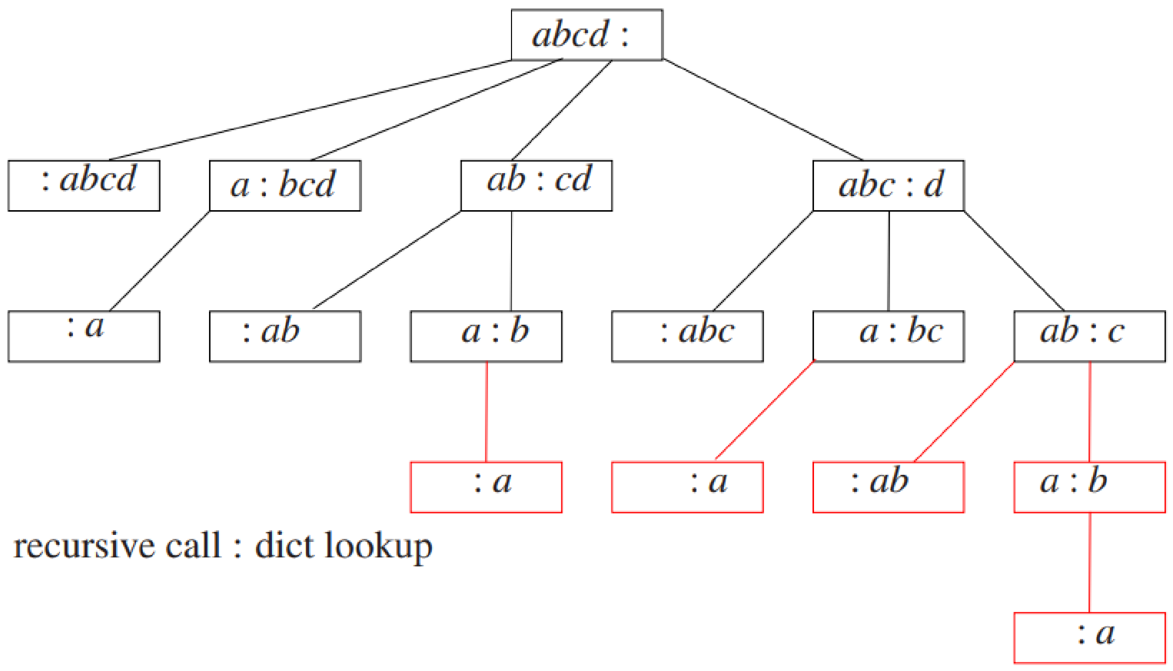

Figure 2.11: Recursion tree for wb1 on a string of length four.

The nodes of the tree represent recursive calls and dictionary lookups. We evaluate in depth-first order. The red nodes are duplicates of nodes which have already been evaluated.

We solve the problem of repeated evaluation of recursive calls using Memoisation:

- Storing previously computed results of recursive calls in an associative array `memo`
- `memo` is initially empty

```
memo = {} # empty associative array
procedure wb2(s):
        if len(s) == 0:
                return true
        else:
                for i = 0 to len(s) - 1:
                        if indict(s[i :]):
                                if memo[s[: i]] undefined:
                                        memo[s[: i]] = wb2(s[: i])
                                if memo[s[: i]]:
                                        return true
        return false
```

The transformation is generic. The recursion tree is now cut off at depth two (the red nodes have been removed).

At level 1 we have n nodes.

At level 2 there are $\sum_{i=0}^{n-1} i$ nodes.

Adding, we see that the worst case on strings of length n is $\sum_{i=0}^{n} i = n(n+1)/2$. The complexity is now $O(n^2)$.

This is a dramatic improvement.

We next develop a ‘bottom-up’ non-recursive solution:

- The idea is to solve increasing sub-problems culminating in the main problem

$$s[:0], s[:1], \ldots, s[:n] = s$$

These problems overlap, which is typical of dynamic programming, in contrast to divide and conquer algorithms where subproblems do not overlap.

We store the result for $s[:i]$ in array $wb[i]$

```
wb3(s):
        n = len(s)
        wb[0] = true
        if n > 0:
                for i = 1 to n:
                        wb[i] = false
                        for j = 0 to i - 1:
                                if wb[j] and indict(s[j : i]):
                                        wb[i] = true
                                        break
        return wb[n]
```

The complexity is again $O(n^2)$

We compare the advantages of the two styles.

A top-down solution with Memoisation is easier to develop, and may be faster if not all subproblems need computing.

A bottom-up non-recursive solution avoids overheads due to recursion.

Dynamic programming typically involves taking a problem with an exponentially large solution space and finding a path to a solution in polynomial time.

Typical problems are to find whether a solution exists (Hamiltonian Circuit, Word Break) or (very often) find the best solution (TSP).

A dynamic programming solution will tend to involve evaluating overlapping subproblems (in contrast with divide and conquer algorithms).

It will also require storing the results of computations of subproblems (requiring more space/memory usage).