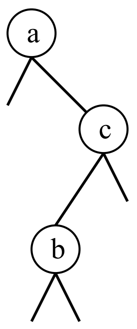
Optimal Binary Search Trees (and a second example of dynamic programming)**:** [ref: Cormen Leiserson, Rivest and Stein section 15.5]

We start with a "simple" problem regarding binary search trees in an environment in which the probabilities of accessing elements and gaps between elements is known. So, for example, we have the elements a, b and c; with probabilities of access .3, .08 and .12 respectively. The probabilities of an access for a value x<a is .25, in the range a<x<b is .05, and .1 for each of the other two "gaps". We want to find the tree that minimizes the expected number of nodes probed on a search. The solution to the case given is:



p(<a) = .25
p(a) = .3
p(a<x<b) = .05
p(b) = .08
p(b<x<c) = .1
p(c) = .12
p(>c) = .1

The expected cost (number of internal nodes inspected in a search) is

.25*1+.3*1+.05*3+.08*3+.1*3+.12*2+.15*2 = 1.78.

Other than by checking out all possible search trees (and there are close to 4^n of them on n nodes), how do we determine the optimal tree?

First, a few thoughts about the problem:

- A balanced tree would have cost Θ(lg n) if the weights are "reasonably balanced", but for an "uneven" distribution the expected search could be far less.
- The order of elements is given (fixed) and searches may end either at internal nodes or at leaves. The problem of finding a Huffman code (recall CS 240) is "similar", but all the nonzero probabilities are at the leaves, and we are free to modify the order of keys in any way we like. Nice ideas, but the change in requirements seems to make the approach for Huffman codes inappropriate.
- Putting the heaviest internal node at the root "clearly" won't always work. Even trying to get a 50-50 split around the root is not necessarily optimal. So it appears that "greedy" does not give us our optimal tree.
- Trying all possible trees is clearly unacceptable.

- If we get the root right, then the left subtree is the optimal tree on the values before the root and the right subtree is the optimal on the values after it! Dynamic programming!! (CS 341 !!!)

Model: Searches must start at root of a binary search tree. Charge for each node inspected.

Static optimal binary search tree: we start with easy an dynamic programming.

Given: p_i = prob. of access for $A_i\,(i = 1, n)$

q_i = prob. of access for value between A_i and $A_{i+1}\,(i = 0, n)$

$[p_0 = p_{n+1} = 0]$

root$[i, j]$ = root of optimal tree on range q_{i-1} to q_j

e$[i, j]$ = cost of tree rooted at *root*$[i, j]$; this cost is the probability of looking for one of the values (or gaps) in the range times the expected cost of doing in that tree.

Algorithm computes *root's* and *e*'s by increasing size, i.e. by increasing value of *(j-i)*.

So

$root[i,i] = i$ *[Initialization, i is the only key value in the range, so it must be the root]*

$e[i,i] = q_{i-1} + p_i + q_i$ [the probability of searching in this tree with one internal node]

If r = root; L = left subtree; R = right subtree;

W[tree] = probability of being in tree = probability of accessing root.

Then

C[tree rooted at r] = *W*[tree] + *e*[L] + *e*[R]

It will clearly be handy to have *W[i,j]*, the probability of accessing any node $q_{i-1}, \ldots, q_j$ or $p_i, \ldots, p_j$. So $W[i, j] = \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k$ These are easy to compute in $\Theta(n^2)$ time by computing $W[i, j+1]$ as $W[i, j] + p_{j+1} + q_{j+1}$.

In our description, if *root*$[i, j]$ is defined, take that value, otherwise compute it recursively as:

```
$root[i, j] = i$
$e[i, j] = W[i, j] + q_{i-1} + e[i+1, j]$ {initialize with i as root}
 for $r = i+1$ to $j$ do {if $r$ is a better root, take it .. this line will be editted later}
$ec = W[i,j] + e[i,r] + p_r + e[r+1,j]$
    if $ec < e[i,j]$ then
    begin
        $e[i, j] = ec$
        $root[i, j] = r$
    end
```

The approach of storing computed values and reusing them as required is known as *memoing*, if a value of $root[i,j]$ *or* $e[i,j]$ has been computed ... use it; otherwise compute it and remember it. Hence an $O(n^3)$ algorithm as each of the $O(n^2)$ values takes $O(n)$ time to compute given values on smaller ranges. It's polynomial time and space but not a great solution.

But one more thing – Do we have to check entire *i,j* range for a root?

Check just from $root[i, j-1]$ *to* $root[i+1, j]$

Lemma: $root[i, j]$ cannot be to the right of $root[i+1, j]$ (similarly not to the left of $root[i, j\text{-}1]$)

Proof: Induction on range size (Basis 2 node trees)

Hence key loop is "shorter" if we just go between the subtree roots. And indeed the runtime is improved to $O(n^2)$ [working through some series telescope … essentially

$$\sum_{i=1}^{n}(Cost_{i+1} - Cost_i) = Cost_{n+1} - Cost_0$$

Fact – after 30 years still best is $\Theta(n^2)$ time and space. There is no known polynomial time $\Theta(n^{2-\epsilon})$ space method.

How good is the tree? Clearly the expected cost can be as high as lg *n*.

Definition: The entropy of a probability distribution $x_1,..\ x_k$ is given by

$$H(x_1,\dots,x_k) = \sum_{i=1}^{k} x_i \lg(1/x_i)$$

Hence the entropy of our distribution is $H(p_1,\dots,p_n,q_0,\dots q_n) = \sum_{i=1}^{n} p_i \lg(1/p_i) + \sum_{i=0}^{n} q_i \lg(1/q_i)$

Theorem: The expected cost, *C*, of the optimal binary search tree satisfies the inequalities:

$$C \geq H - \lg H - \lg e + 1$$

and $$C \leq H + 3.$$

It is interesting to note these bounds can essentially be achieved for the same set of *p*'s and *q*'s by simply permuting the probabilities.

We also note that the optimal tree is expensive to compute and ask whether it can be approximated more cheaply. A natural greedy algorithm would be to put the key with the largest probability at the root. This can be a serious problem, even if all q_i values are 0. Suppose, for example, that all p_i values are virtually the same, but that $p_i > p_{i+1}$. Then the heuristic suggested will give a tree rooted at node 1, in which each node (except the last) has a right child but no left child. The search cost will be linear; whereas the optimal tree is balanced, with expected cost lg *n*.

A different approach is to try to balance the load at each node. That is to choose as root a node so that less than half the weight of the tree is in the left subtree and less than half is in the right. Clearly this is not always possible. For example, consider the case in which p_1=.2 and p_2=.2 and the *q* values are also .2. We must choose one of the internal nodes as root, so the weight of one subtree will be .6 while the other is .2. An achievable version of this approach is to choose as root the internal node that minimizes the weight of the largest subtree, or that minimizes the difference between the weights of the two subtrees. Ties are broken arbitrarily. Both approaches are effective. *The bounds quoted above for the optimal tree also apply to these approximate solutions.* Furthermore, while is may seem tat the minmax approach is the better of the two, it is easy to construct example in which this is not the case.

How long does it take to find this approximate solution … and how much space? A quadratic algorithm using linear space is obvious, and in practice, the quadratic space of the "optimal" algorithms is a bigger issue than the time. An O($n \lg n$) method is easy!

First compute $sum(i) = \Sigma_{j<i}\, q(j) + p(j)$. So the values $sum(i)$ are in nondecreasing order. A simple binary search (on this first step) finds the root, and we recurse. With linear preprocessing, the runtime of the main portion of the method satisfies the recurrence $T(n) = \lg n + T(j-1) + T(n-j)$, for some arbitrary j. If we are "lucky" and j is not too close to 0 or n-1, the runtime is linear; but we are inclined to think this will not be the case, or we wouldn't be that concerned about an optimal (rather than balanced) tree. If j=1 at each step, the method is clearly $\Theta(n \lg n)$.

The problem is that we are paying the same amount for a "bad split" as we are for a "good split". The solution is to pay O(lg j), or O(lg (n-j)) for the binary search. This is easy:

- First check the middle of the array section, now we have to search the first half or the last half.
- Assume without loss of generality that we are doing the first half, and start with position 1, doubling on each step until we have overshot. Complete the binary search in O(lg j) time.

It is not hard to show that the solution to $T(n) = \lg j + T(j-1) + T(n-j)$ is linear, here is a sketch.

Prove by induction that the number of comparisons is $an - b \lg n$. Cleary the base case works for appropriate a and b. The method takes, say $b \lg j$ to break into two parts, then $aj - b \lg j$ and $a(n-j) - b \lg (n-j)$, by the inductive hypothesis for the parts. Note:

$b \lg j + a(j-1) - b \lg (j-1) + a(n-j) - b \lg (n-j) < an - b \lg n$

There are a few issues to tidy up; these are left as an exercise.

Another example of Dynamic Programming

Consider the problem of parsing a string in a context free language (i.e. defined by a context grammar/ BNF grammar). It makes things simpler if we assume the grammar has either one terminal symbol on the right or two arbitrary symbols. This is called Chomsky Normal Form and it is easy to translate an arbitrary context free grammar into this form. For example, suppose we have the following grammar for all non-null strings with the same number of a's as b's.

S ::= ab | ba | SS | aSb | bSa

The last two productions are not in Chomsky Normal Form so we replace

S ::= aSb by S ::= aP and P ::= Sb, and handle bSa similarly. This gives the grammar:

S ::= ab | ba | SS | aP | bQ

P ::= Sb

Q::= Sa

The parsing method dates from the 1960's and is due, independently, to John Cocke, Dan Younger (of our C&O Department) and Tadao Kasami. It demonstrates that any context free language can be recognized in polynomial time. Note that virtually all programming languages have special features that permit much faster parsing; this method remains *almost* the fastest parsing technique for arbitrary context free languages. (V. Pratt showed how to solve the problem by matrix multiplication.)

As we are using dynamic programming, we will determine what symbols can generate the portion of our input from position i to position j, for all $j \geq i$. The string is in the language is the "start symbol", S, generates the entire input string. In our previous example we focused on a recursive approach, where we took an "answer" if it had already been computed, and worked it out otherwise. An equivalent approach, that takes less space (though still $\Theta(n^2)$) simply makes sure that all substrings of the portion under consideration have already been worked out.

Consider the n by n array whose position i,j contains a symbol if and only if that symbol generates that segment (So we need only the portion on or below the diagonal). We start by putting the input string on the diagonal (so the symbol in position i generate the substring of length 1 from i to i), this gives all the substring of length 1.

a									
	b								
		b							
			a						
				a					
					b				
						b			
							b		
								a	
									a

Continue on in turn on each diagonal line moving down (i.e. by increasing j-i+1), and include in position i,j the nonterminal symbol "T" if for some k ($i \leq k < j$) T ::= UV, where U is in position i,k and V is in position k+1,j. Part way through we have the following table:

a	S	P	S	Q	here				
	b	-	-	S	P				
		b	S	Q	S	P			
			a	-	-	-	-		
				a	S	P	-	-	
					b	-	-	-	-

						b	-	-	S
							b	S	Q
								a	-
									a

The next position is a bit more interesting, the substring of length 6 starting in the first position, abbaab, can be generated in 3 different ways:

ab baab	i.e. SS
abba ab	again SS
a bbaab	or aP

So we simply write S in this location, but note that we would realize there are different parses. In this example there is no case in which more than one symbol can generate the same subtstring, but this will happen in general. Continuing we get:

a	S	P	S	Q	S	P	-	-	S
	b	-	-	S	P	-	-	-	-
		b	S	Q	S	P	-	-	-
			a	-	-	-	-	-	-
				a	S	P	-	-	-
					b	-	-	-	-
						b	-	-	S
							b	S	Q
								a	-
									a

An S (with or without any other symbol) in the top right corner indicates the string is in the language. When we write a symbol in a location of the array, we could also include pointers to the pair of symbols from which it was derived. (For example in the bottom left, with the S we could include references to the S in row 1 column 2 and another to the S in row 3 column 10. In this way we can actually produce a parse tree (or all possible parse trees if it is ambiguous) for the string.