Chapter 19

Dynamic Programming

> "An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. This, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities".
>
> Richard Bellman ("Eye of the Hurricane: An autobiography", World Scientific, 1984)

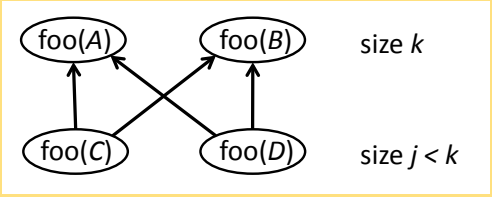
The Bellman-Ford shortest path algorithm covered in Chapter 16 is named after Richard Bellman and Lester Ford. In fact that algorithm can be viewed as a dynamic program. Although the quote is an interesting bit of history it does not tell us much about dynamic programming. But perhaps the quote will make you feel better about the fact that the term has little intuitive meaning.

In this book, as commonly used in computer science, we will use the term dynamic programming to mean an algorithmic technique in which (1) one constructs the solution of a

larger problem instance by composing solutions to smaller instances, and (2) the solution to each smaller instance can be used in multiple larger instances.

Dynamic programming is another example of an inductive technique where an algorithm relies on putting together smaller parts to create a larger solution. The correctness then follows by induction on problem size. The beauty of such techniques is that the proof of correctness parallels the algorithmic structure. So far the inductive techniques we have covered are divide-and-conquer, the greedy method, and contraction. In the greedy method and contraction each instance makes use of only a single smaller instance. In the case of divide-and-conquer, as with dynamic programming, we made use of multiple smaller instances to solve a single larger instance. However in divide-and-conquer we have always assumed the solutions are solved independently, and therefore to calculate the work we added up the work from each recursive call. However, what if two instances of size k, for example, both need the solution to the same instance of size $j < k$?

Example 19.1. Two smaller instance of the problem (function call) `foo` being shared by two larger instances.

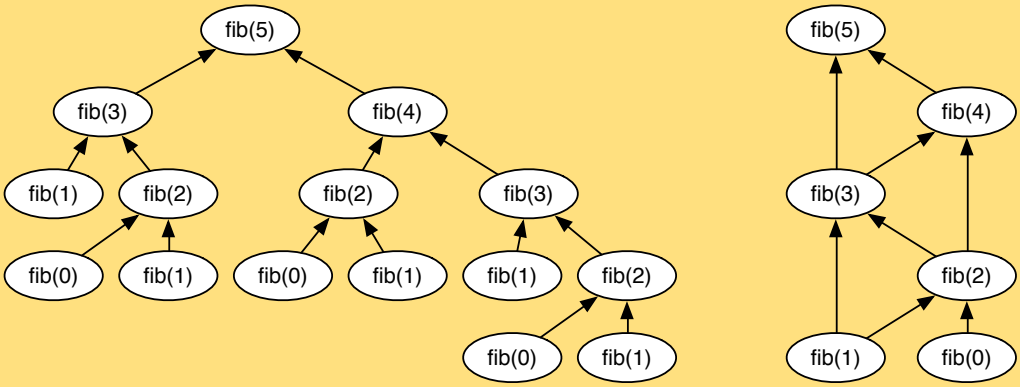


It is all about sharing. Although sharing the results in this simple example will make at most a factor of two difference in work, in general sharing the results of subproblems can make an exponential difference in the work performed. It turns out there are many practical problems where sharing results of subinstances is useful and can make a significant differences in the work used to solve a problem. We will go through several of these examples in this chapter.

Example 19.2. Consider the following algorithm for calculating the Fibonacci numbers.

```
fib(n) =
  if (n ≤ 1) then 1
  else fib(n − 1) + fib(n − 2)
```

This recursive algorithm takes exponential work in n as indicated by the recursion tree below on the left for `fib`(5). If the results from the instances are shared, however, then the algorithm only requires linear work, as illustrated below on the right.



Here many of the calls to `fib` are reused by two other calls. Note that the root of the tree or DAG is the problem we are trying to solve, and the leaves of the tree or DAG are the base cases.

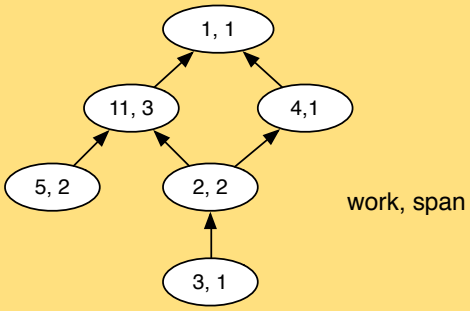
With divide-and-conquer the composition of a problem instance in terms of smaller instances is typically described as a tree, and in particular the so called recursion tree. With dynamic programming, to account for sharing, the composition can instead be viewed as a Directed Acyclic Graph (DAG). Each vertex in the DAG corresponds to a problem instance and each edge goes from an instance of size j to one of size $k > j$—i.e. each directed edge (arc) is directed from a smaller instances to a larger instance that uses it. The edges therefore represent dependences between the source and destination (i.e. the source has to be calculated before the destination can be). The leaves of this DAG (i.e. vertices with no in-edges) are the base cases of our induction (instances that can be solved directly), and the root of the DAG (the vertex with no out-edges) is the instance we are trying to solve. More generally we might actually have multiple roots if we want to solve multiple instances.

Abstractly dynamic programming can therefore be best viewed as evaluating a DAG by propagating values from the leaves (in degree zero) to the root (out degree zero) and performing some calculation at each vertex based on the values of its in-neighbors. Based on this view, calculating the work and span of a dynamic program is relatively straightforward. We can associate with each vertex a work and span required for that vertex. We then have

- The *work* of a dynamic program viewed as a DAG is the sum of the work of the vertices of that DAG, and

- the *span* of a dynamic program viewed as a DAG is the heaviest vertex-weighted path in the DAG—i.e., the weight of each path is the sum of the spans of the vertices along it.

Example 19.3. Consider the following DAG:



where we have written the work and span on each vertex. This DAG does $5 + 11 + 3 + 2 + 4 + 1 = 26$ units of work and has a span of $1 + 2 + 3 + 1 = 7$.

Whether a dynamic programming algorithm has much parallelism (work over span) will depend on the particular DAG. As usual the parallelism is defined as the work divided by the span. If this is large, and grows asymptotically, then the algorithm has significant parallelism Fortunately, most dynamic programs have significant parallelism. Some, however, do not have much parallelism.

The challenging part of developing a dynamic programming algorithm for a problem is in determining what DAG to use. The best way to do this, of course, is to think inductively—how can you solve an instance of a problem by composing the solutions to smaller instances? Once an inductive solution is formulated you can think about whether the solutions can be shared and how much savings can be achieved by sharing. As with all algorithmic techniques, being able to come up with solutions takes practice.

It turns out that most problems that can be tackled with dynamic programming solutions are optimization or decision problems. An *optimization problem* is one in which we are trying to find a solution that optimizes some criteria (e.g. finding a shortest path, or finding the longest contiguous subsequence sum). Sometimes we want to enumerate (list) all optimal solutions, or count the number of such solutions. A *decision problem* is one in which we are trying to find if a solution to a problem exists. Again we might want to count or enumerate the valid solutions. Therefore when you see an optimization or enumeration problem you should think about possible dynamic programming solutions.

Although dynamic programming can be viewed abstractly as a DAG, in practice we need to implement (code) the dynamic program. There are two common ways to do this, which are referred to as the top-down and bottom-up approaches. The *top-down* approach starts at the root(s) of the DAG and uses recursion, as in divide-and-conquer, but remembers solutions to subproblems so that when the algorithm needs to solve the same instance many times, only the first call does the work and the remaining calls just look up the solution. Storing solutions for reuse is called *memoization*. The *bottom-up* approach starts at the leaves of the DAG and

typically processes the DAG in some form of level order traversal—for example, by processing all problems of size 1 and then 2 and then 3, and so on.

Each approach has its advantages and disadvantages. Using the top-down approach (recursion with memoization) can be quite elegant and can be more efficient in certain situations by evaluating only those instances actually needed. The bottom up approach (level order traversal of the DAG) can be easier to parallelize and can be more space efficient, but always requires evaluating all instances. There is also a third technique for solving dynamic programs that works for certain problems, which is to find the shortest path in the DAG where the weighs on edges are defined in some problem specific way.

In summary the approach to coming up with a dynamic programming solution to a problem is as follows.

1. Is it a decision or optimization problem?
2. Define a solution recursively (inductively) by composing the solution to smaller problems.
3. Identify any sharing in the recursive calls, i.e. calls that use the same arguments.
4. Model the sharing as a DAG, and calculate the work and span of the computation based on the DAG.
5. Decide on an implementation strategy: either bottom up top down, or possibly shortest paths.

It is important to remember to first formulate the problem abstractly in terms of the inductive structure, then think about it in terms of how substructure is shared in a DAG, and only then worry about coding strategies.

19.1 Subset Sums

The first problem we cover in this chapter is a decision problem, the subset sum problem. It takes as input a multiset of numbers, i.e. a set that allows duplicate elements, and sees if any subset sums to a target value. More formally:

Problem 19.4. The *subset sum* (SS) problem is, given a multiset of positive integers S and a positive integer value k, determine if there is any $X \subseteq S$ such that $\sum_{x \in X} x = k$.

Example 19.5. `subsetSum`$(\{1, 4, 2, 9\}, 8)$ returns `false` since there is no subset of 1, 4, 2, and 9 that adds up to 8. However, `subsetSum`$(\{1, 4, 2, 9\}, 12)$ returns `true` since $1 + 2 + 9 = 12$.

In the general case when k is unconstrained, the SS problem is a classic NP-hard problem. However, our goal here is more modest. We are going to consider the case where we include the value of k in the work bounds as a variable. We show that as long as k is polynomial in $|S|$ then the work is also polynomial in $|S|$. Solutions of this form are often called *pseudo-polynomial* work (or time) solutions.

The SS problem can be solved using brute force by simply considering all possible subsets. This takes exponential work since there are an $2^{|S|}$ subsets. For a more efficient solution, one should consider an inductive solution to the problem. As greedy algorithms tend to be efficient, you should first consider some form of greedy method that greedily takes elements from S. Unfortunately the greedy method does not work. The problem is that in general there is no way to know for a particular element whether to include it or not. Greedily adding it could be a mistake, and recall that in greedy algorithms once you make a choice you cannot go back and undo your choice.

Since we do not know whether to add an element or not, we could try both cases, i.e. finding a sum with and without that element. This leads to a divide-and-conquer approach for solving SS(S, k) in which we pick one element a out of the set S (any will do), and then make two recursive calls, one with a included in X (the elements of S that sum to k) and one without a. For the call in which we include a we need to subtract the value a from k and in the other case we leave k as is. Here is an algorithm based on this idea. It assumes the input is given as a list (the order of the elements of S in the list does not matter):

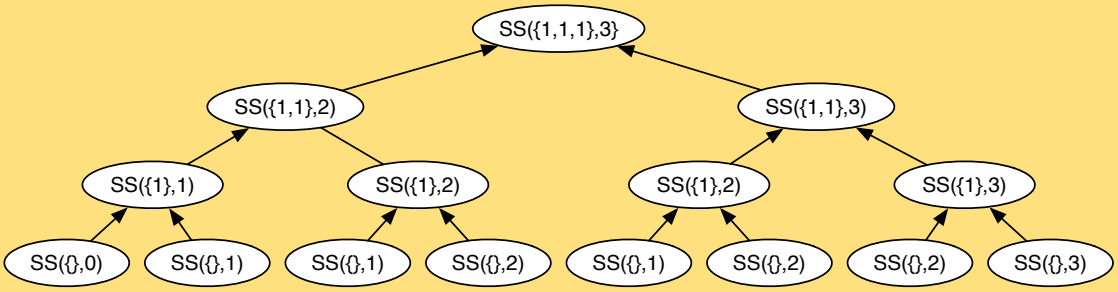
Algorithm 19.6. [Recursive Subset Sum]

```
SS(S, k) =
  case (S, k) of
    (_, 0) => true
  | (Nil, _) => false
  | (Cons(a, R), _) =>
      if (a > k) then SS(R, k)
      else (SS(R, k − a) or SS(R, k))
```

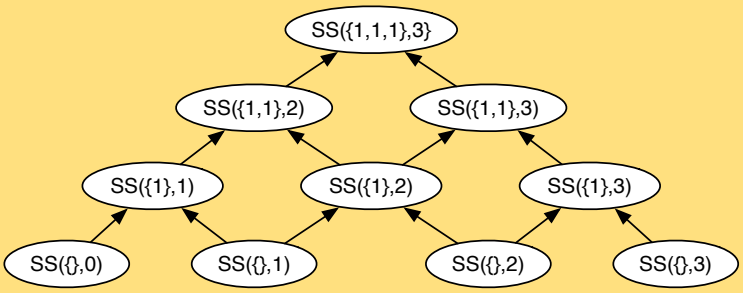
Lines 3 and 4 are the base cases. In particular if $k = 0$ then the result is true since the empty set sums to zero, and the empty set is a subset of any set. If $k \neq 0$ and S is empty, then the result is false since there is no way to get k from an empty set. If S is not empty but its first element a is greater than k, then we clearly can not add a to X, and we need only make one recursive call. The last line is the main inductive case where we either include a or not. In both cases we remove a from S in the recursive call to SS, and therefore use R. In the left case we are including a in the set so we have to subtract its value from k. In the right case we are not, so k remains the same. The algorithm is correct by induction—the base cases are correct, and inductively we assume the subproblems are correct and then note that those are the only two possibilities.

What is the work of this recursive algorithm? Well, it leads to a binary recursion tree that might be $n = |S|$ deep. This would imply something like 2^n work. This is not good. The key

Example 19.7. Consider $\texttt{SS}(\{1,1,1\},3)$. This clearly should return true since $1+1+1=3$. The recursion tree is as follows.



There are many calls to $\texttt{SS}$ in this tree with the same arguments. In the bottom row, for example there are three calls each to $\texttt{SS}(\emptyset,1)$ and $\texttt{SS}(\emptyset,2)$. If we coalesce the common calls we get the following DAG:



observation, however, is that there is a large amount of sharing of subproblems, as can be seen in Example 19.7. The question is how do we calculate how much sharing there is, or more specifically how many distinct subproblems are there in. For an initial instance $\texttt{SS}(S,k)$ there are only $|S|$ distinct lists that are ever used (each suffix of S). Furthermore, the value of the second argument in the recursive calls only decreases and never goes below 0, so it can take on at most $k+1$ values. Therefore the total number of possible instances of $\texttt{SS}$ (vertices in the DAG) is $|S|(k+1) = O(k|S|)$.

To calculate the overall work we need to sum the work over all the vertices of the DAG. However, each vertex only needs to do some constant number of operations (a comparison, a subtract, a logical or, and a few branches). Therefore each node does constant work and we have that the overall work is:

$$W(\texttt{SS}(S,k)) = O(k|S|)$$

To calculate the span we need know the heaviest path in the DAG. Again the span of each vertex is constant, so we only need to count the number of nodes in a path. The length of the longest path is at most $|S|$ since on each level we remove one element from the set. Therefore we have:

$$S(\texttt{SS}(S,k)) = O(|S|)$$

and together this tells us that the parallelism is $O(W/S) = O(k)$.

At this point we have not fully specified the algorithm since we have not explained how to take advantage of the sharing—certainly the recursive code we wrote would not. We will get back to this after one more example. Again we want to emphasize that the first two orders of business are to figure out the inductive structure and figure out what instances can be shared.

To make it easier to determine an upper bound on the number of subproblems in a DP DAG it can be convenient to replace any sequences (or lists) in the argument to the recursive function with an integer indicating our current position in the input sequence(s). For the subset sum problem this leads to the following variant of our previous algorithm:

Algorithm 19.8. [Recursive Subset Sum (Indexed)]

```
SS(S, k) = let
    % Determine SS(S[0, ..., i-1], j)
    % 0 ≤ i ≤ |S| and 0 ≤ j ≤ k, so at most (|S|+1) × (k+1) distinct calls
    SS'(i, j) =
        case (i, j) of
          (_, 0) => true
        | (0, _) => false
        | (i,k) => if (S[i-1] > j) then SS'(i-1, j)
                   else (SS'(i-1, j-S[i-1]) or SS'(i-1, j))
    % Calculates subset sum over all elements of S
    in SS'(|S|, k) end
```

In the algorithm the $i - 1$ represents the element we are currently considering. We start with $i = |S|$ and when $i = 0$ we are done (the algorithm reaches the base case). As we will see later this has a second important advantage—it makes it easier for a program to recognize when arguments are equal so they can be reused.

Remark 19.9. Why do we say the `SS` algorithm we described is pseudo-polynomial? The size of the subset sum problem is defined to be the number of bits needed to represent the input. Therefore, the input size of k is $\log k$. But the work is $O(2^{\log k}|S|)$, which is exponential in the input size. That is, the complexity of the algorithm is measured with respect to the length of the input (in terms of bits) and not on the numeric value of the input. If the value of k, however, is constrained to be a polynomial in $|S|$ (i.e., $k \leq |S|^c$ for some constant c) then the work is $O(k|S|) = O(|S|^{c+1})$ on input of size $c \log |S| + |S|$, and the algorithm is polynomial in the length of the input.

19.2 Minimum Edit Distance

The second problem we consider is a optimization problem, the minimum edit distance problem.

Problem 19.10. The minimum edit distance (MED) problem is, given a character set Σ and two sequences of characters $S = \Sigma^*$ and $T = \Sigma^*$, determine the minimum number of insertions and deletions of single characters required to transform S to T.

Example 19.11. Consider the sequence

$$S = \langle A, B, C, A, D, A \rangle$$

we could convert it to

$$T = \langle A, B, A, D, C \rangle$$

with 3 edits (delete the C in the middle, delete the last A, and insert a C at the end). This is the best that can be done so we have that $\text{MED}(S, T) = 3$.

Finding the minimum edit distance is an important problem that has many applications. For example in version control systems such as `git` or `svn` when you update a file and commit it, the system does not store the new version but instead only stores the "differences" from the previous version[1]. Storing the differences can be quite space efficient since often the user is only making small changes and it would be wasteful to store the whole file. Variants of the minimum edit distance problem are use to find this difference. Edit distance can also be used to reduce communication costs by only communicating the differences from a previous version. It turns out that edit-distance is also closely related to approximate matching of genome sequences. In many of these applications it useful to know in addition to the minimum number of edits, the actual edits. It is easy to extend the approach in this section for this purpose, but we leave it as an exercise.

Remark 19.12. The algorithm used in the Unix "diff" utility was invented and implemented by Eugene Myers, who also was one of the key people involved in the decoding of the human genome at Celera.

To solve the MED problem we might consider trying a greedy method that scans the sequences finding the first difference, fixing it and then moving on. Unfortunately no simple greedy method is known to work. The difficulty is that there are two ways to fix the error—we can either delete the offending character, or insert a new one. If we greedily pick the wrong edit, we might not end up with an optimal solution. Note that this is similar to the subset sum problem where we did not know whether to include an element or not.

[1] Alternatively it might store the new version, but use the differences to encode the old version.

Example 19.13. Consider the sequences

$$S = \langle A, B, C, A, D, A \rangle$$

and

$$T = \langle A, B, A, D, C \rangle .$$

We can match the initial characters $A-A$ and $B-B$ but when we come to $C-A$ in S and T, we have two choices for editing C, delete C or insert A. However, we do not know which leads to an optimal solution because we don't know the rest of the sequences. In the example, if we insert an A, then a suboptimal number of edits will be required.

As with the subset sum problem, since we cannot decide which choice to make (in this case deleting or inserting), why not try both. This again leads to a recursive solution. In the solution we can start at either end of the string, and go along matching characters, and whenever two characters do not match, we try both a deletion and an insertion, recur on the rest of the string, and pick the best of the two choices. This idea leads to the following algorithm (S and T are given as lists, and we start from the front):

Algorithm 19.14. [Recursive solution to the MED problem]

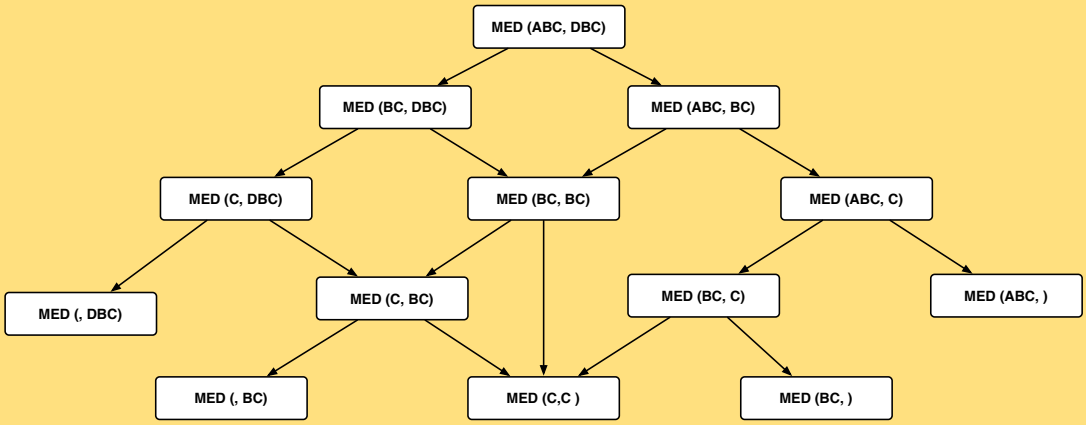
```
MED(S,T) =
  case (S, T) of
    (_, Nil) => |S|
  | (Nil, _) => |T|
  | (Cons(s,S'), Cons(t,T')) =>
      if (s = t) then MED(S',T')
      else 1 + min(MED(S,T'), MED(S',T))
```

In the first base case where T is empty we need to delete all of S to generate an empty string requiring $|S|$ insertions. In the second base case where S is empty we need to insert all of T, requiring $|T|$ insertions. If neither is empty we compare the first character of each string, s and t. If these characters are equal we can just skip them and make a recursive call on the rest of the sequences. If they are different then we need to consider the two cases. The first case ($\texttt{MED}(S, T')$) corresponds to inserting the value t. We pay one edit for the insertion and then need to match up S (which all remains) with the tail of T (we have already matched up the head t with the character we inserted). The second case ($\texttt{MED}(S', T)$) corresponds to deleting the value s. We pay one edit for the deletion and then need to match up the tail of S (the head has been deleted) with all of T.

As with the recursive solution to the subset sum problem, the recursive algorithm for MED performs exponential work. In particular the recursion tree is a full binary tree (each internal node has two children) and has a depth that is linear in the size of S and T. However, again, there are many calls to MED with the same arguments. We thus view the computation as a

DAG in which each vertex corresponds to call to MED with distinct arguments. An edge is placed from u to v if the call v uses u.

Example 19.15. An example MED instance with sharing.



The call to $\texttt{MED}(\langle B, C \rangle, \langle D, B, C \rangle)$, for example, makes recursive calls to $\texttt{MED}(\langle C \rangle, \langle D, B, C \rangle)$ (corresponding to the deletion of B from the first string) and $\texttt{MED}(\langle B, C \rangle, \langle B, C \rangle)$ (corresponding to the insertion of D into the second string). One of the calls is shared with the call to $\texttt{MED}(\langle A, B, C \rangle, \langle B, C \rangle)$

To determine the work we need to know how many vertices there are in the DAG. We can place an upper bound on the number of vertices by bounding the number of distinct arguments. There can be at most $|S| + 1$ possible values of the first argument since in recursive calls we only use suffixes of the original S and there are only $|S| + 1$ such suffixes (including the empty and complete suffixes). Similarly there can be at most $|T| + 1$ possible values for the second argument. Therefore the total number of possible distinct arguments to MED on original strings S and T is $(|T| + 1)(|S| + 1) = O(|S||T|)$. Furthermore the depth of the DAG (heaviest path) is $O(|S| + |T|)$ since each recursive call either removes an element from S or T so after $|S| + |T|$ calls there cannot be any element left. Finally we note that assuming we have constant work operations for removing the head of a sequence (e.g. using a list) then each vertex of the DAG takes constant work and span.

All together this gives us

$$W(\texttt{MED}(S, T)) = O(|S||T|)$$

and

$$S(\texttt{MED}(S, T)) = O(|S| + |T|).$$

As in subset sum we can again replace the lists used in MED with integer indices pointing to where in the sequence we are currently at. This gives the following variant of the MED algorithm:

Algorithm 19.16. [Recursive indexed solution to the MED problem]

```
MED(S,T) = let
  % Return MED(S[0,...,i-1], T[0,...,j-1])
  % 0 ≤ i ≤ |S| and 0 ≤ j ≤ |T|, so at most (|S|+1) × (|T|+1) distinct arguments
  MED'(i,j) =
    case (i,j) of
      (i,0) = i
    | (0,j) = j
    | (i,j) = if (S[i-1] = T[i-1]) then MED'(i-1,j-1)
              else 1 + min(MED'(i,j-1), MED'(i-1,j))
in MED'(|S|,|T|) end
```

This variant starts at the end of the sequence instead of the start, but is otherwise equivalent our previous version. This form makes it more clear that there are only $|S| \times |T|$ distinct arguments, and will make it easier to implement efficiently, as we discuss next.

19.3 Top-Down Dynamic Programming

So far we have assumed some sort of magic identified the shared subproblems in our recursive codes and avoided recomputation. We are now ready to cover one of the techniques, the top-down approach to implementing dynamic-programming algorithms.

The top-down approach is based on running the recursive code as is, generating implicitly the recursion structure from the root of the DAG down to the leaves. Each time a solution to a smaller instance is found for the first time it generates a mapping from the input argument to its solution. This way when we come across the same argument a second time we can just look up the solution. This process is called *memoization*, and the table used to map the arguments to solutions is called a *memo table*.

The tricky part of memoization is checking for equality of arguments since the arguments might not be simple values such as integers. Indeed in our examples so far the arguments have all involved sequences. We could compare the whole sequence element by element, but that would be too expensive.

You might think that we can do it by comparing "pointers" to the values. But this does not work since the sequences can be created separately, and even though the values are equal, there could be two copies in different locations. Comparing pointers would say they are not equal and we would fail to recognize that we have already solved this instance. There is actually a very elegant solution that fixes this issue called *hash consing*. Hash consing guarantees that there is a unique copy of each value by always hashing the contents when allocating a memory location and checking if such a value already exists. This allows equality of values to be done in constant work. The approach only works in purely functional languages and needs to be

implemented as part of the language runtime system (as part of memory allocation). Unfortunately no language does hash consing automatically, so we are left to our own devices.

The most common way to quickly test for equality of arguments is to use a simple type, such as an integer, as a *surrogate* to represent the input values. The property of these surrogates is that there needs to be a 1-to-1 correspondence between the surrogates and the argument values—therefore if the surrogates match, the arguments match. The user is responsible for guaranteeing this 1-to-1 correspondence.

Consider how we might use memoization in the dynamic program we described for minimum edit distance (MED). You have probably covered memoization before, but you most likely did it with side effects. Here we will do it in a purely functional way, which requires that we "thread" the table that maps arguments to results through the computation. Although this threading requires a few extra characters of code, it is safer for parallelism.

Recall that `MED` takes two sequences and on each recursive call, it uses suffixes of the two original sequences. To create integer surrogates we can simply use the length of each suffix. There is clearly a 1-to-1 mapping from these integers to the suffixes. `MED` can work from either end of the string so instead of working front to back and using suffix lengths, it can work back to front and use prefix lengths—we make this switch since it simplifies the indexing. This leads to the following variant of our `MED` code. In this code, we use prefixes instead of suffixes to simplify the indexing.

```
MED(S,T) = let
  MED'(i,j) =
      case (i,j) of
           (i,0) = i
         | (0,j) = j
         | (i,j) = if (S[i-1] = T[j-1]) then MED'(i-1,j-1)
                                        else 1 + min(MED'(i,j-1), MED'(i-1,j))
in
  MED'(|S|,|T|)
end
```

You should compare this with the purely recursive code for `MED`. Apart from the use of prefixes to simplify indexing, the only real difference is replacing S and T with i and j in the definition of `MED`′. The i and j are therefore the surrogates for S and T respectively. They represent the sequences $S\langle 0, \ldots, i-1\rangle$ and $T\langle 0, \ldots, j-1\rangle$ where S and T are the original input strings.

So far we have not added a memo table, but we can now efficiently store our solutions in a memo table based on the pair of indices (i, j). Each pair represents a unique input. In fact since the arguments range from 0 to the length of the sequence we can actually use a two dimensional array (or array of arrays) to store the solutions.

To implement the memoization we define a memoization function:

```
memo f (M,a) =
  case find(M,a) of
    SOME(v) => v
```

```
MED(S,T) = let
  MED'(M,(i,j)) =
      case (i,j) of
        | (M,(0,j)) = (M,j)
        | (M,(i,0)) = (M,i)
        | (M,(i,j)) =
            if (S[i-1] = T[j-1]) then memo MED'(M,(i-1,j-1))
            else let
              (M',v1) = memo MED'(M,(i,j-1))
              (M'',v2) = memo MED'(M',(i-1,j))
            in (M'',1+min(v1,v2)) end
in
  MED'({},(|S|,|T|))
end
```

Figure 19.1: The memoized version of Minimum Edit Distance (MED).

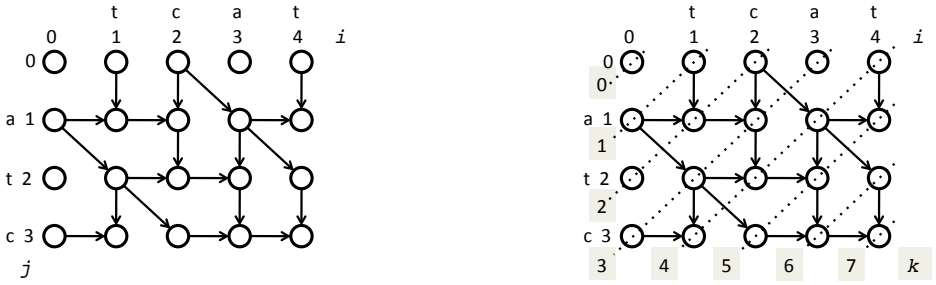
```
| NONE => let (M',v) = f(M,a)
          in (update(M',a,v), v) end
```

In this function f is the function that is being memoized, M is the memo table, and a is the argument to f. This function simply looks up the value a in the memo table. If it exists, then it returns the corresponding result. Otherwise it evaluates the function on the argument, and as well as returning the result it stores it in the memo. We can now write `MED` using memoization as shown in Figure 19.1.

Note that the memo table M is threaded throughout the computation. In particular every call to `MED` not only takes a memo table as an argument, it also returns a memo table as a result (possibly updated). Because of this passing, the code is purely functional. The problem with the top-down approach as described, however, is that it is inherently sequential. By threading the memo state we force a total ordering on all calls to `MED`. It is easy to create a version that uses side effects, as you did in 15-150 or as is typically done in imperative languages. In this case calls to `MED` can be made in parallel. However, then one has to be very careful since there can be race conditions (concurrent threads modifying the same cells). Furthermore if two concurrent threads make a call on `MED` with the same arguments, they can and will often both end up doing the same work. There are ways around this issue which are also fully safe—i.e., from the users point of view all calls look completely functional—but they are beyond the scope of this course.

19.4 Bottom-Up Dynamic Programming

The other technique for implementing dynamic-programming algorithms is the bottom-up technique. Instead of simulating the recursive structure, which starts at the root of the DAG,



(a) (b)

Figure 19.2: The DAG for MED on the strings "tcat" and "atc" (a) and processing it by diagonals (b).

when using this technique, we start at the leaves of the DAG and fills in the results in some order that is consistent with the DAG–i.e. for all edges (u, v) it always calculates the value at a vertex u before working on v. Because of this careful scheduling, all values will be already calculated when they are needed.

The simplest way to implement bottom-up dynamic programming is to do some form of systematic traversal of a DAG. It is therefore useful to understand the structure of the DAG. For example, consider the structure of the DAG for minimum edit distance. In particular let's consider the two strings $S = \texttt{tcat}$ and $T = \texttt{atc}$. We can draw the DAG as follows where all the edges go down and to the right:

The numbers represent the i and the j for that position in the string. We draw the DAG with the root at the bottom right, so that the vertices are structured the same way we might fill an array indexed by i and j. We Consider $\texttt{MED}(4, 3)$. The characters $S[4]$ and $T[3]$ are not equal so the recursive calls are to $\texttt{MED}(3, 3)$ and $\texttt{MED}(4, 2)$. This corresponds to the vertex to the left and the one above. Now if we consider $\texttt{MED}(4, 2)$ the characters $S[4]$ and $T[2]$ are equal so the recursive call is to $\texttt{MED}(3, 1)$. This corresponds to the vertex diagonally above and to the left. In fact whenever the characters $S[i]$ and $T[j]$ are not equal we have edges from directly above and directly to the left, and whenever they are equal we have an edge from the diagonal to the left and above. This tells us quite a bit about the DAG. In particular it tells us that it is safe to process the vertices by first traversing the first row from left to right, and then the second row, and so on. It is also safe to traverse the first column from top to bottom and then the second column and so on. In fact it is safe to process the diagonals in the / direction from top left moving to the bottom right. In this case each diagonal can be processed in parallel.

In general when applying $\texttt{MED}(S, T)$ we can use an $|T| \times |S|$ array to store all the partial results. We can then fill the array either by row, column, or diagonal. Using diagonals can be coded as shown in Figure 19.3.

The code uses a table M to store the array entries. In practice an array might do better. Each round of `diagonals` processes one diagonal and updates the table M, starting at the leaves at top left. The figure below shows these diagonals indexed by k on the left side and at the bottom. We note that the index calculations are a bit tricky (hopefully we got them right). Notice that the size of the diagonals grows and then shrinks.

```
MED(S,T) = let
  MED'(M,(i,j)) =
    case (i,j) of
          (i,0) => i
        | (0,j) => j
        | (i,j) => if (S[i-1] = T[j-1]) then M[i-1,j-1]
                   else 1 + min(M[i,j-1], M[i-1,j])
  diagonals(M,k) =
    if (k > |S| + |T|) then M
    else let
      s = max(0, k - |T|)
      e = min(k, |S|)
      M' = M ∪ {(i,k-i) ↦ MED(M,(i,k-i)) : i ∈ {s,...,e}}
     in
      diagonals(M', k+1)
     end
in
  diagonals({},0)
end
```

Figure 19.3: The dynamic program for MED based on the bottom-up approach using diagonals.

19.5 Optimal Binary Search Trees

We have talked about using BSTs for storing an ordered set or table. The cost of finding an key is proportional to the depth of the key in the tree. In a fully balanced BST of size n the average depth of each key is about $\log n$. Now suppose you have a dictionary where you know probability (or frequency) that each key will be accessed—perhaps the word "of" is accessed much more often than "epistemology". The goal is find a static BST with the lowest overall access cost. That is, make a BST so that the more likely keys are closer to the root and hence the average access cost is reduced. This line of reasoning leads to the following problem:

Definition 19.17. The *optimal binary search tree* (OBST) problem is given an ordered set of keys S and a probability function $p : S \to [0 : 1]$:

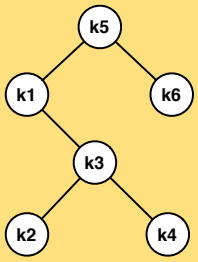
$$\min_{T \in \texttt{Trees}(S)} \left(\sum_{s \in S} d(s,T) \cdot p(s) \right)$$

where $\texttt{Trees}(S)$ is the set of all BSTs on S, and $d(s,T)$ is the depth of the key s in the tree T (the root has depth 1).

Example 19.18. For example we might have the following keys and associated probabilities

key	k_1	k_2	k_3	k_4	k_5	k_6
p(key)	$1/8$	$1/32$	$1/16$	$1/32$	$1/4$	$1/2$

Then the tree below has cost $31/16$, which is optimal. Creating a tree with these two solutions as the left and right children of S_i, respectively, leads to the optimal solution given S_i as a root.



Exercise 19.19. Find another tree with equal cost.

The brute force solution would be to generate every possible binary search tree, compute their cost, and pick the one with the lowest costs. But the number of such trees is $O(4^n)$ which is prohibitive.

Exercise 19.20. Write a recurrence for the total number of distinct binary search trees with n keys.

Since we are considering binary search trees, one of the keys must be the root of the optimal tree. Suppose S_r is that root. An important observation is that both of its subtrees must be optimal, which is a common property of optimization problems: The optimal solution to a problem contains optimal solutions to subproblems. This *optimal substructure* property is often a clue that either a greedy or dynamic programming algorithm might apply.

Which key should be the root of the optimal tree? A greedy approach might be to pick the key k with highest probability and put it at the root and then recur on the two sets less and greater than k. You should convince yourself that this does not work. Since we cannot know in advance which key should be the root, let's try all of them, recursively finding their optimal subtrees, and then pick the best of the $|S|$ possibilities.

With this recursive approach, how should we define the subproblems? Let S be all the keys placed in sorted order. Now any subtree of a BST on S must contain the keys of a contiguous subsequence of S. We can therefore define subproblems in terms of a contiguous subsequence of S. We will use $S_{i,j}$ to indicate the subsequence starting at i and going to j (inclusive of both). Not surprisingly we will use the pair (i, j) to be the surrogate for $S_{i,j}$.

Now Let's consider how to calculate the cost give the solution to two subproblems. For subproblem $S_{i,j}$, assume we pick key S_r ($i \leq r \leq j$) as a the root. We can now solve the OSBT problem on the prefix $S_{i,r-1}$ and suffix $S_{r+1,i}$. We therefore might consider adding these two solutions and the cost of the root ($p(S_r)$) to get the cost of this solution. This, however, is wrong. The problem is that by placing the solutions to the prefix and suffix as children of S_r we have increased the depth of each of their keys by 1. Let T be the tree on the keys $S_{i,j}$ with root S_r, and T_L, T_R be its left and right subtrees. We therefore have:

$$\begin{aligned}
\texttt{Cost}(T) &= \sum_{s \in T} d(s,T) \cdot p(s) \\
&= p(S_r) + \sum_{s \in T_L} (d(s,T_L)+1) \cdot p(s) + \sum_{s \in T_R} (d(s,T_R)+1) \cdot p(s) \\
&= \sum_{s \in T} p(s) + \sum_{s \in T_L} d(s,T_L) \cdot p(s) + \sum_{s \in T_R} d(s,T_R) \cdot p(s) \\
&= \sum_{s \in T} p(s) + \texttt{Cost}(T_L) + \texttt{Cost}(T_R)
\end{aligned}$$

That is, the cost of a subtree T is the probability of accessing the root (i.e., the total probability of accessing the keys in the subtree) plus the cost of searching its left subtree and the cost of searching its right subtree. When we add the base case this leads to the following recursive definition:

Algorithm 19.21. [Recursive Optimal Binary Search Tree]

$\texttt{OBST}(S) =$
if $|S| = 0$ **then** 0
else $\sum_{s \in S} p(s) + \min_{i \in \langle 1 \ldots |S| \rangle} \left(\texttt{OBST}(S_{1,i-1}) + \texttt{OBST}(S_{i+1,|S|}) \right)$

Exercise 19.22. How would you return the optimal tree in addition to the cost of the tree?

As in the examples of subset sum and minimum edit distance, if we execute the recursive program directly `OBST` it will require exponential work. Again, however, we can take advantage of sharing among the calls to `OBST`. To bound the number of vertices in the corresponding DAG we need to count the number of possible arguments to `OBST`. Note that every argument is a contiguous subsequence from the original sequence S. A sequence of length n has only $n(n+1)/2$ contiguous subsequences since there are n possible ending positions and for the i^{th} end position there are i possible starting positions ($\sum_{i=1}^{n} i = n(n+1)/2$). Therefore the number of possible arguments is at most $O(n^2)$. Furthermore the heaviest path of vertices in the DAG is at most $O(n)$ since recursion can at most go n levels (each level removes at least one key).

Unlike our previous examples, however, the cost of each vertex in the DAG (each recursive in our code not including the subcalls) is no longer constant. The subsequence computations

$S_{i,j}$ can be done in $O(1)$ work each (think about how) but there are $O(|S|)$ of them. Similarly the sum of the $p(s)$ will take $O(|S|)$ work. To determine the span of a vertex we note that the $\min$ and sum can be done with a reduce in $O(\log |S|)$ span. Therefore the work of a vertex is $O(|S|) = O(n)$ and the span is $O(\log n)$. Now we simply multiply the number of vertices by the work of each to get the total work, and the heaviest path of vertices by the span of each vertex to get the span. This give $O(n^3)$ work and $O(n \log n)$ span.

This example of the optimal BST is one of several applications of dynamic programming which effectively based on trying all binary trees and determining an optimal tree given some cost criteria. Another such problem is the matrix chain product problem. In this problem one is given a chain of matrices to be multiplied ($A_1 \times A_2 \times \cdots A_n$) and wants to determine the cheapest order to execute the multiplies. For example given the sequence of matrices $A \times B \times C$ it can either be ordered as $(A \times B) \times C$ or as $A \times (B \times C)$. If the matrices have sizes $2 \times 10, 10 \times 2$, and 2×10, respectively, it is much cheaper to calculate $(A \times B) \times C$ than $a \times (B \times C)$. Since $\times$ is a binary operation any way to evaluate our product corresponds to a tree, and hence our goal is to pick the optimal tree. The matrix chain product problem can therefore be solved in a very similar structure as the OBST algorithm and with the same cost bounds.

19.6 Coding optimal BST

As with the MED problem we first replace the sequences in the arguments with integers. In particular we describe any subsequence of the original sorted sequence of keys S to be put in the BST by its offset from the start (i, 1-based) and its length l. We then get the following recursive routine.

Algorithm 19.23. [Recursive Optimal Binary Search Tree (indexed)]

$\texttt{OBST}(S) = \textbf{let}$

% Determine OBST$(S[i, \ldots, i + l - 1])$

$\texttt{OBST'}(i, l) =$

if $l = 0$ **then** 0

else $\sum_{k=0}^{l-1} p(S[i + k]) + \min_{k=0}^{l-1} (\texttt{OBST'}(i, k) + \texttt{OBST'}(i + k + 1, l - k - 1))$

in

$\texttt{OBST}(1, |S|)$

end

This modified version can now more easily be used for either the top-down solution using memoization or the bottom-up solution. In the bottom-up solution we note that we can build a table with the columns corresponding to the i and the rows corresponding to the l. Each of them range from 1 to n ($n = |S|$). It would as follows:

```
  1 2 ... n
1          /
```

```
2         /
.       /
.     /
n   /
```

The table is triangular since as l increases the number of subsequences of that length decreases. This table can be filled up row by row since every row only depends on elements in rows above it. Each row can be done in parallel.

19.7 Problems with Efficient Dynamic Programming Solutions

There are many problems with efficient dynamic programming solutions. Here we list just some of them to give a sense of what these problems are.

1. Fibonacci numbers
2. Using only addition compute (n choose k) in $O(nk)$ work
3. Edit distance between two strings
4. Edit distance between multiple strings
5. Longest common subsequence
6. Maximum weight common subsequence
7. Can two strings S1 and S2 be interleaved into S3
8. Longest palindrome
9. longest increasing subsequence
10. Sequence alignment for genome or protein sequences
11. subset sum
12. knapsack problem (with and without repetitions)
13. weighted interval scheduling
14. line breaking in paragraphs
15. break text into words when all the spaces have been removed
16. chain matrix product
17. maximum value for parenthesizing x1/x2/x3.../xn for positive rational numbers
18. cutting a string at given locations to minimize cost (costs n to make cut)
19. all shortest paths

20. find maximum independent set in trees
21. smallest vertex cover on a tree
22. optimal BST
23. probability of generating exactly k heads with n biased coin tosses
24. triangulate a convex polygon while minimizing the length of the added edges
25. cutting squares of given sizes out of a grid
26. change making
27. box stacking
28. segmented least squares problem
29. counting Boolean parenthesization – true, false, or, and, xor, count how many parenthesization return true
30. balanced partition – given a set of integers up to k, determine most balanced two way partition
31. Largest common subtree

.