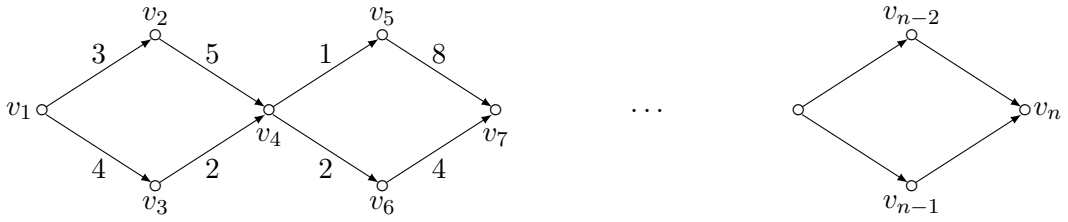
Lecture Notes, Fundamental Algorithms: Dynamic Programming

Dynamic Programming can be viewed as a technique for handling recursive algorithms in which the same subproblems are called repeatedly. The solution is the obvious one: save (or cache) solutions to subproblems as they are computed and reuse them as needed.

To illustrate this, let's consider the following shortest path problem in an acyclic graph (for those of you who don't know graphs, simply consider this to be a tree modified to have shared subtrees – we are going to draw our example graph so it goes from left to right, so if you are thinking of it as a modified tree, it has been laid sideways).



This can be formulated recursively as follows.

$$\text{Shortest}(v) = \begin{cases} \min\{\text{Shortest}(w) + \text{length}(v,w) \mid (v,w) \text{ is an edge of the graph}\} \\ 0 \qquad\qquad\qquad \text{if } v \text{ has no outgoing edge} \end{cases}$$

This yields the following pseudo-code.

```
SP(v)
    if v.shtst has no successor (* or 'child' *)
        then v.shtst ← 0
        else v.shtst ← MaxInt;
    for each successor (* 'child' *) w of v do
        SP(w);
        v.shtst ← min{v.shtst, w.shtst + length(v, w)};
        end (* for *)
```

This code is highly inefficient, however. For the call SP(v_1) will make recursive calls to SP(v_2) and SP(v_3). SP(v_2) will make a call to SP(v_4), and later SP(v_3) will make another call to SP(v_4). Each call to SP(v_4) will make a call to SP(v_5) and to SP(v_6); each call to SP(v_5) will make a call to SP(v_7), as will each call to SP(v_6), resulting in 4 calls to SP(v_7). This will in turn induce 8 calls to SP(v_{10}), and this doubling keeps going, resulting in $2^{(n-1)/3}$ calls to SP(v_n).

But there is no point to more than one recursive call to each node. To ensure this we introduce a global array Done, where Done[v] stores values **True** or **False**, to indicate whether SP[v] has already been computed or not. Initially Done[v] is set to **False** for all v. In addition, as we are introducing arrays, we might as well also store the computed values in an array Shtst. Initially, Shtst[v] is set to MaxInt for all v. This leads to the following code.

```
SP(v)
    if v.shtst has no successor (* or 'child' *)
        then v.shtst ← 0
        else v.shtst ← MaxInt;
    for each successor (* 'child' *) w of v do
        if not Done[w] then SP(w);
        Shtst[v] ← min{Shtst[v], Shtst[w] + length(v, w)};
        end; (* for *)
    Done[v] ← True
```

Now, there is just one recursive call on each node, and the cost of the computation at each node is proportional to the number of successors, which is $O(1)$ for each node in this example, yielding an overall $O(n)$ computation time, a vast improvement on $\Theta(2^{n/3})$.

Often, we would like to know a shortest path and not just its length. To determine this, it is enough to know the next vertex on a shortest path. So let's modify the algorithm by adding an array Next to store this information.

```
SP(v)
    if v.shtst has no successor (* or 'child' *)
        then v.shtst ← 0
        else v.shtst ← MaxInt;
    for each successor (* 'child' *) w of v do
        if not Done[w] then SP(w);
        if Shtst[v] > Shtst[w] + length(v, w)
            then do
                Shtst[v] ← Shtst[w] + length(v, w);
                Next[v] ← w
            end; (* then do *)
        end; (* for *)
    Done[v] ← True
```

The modified $SP(v)$ still runs in linear time. A shortest path can now be printed as follows:

```
Path(v)
    print(v)
    if Next[v] ≠ nil then Path(Next[v])
```

Question: What if we want to compute the number of shortest paths? This should be stored in an array NumSht[v]. Show how to modify SP so as to compute these values.

Creating a Dynamic Programming Algorithm The main challenge is to find the recursive solution. Once this is done, creating an efficient recursive implementation to compute an optimal value (e.g. the length of a shortest path) is an essentially mechanical task, as is augmenting the solution to find an instance generating the optimal value (e.g. an actual shortest path).

We continue with several additional examples of dynamic programming algorithms.

Longest common subsequence (LCS) The input comprises two strings $u = u_n u_{n-1} \dots u_1$ and $v = v_m v_{m-1} \dots v_1$. As we shall see, it is more convenient to index the strings in right to left order. For example $u =$ ACAGTCAGGT and $v =$ CCGACGGAC. One common subsequence is CGAGG (and in fact this is a longest subsequence). Let's introduce the notation $\text{LCS}(n, m)$ for the length of the longest common subsequence.

What we need to do is to identify longest identical subsequences of characters in u and v; we can think of this process as matching pairs of equal characters from u and v, resp. So how do we compute this? We will repeatedly create smaller subproblems by reducing the strings from the left end (this is simpler, notation-wise). Incidentally, if we wanted to reduce from the right end, we would index the strings from left to right.

Suppose that $u_n = v_m$. Then we might as well include u_n and v_m in the LCS (for if not we could change the matched characters to include this pair without reducing the length of the LCS). In this case we then want to find the LCS of $u' = u_{n-1} u_{n-2} \dots u_1$ and $v' = v_{m-1} v_{m-2} \dots v_1$ recursively, or for short $\text{LCS}(n-1, m-1)$.

Otherwise, if $u_n \neq v_m$, one at least of u_n and v_m is not included in the longest common subsequence. Thus, in this case the LCS will be the maximum of $\text{LCS}(n, m-1)$ and $\text{LCS}(n-1, m)$.

This leads to the following recursive definition for LCS.

$$
\begin{aligned}
\text{LCS}(n, m) &= 1 + \text{LCS}(n-1, m-1) && \text{if } u_n = v_m, n, m \geq 1 \\
\text{LCS}(n, m) &= \max\{\text{LCS}(n, m-1), \text{LCS}(n-1, m)\} && \text{if } u_n \neq v_m, n, m \geq 1 \\
\text{LCS}(n, 0) &= 0 \\
\text{LCS}(0, m) &= 0
\end{aligned}
$$

Let's think about the resulting efficient recursive algorithm. It will make at most $(n+1)(m+1)$ recursive calls. Each recursive call results in $O(1)$ non-recursive work, for a total of $O(nm)$ work.

Next, we show the recursive calls that are made on the following strings: $u =$ ACAG and $v =$ CCGA. A Y in entry (i, j) indicates that a recursive call to $\text{LCS}(i, j)$ was made, and N indicates that no call was made.

	C	C	G	A	$m = 0$
A	$Y \rightarrow$	$Y \rightarrow$	$Y \rightarrow$	Y	N
	$\downarrow$	$\downarrow$	$\downarrow$	$\searrow$	
C	Y	Y	$Y \rightarrow$	$Y \rightarrow$	Y
	$\searrow$	$\searrow$	$\downarrow$	$\downarrow$	
A	N	$Y \rightarrow$	$Y \rightarrow$	Y	N
		$\downarrow$	$\downarrow$	$\searrow$	
G	N	$Y \rightarrow$	Y	N	Y
		$\downarrow$	$\searrow$		
$n = 0$	N	Y	N	Y	N

And the LCS values that are computed:

	C	C	G	A	$m = 0$
A	2	2	1	1	$\cdot$
C	2	2	1	1	0
A	$\cdot$	1	1	1	$\cdot$
G	$\cdot$	1	1	$\cdot$	0
$n = 0$	$\cdot$	0	$\cdot$	0	$\cdot$

Finally, if we want to reconstruct an actual longest common subsequence, we have to record the choice of recursive call that yields the longest common subsequence. Let's suppose that we always test removing a character from u first. Then we would get the following array of choices.

	C	C	G	A
A	$\downarrow$	$\downarrow$	$\downarrow$	$\searrow$
C	$\searrow$	$\searrow$	$\downarrow$	$\downarrow$
A	$\cdot$	$\downarrow$	$\downarrow$	$\searrow$
G	$\cdot$	$\rightarrow$	$\searrow$	$\cdot$

Following the arrows (representing choices), with diagonals meaning a common character, we see that the first C in v is matched with the C in u, and the two G's are matched, giving an LCS of CG. (We could also match the C in u with the second C in v, but this is not the match discovered by our algorithm.)

Next, we give pseudo-code for the recursive procedure, which we call FindLCS; it includes the computation of the choices or directions shown in the above example; these directions will be stored in an $n \times m$ array Dir. The procedure also uses $(n+1) \times (m+1)$ arrays Done and LCS; Done$[i, j]$ indicates whether the LCS$[i, j]$ entry has already been computed; the entries for Done are all initialized to **False**. In addition, LCS is initialized to be all zero. The initial call is to FindLCS(n, m).

```
FindLCS(i, j)
    if u_i = v_j then do
        if not Done[i − 1, j − 1] then FindLCS(i − 1, j − 1);
        LCS[i, j] ← LCS[i − 1, j − 1] + 1; Dir[i, j] ← ↘
        end (* then do *)
    else do
        if not Done[i, j − 1] then FindLCS(i, j − 1);
        if not Done[i − 1, j] then FindLCS(i − 1, j);
        if LCS[i − 1, j] ≥ LCS[i, j − 1]
            then LCS[i, j] ← LCS[i − 1, j]; Dir[i, j] ← ↓
            else LCS[i, j] ← LCS[i, j − 1]; Dir[i, j] ← →
        end; (* else do *)
    Done[i, j] ← True
```

Below, we give pseudo-code for a recursive procedure to print an LCS.

```
PrintLCS(i, j)
    if i = 0 or j = 0 then return;
    if Dir[i, j] = ↘
        then do print(u_i); PrintLCS(i − 1, j − 1) end
    else if Dir[i, j] = ↓
        then PrintLCS(i − 1, j)
        else PrintLCS(i, j − 1)
```

Note that if we had indexed u and v in the "normal" way, i.e. from left to right (normal for English that is), then the PrintLCS procedure would report the LCS in reverse order. One way of handling this would be to push the characters onto a stack instead of printing them. Thus if the

LCS was the string w, then reading from top to bottom, the characters on the stack would form the string w. Then repeatedly popping and printing characters from the stack would print w in left to right order.

Edit Distance Again the input consists of two strings $u = u_n u_{n-1} \dots u_1$ and $v = v_m v_{m-1} \dots v_1$. The question being asked is how small a number of edits can transform u to v (as we will see shortly, the problem is symmetric: this is also the number of edits needed to transform v to u.

There are three types of edits: deletion, which removes a specific character of u; insertion, which inserts a character into u; and substitution, which replaces a character in u (e.g. an A) with another character (e.g. a C). We illustrate this further in the following example.

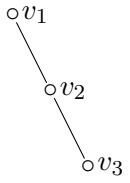
Let $u =$ ACAGTCAGGT and $v =$ CCGACGGAC. Below, we show 6 edits that transform u to v.

A	C	A	G	T	C	A	G	G	T	
sub(C)		del		sub(A)		del			sub(A)	ins(C)
C	C		G	A	C		G	G	A	C

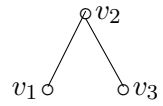
This is quite similar to the LCS problem. We leave the formulation of a recursive solution to the reader.

Optimal Binary Search Tree (BST) Suppose the n values $v_1 < v_2 < \ldots < v_n$ are being stored in a binary search tree. Suppose that the probability of a search on value v_i is p_i (so $\sum_{i=1}^{n} p_i = 1$). The cost of a search for the item with value v_i (item v_i for short) is defined to be the number of nodes on the path from the root of the search tree to the node at which v_i is stored.

For example, suppose there are 3 items, v_1, v_2, v_3, with corresponding search probabilities $3/4, 1/8, 1/8$, resp. Then the following two search trees have average search cost $11/8$ and $15/8$, respectively. The "balanced" search tree is the worse choice in terms of average search cost.



$$\text{cost} = 3/4 \cdot 1 + 1/8 \cdot 2 + 1/8 \cdot 3 = 11/8$$



$$\text{cost} = 3/4 \cdot 2 + 1/8 \cdot 1 + 1/8 \cdot 2 = 15/8$$

The task is to find a BST with minimum expected search cost, i.e. to minimize $\sum_{i=1}^{n} d_i \cdot p_i$, where d_i is the vertex depth of v_i.

If v_j were the item at the root of the BST, the expected search cost would be

$$\Big\{ D(1, j-1) + D(j+1, n) + \sum_{j=1}^{n} p_j \Big\},$$

where $D(1, j-1)$ denotes the average search cost on an optimal tree for items $v_1, \dots, v_{j-1}$, when the search probabilities for these items are $p_1, \dots, p_{j-1}$, and similarly $D(j+1, n)$ denotes the average search cost on the optimal tree for items $v_{j+1}, \dots, v_n$, when the search probabilities for these items are $p_{j+1}, \dots, p_n$. We optimize by choosing a value of j that minimizes the above expression.

When we seek to recurse, we see that in general we will need to recurse on subsets of items of the form $v_i, \ldots, v_k$. Accordingly, we let $D(i,k)$ denote the average search cost on the optimal tree for items $v_i, \ldots, v_k$, when the search probabilities are $p_i, \ldots, p_k$ (in general, the search probabilities may sum to less than 1). $D(i,k)$ can be expressed recursively as follows.

$$D(i,k) = \min_{i \le j \le k} \left\{ D(i, j-1) + D(j+1, n) + \sum_{j=i}^{k} p_j \right\} \qquad i \le k$$

$$D(i,k) = 0 \qquad i > k$$

The running time of an efficient recursive implementation can be deduced as follows. There are $O(n^2)$ recursive problems. The non-recursive work in the computation of $D(i,k)$ requires a minimization over $k - i + 1 \le n$ terms, which takes $O(n)$ time. This is $O(n^3)$ time in total.

What needs to be remembered when $D(i,k)$ is computed in order to construct an optimal binary search tree?

Two Common Patterns

There are two patterns that arise frequently in the recursive subproblems in dynamic programming algorithms.

Reduce the problem size at one end only In this case we create smaller instances of the original problem. LCS(n,m) provides a good example. The recursive problems are all of the form LCS(i,k) with either $i < n$ or $k < m$, or both. The same is true for knapsack problems. What is possibly unfamiliar here is the fact that the problem is expressed in terms of two parameters, n and m. Similarly, in the shortest path problem, the recursive problems are always to 'descendant' nodes.

Reduce the problem at both ends In this case the problem is being generalized. For example, when computing the optimal BST, the original problem instance $D(1,n)$, yields recursive problems $D(i,k)$ with $1 \le i \le k-1 \le n$ and $k - i + 1 < n$ ($k - i + 1$ is the size of the problem). This is also the case for the algorithm for matrix chain multiplication.

You have already seem multiple recursive algorithms in which a problem is reduced at both ends, for example, merge sort, binary search, quicksort, quick select.