CS 624
Lecture 8: Dynamic Programming

We're now going to investigate a general method of attacking a certain class of problems. The method is called *dynamic programming*. It doesn't work on every problem, and it's not needed on many problems. (For instance, we have very good methods of sorting, and we don't need to use the technique of dynamic programming to deal with this problem.) But where dynamic programming is appropriate, it can often give extremely good results.

We'll start out by talking about a particular problem. We'll show how it can be solved using dynamic programming. Then we'll summarize the technique, and see how it can be applied to a couple of other problems.

1 Shortest paths in weighted DAGs

I have to start out with a word of caution. There are many variations of this problem, and there are many different algorithms that have been designed for these variations. So we need to be very particular about exactly what the problem is, and exactly how we are going to approach it.

And I should say from the beginning that even though you may find another algorithm for this *particular* problem, the one I am going to present here is essential for you to understand; and this is for two reasons:

1. It is as efficient as any other algorithm you will find.
2. It is key to understanding dynamic programming in general, and in particular the other two problems we will consider afterwards.

So here is the problem:

1. We have a DAG G.
2. Each edge e of G has associated with it a *weight* $w(e)$. In our problem, it will always be true that for each edge e, $w(e) > 0$. (There are other variations of the problem where this is not true. We are not concerned with those cases.)
3. If $n_1 \to n_2 \to \cdots \to n_p$ is a path in G (where of course each edge $n_i \to n_j$ is a directed edge *from* n_i *to* n_j), we define the *cost* of that path to be the sum of the weights of the edges on that path.

4. Each node of G has a name. One of the nodes of GR is named `start` and another node is named `end`.
5. There is at least one path from `start` to `end`.
6. The problem is to find the path of least cost from `start` to `end`.

Figure 1 shows a very simple example of a DAG like this.

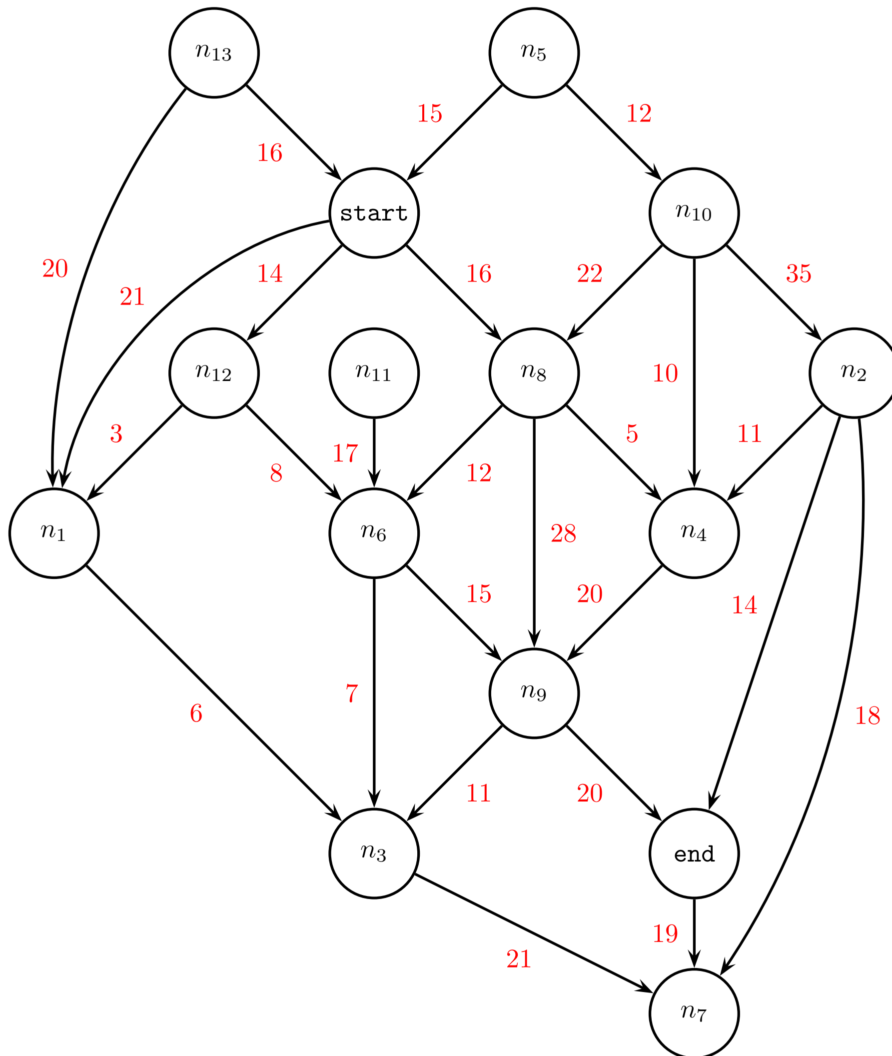


Figure 1: A simple weighted DAG with distinguished `start` and `end` nodes

---

You can think of this kind of problem as a simplified version of what Google maps or the GPS in a car does when you want to find the shortest path from where you are to some destination.

Each edge is a street, and the weight of each edge corresponds to the length traveled on that street, or perhaps to a guess of how long it might take to traverse that street. Of course, this is vastly simplified, because in this problem, all the streets are one-way[1].

In this case, it's not really too hard to find the shortest path between `start` and `end` just by trial and error. But it should be clear that if we had a real graph, with hundreds or thousands of nodes and edges, the difficulty would become immense, and in fact, intractable.

Nevertheless, there is a very effective way of solving this problem. It depends on two essential properties of the problem:

1.1 First essential property: optimal substructure

The problem exhibits what our text calls **optimal substructure**. In this case, this means the following: Suppose the nodes in our graph are denoted as $\{n_i : 1 \le i \le N\}$. Of course one of those nodes will also be named `start` and another node will also be named `end`. And suppose we know that an optimal path P from node n_a to node n_b is the following:

$$n_a \to n_{i_1} \to n_{i_2} \to n_{i_3} \cdots \to n_{i_k} \to \cdots n_{i_r} \to n_b$$

where n_{i_k} is just some node on the path. Suppose the sub-path of this path that starts at n_{i_k} and ends at n_b is denoted by Q.

Then since we know that the whole path P is optimal—that is, we know that P is the shortest path from n_a to n_b—we claim that it follows that Q is also optimal, in that it is the shortest path from n_{i_k} to n_b.

This claim is something that we need to prove. In this case, the proof is quite simple. It is what is called a "cut-and-paste" proof. Here it is:

Suppose that Q was not the shortest path from n_{i_k} to n_b. Then there would be a shorter one—let's call it Q'. But then if we follow the original path P from n_1 to n_{i_k} and then switch to the path Q' from n_{i_k} to n_b, the total cost of all the edges on that path will be *less* than the cost of the original path P. And that is a contradiction, because we assumed that P had the least cost of any path from n_a to n_b.

(Note that optimal paths are not necessarily unique. There might actually be more than one optimal path from n_a to n_b. We are not assuming that the cost of P is *less* than any other path—we are just assuming that *no other path has a cost that is less than the cost of P*. That's all we need for the argument we just made.)

The claim that we just proved is what is called (for this problem) the optimal substructure property. In other words, if a path P is optimal, then every sub-path of P that ends at the end of P is also optimal.

1.1 Exercise *Prove that in fact* any *sub-path of P is optimal. That is, is n_p and n_q are two nodes in P (with n_q occurring later than n_p in the path), and if P is an optimal path from its start to its end nodes, then the sub-path of P from n_p to n_q is an optimal path from n_p to n_q.*

[1] And there are no cycles in the graph—once you leave a vertex, you can't get back to it. And further, the graph itself is very special because it's a *planar* graph. That is, it can be drawn on a plane without any edges crossing each other. That would of course be true for a road map on the surface of the earth (and you might want to convince yourself that this is really true), but it's not necessary for our problem, and our algorithm will not depend on it in any way.

Because the problem exhibits optimal substructure, we can write a recursive algorithm for solving it. Here is how we can do this:

First, we can assume we have a function

WEIGHT(*node*, *child*)

where *node* and *child* are two nodes in the DAG and there is an edge from *node* to *child*. This function simply returns the weight of that edge.

Let us denote by *infinity* some number that is guaranteed to be larger than any number would could encounter in the problem. For instance, it could be 1 more than the sum of the weights of the edges. Then we can write the following procedure which returns the least cost of any path from an arbitrary node to `end`:

```
COST(node)
  if node has no children then
    return infinity
  foreach child of node do
    if child is end then
      just compute WEIGHT(node, child)
    else
      compute WEIGHT(node, child) + COST(child)
  return the minimum of those computed values
```

1.2 Exercise *Prove that* COST(`start`) *does indeed return the least cost of any path from* `start` *to* `end`*.*

Writing this recursive function is the key to our solution.

1.2 Second essential property: overlapping subproblems

There is a problem, however with our recursive procedure: it is far too expensive. And the reason is that the procedure has to compute a cost for each path leaving `start`, and there can be an enormous number of paths. For instance, suppose we have the DAG represented in Figure 2.

1.3 Exercise

1. *Show that the number of paths from* `start` *to* `end` *in the DAG in Figure 2 is 70.*
2. *Let us number the rows in the DAG starting from 0. So the number of the last row is 8. We could of course make larger figures in an exactly similar way, containing more rows. In any case the number of rows would be an even number. (This should be pretty clear, I hope.) Let us denote the number of rows by* $2k$*.*

   Find an expression for the number of paths from `start` *to* `end` *in such a DAG with* $2k$ *rows.*
3. *Show that the number of such paths is*

$$O\left(\frac{1}{\sqrt{k}}2^{2k}\right)$$

(You will need Stirling's formula to show this.)

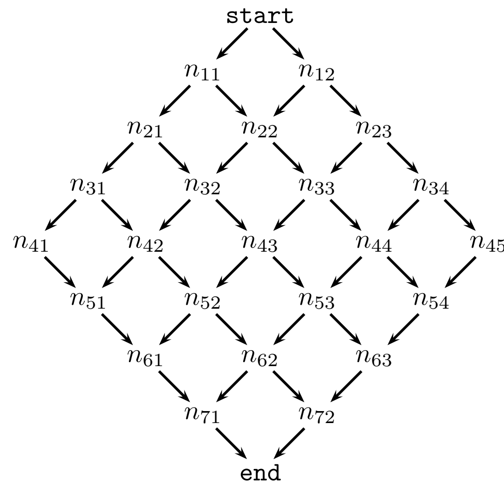


Figure 2: A bigger DAG.

Thus, the number of paths in such a DAG grows exponentially in the number of rows in the graph. The recursive algorithm we have so far come up with is therefore not going to be at all practical—it would be useless even for a moderately sized DAG.

However, we are saved by the second property of this problem, which is called **overlapping subproblems**. In this case, it occurs in this fashion. Consider the node n_{43} for example. There are many paths from `start` to `end` that pass through this node. And when our procedure reaches this node, it continues to compute the least cost of all paths from n_{43} to `end`. And it does this again and again, once for each time it reaches n_{43}. But we don't really have to do this: we can just compute the least cost of all paths from n_{43} to `end` once—the first time we reach n_{43}. And we can put this result in a lookup table. And so every time after this whenever we reach n_{43}, we can just look up that cost without recomputing it.

This technique is called *memoization*[2].

Here is our memoized algorithm:

Cost(*node*)
  if *node* has no children **then**
    return *infinity*
  foreach *child* of *node* **do**
    if *child* is `end` **then**
      just compute Weight(*node*, *child*)
    else

[2]Please be careful about this. The word "memoization" does not have an "r" in it. It is different from the word "memorization", which *does* contain an "r". They mean different things. Memoization (which is a word used only in computer science, so far as I know) refers to the process of saving (i.e., making a "memo") of a intermediate result so that it can be used again without recomputing it. Of course the words "memoize" and "memorize" are related etymologically—that is, they are derived from the same root—but they are different words, and you should not mix them up.

if COST(*child*) has not already been computed **then**
compute COST(*child*) and enter it in the lookup table
compute WEIGHT(*node*, *child*) + LOOKUP-COST(*child*)
return the minimum of those computed values

1.4 Exercise *Show that the cost of the memoized algorithm is* $O(|V| + |E|)$*, where as usual* V *is the set of vertices in the DAG and* E *is the set of edges.*

This is an astonishing improvement. At some point you should actually try this out, writing these two algorithms in your favorite programming language and testing them on a graph of this form of moderate size. You will be amazed at the difference in performance.

This is how dynamic programming works: it can be used for problems that

1. have the optimal substructure property, so that we can write a recursive procedure to compute the result, and
2. have overlapping subproblems, so we can memoize intermediate results to great effect.

2 Longest common subsequence

Definition *A* subsequence *of a sequence* $A = \{a_1, a_2, \ldots, a_n\}$ *is a sequence* $B = \{b_1, b_2, \ldots, b_m\}$ *(with* $m \leq n$*) such that*

- *Each* b_i *is an element of* A*.*
- *If* b_i *occurs before* b_j *in* B *(i.e., if* $i < j$*) then it also occurs before* b_j *in* A*.*

Note that in particular, we do *not* assume that the elements of B are consecutive elements of A. For instance, here is an example, where each sequence is an ordinary string of letters of the alphabet:

- "axdy" is a subsequence of "baxefdoym"

The "longest common subsequence" problem is simply this:

> Given two sequences $X = \{x_1, x_2, \ldots, x_m\}$ and $Y = \{y_1, y_2, \ldots, y_n\}$ (note that the sequences may have different lengths), find a subsequence common to both whose length is longest.

For example:

s p r i n g t i m e

p i o n e e r

We will use the abbreviation LCS to mean "longest common subsequence".

This may seem like a pretty artificial problem, but it is part of a class of what are called *alignment problems*, which are extremely important in modern biology. Now that we can sequence the entire genomes of many organisms, we can use this information to deduce quite accurately how closely related different organisms are, and to infer the real "tree of life". Trees showing the evolutionary development of classes of organisms are called "phylogenetic trees", and they are computed by looking at the DNA sequences of different organisms and comparing them to see how close they are and where the differences are. A lot of this kind of comparison amounts to finding common subsequences, just as we are doing here. This is a tremendously exciting field.

So how do we solve this problem? Suppose we try the obvious approach: list all the subsequences of X and check each to see if it is a subsequence of Y, and pick the longest one that is.

There are 2^m subsequences of X. To check to see if a subsequence of X is also a subsequence of Y will take time $O(n)$. (Is this obvious?) Picking the longest one is really just an $O(1)$ job, since we can keep track as we proceed of the longest subsequence that we have found so far that works. So the cost of this method is $O(n2^m)$.

That's pretty awful. It's so bad, in fact, that it's completely useless. The strings that we are concerned with in biology have hundreds or thousands of elements *at least*. So we really need a better algorithm.

Fortunately, dynamic programming comes to our rescue. Let's see how:

2.1 Optimal substructure

Again, let us say we have two strings, with possibly different lengths:

$$X = \{x_1, x_2, \dots, x_m\}$$
$$Y = \{y_1, y_2, \dots, y_n\}$$

A *prefix* of a string is an initial segment. So we define for each i less than or equal to the length of the string the prefix of length i:

$$X_i = \{x_1, x_2, \dots, x_i\}$$
$$Y_i = \{y_1, y_2, \dots, y_i\}$$

Now the point of what we are going to prove is that a solution of our problem reflects itself in solutions of prefixes of X and Y.

2.1 Theorem *Let $Z = \{z_1, z_2, \dots, z_k\}$ be any LCS of X and Y.*

1. *If $x_m = y_n$, then $z_k = x_m = y_n$, and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.*
2. *If $x_m \neq y_n$, then $z_k \neq x_m \implies Z$ is an LCS of X_{m-1} and Y.*
3. *If $x_m \neq y_n$, then $z_k \neq y_n \implies Z$ is an LCS of X and Y_{n-1}.*

Remark *Please be careful about the notation in item 2. What follows the word "then" is an implication. It does **not** assert that $z_k \neq x_m$. What it says is that **if** $z_k \neq x_m$ **then** Z is an LCS of X_{m-1}. You might find it somewhat clearer if I wrote it like this:*

2. If $x_m \neq y_n$, then ($z_k \neq x_m \implies Z$ is an LCS of X_{m-1} and Y).

or equivalently,

2. If $x_m \neq y_n$ and $z_k \neq x_m$, then Z is an LCS of X_{m-1} and Y.

However, the way I originally wrote it above is quite standard, and you need to get used to it. The same comment applies to item 3.

PROOF.

1. By assumption $x_m = y_n$. If z_k does not equal this value, then Z must be a common subsequence of X_{m-1} and Y_{n-1}, and so the sequence

$$Z' = \{z_1, z_2, \dots, z_k, x_m\}$$

would be a common subsequence of X and Y. But this is a longer common subsequence than Z, and this is a contradiction.

2. If $z_k \neq x_m$, then Z must be a subsequence of X_{m-1}, and so it is a common subsequence of X_{m-1} and Y. If there were a longer one, then it would also be a common subsequence of X and Y, which would be a contradiction.

3. This is really the same as 2. □

Note that conclusions 2 and 3 of the Theorem could be summarized as follows:

2.2 Corollary *If $x_m \neq y_n$, then either*

- *Z is an LCS of X_{m-1} and Y, or*
- *Z is an LCS of X and Y_{n-1}.*

Thus, the LCS problem has what is called the *optimal substructure property*: a solution contains within it the solutions to subproblems—in this case, to subproblems constructed from prefixes of the original data. This is one of the two keys to the success of a dynamic programming solution.

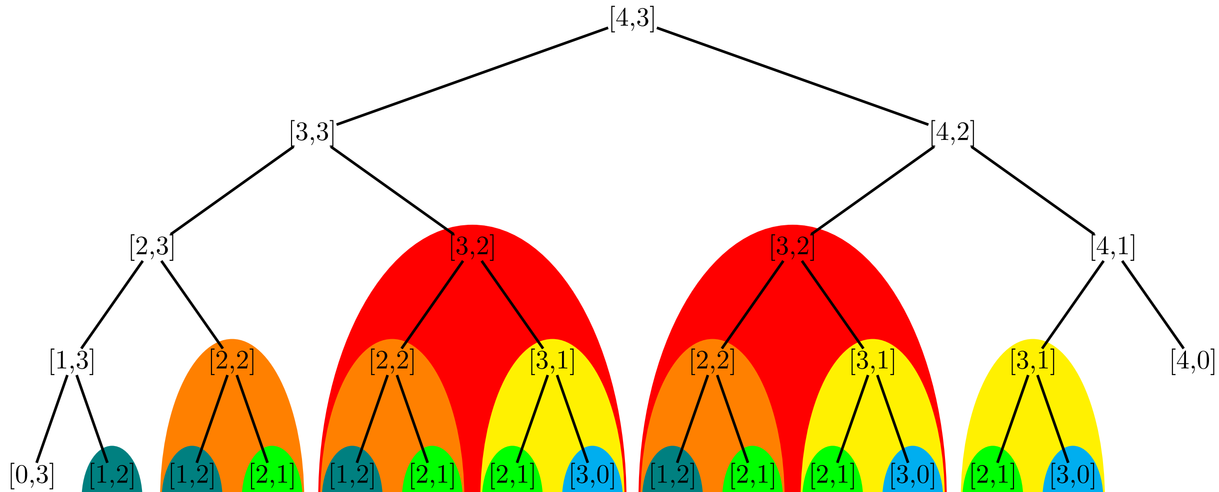
2.1.1 Recursive solution

Let $c[i, j]$ be the length of the LCS of X_i and Y_j. Based on Theorem 2.1, we can write the following recurrence:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

This is nice—the optimal substructure property allows us to write down an elegant recursive algorithm. The way this algorithm works is shown in a simplified form in the tree below.

Now one can see that the cost of this algorithm is still far too great—we can see that there are $\Omega(2^{\min\{m,n\}})$ nodes in the tree, which is still a killer. But at least we have an algorithm.



2.2 Overlapping subproblems

What saves us is that there really aren't that many *distinct* nodes in the tree. In fact, there are only $O(mn)$ distinct nodes. Its just that individual nodes tend to occur lots of times.

2.3 Exercise *Write pseudo-code for a memoized version of the recursive algorithm outlined above. Please note two things:*

1. *The recursive algorithm outlined above is a recursive mathematical formula. I am asking for pseudo-code.*
2. *What you need to write to do this problem is* not *the pseudo-code presented in the book. That's not recursive.*

2.4 Exercise *Show that the cost of doing all this is $O(mn)$.*

3 Optimal Binary Search Trees

Now we'll look at a third problem for which dynamic programming really serves us well.

Suppose we are making a binary search tree to use as a dictionary—say we are translating English to French, for example. So the key of each node in the tree will be an English word, and the contents of each node will be its French equivalent[3].

Now it's easy enough to make a binary search tree that does this—we've already seen how. But of course we want to make it as efficient as possible. In other words, we would like the average cost of a lookup to be as small as possible.

[3]to the extent that there is one, of course—in general, there is really no way to translate one natural language to another this way. We're just using this as an over-simplified example.

How can this be done? Intuitively, we would want the lookup paths to be as short as possible, so we might think that the thing to do is create a binary search tree that was as balanced as possible.

In this case, however, we have some additional information that changes the nature of the problem significantly: Some words are much less likely to be looked up than others.

So in general, we would expect that the words most often looked up should be near the top of the tree. It turns out that even with this notion, finding the best tree is not trivial.

Let us consider a simple example. Suppose we have a binary search tree containing 5 words. Here they are (symbolically, anyway), arranged in alphabetical order:

$$k_1 \quad k_2 \quad k_3 \quad k_4 \quad k_5$$

The following table shows the probabilities of searching for these different nodes:

i	1	2	3	4	5
p_i	0.25	0.20	0.05	0.20	0.30

where p_i is the probability of searching for the node k_i.

Of course we must have

$$\sum_{i=1}^{n} p_i = 1 \tag{1}$$

Suppose then we have created a binary search tree T holding these nodes, and as usual, let us denote by $depth_T(k_i)$ the depth of the node k_i in the tree T. Then the cost of looking up the node k_i in the tree is just

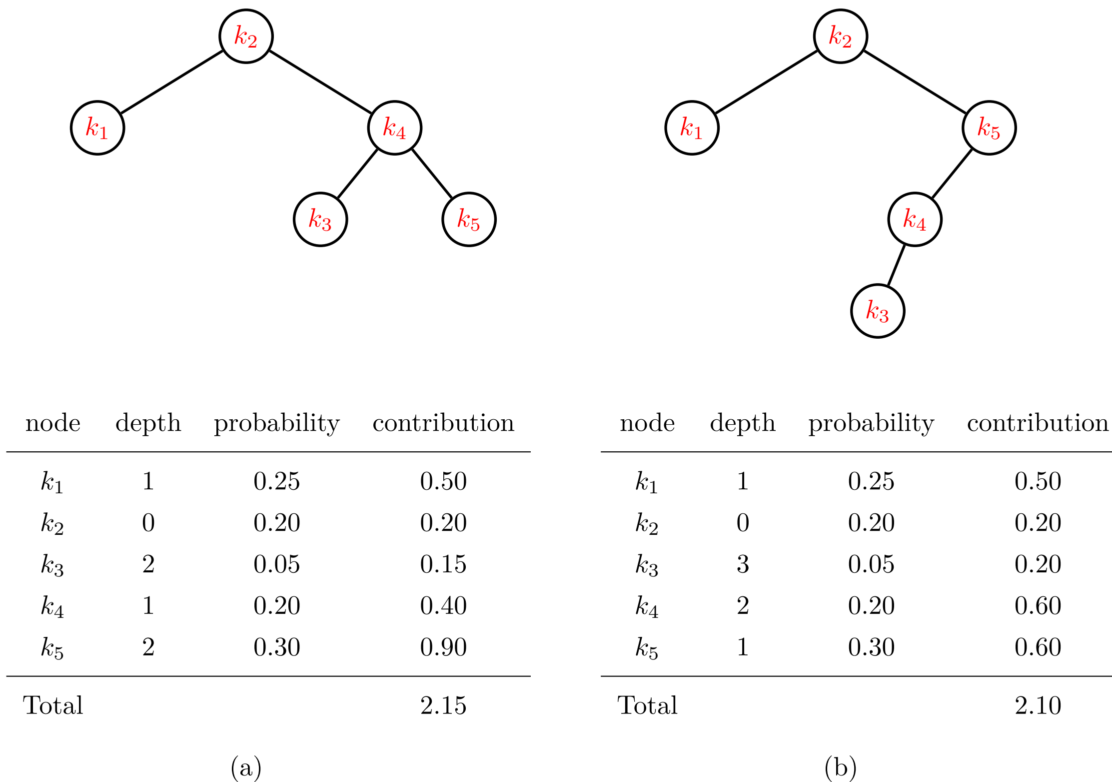
$$depth_T(k_i) + 1$$

and so the expected cost of looking up any node is the given in the usual way as the weighted average, which we will denote[4] by $E(T)$:

$$\begin{aligned} E(T) &= \sum_{i=1}^{n} p_i\big(depth_T(k_i) + 1\big) \\ &= \sum_{i=1}^{n} p_i \, depth_T(k_i) + \sum_{i=1}^{n} p_i \\ &= 1 + \sum_{i=1}^{n} p_i \, depth_T(k_i) \qquad \text{(since the probabilities sum to 1)} \end{aligned} \tag{2}$$

Figure 3 shows two possible binary search trees for this situation. For each tree T, we have shown the computation of the expected search cost $E(T)$.

[4]Perhaps this notation is a bit sloppy, but this is the way the book does it, and it's not really so bad.



node	depth	probability	contribution
k_1	1	0.25	0.50
k_2	0	0.20	0.20
k_3	2	0.05	0.15
k_4	1	0.20	0.40
k_5	2	0.30	0.90
Total			2.15

(a)

node	depth	probability	contribution
k_1	1	0.25	0.50
k_2	0	0.20	0.20
k_3	3	0.05	0.20
k_4	2	0.20	0.60
k_5	1	0.30	0.60
Total			2.10

(b)

Figure 3: Two binary search trees for a set of $n = 5$ keys.

We can see that the tree on the right, which actually has greater depth, has a smaller expected search cost. In fact, as we'll see below, this is the tree with the least expected search cost. And we also see that the node with greatest probability of being picked is not the root node. So we can see that putting nodes in the tree so that the nodes of higher probability occur higher up is not necessarily the best thing to do. So the problem is not a trivial one to solve at all.

We could of course examine each binary search tree on n nodes, compute its cost, and take the minimum. This is a "brute force" solution—that is, it is a solution based on exhaustive search—and it is completely impractical for the following reason: the number of binary trees with n nodes is

$$\frac{1}{n+1}\binom{2n}{n} = \frac{4^n}{\sqrt{\pi}n^{3/2}}\bigl(1 + O(1/n)\bigr)$$

which grows exponentially[5]. So certainly exhaustive search is not a useful way of finding the best tree in this problem. To do better, we need to rely on some special structural properties of the problem.

[5]This formula can be derived very nicely using a generating function.

3.1 Optimal substructure

Well as you might have guessed, the special structural property of this problem is optimal substructure. What does this mean in this case? Since we are dealing with trees, the substructures are naturally going to be subtrees.

The optimal substructure property that our problem possesses is this:

3.1 Theorem (Optimal substructure for the optimal binary search tree problem) *If T is an optimal binary search tree and if T' is any subtree of T, then T' is an optimal binary search tree for its nodes.*

The proof of this is a standard cut-and-paste argument. The idea is this: if T' were not an optimal binary search tree for its nodes, then we could replace it (in T) by a better one, and that would drive down the cost of the T itself, which would show that T was not optimal. And that would be a contradiction.

The idea is quite simple, but it is a little tricky to write out correctly. And in fact, all we really need (as you will see below) is to consider the case in which T' is a subtree rooted at one of the children of the root of T.

Note in such a case, if k_i is in T' (and so of course is also in T), then

$$depth_T(k_i) = 1 + depth_{T'}(k_i)$$

3.2 Exercise *Fill out the proof of this theorem in the case that T' is an immediate subtree of T—that is that the root of T' is a child of the root of T.*

You have to be careful when you do this. Remember that this is a question about a data structure, not about algorithms. So you can't say anything like "the algorithm would put this node here".

The idea is simple enough, but note that you will almost certainly be using the formula (2), and you will have to deal both with expressions like $depth_T(k_i)$ and $depth_{T'}(k_i)$ (and possibly other, similar, ones). I have already showed you how these expressions are related. You have to make this all explicit. It's not super-difficult, but you have to do it[6].

3.2 A recursive solution

As usual, we can use the optimal substructure property of this problem to construct a recursive algorithm to solve it. Here's how:

First, our data structures will be greatly simplified by the following bit of knowledge, which I am putting here as an exercise with an important hint:

3.3 Exercise *Prove that the keys in any subtree form a contiguous*[7] *sequence. For instance, the subtree of the tree (a) in Figure 3 whose root is k_4 consists of the contiguous sequence $\{k_3, k_4, k_5\}$ of keys.*

Here is the hint: You might find some of the ideas in Lecture 7 helpful. In particular, think of least common ancestors.

[6] For instance, if I don't see any references to the expressions $depth_T(k_i)$ and $depth_{T'}(k_i)$, then I know without even looking any farther that what you have written can't possibly be correct. And in addition, you have to write it up so I can understand it. I'm not a mind-reader. Please!

[7] Before you even start this exercise, make sure you know what the term "contiguous" means. It is *not* the same as the term "continuous".

Let us consider *any* subtree S of T, where T is assumed to be an optimal binary search tree for all our nodes. We know by Exercise 3.3 that the keys for the nodes in S are contiguous. Say they are $\{k_i, \ldots, k_j\}$. Simply as a matter of notation, we will then denote this subtree S by $T_{i,j}$. Of course the original tree T is just $T_{1,n}$.

Now given such a subtree $T_{i,j}$, let us define $e[i, j]$ to be the expected cost of searching an optimal binary search tree containing the keys $\{k_i, \ldots, k_j\}$. That is, $e[i, j]$ is just what we have already been calling $E(T_{i,j})$.

Suppose the optimal binary search tree $T_{i,j}$ for this subproblem has k_r as its root. Then we have essentially divided the cost of the problem into three parts:

- The expected cost of searching the tree $T_{i,r-1}$ built from the nodes $\{i, \ldots, r-1\}$, adjusted for the fact that this is a subtree of our original tree $T_{i,j}$ and so all the depths really should be 1 greater than they are in the subtree.
- The cost of searching for the root k_r.
- The expected cost of searching the tree $T_{r+1,j}$ built from the nodes $\{k_{r+1}, \ldots, k_j\}$, adjusted for the fact that this is a subtree of the original tree $T_{i,j}$, and so all the depths really should be 1 greater than they are in the subtree.

Note that r can take any of the values $\{i, \ldots, j\}$. If $r = i$ then the first subtree $T_{i,r-1}$ doesn't contain any keys.

Similarly, if $r = j$ then the second subtree $T_{r+1,j}$ doesn't contain any keys.

Now let us assume for the moment that $i \leq j$ (since we know that $e[i, j] = 0$ when $j = i - 1$).

Let us set

$$w(i, j) = \sum_{l=i}^{j} p_l$$

This is the sum of the probabilities of all the nodes in the tree $T_{i,j}$ built from the nodes $\{k_i, \ldots, k_j\}$. We know that $w(1, n) = 1$, but of course all we can say in general is that $0 \leq w(i, j) \leq 1$.

Now the first tree $T_{i,r-1}$ we mentioned above has cost $e[i, r-1]$, but as a subtree of $T_{i,r}$, its cost has to be increased by increasing each depth number in equation (2) by 1. This amounts to adding $w(i, r-1)$. So the expected cost that the subtree $T_{i,r-1}$ contributes to the expected cost of $T_{i,j}$ is $e[i, r-1] + w(i, r-1)$. A similar argument applies to the other subtree $T_{r+1,j}$. And so we get

$$\begin{aligned} e[i, j] &= E(T_{i,j}) \\ &= \big(E(T_{i,r-1}) + w(i, r-1)\big) + p_r + \big(E(T_{r+1,j}) + w(r+1, j)\big) \\ &= p_r + \big(e[i, r-1] + w(i, r-1)\big) + \big(e[r+1, j] + w(r+1, j)\big) \end{aligned}$$

This can be simplified a little: note that

$$w(i, j) = w(i, r-1) + p_r + w(r+1, j)$$

So we have

$$e[i,j] = w(i,j) + e[i,r-1] + e[r+1,j]$$

Now all this assumed that we knew which node k_r to pick for the root. Of course, we don't know, so we have to take the minimum over all possible choices of r. Thus we have

$$e[i,j] = \begin{cases} 0 & \text{if } j = i-1 \\ w(i,j) + \min_{i \le r \le j}\{e[i,r-1] + e[r+1,j]\} & \text{if } i \le j \end{cases}$$

This can be easily turned into a recursive procedure and will correctly compute $e[i,j]$ for any i and j. In particular, it will compute $e[1,n]$, so we can find the expected cost of searching an optimal binary search tree. And in fact, we could extend the algorithm easily enough to return the optimal binary search tree itself as well.

The only problem is, that this algorithm is still exponential in cost. We haven't really cut the cost down any—we have just managed to come up with an algorithm that has more structure to it than simple exhaustive search, but isn't really any cheaper.

3.3 Efficient computation of the expected search cost of an optimal binary search tree

The reason we can do better is that this problem also exhibits the property of *overlapping subproblems*. To be precise, look at what we are computing: There are only $O(n^2)$ values $e[i,j]$ with $1 \le i \le n+1$ and $0 \le j \le n$. (And in fact, we also have $j \ge i-1$, which cuts the total down by about a factor of 2.) These values would be computed over and over again in a naive recursive algorithm. So we can memoize them—we just store them in an array $e[1\,..\,n+1, 0\,..\,n]$ and that way we compute each value only once. We can also store the values $w(i,j)$ in a table $w[1\,..\,n+1, 0\,..\,n]$. We have

$$w[i,j] = \begin{cases} 0 & \text{if } j = i-1 \\ w[i,j-1] + p_j & \text{otherwise} \end{cases}$$

Thus there are $O(n^2)$ values of $w[i,j]$ and each one takes a constant time to compute, so the total cost of computing the w array is $O(n^2)$.

Then the cost of computing each value of $e[i,j]$ is $O(n)$ and there are $O(n^2)$ such values, so the cost of computing all the values of $e[i,j]$ is $O(n^3)$. So the total cost of computing the w array first and then the e array is

$$O(n^2) + O(n^3) = O(n^3)$$