Exercises

5.1. Consider the following graph.

(a) What is the cost of its minimum spanning tree?

(b) How many minimum spanning trees does it have?

(c) Suppose Kruskal’s algorithm is run on this graph. In what order are the edges added to the MST? For each edge in this sequence, give a cut that justifies its addition.

5.2. Suppose we want to find the minimum spanning tree of the following graph.

(a) Run Prim’s algorithm; whenever there is a choice of nodes, always use alphabetic ordering (e.g., start from node A). Draw a table showing the intermediate values of the cost array.

(b) Run Kruskal’s algorithm on the same graph. Show how the disjoint-sets data structure looks at every intermediate stage (including the structure of the directed trees), assuming path compression is used.

5.3. Design a linear-time algorithm for the following task.

*Input:* A connected, undirected graph $G$.

*Question:* Is there an edge you can remove from $G$ while still leaving $G$ connected?

Can you reduce the running time of your algorithm to $O(|V|)$?

5.4. Show that if an undirected graph with $n$ vertices has $k$ connected components, then it has at least $n - k$ edges.

5.5. Consider an undirected graph $G = (V, E)$ with nonnegative edge weights $w_e \geq 0$. Suppose that you have computed a minimum spanning tree of $G$, and that you have also computed shortest paths to all nodes from a particular node $s \in V$.

Now suppose each edge weight is increased by 1: the new weights are $w'_e = w_e + 1$.

(a) Does the minimum spanning tree change? Give an example where it changes or prove it cannot change.

(b) Do the shortest paths change? Give an example where they change or prove they cannot change.
5.6. Let $G = (V, E)$ be an undirected graph. Prove that if all its edge weights are distinct, then it has a unique minimum spanning tree.

5.7. Show how to find the maximum spanning tree of a graph, that is, the spanning tree of largest total weight.

5.8. Suppose you are given a weighted graph $G = (V, E)$ with a distinguished vertex $s$ and where all edge weights are positive and distinct. Is it possible for a tree of shortest paths from $s$ and a minimum spanning tree in $G$ to not share any edges? If so, give an example. If not, give a reason.

5.9. The following statements may or may not be correct. In each case, either prove it (if it is correct) or give a counterexample (if it isn’t correct). Always assume that the graph $G = (V, E)$ is undirected. Do not assume that edge weights are distinct unless this is specifically stated.

(a) If graph $G$ has more than $|V| - 1$ edges, and there is a unique heaviest edge, then this edge cannot be part of a minimum spanning tree.

(b) If $G$ has a cycle with a unique heaviest edge $e$, then $e$ cannot be part of any MST.

(c) Let $e$ be any edge of minimum weight in $G$. Then $e$ must be part of some MST.

(d) If the lightest edge in a graph is unique, then it must be part of every MST.

(e) If $e$ is part of some MST of $G$, then it must be a lightest edge across some cut of $G$.

(f) If $G$ has a cycle with a unique lightest edge $e$, then $e$ must be part of every MST.

(g) The shortest-path tree computed by Dijkstra’s algorithm is necessarily an MST.

(h) The shortest path between two nodes is necessarily part of some MST.

(i) Prim’s algorithm works correctly when there are negative edges.

(j) (For any $r > 0$, define an $r$-path to be a path whose edges all have weight $< r$.) If $G$ contains an $r$-path from node $s$ to $t$, then every MST of $G$ must also contain an $r$-path from node $s$ to node $t$.

5.10. Let $T$ be an MST of graph $G$. Given a connected subgraph $H$ of $G$, show that $T \cap H$ is contained in some MST of $H$.

5.11. Give the state of the disjoint-sets data structure after the following sequence of operations, starting from singleton sets $\{1\}, \ldots, \{8\}$. Use path compression. In case of ties, always make the lower numbered root point to the higher numbered one.

\[
\text{union}(1, 2), \text{union}(3, 4), \text{union}(5, 6), \text{union}(7, 8), \text{union}(1, 4), \text{union}(6, 7), \text{union}(4, 5), \text{find}(1)
\]

5.12. Suppose you implement the disjoint-sets data structure using union-by-rank but not path compression. Give a sequence of $m$ union and find operations on $n$ elements that take $\Omega(m \log n)$ time.

5.13. A long string consists of the four characters $A, C, G, T$; they appear with frequency 31%, 20%, 9%, and 40%, respectively. What is the Huffman encoding of these four characters?

5.14. Suppose the symbols $a, b, c, d, e$ occur with frequencies $1/2, 1/4, 1/8, 1/16, 1/16, 1/16$, respectively.

(a) What is the Huffman encoding of the alphabet?

(b) If this encoding is applied to a file consisting of 1,000,000 characters with the given frequencies, what is the length of the encoded file in bits?
5.15. We use Huffman’s algorithm to obtain an encoding of alphabet \{a, b, c\} with frequencies \(f_a, f_b, f_c\). In each of the following cases, either give an example of frequencies \(f_a, f_b, f_c\) that would yield the specified code, or explain why the code cannot possibly be obtained (no matter what the frequencies are).

(a) Code: \{0, 10, 11\}
(b) Code: \{0, 1, 00\}
(c) Code: \{10, 01, 00\}

5.16. Prove the following two properties of the Huffman encoding scheme.

(a) If some character occurs with frequency more than \(2/5\), then there is guaranteed to be a codeword of length 1.
(b) If all characters occur with frequency less than \(1/3\), then there is guaranteed to be no codeword of length 1.

5.17. Under a Huffman encoding of \(n\) symbols with frequencies \(f_1, f_2, \ldots, f_n\), what is the longest a codeword could possibly be? Give an example set of frequencies that would produce this case.

5.18. The following table gives the frequencies of the letters of the English language (including the blank for separating words) in a particular corpus.

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>blank</td>
<td>18.3%</td>
</tr>
<tr>
<td>e</td>
<td>10.2%</td>
</tr>
<tr>
<td>r</td>
<td>4.8%</td>
</tr>
<tr>
<td>t</td>
<td>7.7%</td>
</tr>
<tr>
<td>y</td>
<td>1.6%</td>
</tr>
<tr>
<td>a</td>
<td>6.8%</td>
</tr>
<tr>
<td>i</td>
<td>5.9%</td>
</tr>
<tr>
<td>n</td>
<td>5.5%</td>
</tr>
<tr>
<td>s</td>
<td>5.1%</td>
</tr>
<tr>
<td>h</td>
<td>4.9%</td>
</tr>
<tr>
<td>d</td>
<td>3.5%</td>
</tr>
<tr>
<td>l</td>
<td>3.4%</td>
</tr>
<tr>
<td>p</td>
<td>1.6%</td>
</tr>
<tr>
<td>b</td>
<td>1.3%</td>
</tr>
<tr>
<td>c</td>
<td>2.6%</td>
</tr>
<tr>
<td>v</td>
<td>0.9%</td>
</tr>
<tr>
<td>k</td>
<td>0.6%</td>
</tr>
<tr>
<td>j</td>
<td>0.2%</td>
</tr>
<tr>
<td>x</td>
<td>0.2%</td>
</tr>
<tr>
<td>q</td>
<td>0.1%</td>
</tr>
<tr>
<td>z</td>
<td>0.1%</td>
</tr>
</tbody>
</table>

(a) What is the optimum Huffman encoding of this alphabet?
(b) What is the expected number of bits per letter?
(c) Suppose now that we calculate the entropy of these frequencies

\[ H = \sum_{i=0}^{26} p_i \log \frac{1}{p_i} \]

(see the box in page 156). Would you expect it to be larger or smaller than your answer above? Explain.
(d) Do you think that this is the limit of how much English text can be compressed? What features of the English language, besides letters and their frequencies, should a better compression scheme take into account?

5.19. Entropy. Consider a distribution over \(n\) possible outcomes, with probabilities \(p_1, p_2, \ldots, p_n\).
(a) Just for this part of the problem, assume that each \( p_i \) is a power of 2 (that is, of the form \( 1/2^k \)). Suppose a long sequence of \( m \) samples is drawn from the distribution and that for all \( 1 \leq i \leq n \), the \( i^{th} \) outcome occurs exactly \( mp_i \) times in the sequence. Show that if Huffman encoding is applied to this sequence, the resulting encoding will have length

\[
\sum_{i=1}^{n} mp_i \log \frac{1}{p_i}.
\]

(b) Now consider arbitrary distributions—that is, the probabilities \( p_i \) are not restricted to powers of 2. The most commonly used measure of the amount of randomness in the distribution is the entropy

\[
\sum_{i=1}^{n} p_i \log \frac{1}{p_i}.
\]

For what distribution (over \( n \) outcomes) is the entropy the largest possible? The smallest possible?

5.20. Give a linear-time algorithm that takes as input a tree and determines whether it has a perfect matching: a set of edges that touches each node exactly once.

A feedback edge set of an undirected graph \( G = (V, E) \) is a subset of edges \( E' \subseteq E \) that intersects every cycle of the graph. Thus, removing the edges \( E' \) will render the graph acyclic.

Give an efficient algorithm for the following problem:

Input: Undirected graph \( G = (V, E) \) with positive edge weights \( w_e \)
Output: A feedback edge set \( E' \subseteq E \) of minimum total weight \( \sum_{e \in E'} w_e \)

5.21. In this problem, we will develop a new algorithm for finding minimum spanning trees. It is based upon the following property:

Pick any cycle in the graph, and let \( e \) be the heaviest edge in that cycle. Then there is a minimum spanning tree that does not contain \( e \).

(a) Prove this property carefully.

(b) Here is the new MST algorithm. The input is some undirected graph \( G = (V, E) \) (in adjacency list format) with edge weights \( \{w_e\} \).

\[
\text{sort the edges according to their weights}
\]

for each edge \( e \in E \), in decreasing order of \( w_e \):
if \( e \) is part of a cycle of \( G \):
\( G = G - e \) (that is, remove \( e \) from \( G \))
return \( G \)

Prove that this algorithm is correct.

(c) On each iteration, the algorithm must check whether there is a cycle containing a specific edge \( e \). Give a linear-time algorithm for this task, and justify its correctness.

(d) What is the overall time taken by this algorithm, in terms of \(|E|\)? Explain your answer.
5.22. You are given a graph $G = (V, E)$ with positive edge weights, and a minimum spanning tree $T = (V, E')$ with respect to these weights; you may assume $G$ and $T$ are given as adjacency lists. Now suppose the weight of a particular edge $e \in E$ is modified from $w(e)$ to a new value $\tilde{w}(e)$. You wish to quickly update the minimum spanning tree $T$ to reflect this change, without recomputing the entire tree from scratch. There are four cases. In each case give a linear-time algorithm for updating the tree.

(a) $e \notin E'$ and $\tilde{w}(e) > w(e)$.
(b) $e \notin E'$ and $\tilde{w}(e) < w(e)$.
(c) $e \in E'$ and $\tilde{w}(e) < w(e)$.
(d) $e \in E'$ and $\tilde{w}(e) > w(e)$.

5.23. Sometimes we want light spanning trees with certain special properties. Here's an example.

**Input:** Undirected graph $G = (V, E)$; edge weights $w_e$; subset of vertices $U \subset V$

**Output:** The lightest spanning tree in which the nodes of $U$ are leaves (there might be other leaves in this tree as well).

(The answer isn't necessarily a minimum spanning tree.)

Give an algorithm for this problem which runs in $O(|E| \log |V|)$ time. (Hint: When you remove nodes $U$ from the optimal solution, what is left?)

5.24. A binary counter of unspecified length supports two operations: increment (which increases its value by one) and reset (which sets its value back to zero). Show that, starting from an initially zero counter, any sequence of $n$ increment and reset operations takes time $O(n)$; that is, the amortized time per operation is $O(1)$.

5.25. Here’s a problem that occurs in automatic program analysis. For a set of variables $x_1, \ldots, x_n$, you are given some equality constraints, of the form “$x_i = x_j$” and some disequality constraints, of the form “$x_i \neq x_j$.” Is it possible to satisfy all of them?

For instance, the constraints

$$x_1 = x_2, x_2 = x_3, x_3 = x_4, x_1 \neq x_4$$

cannot be satisfied. Give an efficient algorithm that takes as input $m$ constraints over $n$ variables and decides whether the constraints can be satisfied.

5.26. Graphs with prescribed degree sequences. Given a list of $n$ positive integers $d_1, d_2, \ldots, d_n$, we want to efficiently determine whether there exists an undirected graph $G = (V, E)$ whose nodes have degrees precisely $d_1, d_2, \ldots, d_n$. That is, if $V = \{v_1, \ldots, v_n\}$, then the degree of $v_i$ should be exactly $d_i$. We call $(d_1, \ldots, d_n)$ the **degree sequence** of $G$. This graph $G$ should not contain self-loops (edges with both endpoints equal to the same node) or multiple edges between the same pair of nodes.

(a) Give an example of $d_1, d_2, d_3, d_4$ where all the $d_i \leq 3$ and $d_1 + d_2 + d_3 + d_4$ is even, but for which no graph with degree sequence $(d_1, d_2, d_3, d_4)$ exists.

(b) Suppose that $d_1 \geq d_2 \geq \cdots \geq d_n$ and that there exists a graph $G = (V, E)$ with degree sequence $(d_1, \ldots, d_n)$. We want to show that there must exist a graph that has this degree sequence and where in addition the neighbors of $v_1$ are $v_2, v_3, \ldots, v_{d_1+1}$. The idea is to gradually transform $G$ into a graph with the desired additional property.

i. Suppose the neighbors of $v_1$ in $G$ are not $v_2, v_3, \ldots, v_{d_1+1}$. Show that there exists $i < j \leq n$ and $u \in V$ such that $\{v_1, v_i\}, \{u, v_j\} \notin E$ and $\{v_1, v_i\}, \{u, v_i\} \in E$. 

ii. Specify the changes you would make to $G$ to obtain a new graph $G' = (V, E')$ with the same degree sequence as $G$ and where $(v_1, v_i) \in E'$.

iii. Now show that there must be a graph with the given degree sequence but in which $v_1$ has neighbors $v_2, v_3, \ldots, v_{d_1+1}$.

(c) Using the result from part (b), describe an algorithm that on input $d_1, \ldots, d_n$ (not necessarily sorted) decides whether there exists a graph with this degree sequence. Your algorithm should run in time polynomial in $n$ and in $m = \sum_{i=1}^{n} d_i$.

5.27. Alice wants to throw a party and is deciding whom to call. She has $n$ people to choose from, and she has made up a list of which pairs of these people know each other. She wants to pick as many people as possible, subject to two constraints: at the party, each person should have at least five other people whom they know and five other people whom they don’t know.

Give an efficient algorithm that takes as input the list of $n$ people and the list of pairs who know each other and outputs the best choice of party invitees. Give the running time in terms of $n$.

5.28. A prefix-free encoding of a finite alphabet $\Gamma$ assigns each symbol in $\Gamma$ a binary codeword, such that no codeword is a prefix of another codeword.

Show that such an encoding can be represented by a full binary tree in which each leaf corresponds to a unique element of $\Gamma$, whose codeword is generated by the path from the root to that leaf (interpreting a left branch as 0 and a right branch as 1).

5.29. Ternary Huffman. Trimedia Disks Inc. has developed “ternary” hard disks. Each cell on a disk can now store values 0, 1, or 2 (instead of just 0 or 1). To take advantage of this new technology, provide a modified Huffman algorithm for compressing sequences of characters from an alphabet of size $n$, where the characters occur with known frequencies $f_1, f_2, \ldots, f_n$. Your algorithm should encode each character with a variable-length codeword over the values 0, 1, 2 such that no codeword is a prefix of another codeword and so as to obtain the maximum possible compression. Prove that your algorithm is correct.

5.30. The basic intuition behind Huffman’s algorithm, that frequent blocks should have short encodings and infrequent blocks should have long encodings, is also at work in English, where typical words like I, you, is, and, to, from, and so on are short, and rarely used words like velociraptor are longer.

However, words like fire!, help!, and run! are short not because they are frequent, but perhaps because time is precious in situations where they are used.

To make things theoretical, suppose we have a file composed of $m$ different words, with frequencies $f_1, \ldots, f_m$. Suppose also that for the $i$th word, the cost per bit of encoding is $c_i$. Thus, if we find a prefix-free code where the $i$th word has a codeword of length $l_i$, then the total cost of the encoding will be $\sum_i f_i \cdot c_i \cdot l_i$.

Show how to modify Huffman’s algorithm to find the prefix-free encoding of minimum total cost.

5.31. A server has $n$ customers waiting to be served. The service time required by each customer is known in advance: it is $t_i$ minutes for customer $i$. So if, for example, the customers are served in order of increasing $i$, then the $i$th customer has to wait $\sum_{j=1}^{i} t_j$ minutes.

We wish to minimize the total waiting time

$$T = \sum_{i=1}^{n} (\text{time spent waiting by customer } i).$$

Give an efficient algorithm for computing the optimal order in which to process the customers.
5.32. Show how to implement the stingy algorithm for Horn formula satisfiability (Section 5.3) in time that is linear in the length of the formula (the number of occurrences of literals in it). (*Hint:* Use a directed graph, with one node per variable, to represent the implications.)

5.33. Show that for any integer $n$ that is a power of 2, there is an instance of the set cover problem (Section 5.4) with the following properties:

   i. There are $n$ elements in the base set.
   ii. The optimal cover uses just two sets.
   iii. The greedy algorithm picks at least $\log n$ sets.

Thus the approximation ratio we derived in the chapter is tight.
Chapter 6

Dynamic programming

In the preceding chapters we have seen some elegant design principles—such as divide-and-conquer, graph exploration, and greedy choice—that yield definitive algorithms for a variety of important computational tasks. The drawback of these tools is that they can only be used on very specific types of problems. We now turn to the two 

sledgehammers

of the algorithms craft, dynamic programming and linear programming, techniques of very broad applicability that can be invoked when more specialized methods fail. Predictably, this generality often comes with a cost in efficiency.

6.1 Shortest paths in dags, revisited

At the conclusion of our study of shortest paths (Chapter 4), we observed that the problem is especially easy in directed acyclic graphs (dags). Let’s recapitulate this case, because it lies at the heart of dynamic programming.

The special distinguishing feature of a dag is that its nodes can be linearized; that is, they can be arranged on a line so that all edges go from left to right (Figure 6.1). To see why this helps with shortest paths, suppose we want to figure out distances from node $S$ to the other nodes. For concreteness, let’s focus on node $D$. The only way to get to it is through its

**Figure 6.1** A dag and its linearization (topological ordering).
predecessors, $B$ or $C$; so to find the shortest path to $D$, we need only compare these two routes:

$$\text{dist}(D) = \min\{\text{dist}(B) + 1, \text{dist}(C) + 3\}.$$ 

A similar relation can be written for every node. If we compute these dist values in the left-to-right order of Figure 6.1, we can always be sure that by the time we get to a node $v$, we already have all the information we need to compute $\text{dist}(v)$. We are therefore able to compute all distances in a single pass:

- initialize all $\text{dist}(\cdot)$ values to $\infty$
- $\text{dist}(s) = 0$
- for each $v \in V \setminus \{s\}$, in linearized order:
  $$\text{dist}(v) = \min_{(u,v) \in E}\{\text{dist}(u) + l(u,v)\}$$

Notice that this algorithm is solving a collection of subproblems, $\{\text{dist}(u) : u \in V\}$. We start with the smallest of them, $\text{dist}(s)$, since we immediately know its answer to be 0. We then proceed with progressively “larger” subproblems—distances to vertices that are further and further along in the linearization—where we are thinking of a subproblem as large if we need to have solved a lot of other subproblems before we can get to it.

This is a very general technique. At each node, we compute some function of the values of the node’s predecessors. It so happens that our particular function is a minimum of sums, but we could just as well make it a maximum, in which case we would get longest paths in the dag. Or we could use a product instead of a sum inside the brackets, in which case we would end up computing the path with the smallest product of edge lengths.

*Dynamic programming* is a very powerful algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling them one by one, smallest first, using the answers to small problems to help figure out larger ones, until the whole lot of them is solved. In dynamic programming we are not given a dag; the dag is *implicit*. Its nodes are the subproblems we define, and its edges are the dependencies between the subproblems: if to solve subproblem $B$ we need the answer to subproblem $A$, then there is a (conceptual) edge from $A$ to $B$. In this case, $A$ is thought of as a smaller subproblem than $B$—and it will always be smaller, in an obvious sense.

But it’s time we saw an example.

### 6.2 Longest increasing subsequences

In the *longest increasing subsequence* problem, the input is a sequence of numbers $a_1, \ldots, a_n$. A *subsequence* is any subset of these numbers taken in order, of the form $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, and an *increasing* subsequence is one in which the numbers are getting strictly larger. The task is to find the increasing subsequence of greatest length. For instance, the longest increasing subsequence of 5, 2, 8, 6, 3, 6, 9, 7 is 2, 3, 6, 9:
In this example, the arrows denote transitions between consecutive elements of the optimal solution. More generally, to better understand the solution space, let's create a graph of all permissible transitions: establish a node \( i \) for each element \( a_i \), and add directed edges \((i, j)\) whenever it is possible for \( a_i \) and \( a_j \) to be consecutive elements in an increasing subsequence, that is, whenever \( i < j \) and \( a_i < a_j \) (Figure 6.2).

Notice that (1) this graph \( G = (V, E) \) is a dag, since all edges \((i, j)\) have \( i < j \), and (2) there is a one-to-one correspondence between increasing subsequences and paths in this dag. Therefore, our goal is simply to find the longest path in the dag.

Here is the algorithm:

\[
\text{for } j = 1, 2, \ldots, n: \\
L(j) = 1 + \max \{ L(i) : (i, j) \in E \} \\
\text{return } \max_j L(j)
\]

\( L(j) \) is the length of the longest path—the longest increasing subsequence—ending at \( j \) (plus 1, since strictly speaking we need to count nodes on the path, not edges). By reasoning in the same way as we did for shortest paths, we see that any path to node \( j \) must pass through one of its predecessors, and therefore \( L(j) \) is 1 plus the maximum \( L(\cdot) \) value of these predecessors. If there are no edges into \( j \), we take the maximum over the empty set, zero. And the final answer is the largest \( L(j) \), since any ending position is allowed.

This is dynamic programming. In order to solve our original problem, we have defined a collection of subproblems \( \{ L(j) : 1 \leq j \leq n \} \) with the following key property that allows them to be solved in a single pass:

(*) There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to “smaller” subproblems, that is, subproblems that appear earlier in the ordering.

In our case, each subproblem is solved using the relation

\[
L(j) = 1 + \max \{ L(i) : (i, j) \in E \},
\]
an expression which involves only smaller subproblems. How long does this step take? It requires the predecessors of \( j \) to be known; for this the adjacency list of the reverse graph \( G^R \), constructible in linear time (recall Exercise 3.5), is handy. The computation of \( L(j) \) then takes time proportional to the indegree of \( j \), giving an overall running time linear in \(|E|\). This is at most \( O(n^2) \), the maximum being when the input array is sorted in increasing order. Thus the dynamic programming solution is both simple and efficient.

There is one last issue to be cleared up: the \( L \)-values only tell us the length of the optimal subsequence, so how do we recover the subsequence itself? This is easily managed with the same bookkeeping device we used for shortest paths in Chapter 4. While computing \( L(j) \), we should also note down \( \text{prev}(j) \), the next-to-last node on the longest path to \( j \). The optimal subsequence can then be reconstructed by following these backpointers.
Recursion? No, thanks.

Returning to our discussion of longest increasing subsequences: the formula for $L(j)$ also suggests an alternative, recursive algorithm. Wouldn’t that be even simpler?

Actually, recursion is a very bad idea: the resulting procedure would require exponential time! To see why, suppose that the dag contains edges $(i, j)$ for all $i < j$—that is, the given sequence of numbers $a_1, a_2, \ldots, a_n$ is sorted. In that case, the formula for subproblem $L(j)$ becomes

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}.$$ 

The following figure unravels the recursion for $L(5)$. Notice that the same subproblems get solved over and over again!

![Recursion Tree](attachment:image.png)

For $L(n)$ this tree has exponentially many nodes (can you bound it?), and so a recursive solution is disastrous.

Then why did recursion work so well with divide-and-conquer? The key point is that in divide-and-conquer, a problem is expressed in terms of subproblems that are substantially smaller, say half the size. For instance, mergesort sorts an array of size $n$ by recursively sorting two subarrays of size $n/2$. Because of this sharp drop in problem size, the full recursion tree has only logarithmic depth and a polynomial number of nodes.

In contrast, in a typical dynamic programming formulation, a problem is reduced to subproblems that are only slightly smaller—for instance, $L(j)$ relies on $L(j-1)$. Thus the full recursion tree generally has polynomial depth and an exponential number of nodes. However, it turns out that most of these nodes are repeats, that there are not too many distinct subproblems among them. Efficiency is therefore obtained by explicitly enumerating the distinct subproblems and solving them in the right order.
The origin of the term *dynamic programming* has very little to do with writing code. It was first coined by Richard Bellman in the 1950s, a time when computer programming was an esoteric activity practiced by so few people as to not even merit a name. Back then programming meant “planning,” and “dynamic programming” was conceived to optimally plan multistage processes. The dag of Figure 6.2 can be thought of as describing the possible ways in which such a process can evolve: each node denotes a state, the leftmost node is the starting point, and the edges leaving a state represent possible actions, leading to different states in the next unit of time.

The etymology of *linear programming*, the subject of Chapter 7, is similar.

### 6.3 Edit distance

When a spell checker encounters a possible misspelling, it looks in its dictionary for other words that are close by. What is the appropriate notion of closeness in this case?

A natural measure of the distance between two strings is the extent to which they can be aligned, or matched up. Technically, an alignment is simply a way of writing the strings one above the other. For instance, here are two possible alignments of *SNOWY* and *SUNNY*:

```
S - N O W Y
S U N N - Y
Cost: 3
```

```
S N O W Y
S U N N - Y
Cost: 5
```

The “−” indicates a “gap”; any number of these can be placed in either string. The *cost* of an alignment is the number of columns in which the letters differ. And the *edit distance* between two strings is the cost of their best possible alignment. Do you see that there is no better alignment of *SNOWY* and *SUNNY* than the one shown here with a cost of 3?

Edit distance is so named because it can also be thought of as the minimum number of *edits*—insertions, deletions, and substitutions of characters—needed to transform the first string into the second. For instance, the alignment shown on the left corresponds to three edits: insert *U*, substitute *O* → *N*, and delete *W*.

In general, there are so many possible alignments between two strings that it would be terribly inefficient to search through all of them for the best one. Instead we turn to dynamic programming.

#### A dynamic programming solution

When solving a problem by dynamic programming, the most crucial question is, *What are the subproblems?* As long as they are chosen so as to have the property (*) from page 171, it is an easy matter to write down the algorithm: iteratively solve one subproblem after the other, in order of increasing size.

Our goal is to find the edit distance between two strings $x[1 \cdots m]$ and $y[1 \cdots n]$. What is a good subproblem? Well, it should go part of the way toward solving the whole problem; so how
Figure 6.3 The subproblem $E(7,5)$.

<table>
<thead>
<tr>
<th>EXPONENTIAL</th>
<th>POLYNOMIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

about looking at the edit distance between some prefix of the first string, $x[1 \cdots i]$, and some prefix of the second, $y[1 \cdots j]$? Call this subproblem $E(i, j)$ (see Figure 6.3). Our final objective, then, is to compute $E(m, n)$.

For this to work, we need to somehow express $E(i, j)$ in terms of smaller subproblems. Let's see—what do we know about the best alignment between $x[1 \cdots i]$ and $y[1 \cdots j]$? Well, its rightmost column can only be one of three things:

- $x[i]$
- $y[j]$
- $x[i] y[j]$

The first case incurs a cost of 1 for this particular column, and it remains to align $x[1 \cdots i]$ with $y[1 \cdots j]$. But this is exactly the subproblem $E(i - 1, j)$! We seem to be getting somewhere. In the second case, also with cost 1, we still need to align $x[1 \cdots i]$ with $y[1 \cdots j - 1]$. This is again another subproblem, $E(i, j - 1)$. And in the final case, which either costs 1 (if $x[i] \neq y[j]$) or 0 (if $x[i] = y[j]$), what's left is the subproblem $E(i - 1, j - 1)$. In short, we have expressed $E(i, j)$ in terms of three smaller subproblems $E(i - 1, j)$, $E(i, j - 1)$, $E(i - 1, j - 1)$. We have no idea which of them is the right one, so we need to try them all and pick the best:

$$E(i, j) = \min \{1 + E(i - 1, j), 1 + E(i, j - 1), \text{diff}(i, j) + E(i - 1, j - 1)\}$$

where for convenience $\text{diff}(i, j)$ is defined to be 0 if $x[i] = y[j]$ and 1 otherwise.

For instance, in computing the edit distance between EXPONENTIAL and POLYNOMIAL, subproblem $E(4,3)$ corresponds to the prefixes EXPO and POL. The rightmost column of their best alignment must be one of the following:

- $O$
- $-$
- $L$
- $O$

Thus, $E(4,3) = \min \{1 + E(3,3), 1 + E(4,2), 1 + E(3,2)\}$.

The answers to all the subproblems $E(i, j)$ form a two-dimensional table, as in Figure 6.4. In what order should these subproblems be solved? Any order is fine, as long as $E(i - 1, j)$, $E(i, j - 1)$, and $E(i - 1, j - 1)$ are handled before $E(i, j)$. For instance, we could fill in the table one row at a time, from top row to bottom row, and moving left to right across each row. Or alternatively, we could fill it in column by column. Both methods would ensure that by the time we get around to computing a particular table entry, all the other entries we need are already filled in.
With both the subproblems and the ordering specified, we are almost done. There just remain the “base cases” of the dynamic programming, the very smallest subproblems. In the present situation, these are \( E(0, \cdot) \) and \( E(\cdot, 0) \), both of which are easily solved. \( E(0, j) \) is the edit distance between the 0-length prefix of \( x \), namely the empty string, and the first \( j \) letters of \( y \): clearly, \( j \). And similarly, \( E(i, 0) = i \).

At this point, the algorithm for edit distance basically writes itself.

```plaintext
for i = 0, 1, 2, \ldots, m:
    \( E(i, 0) = i \)
for j = 1, 2, \ldots, n:
    \( E(0, j) = j \)
for i = 1, 2, \ldots, m:
    for j = 1, 2, \ldots, n:
        \( E(i, j) = \min\{E(i - 1, j) + 1, E(i, j - 1) + 1, E(i - 1, j - 1) + \text{diff}(i, j)\} \)
return \( E(m, n) \)
```

This procedure fills in the table row by row, and left to right within each row. Each entry takes constant time to fill in, so the overall running time is just the size of the table, \( O(mn) \).

And in our example, the edit distance turns out to be 6:

```
EXPONENTIAL
- POLYNOMIAL
```
The underlying dag

Every dynamic program has an underlying dag structure: think of each node as representing a subproblem, and each edge as a precedence constraint on the order in which the subproblems can be tackled. Having nodes $u_1, \ldots, u_k$ point to $v$ means “subproblem $v$ can only be solved once the answers to $u_1, \ldots, u_k$ are known.”

In our present edit distance application, the nodes of the underlying dag correspond to subproblems, or equivalently, to positions $(i,j)$ in the table. Its edges are the precedence constraints, of the form $(i-1,j) \rightarrow (i,j)$, $(i,j-1) \rightarrow (i,j)$, and $(i-1,j-1) \rightarrow (i,j)$ (Figure 6.5). In fact, we can take things a little further and put weights on the edges so that the edit distances are given by shortest paths in the dag! To see this, set all edge lengths to 1, except for $\{(i-1,j-1) \rightarrow (i,j) : x[i] = y[j]\}$ (shown dotted in the figure), whose length is 0. The final answer is then simply the distance between nodes $s = (0,0)$ and $t = (m,n)$. One possible shortest path is shown, the one that yields the alignment we found earlier. On this path, each move down is a deletion, each move right is an insertion, and each diagonal move is either a match or a substitution.

By altering the weights on this dag, we can allow generalized forms of edit distance, in which insertions, deletions, and substitutions have different associated costs.
Common subproblems

Finding the right subproblem takes creativity and experimentation. But there are a few standard choices that seem to arise repeatedly in dynamic programming.

i. The input is $x_1, x_2, \ldots, x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.

\[
\begin{array}{cccccccccc}
  x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} \\
\end{array}
\]

The number of subproblems is therefore linear.

ii. The input is $x_1, \ldots, x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$.

\[
\begin{array}{cccccccc}
  x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
  y_1 & y_2 & y_3 & y_4 & y_5 & y_6 & y_7 & y_8 \\
\end{array}
\]

The number of subproblems is $O(mn)$.

iii. The input is $x_1, \ldots, x_n$ and a subproblem is $x_i, x_{i+1}, \ldots, x_j$.

\[
\begin{array}{cccccccccc}
  x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} \\
\end{array}
\]

The number of subproblems is $O(n^2)$.

iv. The input is a rooted tree. A subproblem is a rooted subtree.

If the tree has $n$ nodes, how many subproblems are there?

We’ve already encountered the first two cases, and the others are coming up shortly.
**Of mice and men**

Our bodies are extraordinary machines: flexible in function, adaptive to new environments, and able to interact and reproduce. All these capabilities are specified by a program unique to each of us, a string that is 3 billion characters long over the alphabet \( \{A, C, G, T\} \)—our DNA.

The DNA sequences of any two people differ by only about 0.1%. However, this still leaves 3 million positions on which they vary, more than enough to explain the vast range of human diversity. These differences are of great scientific and medical interest—for instance, they might help predict which people are prone to certain diseases.

DNA is a vast and seemingly inscrutable program, but it can be broken down into smaller units that are more specific in their role, rather like subroutines. These are called genes. Computers have become a crucial tool in understanding the genes of humans and other organisms, to the extent that computational genomics is now a field in its own right. Here are examples of typical questions that arise.

1. When a new gene is discovered, one way to gain insight into its function is to find known genes that match it closely. This is particularly helpful in transferring knowledge from well-studied species, such as mice, to human beings.

   A basic primitive in this search problem is to define an efficiently computable notion of when two strings approximately match. The biology suggests a generalization of edit distance, and dynamic programming can be used to compute it.

   Then there’s the problem of searching through the vast thicket of known genes: the database GenBank already has a total length of over \( 10^{10} \), and this number is growing rapidly. The current method of choice is BLAST, a clever combination of algorithmic tricks and biological intuitions that has made it the most widely used software in computational biology.

2. Methods for sequencing DNA (that is, determining the string of characters that constitute it) typically only find fragments of 500–700 characters. Billions of these randomly scattered fragments can be generated, but how can they be assembled into a coherent DNA sequence? For one thing, the position of any one fragment in the final sequence is unknown and must be inferred by piecing together overlapping fragments.

   A showpiece of these efforts is the draft of human DNA completed in 2001 by two groups simultaneously: the publicly funded Human Genome Consortium and the private Celera Genomics.

3. When a particular gene has been sequenced in each of several species, can this information be used to reconstruct the evolutionary history of these species?

We will explore these problems in the exercises at the end of this chapter. Dynamic programming has turned out to be an invaluable tool for some of them and for computational biology in general.