Matrix-chain multiplication problem: given a chain \( \langle A_1, A_2, \ldots, A_n \rangle \) of \( n \) matrices, where matrix \( A_i \) has dimension \( p_{i-1} \times p_i \), fully parenthesize the product \( A_1 A_2 \cdots A_n \) in a way that minimizes the number of scalar multiplications.

- We are not actually multiplying matrices.
- Our goal is only to determine an order for multiplying matrices that has the lowest cost.
- Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications.

Counting the number of parenthesizations

- Exhaustive checking of all possible parentheses combinations doesn’t yield an efficient algorithm.
- Let the number of alternative parenthesizations of a sequence of \( n \) matrices be \( P(n) \).
- When \( n = 1 \), we have just one matrix and only one way to fully parenthesize the matrix product.
- When \( n \geq 2 \), a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two may occur between the \( k \)th and \( (k + 1) \)st matrices \( \forall k \).
Thus, we obtain the recurrence
\[
P(n) = \begin{cases} 
1 & \text{if } n = 1 \\
\sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 
\end{cases}
\]

- We leave it as an exercise to show that the solution to the recurrence is \( \Omega(2^n) \)
- The number of solutions is exponential in \( n \)
- The brute-force method of exhaustive search makes for a poor strategy when determining how to optimally parenthesize a matrix chain

**Applying dynamic programming**

**Step 1: The structure of an optimal parenthesization**

- Let us adopt the notation \( A_{i..j} \), \( i \leq j \), for the matrix that results from evaluating the product \( A_iA_{i+1} \cdots A_j \)
- Observe that if \( i < j \) then to parenthesize the product \( A_iA_{i+1} \cdots A_j \), we must split the product between \( A_k \) and \( A_{k+1} \) for some \( i \leq k < j \)
- I.e., for some value of \( k \), we first compute the matrices \( A_{i..k} \) and \( A_{k+1..j} \) and then multiply them together to produce the final product \( A_{i..j} \)
- The cost: cost of computing the matrix $A_{i,k} + \text{cost of computing } A_{k+1,j} + \text{cost of multiplying them together}$

- The optimal substructure of this problem:
  - Suppose that to optimally parenthesize $A_i A_{i+1} \cdots A_j$, we split the product between $A_k$ and $A_{k+1}$.
  - Then the way we parenthesize $A_i A_{i+1} \cdots A_k$ within this optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \cdots A_k$.
  - If there were a less costly way to parenthesize $A_i A_{i+1} \cdots A_k$, then substitute that parenthesization in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ to produce a way to parenthesize $A_i A_{i+1} \cdots A_j$ with lower cost than the optimum: a contradiction.

- We can construct an optimal solution to the problem from optimal solutions to subproblems.
  - Solution to a nontrivial instance requires us to split the product, and any optimal solution contains within it optimal solutions to subproblem instances.
  - We can build an optimal solution by splitting the problem into two subproblems (optimally parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$), finding optimal solutions to these, and then combining these optimal subproblem solutions.
  - We must ensure that when we search for the correct place to split the product, we have considered all possible places, so that we are sure of having examined the optimal one.
Step 2: A recursive solution

- Next, we define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems.
- For the matrix-chain multiplication problem, we pick as our subproblems the problems of determining the minimum cost of parenthesizing $A_iA_{i+1}\cdots A_j$ for $1 \leq i \leq j \leq n$.
- Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$ for the full problem, the lowest-cost way to compute $A_{1..n}$ would thus be $m[1, n]$.

- We define $m[i, j]$ recursively as follows:
  - If $i = j$, no scalar multiplications are necessary.
  - Thus, $m[i, i] = 0$ for $i = 1, 2, \ldots, n$.
  - To compute $m[i, j]$, $i < j$, we take advantage of the structure of an optimal solution from step 1.
  - Assume that to optimally parenthesize, we split the product $A_iA_{i+1}\cdots A_j$ between $A_k$ and $A_{k+1}$.
  - Then, $m[i, j] = \text{the minimum cost for computing the subproducts } A_{i,k} \text{ and } A_{k+1,j}$, plus the cost of multiplying these two matrices together.
  - Recalling that each matrix $A_i$ is $p_{i-1} \times p_i$, we see that computing the matrix product $A_{i,k}A_{k+1,j}$ takes $p_{i-1}p_kp_j$ scalar multiplications.
Thus, we obtain

\[ m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j \]

This recursive equation assumes that we know the value of \( k \), which we do not.

There are only \( j - i \) possible values for \( k \), however, namely \( k = i, i + 1, \ldots, j - 1 \).

The optimal parenthesization must use one of these values, we need only check them all to find the best

Thus, our recursive definition for the minimum cost of parenthesizing the product \( A_iA_{i+1} \cdots A_j \) becomes

\[
m[i, j] = \begin{cases} 
0 & \text{if } i = j \\
\min_{i \leq k < j} \{ m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j \} & \text{if } i < j
\end{cases}
\]

The \( m[i, j] \) values give the costs of optimal solutions to subproblems, but they do not provide an optimal solution.

To help us do so, we define \( s[i, j] \) to be a value of \( k \) at which we split the product \( A_iA_{i+1} \cdots A_j \) in an optimal parenthesization.

That is, \( s[i, j] \) equals a value \( k \) such that

\[ m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j \]
Step 3: Computing the optimal costs

- Now we could easily write a recursive algorithm based on the recurrence to compute the minimum cost $m[1,n]$ for multiplying $A_1 A_2 \cdots A_n$
- This recursive algorithm takes exponential time
- It is no better than the brute-force method of checking each way of parenthesizing a product
- We have relatively few distinct subproblems: one subproblem for each choice of $i$ and $j$ satisfying $1 \leq i \leq j \leq n$, or $\binom{n}{2} + n = \Theta(n^2)$ in all
- We encounter each subproblem many times in different branches of its recursion tree

We implement the tabular, bottom-up method in MATRIX-CHAIN-ORDER, which assumes that matrix $A_i$ has dimensions $p_{i-1} \times p_i$
- Its input is a sequence $p = (p_0,p_1,\ldots,p_n)$, where $p.length = n + 1$
- The procedure uses
  - table $m[1..n,1..n]$ for storing the $m[i,j]$ costs and
  - another table $s[1..n-1,2..n]$ that records which index of $k$ achieved the optimal cost in computing $m[i,j]$
- We use the table $s$ to construct an optimal solution
By the recurrence the cost $m[i,j]$ of computing a matrix-chain product of $j - i + 1$ matrices depends only on the costs of computing matrix-chain products of $\leq j - i + 1$ matrices.

I.e., for $k = i, i + 1, \ldots, j - 1$, $A_{i,k}$ is a product of $k - i + 1 < j - i + 1$ matrices and $A_{k+1,j}$ is a product of $j - k < j - i + 1$ matrices.

Thus, we should fill in $m$ in a manner that corresponds to solving the problem on matrix chains of increasing length.

For $A_i A_{i+1} \cdots A_j$, we consider the subproblem size to be the length $j - i + 1$ of the chain.

---

**Matrix-Chain-Order** ($p$)

1. $n = p.length - 1$
2. let $m[1..n, 1..n]$ and $s[1..n - 1, 2..n]$ be new tables
3. for $i = 1$ to $n$
4.  $m[i,i] = 0$
5. for $l = 2$ to $n$  // $l$ is the chain length
6.  for $i = 1$ to $n - l + 1$
7.    $j = i + l - 1$
8.    $m[i,j] = \infty$
9.    for $k = i$ to $j - 1$
10.      $q = m[i,k] + m[k + 1,j] + p_{i-1}p_kp_j$
11.      if $q < m[i,j]$
12.          $m[i,j] = q$
13.          $s[i,j] = k$
14. return $m$ and $s$
The minimum number of scalar multiplications to multiply the 6 matrices is $m[1,6] = 15,125$

Of the darker entries, the pairs that have the same shading are taken together in line 10 when computing

\[ m[2,5] = \min \left\{ m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13,000, \right. \]
\[ \left. m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \right\} \]

The nested loop structure of MATRIX-CHAIN-ORDER yields a running time of $O(n^3)$ for the algorithm

- The loops are nested three deep, and each loop index ($l$, $i$, and $k$) takes on at most $n - 1$ values

- The running time of this algorithm is in fact also $\Omega(n^3)$

- The algorithm requires $\Theta(n^2)$ space to store the two tables

- MATRIX-CHAIN-ORDER is much more efficient than the exponential-time method of enumerating all possible parenthesizations and checking each one
Step 4: Constructing an optimal solution

- Table $s[1..n - 1, 2..n]$ gives us the information we need to multiply the matrices.
- Entry $s[i, j]$ records a value of $k$ s.t. an optimal parenthesization of $A_iA_{i+1} \cdots A_j$ splits the product between $A_k$ and $A_{k+1}$.
- Thus, we know that the final matrix multiplication in computing $A_{1..n}$ optimally is $A_{1..s[1,n]}A_{s[1,n]+1..n}$.
- $s[1, s[1,n]]$ determines the last matrix multiplication when computing $A_{1..s[1,n]}$ and $s[s[1,n] + 1, n]$ determines the last matrix multiplication when computing $A_{s[1,n]+1..n}$.

The following procedure prints an optimal parenthesization of $\langle A_i, A_{i+1}, \ldots, A_j \rangle$, given the $s$ table computed by MATRIX-CHAIN-ORDER and the indices $i$ and $j$.

The call \texttt{PRINT-OPTIMAL-PARENS}(s, 1, n) prints an optimal parenthesization of $\langle A_1, A_2, \ldots, A_n \rangle$.

\textbf{PRINT-OPTIMAL-PARENS}(s, i, j)
1. \textbf{if} $i == j$
2. \hspace{1em} print “$A_i$”
3. \textbf{else} print “(”
4. \hspace{1em} \textbf{PRINT-OPTIMAL-PARENS}(s, i, s[i, j])
5. \hspace{1em} \textbf{PRINT-OPTIMAL-PARENS}(s, s[i, j] + 1, j)
6. \hspace{1em} print “)”
15.4 Longest common subsequence

- Biological applications often need to compare the DNA of two (or more) different organisms.
- A strand of DNA consists of a string of molecules called **bases**, where the possible bases are Adenine, Guanine, Cytosine, and Thymine.
- We express a strand of DNA as a string over the alphabet \{A, C, G, T\}.
- E.g., the DNA of two organisms may be
  - $S_1 = \text{ACCGTCCGAGTCGGGAGCCGGCCGAA}$
  - $S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$

- By comparing two strands of DNA we determine how “similar” they are, as some measure of how closely related the two organisms are.
- We can define similarity in many different ways.
- E.g., we can say that two DNA strands are similar if one is a substring of the other.
  - Neither $S_1$ nor $S_2$ is a substring of the other.
- Alternatively, we could say that two strands are similar if the number of changes needed to turn one into the other is small.
Yet another way to measure the similarity of $S_1$ and $S_2$ is by finding a third strand $S_3$ in which the bases in $S_3$ appear in each of $S_1$ and $S_2$

- these bases must appear in the same order, but not necessarily consecutively

- The longer the strand $S_3$ we can find, the more similar $S_1$ and $S_2$ are

In our example, the longest strand $S_3$ is

- $S_1 = $ACCGGGTCGAGTGCAGGAAAGCCCAGCCGAA
- $S_2 = $GTCGTTCCGAAATGCCGTGCCTGGTAAA
- $S_3 = $GTCGTTCCGAAAGCCAGCCCGAA

We formalize this last notion of similarity as the longest-common-subsequence problem

A subsequence is just the given sequence with zero or more elements left out

Formally, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a subsequence of $X$ if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of $X$ such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$

For example, $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$
We say that a sequence $Z$ is a **common subsequence** of $X$ and $Y$ if $Z$ is a subsequence of both $X$ and $Y$

For example, if $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, the sequence $\langle B, C, A \rangle$ is a common subsequence of both $X$ and $Y$.

$\langle B, C, A \rangle$ is not the longest common subsequence (LCS) of $X$ and $Y$.

The sequence $\langle B, C, B, A \rangle$, which is also common to both $X$ and $Y$, has length 4.

This sequence is an LCS of $X$ and $Y$, as is $\langle B, D, A, B \rangle$; $X$ and $Y$ have no common subsequence of length 5 or greater.

In the **longest-common-subsequence problem**, we are given sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ and wish to find a maximum-length common subsequence of $X$ and $Y$.

**Step 1: Characterizing a longest common subsequence**

- In a brute-force approach, we would enumerate all subsequences of $X$ and check each of them to see whether it is also a subsequence of $Y$, keeping track of the longest subsequence we find.

- Each subsequence of $X$ corresponds to a subset of the indices $\langle 1, 2, \ldots, m \rangle$ of $X$.

- Because $X$ has $2^m$ subsequences, this approach requires exponential time, making it impractical for long sequences.
The LCS problem has an optimal-substructure property, however, as the following theorem shows.

We shall see that the natural classes of subproblems correspond to pairs of “prefixes” of the two input sequences.

Precisely, given a sequence $X = \langle x_1, x_2, ..., x_m \rangle$, we define the $i$th prefix of $X$, for $i = 0, 1, ..., m$, as $X_i = \langle x_1, x_2, ..., x_i \rangle$.

For example, if $X = \langle A, B, C, B, D, A, B \rangle$, then $X_4 = \langle A, B, C, B \rangle$ and $X_0$ is the empty sequence.

**Theorem 15.1** (Optimal substructure of LCS)

Let $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, ..., z_k \rangle$ 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 that $Z$ is an LCS of $X_{m-1}$ and $Y$.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z$ is an LCS of $X$ and $Y_{n-1}$.
Proof (1) If $z_k \neq x_m$, then we could append $x_m = y_n$ to $Z$ to obtain a common subsequence of $X$ and $Y$ of length $k + 1$, contradicting the supposition that $Z$ is a LCS of $X$ and $Y$. Thus, we must have $z_k = x_m = y_n$. Now, the prefix $Z_{k-1}$ is a length-$(k - 1)$ common subsequence of $X_{m-1}$ and $Y_{n-1}$. We wish to show that it is an LCS. Suppose for the purpose of contradiction that there exists a common subsequence $W$ of $X_{m-1}$ and $Y_{n-1}$ with length greater than $k - 1$. Then, appending $x_m = y_n$ to $W$ produces a common subsequence of $X$ and $Y$ whose length is greater than $k$, which is a contradiction.

(2) If $z_k \neq x_m$, then $Z$ is a common subsequence of $X_{m-1}$ and $Y$. If there were a common subsequence $W$ with length greater than $k$, then $W$ would also be a common subsequence of $X_m$ and $Y$, contradicting the assumption that $Z$ is an LCS of $X$ and $Y$.

(3) The proof is symmetric to (2). ■

- Theorem 15.1 tells us that an LCS of two sequences contains within it an LCS of prefixes of the two sequences
- Thus, the LCS problem has an optimal-substructure property
Step 2: A recursive solution

- We examine either one or two subproblems when finding an LCS of $X$ and $Y$
- If $x_m = y_n$, we find an LCS of $X_{m-1}$ and $Y_{n-1}$
- Appending $x_m = y_n$ yields an LCS of $X$ and $Y$
- If $x_m \neq y_n$, then we (1) find an LCS of $X_{m-1}$ and $Y$ and (2) find an LCS of $X$ and $Y_{n-1}$
- Whichever of these two LCSs is longer is an LCS of $X$ and $Y$
- These cases exhaust all possibilities, and we know that one of the optimal subproblem solutions must appear within an LCS of $X$ and $Y$

To find an LCS of $X$ and $Y$, we may need to find the LCSs of $X$ and $Y_{n-1}$ and of $X_{m-1}$ and $Y$

Each subproblem has the subsubproblem of finding an LCS of $X_{m-1}$ and $Y_{n-1}$

Many other subproblems share subsubproblems

As in the matrix-chain multiplication, recursive solution to the LCS problem involves a recurrence for the value of an optimal solution

Let us define $c[i, j]$ to be the length of an LCS of the sequences $X_i$ and $Y_j$

If either $i = 0$ or $j = 0$, one of the sequences has length 0, and so the LCS has length 0
The optimal substructure of the LCS problem gives
\[
c[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
\max(c[i - 1, j - 1], c[i, j - 1]) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\
\max(c[i - 1, j - 1], c[i - 1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j 
\end{cases}
\]

- Observe that a condition in the problem restricts which subproblems we may consider.
- When \(x_i = y_j\), we consider finding an LCS of \(X_{i-1}\) and \(Y_{j-1}\).
- Otherwise, we instead consider the two subproblems of finding an LCS of \(X_i\) and \(Y_{j-1}\) and of \(X_{i-1}\) and \(Y_j\).

In the previous dynamic-programming algorithms — for rod cutting and matrix-chain multiplication — we ruled out no subproblems due to conditions in the problem.

**Step 3: Computing the length of an LCS**

- Since the LCS problem has only \(\Theta(mn)\) distinct subproblems, we can use dynamic programming to compute the solutions bottom up.
- LCS-LENGTH stores the \(c[i, j]\) values in \(c[0..m, 0..n]\), and it computes the entries in *row-major* order — i.e., the procedure fills in the first row of \(c\) from left to right, then the second row, and so on.
- The procedure also maintains the table \(b[1..m, 1..n]\).
- Intuitively, \(b[i, j]\) points to the table entry corresponding to the optimal subproblem solution chosen when computing \(c[i, j]\).
- \(c[m, n]\) contains the length of an LCS of \(X\) and \(Y\).
LCS-LENGTH($X, Y$)

1. $m = X\. length$
2. $n = Y\. length$
3. let $b[1..m, 1..n]$ and $c[0..m, 0..n]$ be new tables
4. for $i = 1$ to $m$
5. $c[i, 0] = 0$
6. for $j = 0$ to $n$
7. $c[0,j] = 0$
8. for $i = 1$ to $m$
9. for $j = 1$ to $n$
10. if $x_i == y_j$
11. $c[i, j] = c[i-1, j-1] + 1$
12. $b[i, j] = \text{“}\&\text{“} $
13. elseif $c[i-1, j] \geq c[i, j-1]$
14. $c[i, j] = c[i-1, j]$
15. $b[i, j] = \text{“}\uparrow\text{“} $
16. else $c[i, j] = c[i, j-1]$
17. $b[i, j] = \text{“}\leftarrow\text{“} $
18. return $c$ and $b$

Running time: $\Theta(mn)$
Step 4: Constructing an LCS

- The $b$ table returned by LCS-LENGTH enables us to quickly construct an LCS of $X$ and $Y$.
- We simply begin at $b[m, n]$ and trace through the table by following the arrows.
- Whenever we encounter a “$\star$” in entry $b[i, j]$, it implies that $x_i = y_j$ is an element of the LCS that LCS-LENGTH found.
- With this method, we encounter the elements of this LCS in reverse order.
- A recursive procedure prints out an LCS of $X$ and $Y$ in the proper, forward order.

- The square in row $i$ and column $j$ contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$.
- The entry 4 in $c[7, 6]$ — the lower right-hand corner of the table — is the length of an LCS $\{B, C, B, A\}$.
- For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i - 1, j]$, $c[i, j - 1]$, and $c[i - 1, j - 1]$, which are computed before $c[i, j]$.
- To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner.
- Each “$\star$” on the shaded sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.
Improving the code

- Each $c[i,j]$ entry depends on only 3 other $c$ table entries: $c[i-1,j]$, $c[i,j-1]$, and $c[i-1,j-1]$
- Given the value of $c[i,j]$, we can determine in $O(1)$ time which of these three values was used to compute $c[i,j]$, without inspecting table $b$
- We can reconstruct an LCS in $O(m + n)$ time
- The auxiliary space requirement for computing an LCS does not asymptotically decrease, since we need $\theta(mn)$ space for the $c$ table anyway

- We can, however, reduce the asymptotic space requirements for LCS-LENGTH, since it needs only two rows of table $c$ at a time
  - the row being computed and the previous row
- This improvement works if we need only the length of an LCS
  - if we need to reconstruct the elements of an LCS, the smaller table does not keep enough information to retrace our steps in $O(m + n)$ time
15.5 Optimal binary search trees

- We are designing a program to translate text
- Perform lookup operations by building a BST with $n$ words as keys and their equivalents as satellite data
- We can ensure an $O(\log n)$ search time per occurrence by using a RBT or any other balanced BST
- A frequently used word may appear far from the root while a rarely used word appears near the root
- We want frequent words to be placed nearer the root
- How do we organize a BST so as to minimize the number of nodes visited in all searches, given that we know how often each word occurs?

- What we need is known as an **optimal binary search tree**
- Formally, given a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of $n$ distinct sorted keys ($k_1 < k_2 < \ldots < k_n$), we wish to build a BST from these keys
- For each key $k_i$, we have a probability $p_i$ that a search will be for $k_i$
- Some searches may be for values not in $K$, so we also have $n + 1$ “dummy keys” $d_0, d_1, \ldots, d_n$ representing values not in $K$
- In particular, $d_0$ represents all values less than $k_1$, $d_n$ represents all values greater than $k_n$
For \( i = 1, 2, \ldots, n - 1 \), the dummy key \( d_i \) represents all values between \( k_i \) and \( k_{i+1} \).

For each dummy key \( d_i \), we have a probability \( q_i \) that a search will correspond to \( d_i \).

Each key \( k_i \) is an internal node, and each dummy key \( d_i \) is a leaf.

Every search is either successful (finds a key \( k_i \)) or unsuccessful (finds a dummy key \( d_i \)), and so we have

\[
\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1
\]

Because we have probabilities of searches for each key and each dummy key, we can determine the expected cost of a search in a given BST \( T \).
• Let us assume that the actual cost of a search equals the number of nodes examined, i.e., the depth of the node found by the search in $T + 1$.
• Then the expected cost of a search in $T$, $E_{\text{search cost in } T} =$
\[
\sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i
\]
\[= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i,
\]
• where $\text{depth}_T$ denotes a node’s depth in the tree $T$.

<table>
<thead>
<tr>
<th>Node</th>
<th>Depth</th>
<th>Probability</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_1$</td>
<td>1</td>
<td>0.15</td>
<td>0.30</td>
</tr>
<tr>
<td>$k_2$</td>
<td>0</td>
<td>0.10</td>
<td>0.10</td>
</tr>
<tr>
<td>$k_3$</td>
<td>2</td>
<td>0.05</td>
<td>0.15</td>
</tr>
<tr>
<td>$k_4$</td>
<td>1</td>
<td>0.10</td>
<td>0.20</td>
</tr>
<tr>
<td>$k_5$</td>
<td>2</td>
<td>0.20</td>
<td>0.60</td>
</tr>
<tr>
<td>$d_0$</td>
<td>2</td>
<td>0.05</td>
<td>0.15</td>
</tr>
<tr>
<td>$d_1$</td>
<td>2</td>
<td>0.10</td>
<td>0.30</td>
</tr>
<tr>
<td>$d_2$</td>
<td>3</td>
<td>0.05</td>
<td>0.20</td>
</tr>
<tr>
<td>$d_3$</td>
<td>3</td>
<td>0.05</td>
<td>0.20</td>
</tr>
<tr>
<td>$d_4$</td>
<td>3</td>
<td>0.05</td>
<td>0.20</td>
</tr>
<tr>
<td>$d_5$</td>
<td>3</td>
<td>0.10</td>
<td>0.40</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td><strong>2.80</strong></td>
</tr>
</tbody>
</table>
For a given set of probabilities, we wish to construct a BST whose expected search cost is smallest. We call such a tree an **optimal binary search tree**. An optimal BST for the probabilities given has expected cost 2.75. An optimal BST is not necessarily a tree whose overall height is smallest. Nor can we necessarily construct an optimal BST by always putting the key with the greatest probability at the root. The lowest expected cost of any BST with $k_5$ at the root is 2.85.