

We look at the solutions to four similar dynamic programming problems.

- Maximum contiguous subsequence.
- Maximum independent subsequence.
- Longest increasing subsequence.
- Longest common subsequence.

One lesson we shall learn is that it is not always obvious what subproblem we should define as the basis for our dynamic programming recurrence relation. The most obvious choice of a subproblem may not work, in the sense that we may not be able to describe the solution of the obvious subproblem in terms of other subproblems. Sometimes it will be a rather non-obvious choice of a subproblem that can be solved in terms of other subproblems.

### 1. Maximum contiguous subsequence

Suppose we are given an array,  $A$ , of both positive and negative numbers

$$A = [a_1, a_2, \dots, a_n].$$

Two integers  $i$  and  $j$  with  $1 \leq i \leq j \leq n$  define a **contiguous subsequence** of  $A$  which is all the elements of  $A$  between index  $i$  and index  $j$ , that is  $[a_i, a_{i+1}, a_{i+2}, \dots, a_{j-1}, a_j]$ .

Given integers  $i$  and  $j$  that define a contiguous subsequence, we can form the sum of the numbers in this subsequence,

$$\sum_{k=i}^j a_k.$$

The “maximum contiguous subsequence” problem is to find the maximum possible sum over all possible contiguous subsequences from the given array  $A$ .

For example, consider the following array  $A$ .

|       |   |   |    |   |          |   |    |   |    |   |
|-------|---|---|----|---|----------|---|----|---|----|---|
| 1     | 2 | 3 | -7 | 3 | 1        | 3 | -4 | 6 | -3 | 1 |
| $a_1$ |   |   |    |   | $a_{11}$ |   |    |   |    |   |

The maximum contiguous subsequence sum is 9, and it is the sum of the subsequence from  $a_5$  to  $a_9$ .

Now let us consider how to solve this problem using dynamic programming. We need to come up with a definition of a subproblem for which we can write a recurrence relation relating the solution of the subproblem to the solutions of other (smaller) subproblems. The most obvious choice for a subproblem is

$$M(j) = \text{maximum contiguous subsequence sum between } a_1 \text{ and } a_j.$$

But this is not a good definition for a subproblem. Look at our example array  $A$  above. For that array,  $M(7)$  is 7 and that sum is from the subsequence from  $a_5$  to  $a_7$ . Now look

at  $M(6)$ . It is 6, and it is defined by the subsequence from  $a_1$  to  $a_3$ . Notice that the solution to the subproblem  $M(7)$  does not really have anything to do with the solution to the (smaller) subproblem  $M(6)$ . This is not good. Here is another way to think about this. Suppose we know the solution to subproblem  $M(j-1)$ . Can we use that information to construct the solution to subproblem  $M(j)$ ? Can we use element  $a_j$  to extend the solution to  $M(j-1)$  into a solution to  $M(j)$ ? The answer is no, because we do not know where the subsequence defined by  $M(j-1)$  ends. We cannot write a recurrence relation relating the subproblem  $M(j)$  to any of its (smaller) subproblems  $M(i)$  where  $i < j$ .

Here is a more useful definition of a subproblem.

$ME(j)$  = maximum contiguous subsequence sum of subsequences that *end* at  $a_j$ .

This definition of a subproblem has the advantage that we can define a recurrence relation using it, but it has a slight disadvantage in that  $ME(n)$  is *not* the solution of the problem we started out with (why?).

Suppose we know the value of  $ME(j-1)$ . Here is how we can use it to compute  $ME(j)$ . If  $ME(j-1)$  is a positive number, then we can just tack on  $a_j$  to the maximum subsequence that ends at  $a_{j-1}$  to get the maximum subsequence that ends at  $a_j$  and  $ME(j) = ME(j-1) + a_j$ . If  $ME(j-1)$  is negative, then the best we can do for  $ME(j)$  is  $ME(j) = a_j$ , in other words, don't make any use of the maximum subsequence that ends at  $a_{j-1}$ . We can write this as the following recurrence relation.

$$ME(j) = \max\{a_j + ME(j-1), a_j\} \quad \text{with } ME(1) = a_1$$

The problem we want to solve is  $M(n)$ , where  $M$  still has the definition it was given above. We can define  $M(n)$  in terms of the values of  $ME(j)$ . Since the solution to  $M(n)$  must be a subsequence that ends somewhere, that maximizing subsequence must be a solution to one of the subproblems  $ME(j)$  for some  $j$  between 1 and  $n$ . So

$$M(n) = \max_{1 \leq j \leq n} \{ ME(j) \}.$$

We can also evaluate  $M$  in the following way.

$$\begin{aligned} M(j) &= \max\{ M(j-1), ME(j) \} \\ &= \max\{ M(j-1), ME(j-1) + a_j, a_j \} \end{aligned}$$

We can implement the recurrence relation for  $ME$  as a bottom-up (or top-down) dynamic programming solution. You should convince yourself that this would be an  $\Theta(n)$  algorithm. The calculation of  $M(n)$  from the table of  $ME(j)$  values is another  $\Theta(n)$  procedure. So the whole algorithm is  $\Theta(n)$ .

Notice that we have only provided the maximum *value* of the maximal contiguous subsequence sum. We have not calculated the location in the array of this maximal contiguous subsequence, nor have we determined if there might be more than one contiguous subsequence that gives this maximal sum.

## 2. Maximum independent subsequence

Suppose we are given an array,  $A$ , of  $n$  positive numbers

$$A = [a_1, a_2, \dots, a_n].$$

An **independent subsequence** of  $A$  is a selection of elements from  $A$  such that no two elements in the selection are adjacent in the array. An independent subsequence is kind of the opposite of a contiguous subsequence in the sense that an independent subsequence must be as non-contiguous as possible; no two elements of an independent subsequence can be contiguous (i.e., next to each other).

Given an independent subsequence of  $A$ , we can form the sum of the numbers in this subsequence. The “maximum independent subsequence” problem is to find the maximum possible sum over all possible independent subsequences from the given array  $A$ .

Let us consider how to solve this problem using dynamic programming. We need to come up with a definition of a subproblem for which we can write a recurrence relation relating the solution of the subproblem to the solutions of other (smaller) subproblems. The most obvious choice for a subproblem is

$$M(j) = \text{maximum independent subsequence sum between } a_1 \text{ and } a_j.$$

Unlike the maximum contiguous sum problem, this time this obvious subproblem works. We can write a recurrence relation relating subproblem  $M(j)$  to its (smaller) subproblems  $M(i)$  with  $i < j$ .

The idea behind the recurrence relation is this. The solution to subproblem  $M(j)$  exists, and the independent subsequence that realizes that solution either uses element  $a_j$  or it does not. If the solution to  $M(j)$  does not use element  $a_j$ , then the solution to  $M(j)$  is in fact also the solution to subproblem  $M(j-1)$  (why?). On the other hand, if the solution to  $M(j)$  does use element  $a_j$ , then  $M(j) = M(j-2) + a_j$ . (Notice that we used the subproblem  $M(j-2)$  so that we are sure that we do not violate the definition of an independent subsequence.) In general, for any given subproblem  $M(j)$ , we do not know which of these two cases is the correct one. But the correct case will always be the one that returns the greater value for  $M(j)$ . We can express this as the following recurrence relation.

$$M(j) = \max\{M(j-1), M(j-2) + a_j\} \quad \text{with } M(1) = a_1$$

The solution to the original problem is then  $M(n)$ .

We can implement the recurrence relation for  $M$  as a bottom-up (or top-down) dynamic programming solution. You should convince yourself that this would be an  $\Theta(n)$  algorithm.

Notice that, as with the maximum contiguous sum problem, we have only provided the maximum *value* of the maximal independent subsequence sum. We have not calculated the location in the array of this maximal independent subsequence, nor have we determined if there might be more than one independent subsequence that gives this maximal sum.

### 3. Longest increasing subsequence

Suppose we are given an array,  $A$ , of  $n$  positive numbers (or a string of  $n$  characters)

$$A = [a_1, a_2, \dots, a_n].$$

An **increasing subsequence** of  $A$  is a subsequence,  $[a_{i_1}, a_{i_2}, \dots, a_{i_k}]$ , of elements from  $A$  such that  $i_j < i_{j+1}$  and  $a_{i_j} < a_{i_{j+1}}$  for each  $1 \leq j < k$ . Notice that the elements in the increasing subsequence need not be contiguous in  $A$ .

Given an increasing subsequence of  $A$ , we can compute the length of this subsequence. The “longest increasing subsequence” problem is to find the maximum possible length over all possible increasing subsequences from the given array (or string)  $A$ .

Let us consider how to solve this problem using dynamic programming. We need to come up with a definition of a subproblem for which we can write a recurrence relation relating the solution of the subproblem to the solutions of other (smaller) subproblems. The most obvious choice for a subproblem is

$$L(j) = \text{length of the longest increasing subsequence between } a_1 \text{ and } a_j.$$

However, as with the maximum contiguous sum problem, this is not a good definition for a subproblem. Suppose we know the solution to subproblem  $L(j-1)$ . Can we use that information to construct the solution to subproblem  $L(j)$ ? Can we use element  $a_j$  to extend the solution to  $L(j-1)$  into a solution to  $L(j)$ ? The answer is no, because even though we know the length,  $L(j-1)$ , of the longest increasing subsequence between  $a_1$  and  $a_{j-1}$ , we do not know at which element  $a_i$  that subsequence ended, so we don’t know if  $a_j$  can be used to extend that subsequence. So we cannot write a recurrence relation relating the subproblem  $L(j)$  to any of its (smaller) subproblems  $L(i)$  where  $i < j$ .

Here is a more useful definition of a subproblem.

$$LE(j) = \text{length of the longest increasing subsequence that ends at } a_j.$$

This definition of a subproblem has the advantage that we can define a recurrence relation using it, but it has a slight disadvantage in that  $LE(n)$  is *not* the solution of the problem we started out with (why?).

The idea behind the recurrence relation for  $LE(j)$  is this. Given  $L(j-1)$ , we know both the length of an increasing subsequence and where that subsequence ends. We can use  $a_j$  to increase the length of that subsequence if  $a_j > a_{j-1}$ , and the new increasing subsequence will have length  $1+L(j-1)$ . But even if we can extend the subsequence given by  $L(j-1)$ , the resulting subsequence need not be the longest increasing subsequence that ends at  $a_j$ . It may be that we can extend some other, longer, increasing subsequence,  $L(i)$  where  $i < j-1$  (and  $a_i < a_j$ ). So to find the longest possible increasing subsequence that ends at  $a_j$ , we need to take a maximum over all of the sequences that  $a_j$  can extend. This leads to the following recurrence relation.

$$LE(j) = 1 + \max_{a_i < a_j} \{ LE(i) \} \quad \text{with } LE(1) = 1$$

where we use the convention that the maximum over an empty set is zero. (If  $a_j \leq a_i$  for all  $1 \leq i < j$ , then the above maximum is over an empty set of indices. This can happen, for example, when the array  $A$  is sorted in decreasing order.)

The problem we want to solve is  $L(n)$ , where  $L$  still has the definition it was given above. We can define  $L(n)$  in terms of the values of  $LE(j)$ . Since the solution to  $L(n)$  must be a subsequence that ends somewhere, that maximizing subsequence must be a solution to one of the subproblems  $LE(j)$  for some  $j$  between 1 and  $n$ . So

$$L(n) = \max_{1 \leq j \leq n} \{ LE(j) \}.$$

We can implement the recurrence relation for  $LE$  as a bottom-up (or top-down) dynamic programming solution. You should convince yourself that this would be an  $\Theta(n^2)$  algorithm (because the calculation of each  $LE(j)$  requires searching back through the array and looking at each element  $a_i$  for  $1 \leq i < j$ ). The calculation of  $L(n)$  from the table of  $LE(j)$  values is a  $\Theta(n)$  procedure. So the whole algorithm is  $\Theta(n^2)$ .

There is a  $\Theta(n \lg(n))$  dynamic programming algorithm that solves this problem. It uses an even trickier definition of a subproblem than the one we used here. See *Introduction to Algorithms: A Creative Approach*, by Udi Manber, pages 167–169.

Notice that we have only provided the maximum *length* of the longest increasing subsequence. We have not calculated the location in the array of this maximal increasing subsequence, nor have we determined if there might be more than one increasing subsequence that gives this maximal length.

#### 4. Longest common subsequence

Suppose we are given two arrays,  $A$  and  $B$ , of  $n$  and  $m$  positive numbers (or two strings of  $n$  and  $m$  characters)

$$A = [a_1, a_2, \dots, a_n],$$
$$B = [b_1, b_2, \dots, b_m].$$

A **common subsequence** of  $A$  and  $B$  is a subsequence  $[a_{i_1}, a_{i_2}, \dots, a_{i_k}]$  of elements from  $A$  and a subsequence  $[b_{j_1}, b_{j_2}, \dots, b_{j_k}]$  of elements from  $B$  such that  $a_{i_\nu} = b_{j_\nu}$  for each  $1 \leq \nu \leq k$ . Notice that the elements in the common subsequence need not be contiguous in  $A$  or  $B$ .

Given an common subsequence of  $A$  and  $B$ , we can compute its length. The “longest common subsequence” problem is to find the maximum possible length over all possible common subsequences from the given arrays (or strings)  $A$  and  $B$ .

Let us consider how to solve this problem using dynamic programming. We need to come up with a definition of a subproblem for which we can write a recurrence relation relating the solution of the subproblem to the solutions of other (smaller) subproblems. The most obvious choice for a subproblem is

$$L(i, j) = \text{length of the longest subsequence common to } [a_1, \dots, a_i] \text{ and } [b_1, \dots, b_j].$$

As we shall now show, this obvious choice of subproblem is a good choice. We can write a recurrence relation relating subproblem  $L(i, j)$  to its (smaller) subproblems  $L(r, s)$  with  $r < i$  and  $s < j$ .

The idea behind the recurrence relation is this. Given indices  $i$  and  $j$ , consider elements  $a_i$  from  $A$  and  $b_j$  from  $B$ . We have two mutually exclusive cases, either  $a_i = b_j$  or  $a_i \neq b_j$ . Let us consider these two cases one at a time.

Suppose that we have  $a_i = b_j$ . Then the element  $a_i$  (and also  $b_j$ ) *must* be the tail end of the common subsequence that solves the subproblem  $L(i, j)$  (why?). What about the rest of this common subsequence? It must be  $L(i-1, j-1)$ . That is, the common subsequence that solves  $L(i, j)$  must have  $a_i$  tacked on to the end of the common subsequence that solves  $L(i-1, j-1)$ . So in this case we have

$$L(i, j) = 1 + L(i-1, j-1) \quad \text{if } a_i = b_j.$$

That is what our recurrence relation looks like in this case.

Now suppose that we are in the other case, where  $a_i \neq b_j$ . In this case we can say for sure that we cannot have *both*  $a_i$  and  $b_j$  as part of the solution to  $L(i, j)$  (if both  $a_i$  and  $b_j$  were part of the solution to  $L(i, j)$ , then, since  $a_i$  is the last item we can choose from  $A$  and  $b_j$  is the last item we can choose from  $B$ , it must be that  $a_i$  and  $b_j$  are the last item in the common subsequence, but that implies  $a_i = b_j$ , a contradiction). If we cannot have both  $a_i$  and  $b_j$  as part of the solution to  $L(i, j)$ , that gives us three (mutually exclusive) cases. Either  $a_i$  is part of the solution to  $L(i, j)$  and  $b_j$  is not, or  $b_j$

is part of the solution to  $L(i, j)$  and  $a_i$  is not, or neither  $a_i$  nor  $b_j$  is part of the solution to  $L(i, j)$ . Let us look at these three cases one at a time.

Suppose that  $a_i \neq b_j$ , and  $a_i$  is not part of the solution to  $L(i, j)$  but  $b_j$  is part of this solution. Since we know that  $L(i, j)$  doesn't make use of  $a_i$ , we can ignore it, and conclude that  $L(i, j) = L(i - 1, j)$ . That is, the common subsequence that solves  $L(i, j)$  is the same subsequence that solves  $L(i - 1, j)$ .

Similarly, if  $a_i \neq b_j$ , and  $b_j$  is not part of the solution to  $L(i, j)$  but  $a_i$  is part of this solution, then  $L(i, j) = L(i, j - 1)$ .

For the third case, if  $a_i \neq b_j$ , and neither  $a_i$  nor  $b_j$  is part of the solution to  $L(i, j)$ , then  $L(i, j) = L(i - 1, j - 1)$  (that is, we can ignore both  $a_i$  and  $b_j$ ).

In general, when  $a_i \neq b_j$  we do not know which of the above three cases we are in, so we define our recurrence relation to take the maximum of the values returned by these three cases. That is

$$L(i, j) = \max\{L(i - 1, j), L(i, j - 1), L(i - 1, j - 1)\} \quad \text{if } a_i \neq b_j.$$

Now we can put our two main cases together, to get the whole recurrence relation.

$$L(i, j) = \begin{cases} 1 + L(i - 1, j - 1) & \text{if } a_i = b_j, \\ \max\{L(i - 1, j), L(i, j - 1), L(i - 1, j - 1)\} & \text{if } a_i \neq b_j. \end{cases}$$

This recurrence relation can also be rewritten as

$$L(i, j) = \begin{cases} 1 + L(i - 1, j - 1) & \text{if } a_i = b_j, \\ \max\{L(i - 1, j), L(i, j - 1)\} & \text{if } a_i \neq b_j. \end{cases}$$

The initial conditions are

$$L(i, 0) = 0 \text{ for } 0 \leq i \leq n \quad \text{and} \quad L(0, j) = 0 \text{ for } 0 \leq j \leq m$$

since there can be no common subsequence if one of the sequences is empty.

Notice that we have only provided the maximum *length* of the longest common subsequence. We have not calculated its location in the two arrays, nor have we determined if there might be more than one common subsequence that gives this maximal length.