Actually, it suffices to make the substitution penalty greater than that of an insertion plus a deletion for substitution to lose any allure as a possible edit operation.

- *Maximum Monotone Subsequence* – A numerical sequence is *monotonically increasing* if the ith element is at least as big as the $(i-1)$st element. The *maximum monotone subsequence* problem seeks to delete the fewest number of elements from an input string S to leave a monotonically increasing subsequence. A longest increasing subsequence of 243517698 is 23568.

  In fact, this is just a longest common subsequence problem, where the second string is the elements of S sorted in increasing order. Any common sequence of these two must (a) represent characters in proper order in S, and (b) use only characters with increasing position in the collating sequence—so, the longest one does the job. Of course, this approach can be modified to give the longest decreasing sequence by simply reversing the sorted order.

As you can see, our edit distance routine can be made to do many amazing things easily. The trick is observing that your problem is just a special case of approximate string matching.

The alert reader may notice that it is unnecessary to keep all $O(mn)$ cells to compute the cost of an alignment. If we evaluate the recurrence by filling in the columns of the matrix from left to right, we will never need more than two columns of cells to store what is necessary for the computation. Thus, $O(m)$ space is sufficient to evaluate the recurrence without changing the time complexity. Unfortunately, we cannot reconstruct the alignment without the full matrix.

Saving space in dynamic programming is very important. Since memory on any computer is limited, using $O(nm)$ space proves more of a bottleneck than $O(nm)$ time. Fortunately, there is a clever divide-and-conquer algorithm that computes the actual alignment in $O(nm)$ time and $O(m)$ space. It is discussed in Section 18.4 (page 631).

8.3 Longest Increasing Sequence

There are three steps involved in solving a problem by dynamic programming:

1. Formulate the answer as a recurrence relation or recursive algorithm.

2. Show that the number of different parameter values taken on by your recurrence is bounded by a (hopefully small) polynomial.

3. Specify an order of evaluation for the recurrence so the partial results you need are always available when you need them.

To see how this is done, let's see how we would develop an algorithm to find the longest monotonically increasing subsequence within a sequence of n numbers. Truth be told, this was described as a special case of edit distance in the previous section, where it was called *maximum monotone subsequence*. Still, it is instructive to work it out from scratch. Indeed, dynamic programming algorithms are often easier to reinvent than look up.

We distinguish an increasing sequence from a *run*, where the elements must be physical neighbors of each other. The selected elements of both must be sorted in increasing order from left to right. For example, consider the sequence

$$S = \{2, 4, 3, 5, 1, 7, 6, 9, 8\}$$

The longest increasing subsequence of S has length 5, including {2,3,5,6,8}. In fact, there are eight of this length (can you enumerate them?). There are four longest increasing runs of length 2: $(2, 4)$, $(3, 5)$, $(1, 7)$, and $(6, 9)$.

Finding the longest increasing run in a numerical sequence is straightforward. Indeed, you should be able to devise a linear-time algorithm fairly easily. But finding the longest increasing subsequence is considerably trickier. How can we identify which scattered elements to skip? To apply dynamic programming, we need to construct a recurrence that computes the length of the longest sequence. To find the right recurrence, ask what information about the first $n - 1$ elements of S would help you to find the answer for the entire sequence?

- The length of the longest increasing sequence in $s_1, s_2, \ldots, s_{n-1}$ seems a useful thing to know. In fact, this will be the longest increasing sequence in S, unless s_n extends some increasing sequence of the same length.

  Unfortunately, the length of this sequence is not enough information to complete the full solution. Suppose I told you that the longest increasing sequence in $s_1, s_2, \ldots, s_{n-1}$ was of length 5 and that $s_n = 9$. Will the length of the final longest increasing subsequence of S be 5 or 6?

- We need to know the length of the longest sequence that s_n will extend. To be certain we know this, we really need the length of the longest sequence that *any* possible value for s_n can extend.

This provides the idea around which to build a recurrence. Define l_i to be the length of the longest sequence ending with s_i.

The longest increasing sequence containing the nth number will be formed by appending it to the longest increasing sequence to the left of n that ends on a number smaller than s_n. The following recurrence computes l_i:

$$\begin{aligned} l_i &= \max_{0<j<i} l_j + 1, \text{ where } (s_j < s_i), \\ l_0 &= 0 \end{aligned}$$

These values define the length of the longest increasing sequence ending at each number. The length of the longest increasing subsequence of the entire permutation is given by $\max_{1 \leq i \leq n} l_i$, since the winning sequence will have to end somewhere.

Here is the table associated with our previous example:

Sequence s_i	2	4	3	5	1	7	6	9	8
Length l_i	1	2	3	3	1	4	4	5	5
Predecessor p_i	–	1	1	2	–	4	4	6	6

What auxiliary information will we need to store to reconstruct the actual sequence instead of its length? For each element s_i, we will store its *predecessor*—the index p_i of the element that appears immediately before s_i in the longest increasing sequence ending at s_i. Since all of these pointers go towards the left, it is a simple matter to start from the last value of the longest sequence and follow the pointers so as to reconstruct the other items in the sequence.

What is the time complexity of this algorithm? Each one of the n values of l_i is computed by comparing s_i against (up to) $i - 1 \leq n$ values to the left of it, so this analysis gives a total of $O(n^2)$ time. In fact, by using dictionary data structures in a clever way, we can evaluate this recurrence in $O(n \lg n)$ time. However, the simple recurrence would be easy to program and therefore is a good place to start.

Take-Home Lesson: Once you understand dynamic programming, it can be easier to work out such algorithms from scratch than to try to look them up.

8.4 War Story: Evolution of the Lobster

I caught the two graduate students lurking outside my office as I came in to work that morning. There they were, two future PhDs working in the field of high-performance computer graphics. They studied new techniques for rendering pretty computer images, but the picture they painted for me that morning was anything but pretty.

"You see, we want to build a program to morph one image into another," they explained.

"What do you mean by morph?" I asked.

"For special effects in movies, we want to construct the intermediate stages in transforming one image into another. Suppose we want to turn you into Humphrey Bogart. For this to look realistic, we must construct a bunch of in-between frames that start out looking like you and end up looking like him."

"If you can realistically turn me into Bogart, you have something," I agreed.

"But our problem is that it isn't very realistic." They showed me a dismal morph between two images. "The trouble is that we must find the right corre-

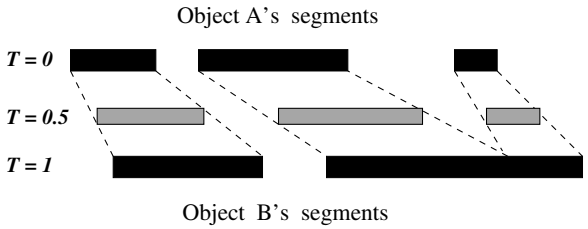

Figure 8.6: A successful alignment of two lines of pixels

spondence between features in the two images. It looks real bad when we get the correspondence wrong and try to morph a lip into an ear."

"I'll bet. So you want me to give you an algorithm for matching up lips?"

"No, even simpler. We morph each row of the initial image into the identical row of the final image. You can assume that we give you two lines of pixels, and you have to find the best possible match between the dark pixels in a row from object A to the dark pixels in the corresponding row of object B. Like this," they said, showing me images of successful matchings like Figure 8.6.

"I see," I said. "You want to match big dark regions to big dark regions and small dark regions to small dark regions."

"Yes, but only if the matching doesn't shift them too much to the left or the right. We might prefer to merge or break up regions rather than shift them too far away, since that might mean matching a chin to an eyebrow. What is the best way to do it?"

"One last question. Will you ever want to match two intervals to each other in such a way that they cross?"

"No, I guess not. Crossing intervals can't match. It would be like switching your left and right eyes."

I scratched my chin and tried to look puzzled, but I'm just not as good an actor as Bogart. I'd had a hunch about what needed to be done the instant they started talking about lines of pixels. They want to transform one array of pixels into another array, using the minimum amount of changes. That sounded like editing one string of pixels into another string, which is a classic application of dynamic programming. See Sections 8.2 and 18.4 for discussions of approximate string matching.

The fact that the intervals couldn't cross settled the issue. It meant that whenever a stretch of dark pixels from A was mapped to a stretch from B, the problem would be split into two smaller subproblems—i.e. , the pixels to the left of the match and the pixels to the right of the match. The cost of the global match would ultimately be the cost of this match plus those of matching all the pixels to the left and of matching all the pixels to the right. Constructing the optimal match on the left side is a smaller problem and hence simpler. Further, there could be only

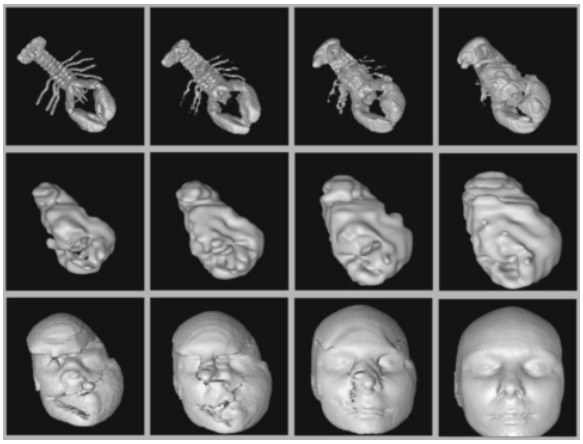

Figure 8.7: Morphing a lobster into a head via dynamic programming

$O(n^2)$ possible left subproblems, since each is completely described by the pair of one of n top pixels and one of n bottom pixels.

"Your algorithm will be based on dynamic programming," I pronounced. "However, there are several possible ways to do things, depending upon whether you want to edit pixels or runs. I would probably convert each row into a list of black pixel runs, with the runs sorted by right endpoint. Label each run with its starting position and length. You will maintain the cost of the cheapest match between the leftmost i runs and the leftmost j runs for all i and j. The possible edit operations are:

- *Full run match* – We may match top run i to run bottom j for a cost that is a function of the difference in the lengths of the two runs and their positions.

- *Merging runs* – We may match a string of consecutive top runs to a bottom run. The cost will be a function of the number of runs, their relative positions, and their lengths.

- *Splitting runs* – We may match a top run to a string of consecutive bottom runs. This is just the converse of the merge. Again, the cost will be a function of the number of runs, their relative positions, and their lengths.

"For each pair of runs (i, j) and all the cases that apply, we compute the cost of the edit operation and add to the (already computed and stored) edit cost to the left of the start of the edit. The cheapest of these cases is what we will take for the cost of $c[i, j]$."

The pair of graduate students scribbled this down, then frowned. "So we will have a cost measure for matching two runs as a function of their lengths and positions. How do we decide what the relative costs should be?"

"That is your business. The dynamic programming serves to optimize the matchings *once* you know the cost functions. It is up to your aesthetic sense to decide the penalties for line length changes and offsets. My recommendation is that you implement the dynamic programming and try different penalty values on each of several different images. Then, pick the setting that seems to do what you want."

They looked at each other and smiled, then ran back into the lab to implement it. Using dynamic programming to do their alignments, they completed their morphing system. It produced the images in Figure 8.7, morphing a lobster into a man. Unfortunately, they never got around to turning me into Humphrey Bogart.

8.5 The Partition Problem

Suppose that three workers are given the task of scanning through a shelf of books in search of a given piece of information. To get the job done fairly and efficiently, the books are to be partitioned among the three workers. To avoid the need to rearrange the books or separate them into piles, it is simplest to divide the shelf into three regions and assign each region to one worker.

But what is the fairest way to divide up the shelf? If all books are the same length, the job is pretty easy. Just partition the books into equal-sized regions,

100 100 100 | 100 100 100 | 100 100 100

so that everyone has 300 pages to deal with.

But what if the books are not the same length? Suppose we used the same partition when the book sizes looked like this:

100 200 300 | 400 500 600 | 700 800 900

I, would volunteer to take the first section, with only 600 pages to scan, instead of the last one, with 2,400 pages. The fairest possible partition for this shelf would be

100 200 300 400 500 | 600 700 | 800 900

where the largest job is only 1,700 pages and the smallest job 1,300.

In general, we have the following problem:

Problem: Integer Partition without Rearrangement
Input: An arrangement S of nonnegative numbers $\{s_1, \ldots, s_n\}$ and an integer k.
Output: Partition S into k or fewer ranges, to minimize the maximum sum over all the ranges, without reordering any of the numbers.

This so-called *linear partition* problem arises often in parallel process. We seek to balance the work done across processors to minimize the total elapsed run time.

The bottleneck in this computation will be the processor assigned the most work. Indeed, the war story of Section 7.10 (page 268) revolves around a botched solution to this problem.

Stop for a few minutes and try to find an algorithm to solve the linear partition problem.

A novice algorist might suggest a heuristic as the most natural approach to solving the partition problem. Perhaps they would compute the average size of a partition, $\sum_{i=1}^{n} s_i/k$, and then try to insert the dividers to come close to this average. However, such heuristic methods are doomed to fail on certain inputs because they do not systematically evaluate all possibilities.

Instead, consider a recursive, exhaustive search approach to solving this problem. Notice that the kth partition starts right after we placed the $(k-1)$st divider. Where can we place this last divider? Between the ith and $(i+1)$st elements for some i, where $1 \leq i \leq n$. What is the cost of this? The total cost will be the larger of two quantities—(1) the cost of the last partition $\sum_{j=i+1}^{n} s_j$, and (2) the cost of the largest partition formed to the left of i. What is the size of this left partition? To minimize our total, we want to use the $k-2$ remaining dividers to partition the elements $\{s_1, \ldots, s_i\}$ as equally as possible. *This is a smaller instance of the same problem, and hence can be solved recursively!*

Therefore, let us define $M[n,k]$ to be the minimum possible cost over all partitionings of $\{s_1, \ldots, s_n\}$ into k ranges, where the cost of a partition is the largest sum of elements in one of its parts. Thus defined, this function can be evaluated:

$$M[n,k] = \min_{i=1}^{n} \max(M[i,k-1], \sum_{j=i+1}^{n} s_j)$$

We must specify the boundary conditions of the recurrence relation. These boundary conditions always settle the smallest possible values for each of the arguments of the recurrence. For this problem, the smallest reasonable value of the first argument is $n = 1$, meaning that the first partition consists of a single element. We can't create a first partition smaller than s_1 regardless of how many dividers are used. The smallest reasonable value of the second argument is $k = 1$, implying that we do not partition S at all. In summary:

$$M[1,k] = s_1, \text{ for all } k > 0 \text{ and,}$$

$$M[n,1] = \sum_{i=1}^{n} s_i$$

By definition, this recurrence must return the size of the optimal partition. How long does it take to compute this when we store the partial results? A total of $k \cdot n$ cells exist in the table. How much time does it take to compute the result

$M[n', k']$? Calculating this quantity involves finding the minimum of n' quantities, each of which is the maximum of the table lookup and a sum of at most n' elements. If filling each of kn boxes takes at most n^2 time per box, the total recurrence can be computed in $O(kn^3)$ time.

The evaluation order computes the smaller values before the bigger values, so that each evaluation has what it needs waiting for it. Full details are provided in the code below:

```
partition(int s[], int n, int k)
{
        int m[MAXN+1][MAXK+1];                  /* DP table for values */
        int d[MAXN+1][MAXK+1];                  /* DP table for dividers */
        int p[MAXN+1];                          /* prefix sums array */
        int cost;                               /* test split cost */
        int i,j,x;                              /* counters */

        p[0] = 0;                               /* construct prefix sums */
        for (i=1; i<=n; i++) p[i]=p[i-1]+s[i];

        for (i=1; i<=n; i++) m[i][1] = p[i];    /* initialize boundaries */
        for (j=1; j<=k; j++) m[1][j] = s[1];

        for (i=2; i<=n; i++)                    /* evaluate main recurrence */
                for (j=2; j<=k; j++) {
                        m[i][j] = MAXINT;
                        for (x=1; x<=(i-1); x++) {
                                cost = max(m[x][j-1], p[i]-p[x]);
                                if (m[i][j] > cost) {
                                        m[i][j] = cost;
                                        d[i][j] = x;
                                }
                        }
                }

        reconstruct_partition(s,d,n,k);         /* print book partition */
}
```

This implementation above, in fact, runs faster than advertised. Our original analysis assumed that it took $O(n^2)$ time to update each cell of the matrix. This is because we selected the best of up to n possible points to place the divider, each of which requires the sum of up to n possible terms. In fact, it is easy to avoid the need to compute these sums by storing the set of n prefix sums $p[i] = \sum_{k=1}^{i} s_k$,

M	k		
n	1	2	3
1	1	1	1
1	2	1	1
1	3	2	1
1	4	2	2
1	5	3	2
1	6	3	2
1	7	4	3
1	8	4	3
1	9	5	3

D	k		
n	1	2	3
1	–	–	–
1	–	1	1
1	–	1	2
1	–	2	2
1	–	2	3
1	–	3	4
1	–	3	4
1	–	4	5
1	–	4	6

M	k		
n	1	2	3
1	1	1	1
2	3	2	2
3	6	3	3
4	10	6	4
5	15	9	6
6	21	11	9
7	28	15	11
8	36	21	15
9	45	24	17

D	k		
n	1	2	3
1	–	–	–
2	–	1	1
3	–	2	2
4	–	3	3
5	–	3	4
6	–	4	5
7	–	5	6
8	–	5	6
9	–	6	7

Figure 8.8: Dynamic programming matrices M and D for two input instances. Partitioning $\{1,1,1,1,1,1,1,1,1\}$ into $\{\{1,1,1\},\{1,1,1\},\{1,1,1\}\}$ (l) . Partitioning $\{1,2,3,4,5,6,7,8,9\}$ into $\{\{1,2,3,4,5\},\{6,7\},\{8,9\}\}$ (r).

since $\sum_{k=i}^{j} s_k = p[j] - p[k]$. This enables us to evaluate the recurrence in linear time per cell, yielding an $O(kn^2)$ algorithm.

By studying the recurrence relation and the dynamic programming matrices of Figure 8.8, you should be able to convince yourself that the final value of $M(n,k)$ will be the cost of the largest range in the optimal partition. For most applications, however, what we need is the actual partition that does the job. Without it, all we are left with is a coupon with a great price on an out-of-stock item.

The second matrix, D, is used to reconstruct the optimal partition. Whenever we update the value of $M[i,j]$, we record which divider position was required to achieve that value. To reconstruct the path used to get to the optimal solution, we work backward from $D[n,k]$ and add a divider at each specified position. This backwards walking is best achieved by a recursive subroutine:

```
reconstruct_partition(int s[],int d[MAXN+1][MAXK+1], int n, int k)
{
        if (k==1)
                print_books(s,1,n);
        else {
                reconstruct_partition(s,d,d[n][k],k-1);
                print_books(s,d[n][k]+1,n);
        }
}


print_books(int s[], int start, int end)
{
        int i;                  /* counter */

        for (i=start; i<=end; i++) printf(" %d ",s[i]);
        printf("\n");
}
```

sentence ::= noun–phrase
verb–phrase
noun–phrase ::= article noun
verb–phrase ::= verb noun–phrase
article ::= *the, a*
noun ::= *cat, milk*
verb ::= *drank*

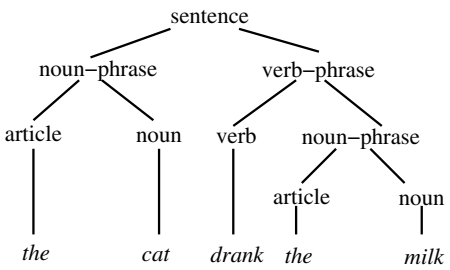

Figure 8.9: A context-free grammar (l) with an associated parse tree (r)

8.6 Parsing Context-Free Grammars

Compilers identify whether the given program is legal in the programming language, and reward you with syntax errors if not. This requires a precise description of the language syntax typically given by a *context-free grammar* as shown in Figure 8.9(l). Each *rule* or *production* of the grammar defines an interpretation for the named symbol on the left side of the rule as a sequence of symbols on the right side of the rule. The right side can be a combination of *nonterminals* (themselves defined by rules) or *terminal* symbols defined simply as strings, such as "the", "a", "cat", "milk", and "drank."

Parsing a given text string S according to a given context-free grammar G is the algorithmic problem of constructing a *parse tree* of rule substitutions defining S as a single nonterminal symbol of G. Figure 8.9(r) gives the parse tree of a simple sentence using our sample grammar.

Parsing seemed like a horribly complicated subject when I took a compilers course as a graduate student. But, a friend easily explained it to me over lunch a few years ago. The difference is that I now understand dynamic programming much better than when I was a student.

We assume that the text string S has length n while the grammar G itself is of constant size. This is fair, since the grammar defining a particular programming language (say C or Java) is of fixed length regardless of the size of the program we are trying to compile.

Further, we assume that the definitions of each rule are in *Chomsky normal form*. This means that the right sides of every nontrivial rule consists of (a) exactly two nonterminals, i.e. $X \rightarrow YZ$, or (b) exactly one terminal symbol, $X \rightarrow \alpha$. Any context-free grammar can be easily and mechanically transformed into Chomsky normal form by repeatedly shortening long right-hand sides at the cost of adding extra nonterminals and productions. Thus, there is no loss of generality with this assumption.

So how can we efficiently parse a string S using a context-free grammar where each interesting rule consists of two nonterminals? The key observation is that the rule applied at the root of the parse tree (say $X \rightarrow YZ$) splits S at some position i such that the left part of the string ($S[1, i]$) must be *generated* by nonterminal Y, and the right part ($S[i + 1, n]$) generated by Z.

This suggests a dynamic programming algorithm, where we keep track of all of the nonterminals generated by each substring of S. Define $M[i, j, X]$ to be a boolean function that is true iff substring $S[i, j]$ is generated by nonterminal X. This is true if there exists a production $X \rightarrow YZ$ and breaking point k between i and j such that the left part generates Y and the right part Z. In other words,

$$M[i, j, X] = \bigvee_{(X \rightarrow YZ) \in G} \left(\bigvee_{i=k}^{j} M[i, k, Y] \cdot M[k + 1, j, Z] \right)$$

where $\vee$ denotes the logical *or* over all productions and split positions, and $\cdot$ denotes the logical *and* of two boolean values.

The one-character terminal symbols define the boundary conditions of the recurrence. In particular, $M[i, i, X]$ is true iff there exists a production $X \rightarrow \alpha$ such that $S[i] = \alpha$.

What is the complexity of this algorithm? The size of our state-space is $O(n^2)$, as there are $n(n + 1)/2$ substrings defined by (i, j) pairs. Multiplying this by the number of nonterminals (say v) has no impact on the big-Oh, because the grammar was defined to be of constant size. Evaluating the value $M[i, j, X]$ requires testing all intermediate values k, so it takes $O(n)$ in the worst case to evaluate each of the $O(n^2)$ cells. This yields an $O(n^3)$ or cubic-time algorithm for parsing.

Stop and Think: Parsimonious Parserization

Problem: Programs often contain trivial syntax errors that prevent them from compiling. Given a context-free grammar G and input string S, find the smallest number of character substitutions you must make to S so that the resulting string is accepted by G.

Solution: This problem seemed extremely difficult when I first encountered it. But on reflection, it seemed like a very general version of edit distance, which is addressed naturally by dynamic programming. Parsing initially sounded hard, too, but fell to the same technique. Indeed, we can solve the combined problem by generalizing the recurrence relation we used for simple parsing.

Define $M'[i, j, X]$ to be an *integer* function that reports the minimum number of changes to substring $S[i, j]$ so it can be generated by nonterminal X. This symbol will be generated by some production $x \rightarrow yz$. Some of the changes to s may be

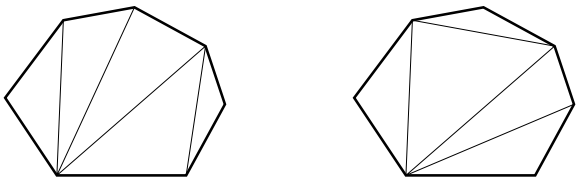

Figure 8.10: Two different triangulations of a given convex seven-gon

to the left of the breaking point and some to the right, but all we care about is minimizing the sum. In other words,

$$M'[i,j,X] = \min_{(X \to YZ) \in G} \left(\min_{i=k}^{j} M'[i,k,Y] + M'[k+1,j,Z] \right)$$

The boundary conditions also change mildly. If there exists a production $X \to \alpha$, the cost of matching at position i depends on the contents of $S[i]$, where $S[i] = \alpha$, $M[i,i,X] = 0$. Otherwise, it is one substitution away, so $M[i,i,X] = 1$ if $S[i] \neq \alpha$. If the grammar does not have a production of the form $X \to \alpha$, there is no way to substitute a single character string into something generating X, so $M[i,i,X] = \infty$ for all i. ∎

8.6.1 Minimum Weight Triangulation

The same basic recurrence relation encountered in the parsing algorithm above can also be used to solve an interesting computational geometry problem. A *triangulation* of a polygon $P = \{v_1, \ldots, v_n, v_1\}$ is a set of nonintersecting diagonals that partitions the polygon into triangles. We say that the *weight* of a triangulation is the sum of the lengths of its diagonals. As shown in Figure 8.10, any given polygon may have many different triangulations. We seek to find its minimum weight triangulation for a given polygon p. Triangulation is a fundamental component of most geometric algorithms, as discussed in Section 17.3 (page 572).

To apply dynamic programming, we need a way to carve up the polygon into smaller pieces. Observe that every edge of the input polygon must be involved in exactly one triangle. Turning this edge into a triangle means identifying the third vertex, as shown in Figure 8.11. Once we find the correct connecting vertex, the polygon will be partitioned into two smaller pieces, both of which need to be triangulated optimally. Let $T[i,j]$ be the cost of triangulating from vertex v_i to vertex v_j, ignoring the length of the chord d_{ij} from v_i to v_j. The latter clause avoids double counting these internal chords in the following recurrence:

$$T[i,j] \quad = \quad \min_{k=i+1}^{j-1} (T[i,k] + T[k,j] + d_{ik} + d_{kj})$$

The basis condition applies when i and j are immediate neighbors, as $T[i,i+1] = 0$.

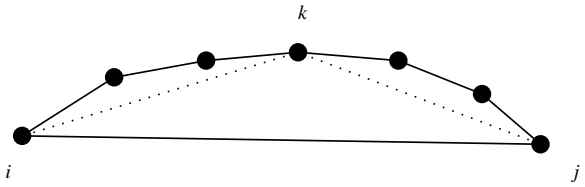

Figure 8.11: Selecting the vertex k to pair with an edge (i, j) of the polygon

Since the number of vertices in each subrange of the right side of the recurrence is smaller than that on the left side, evaluation can proceed in terms of the gap size from i to j:

```
Minimum-Weight-Triangulation(P)
    for i = 1 to n − 1 do T[i, i + 1] = 0
    for gap = 2 to n − 1
        for i = 1 to n − gap do
            j = i + gap
            T[i, j] = min_{k=i+1}^{j−1}(T[i, k] + T[k, j] + d_{P_i,P_k} + d_{P_k,P_j})
    return T[1, n]
```

There are $\binom{n}{2}$ values of T, each of which takes $O(j - i)$ time if we evaluate the sections in order of increasing size. Since $j - i = O(n)$, complete evaluation takes $O(n^3)$ time and $O(n^2)$ space.

What if there are points in the interior of the polygon? Then dynamic programming does not apply in the same way, because triangulation edges do not necessarily cut the boundary into two distinct pieces as before. Instead of only $\binom{n}{2}$ possible subregions, the number of subregions now grows exponentially. In fact, the more general version of this problem is known to be NP-complete.

Take-Home Lesson: For any optimization problem on left-to-right objects, such as characters in a string, elements of a permutation, points around a polygon, or leaves in a search tree, dynamic programming likely leads to an efficient algorithm to find the optimal solution.

8.7 Limitations of Dynamic Programming: TSP

Dynamic programming doesn't always work. It is important to see why it can fail, to help avoid traps leading to incorrect or inefficient algorithms.

Our algorithmic poster child will once again be the traveling salesman, where we seek the shortest tour visiting all the cities in a graph. We will limit attention here to an interesting special case:

Problem: Longest Simple Path
Input: A weighted graph G, with specified start and end vertices s and t.
Output: What is the most expensive path from s to t that does not visit any vertex more than once?

This problem differs from TSP in two quite unimportant ways. First, it asks for a path instead of a closed tour. This difference isn't substantial: we get a closed tour by simply including the edge (t, s). Second, it asks for the most expensive path instead of the least expensive tour. Again this difference isn't very significant: it encourages us to visit as many vertices as possible (ideally all), just as in TSP. The big word in the problem statement is *simple*, meaning we are not allowed to visit any vertex more than once.

For *unweighted* graphs (where each edge has cost 1), the longest possible simple path from s to t is $n - 1$. Finding such *Hamiltonian paths* (if they exist) is an important graph problem, discussed in Section 16.5 (page 538).

8.7.1 When are Dynamic Programming Algorithms Correct?

Dynamic programming algorithms are only as correct as the recurrence relations they are based on. Suppose we define $LP[i, j]$ as a function denoting the length of the longest simple path from i to j. Note that the longest simple path from i to j had to visit some vertex x right before reaching j. Thus, the last edge visited must be of the form (x, j). This suggests the following recurrence relation to compute the length of the longest path, where $c(x, j)$ is the cost/weight of edge (x, j):

$$LP[i, j] = \max_{(x,j) \in E} LP[i, x] + c(x, j)$$

The idea seems reasonable, but can you see the problem? I see at least two of them.

First, this recurrence does nothing to enforce simplicity. How do we know that vertex j has not appeared previously on the longest simple path from i to x? If it did, adding the edge (x, j) will create a cycle. To prevent such a thing, we must define a different recursive function that explicitly remembers where we have been. Perhaps we could define $LP'[i, j, k]$ to be the function denoting the length of the longest path from i to j avoiding vertex k? This would be a step in the right direction, but still won't lead to a viable recurrence.

A second problem concerns evaluation order. What can you evaluate first? Because there is no left-to-right or smaller-to-bigger ordering of the vertices on the graph, it is not clear what the *smaller* subprograms are. Without such an ordering, we get are stuck in an infinite loop as soon as we try to do anything.

Dynamic programming can be applied to any problem that observes the *principle of optimality*. Roughly stated, this means that partial solutions can be optimally extended with regard to the *state* after the partial solution, instead of the specifics of the partial solution itself. For example, in deciding whether to extend an approximate string matching by a substitution, insertion, or deletion, we did not need to

know which sequence of operations had been performed to date. In fact, there may be several different edit sequences that achieve a cost of C on the first p characters of pattern P and t characters of string T. Future decisions are made based on the *consequences* of previous decisions, not the actual decisions themselves.

Problems do not satisfy the principle of optimality when the specifics of the operations matter, as opposed to just the cost of the operations. Such would be the case with a form of edit distance where we are not allowed to use combinations of operations in certain particular orders. Properly formulated, however, many combinatorial problems respect the principle of optimality.

8.7.2 When are Dynamic Programming Algorithms Efficient?

The running time of any dynamic programming algorithm is a function of two things: (1) number of partial solutions we must keep track of, and (2) how long it take to evaluate each partial solution. The first issue—namely the size of the state space—is usually the more pressing concern.

In all of the examples we have seen, the partial solutions are completely described by specifying the stopping *places* in the input. This is because the combinatorial objects being worked on (strings, numerical sequences, and polygons) have an implicit order defined upon their elements. This order cannot be scrambled without completely changing the problem. Once the order is fixed, there are relatively few possible stopping places or states, so we get efficient algorithms.

When the objects are not firmly ordered, however, we get an exponential number of possible partial solutions. Suppose the state of our partial solution is entire path P taken from the start to end vertex. Thus $LP[i, j, P]$ denotes the longest simple path from i to j, where P is the exact sequence of intermediate vertices between i and j on this path. The following recurrence relation works to compute this, where $P + x$ denotes appending x to the end of P:

$$LP[i, j, P + x] = \max_{(x,j) \in E, x, j \notin P} LP[i, x, P] + c(x, j)$$

This formulation is correct, but how efficient is it? The path P consists of an ordered sequence of up to $n - 3$ vertices. There can be up to $(n - 3)!$ such paths! Indeed, this algorithm is really using combinatorial search (*a la* backtracking) to construct all the possible intermediate paths. In fact, the max is somewhat misleading, as there can only be one value of x and one value of P to construct the state $LP[i, j, P + x]$.

We can do something better with this idea, however. Let $LP'[i, j, S]$ denote the longest simple path from i to j, where the intermediate vertices on this path are exactly those in the *subset* S. Thus, if $S = \{a, b, c\}$, there are exactly six paths consistent with S: *iabcj*, *iacbj*, *ibacj*, *ibcaj*, *icabj*, and *icbaj*. This state space is at most 2^n, and thus smaller than enumerating the paths. Further, this function can be evaluated using the following recurrence relation:

$$LP'[i, j, S \cup \{x\}] = \max_{(x,j) \in E, x, j \notin S} LP[i, x, S] + c(x, j)$$

where $S \cup \{x\}$ denotes unioning S with x.

The longest simple path from i to j can then be found by maximizing over all possible intermediate vertex subsets:

$$LP[i,j] = \max_S LP'[i,j,S]$$

There are only 2^n subsets of n vertices, so this is a big improvement over enumerating all $n!$ tours. Indeed, this method could certainly be used to solve TSPs for up to thirty vertices or so, where $n = 20$ would be impossible using the $O(n!)$ algorithm. Still, dynamic programming is most effective on well-ordered objects.

> *Take-Home Lesson:* Without an inherent left-to-right ordering on the objects, dynamic programming is usually doomed to require exponential space and time.

8.8 War Story: What's Past is Prolog

"But our heuristic works very, very well in practice." My colleague was simultaneously boasting and crying for help.

Unification is the basic computational mechanism in logic programming languages like Prolog. A Prolog program consists of a set of rules, where each rule has a head and an associated action whenever the rule head matches or unifies with the current computation.

An execution of a Prolog program starts by specifying a goal, say $p(a, X, Y)$, where a is a constant and X and Y are variables. The system then systematically matches the head of the goal with the head of each of the rules that can be *unified* with the goal. Unification means binding the variables with the constants, if it is possible to match them. For the nonsense program below, $p(X, Y, a)$ unifies with either of the first two rules, since X and Y can be bound to match the extra characters. The goal $p(X, X, a)$ would only match the first rule, since the variable bound to the first and second positions must be the same.

$$p(a,a,a) := h(a);$$
$$p(b,a,a) := h(a) * h(b);$$
$$p(c,b,b) := h(b) + h(c);$$
$$p(d,b,b) := h(d) + h(b);$$

"In order to speed up unification, we want to preprocess the set of rule heads so that we can quickly determine which rules match a given goal. We must organize the rules in a trie data structure for fast unification."

Tries are extremely useful data structures in working with strings, as discussed in Section 12.3 (page 377). Every leaf of the trie represents one string. Each node on the path from root to leaf is labeled with exactly one character of the string, with the ith node of the path corresponding to the ith character of the string.

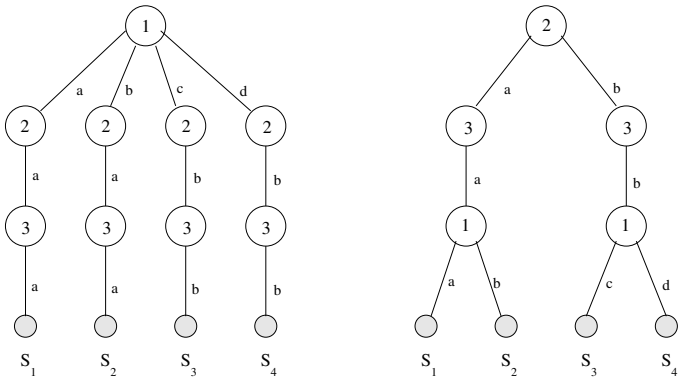

Figure 8.12: Two different tries for the same set of rule heads.

"I agree. A trie is a natural way to represent your rule heads. Building a trie on a set of strings of characters is straightforward: just insert the strings starting from the root. So what is your problem?" I asked.

"The efficiency of our unification algorithm depends very much on minimizing the number of edges in the trie. Since we know all the rules in advance, we have the freedom to reorder the character positions in the rules. Instead of the root node always representing the first argument in the rule, we can choose to have it represent the third argument. We would like to use this freedom to build a minimum-size trie for a set of rules."

He showed me the example in Figure 8.12. A trie constructed according to the original string position order $(1, 2, 3)$ uses a total of 12 edges. However, by permuting the character order to $(2, 3, 1)$, we can obtain a trie with only 8 edges.

"Interesting..." I started to reply before he cut me off again.

"There's one other constraint. We must keep the leaves of the trie ordered, so that the leaves of the underlying tree go left-to-right in the same order as the rules appear on the page."

"But why must you keep the leaves of the trie in the given order?" I asked.

"The order of rules in Prolog programs is very, very important. If you change the order of the rules, the program returns different results."

Then came my mission.

"We have a greedy heuristic for building good, but not optimal, tries based on picking as the root the character position that minimizes the degree of the root. In other words, it picks the character position that has the smallest number of distinct characters in it. This heuristic works very, very well in practice. But we need you to prove that finding the best trie is NP-complete so our paper is, well, complete."

I agreed to try to prove the hardness of the problem, and chased him from my office. The problem did seem to involve some nontrivial combinatorial optimization to build the minimal tree, but I couldn't see how to factor the left-to-right order of the rules into a hardness proof. In fact, I couldn't think of any NP-complete problem that had such a left-right ordering constraint. After all, if a given set of rules contained a character position in common to all the rules, this character position must be probed first in any minimum-size tree. Since the rules were ordered, each node in the subtree must represent the root of a run of consecutive rules. Thus there were only $\binom{n}{2}$ possible nodes to choose from for this tree....

Bingo! That settled it.

The next day I went back to the professor and told him. "I can't prove that your problem is NP-complete. But how would you feel about an efficient dynamic programming algorithm to find the best trie!" It was a pleasure watching his frown change to a smile as the realization took hold. An efficient algorithm to compute what he needed was infinitely better than a proof saying you couldn't do it!

My recurrence looked something like this. Suppose that we are given n ordered rule heads $s_1, \ldots, s_n$, each with m arguments. Probing at the pth position, $1 \leq p \leq m$, partitioned the rule heads into runs $R_1, \ldots, R_r$, where each rule in a given run $R_x = s_i, \ldots, s_j$ had the same character value of $s_i[p]$. The rules in each run must be consecutive, so there are only $\binom{n}{2}$ possible runs to worry about. The cost of probing at position p is the cost of finishing the trees formed by each created run, plus one edge per tree to link it to probe p:

$$C[i,j] = \min_{p=1}^{m} \sum_{k=1}^{r} (C[i_k, j_k] + 1)$$

A graduate student immediately set to work implementing this algorithm to compare with their heuristic. On many inputs, the optimal and greedy algorithms constructed the exact same trie. However, for some examples, dynamic programming gave a 20% performance improvement over greedy—i.e. , 20% better than very, very well in practice. The run time spent in doing the dynamic programming was a bit larger than with greedy, but in compiler optimization you are always happy to trade off a little extra compilation time for better execution time in the performance of your program. Is a 20% improvement worth this effort? That depends upon the situation. How useful would you find a 20% increase in your salary?

The fact that the rules had to remain ordered was the crucial property that we exploited in the dynamic programming solution. Indeed, without it I was able to prove that the problem *was* NP-complete, something we put in the paper to make it complete.

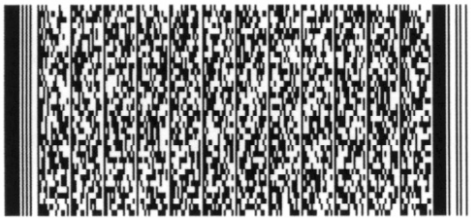

Figure 8.13: A two-dimensional bar-code label of the Gettysburg Address using PDF-417.

Take-Home Lesson: The global optimum (found, for example, using dynamic programming) is often noticeably better than the solution found by typical heuristics. How important this improvement is depends on your application, but it can never hurt.

8.9 War Story: Text Compression for Bar Codes

Ynjiun waved his laser wand over the torn and crumpled fragments of a bar code label. The system hesitated for a few seconds, then responded with a pleasant *blip* sound. He smiled at me in triumph. "Virtually indestructible."

I was visiting the research laboratories of Symbol Technologies, the world's leading manufacturer of bar code scanning equipment. Next time you are in the checkout line at a grocery store, check to see what type of scanning equipment they are using. Likely it will say Symbol on the housing.

Although we take bar codes for granted, there is a surprising amount of technology behind them. Bar codes exist because conventional optical character recognition (OCR) systems are not sufficiently reliable for inventory operations. The bar code symbology familiar to us on each box of cereal or pack of gum encodes a ten-digit number with enough error correction that it is virtually impossible to scan the wrong number, even if the can is upside-down or dented. Occasionally, the cashier won't be able to get a label to scan at all, but once you hear that *blip* you know it was read correctly.

The ten-digit capacity of conventional bar code labels provides room enough only to store a single ID number in a label. Thus any application of supermarket bar codes must have a database mapping 11141-47011 to a particular size and brand of soy sauce. The holy grail of the bar code world has long been the development of higher-capacity bar code symbologies that can store entire documents, yet still be read reliably.

"PDF-417 is our new, two-dimensional bar code symbology," Ynjiun explained. A sample label is shown in Figure 8.13.

"How much data can you fit in a typical one-inch square label?" I asked him.

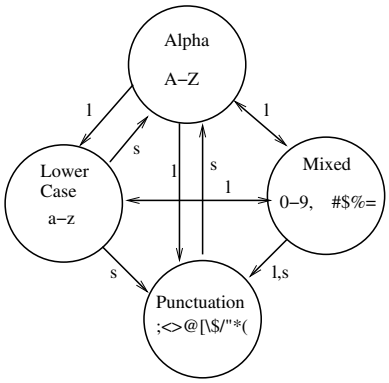

Figure 8.14: Mode switching in PDF-417

"It depends upon the level of error correction we use, but about 1,000 bytes. That's enough for a small text file or image," he said.

"Interesting. You will probably want to use some data compression technique to maximize the amount of text you can store in a label." See Section 18.5 (page 637) for a discussion of standard data compression algorithms.

"We do incorporate a data compaction method," he explained. "We think we understand the different types of files our customers will want to make labels for. Some files will be all in uppercase letters, while others will use mixed-case letters and numbers. We provide four different text modes in our code, each with a different subset of alphanumeric characters available. We can describe each character using only five bits as long as we stay within a mode. To switch modes, we issue a mode switch command first (taking an extra five bits) and then the new character code."

"I see. So you designed the mode character sets to minimize the number of mode switch operations on typical text files." The modes are illustrated in Figure 8.14.

"Right. We put all the digits in one mode and all the punctuation characters in another. We also included both mode *shift* and mode *latch* commands. In a mode shift, we switch into a new mode just for the next character, say to produce a punctuation mark. This way, we don't pay a cost for returning back to text mode after a period. Of course, we can also latch permanently into a different mode if we will be using a run of several characters from there."

"Wow!" I said. "With all of this mode switching going on, there must be many different ways to encode any given text as a label. How do you find the smallest of such encoding."

"We use a greedy algorithm. We look a few characters ahead and then decide which mode we would be best off in. It works fairly well."

I pressed him on this. "How do you know it works fairly well? There might be significantly better encodings that you are simply not finding."

"I guess I don't know. But it's probably NP-complete to find the optimal coding." Ynjiun's voice trailed off. "Isn't it?"

I started to think. Every encoding started in a given mode and consisted of a sequence of intermixed character codes and mode shift/latch operations. At any given position in the text, we could output the next character code (if it was available in our current mode) or decide to shift. As we moved from left to right through the text, our current state would be completely reflected by our current character position and current mode state. For a given position/mode pair, we would have been interested in the cheapest way of getting there, over all possible encodings getting to this point....

My eyes lit up so bright they cast shadows on the walls.

"The optimal encoding for any given text in PDF-417 can be found using dynamic programming. For each possible mode $1 \leq m \leq 4$, and each character position $1 \leq i \leq n$, we will maintain the cheapest encoding found of the first i characters ending in mode m. Our next move from each mode/position is either match, shift, or latch, so there are only a few possible operations to consider."

Basically,

$$M[i,j] = \min_{1 \leq m \leq 4} (M[i-1,m] + c(S_i, m, j))$$

where $c(S_i, m, j)$ is the cost of encoding character S_i and switching from mode m to mode j. The cheapest possible encoding results from tracing back from $M[n,m]$, where m is the value of i that minimizes $\min_{1 \leq i \leq 4} M[n,i]$. Each of the $4n$ cells can be filled in constant time, so it takes time linear in the length of the string to find the optimal encoding.

Ynjiun was skeptical, but he encouraged us to implement an optimal encoder. A few complications arose due to weirdnesses of PDF-417 mode switching, but my student Yaw-Ling Lin rose to the challenge. Symbol compared our encoder to theirs on 13,000 labels and concluded that dynamic programming lead to an 8% tighter encoding on average. This was significant, because no one wants to waste 8% of their potential storage capacity, particularly in an environment where the capacity is only a few hundred bytes. For certain applications, this 8% margin permitted one bar code label to suffice where previously two had been required. Of course, an 8% *average* improvement meant that it did much better than that on certain labels. While our encoder took slightly longer to run than the greedy encoder, this was not significant, since the bottleneck would be the time needed to print the label anyway.

Our observed impact of replacing a heuristic solution with the global optimum is probably typical of most applications. Unless you really botch your heuristic, you are probably going to get a decent solution. Replacing it with an optimal result, however, usually gives a small but nontrivial improvement, which can have pleasing consequences for your application.

Chapter Notes

Bellman [Bel58] is credited with developing the technique of dynamic programming. The edit distance algorithm is originally due to Wagner and Fischer [WF74]. A faster algorithm for the book partition problem appears in [KMS97].

The computational complexity of finding the minimum weight triangulation of disconnected point sets (as opposed to polygons) was a longstanding open problem that finally fell to Mulzer and Rote [MR06].

Techniques such as dynamic programming and backtracking searches can be used to generate worst-case efficient (although still non-polynomial) algorithms for many NP-complete problems. See Woeginger [Woe03] for a nice survey of such techniques.

The morphing system that was the subject of the war story in Section 8.4 (page 291) is described in [HWK94]. See our paper [DRR+95] for more on the Prolog trie minimization problem, subject of the war story of Section 8.8 (page 304). Two-dimensional bar codes, subject of the war story in Section 8.9 (page 307), were developed largely through the efforts of Theo Pavlidis and Ynjiun Wang at Stony Brook [PSW92].

The dynamic programming algorithm presented for parsing is known as the *CKY* algorithm after its three independent inventors (Cocke, Kasami, and Younger) [You67]. The generalization of parsing to edit distance is due to Aho and Peterson [AP72].

8.10 Exercises

Edit Distance

8-1. *[3]* Typists often make transposition errors exchanging neighboring characters, such as typing "setve" when you mean "steve." This requires two substitutions to fix under the conventional definition of edit distance.

Incorporate a swap operation into our edit distance function, so that such neighboring transposition errors can be fixed at the cost of one operation.

8-2. *[4]* Suppose you are given three strings of characters: X, Y, and Z, where $|X| = n$, $|Y| = m$, and $|Z| = n + m$. Z is said to be a *shuffle* of X and Y iff Z can be formed by interleaving the characters from X and Y in a way that maintains the left-to-right ordering of the characters from each string.

(a) Show that *cchocohilaptes* is a shuffle of *chocolate* and *chips*, but *chocochilatspe* is not.

(b) Give an efficient dynamic-programming algorithm that determines whether Z is a shuffle of X and Y. Hint: the values of the dynamic programming matrix you construct should be Boolean, not numeric.

8-3. *[4]* The longest common *substring* (not subsequence) of two strings X and Y is the longest string that appears as a run of consecutive letters in both strings. For example, the longest common substring of *photograph* and *tomography* is *ograph*.

(a) Let $n = |X|$ and $m = |Y|$. Give a $\Theta(nm)$ dynamic programming algorithm for longest common substring based on the longest common subsequence/edit distance algorithm.

(b) Give a simpler $\Theta(nm)$ algorithm that does not rely on dynamic programming.

8-4. *[6]* The *longest common subsequence (LCS)* of two sequences T and P is the longest sequence L such that L is a subsequence of both T and P. The *shortest common supersequence (SCS)* of T and P is the smallest sequence L such that both T and P are a subsequence of L.

(a) Give efficient algorithms to find the LCS and SCS of two given sequences.

(b) Let $d(T, P)$ be the minimum edit distance between T and P when no substitutions are allowed (i.e., the only changes are character insertion and deletion). Prove that $d(T, P) = |SCS(T, P)| - |LCS(T, P)|$ where $|SCS(T, P)|$ ($|LCS(T, P)|$) is the size of the shortest SCS (longest LCS) of T and P.

Greedy Algorithms

8-5. *[4]* Let $P_1, P_2, \ldots, P_n$ be n programs to be stored on a disk with capacity D megabytes. Program P_i requires s_i megabytes of storage. We cannot store them all because $D < \sum_{i=1}^{n} s_i$

(a) Does a greedy algorithm that selects programs in order of nondecreasing s_i maximize the number of programs held on the disk? Prove or give a counterexample.

(b) Does a greedy algorithm that selects programs in order of nonincreasing order s_i use as much of the capacity of the disk as possible? Prove or give a counterexample.

8-6. *[5]* Coins in the United States are minted with denominations of 1, 5, 10, 25, and 50 cents. Now consider a country whose coins are minted with denominations of $\{d_1, \ldots, d_k\}$ units. We seek an algorithm to make change of n units using the minimum number of coins for this country.

(a) The greedy algorithm repeatedly selects the biggest coin no bigger than the amount to be changed and repeats until it is zero. Show that the greedy algorithm does not always use the minimum number of coins in a country whose denominations are $\{1, 6, 10\}$.

(b) Give an efficient algorithm that correctly determines the minimum number of coins needed to make change of n units using denominations $\{d_1, \ldots, d_k\}$. Analyze its running time.

8-7. *[5]* In the United States, coins are minted with denominations of 1, 5, 10, 25, and 50 cents. Now consider a country whose coins are minted with denominations of $\{d_1, \ldots, d_k\}$ units. We want to count how many distinct ways $C(n)$ there are to make change of n units. For example, in a country whose denominations are $\{1, 6, 10\}$, $C(5) = 1$, $C(6)$ to $C(9) = 2$, $C(10) = 3$, and $C(12) = 4$.

(a) How many ways are there to make change of 20 units from $\{1, 6, 10\}$?

(b) Give an efficient algorithm to compute $C(n)$, and analyze its complexity. (Hint: think in terms of computing $C(n, d)$, the number of ways to make change of n units with highest denomination d. Be careful to avoid overcounting.)

8-8. *[6]* In the *single-processor scheduling problem*, we are given a set of n jobs J. Each job i has a processing time t_i, and a deadline d_i. A feasible schedule is a permutation of the jobs such that when the jobs are performed in that order, every job is finished before its deadline. The greedy algorithm for single-processor scheduling selects the job with the earliest deadline first.

Show that if a feasible schedule exists, then the schedule produced by this greedy algorithm is feasible.

Number Problems

8-9. *[6]* The *knapsack problem* is as follows: given a set of integers $S = \{s_1, s_2, \ldots, s_n\}$, and a given target number T, find a subset of S that adds up exactly to T. For example, within $S = \{1, 2, 5, 9, 10\}$ there is a subset that adds up to $T = 22$ but not $T = 23$.

Give a correct programming algorithm for knapsack that runs in $O(nT)$ time.

8-10. *[6]* The *integer partition* takes a set of positive integers $S = s_1, \ldots, s_n$ and asks if there is a subset $I \in S$ such that

$$\sum_{i \in I} s_i = \sum_{i \notin I} s_i$$

Let $\sum_{i \in S} s_i = M$. Give an $O(nM)$ dynamic programming algorithm to solve the integer partition problem.

8-11. *[5]* Assume that there are n numbers (some possibly negative) on a circle, and we wish to find the maximum contiguous sum along an arc of the circle. Give an efficient algorithm for solving this problem.

8-12. *[5]* A certain string processing language allows the programmer to break a string into two pieces. It costs n units of time to break a string of n characters into two pieces, since this involves copying the old string. A programmer wants to break a string into many pieces, and the order in which the breaks are made can affect the total amount of time used. For example, suppose we wish to break a 20-character string after characters 3, 8, and 10. If the breaks are made in left-right order, then the first break costs 20 units of time, the second break costs 17 units of time, and the third break costs 12 units of time, for a total of 49 steps. If the breaks are made in right-left order, the first break costs 20 units of time, the second break costs 10 units of time, and the third break costs 8 units of time, for a total of only 38 steps.

Give a dynamic programming algorithm that takes a list of character positions after which to break and determines the cheapest break cost in $O(n^3)$ time.

8-13. *[5]* Consider the following data compression technique. We have a table of m text strings, each at most k in length. We want to encode a data string D of length n using as few text strings as possible. For example, if our table contains *(a,ba,abab,b)* and the data string is *bababbaababa*, the best way to encode it is *(b,abab,ba,abab,a)*—a total of five code words. Give an $O(nmk)$ algorithm to find the length of the best

encoding. You may assume that every string has at least one encoding in terms of the table.

8-14. *[5]* The traditional world chess championship is a match of 24 games. The current champion retains the title in case the match is a tie. Each game ends in a win, loss, or draw (tie) where wins count as 1, losses as 0, and draws as 1/2. The players take turns playing white and black. White has an advantage, because he moves first. The champion plays white in the first game. He has probabilities w_{w}, w_{d}, and w_{l} of winning, drawing, and losing playing white, and has probabilities b_{w}, b_{d}, and b_{l} of winning, drawing, and losing playing black.

(a) Write a recurrence for the probability that the champion retains the title. Assume that there are g games left to play in the match and that the champion needs to win i games (which may end in a 1/2).

(b) Based on your recurrence, give a dynamic programming to calculate the champion's probability of retaining the title.

(c) Analyze its running time for an n game match.

8-15. *[8]* Eggs break when dropped from great enough height. Specifically, there must be a floor f in any sufficiently tall building such that an egg dropped from the fth floor breaks, but one dropped from the $(f-1)$st floor will not. If the egg always breaks, then $f = 1$. If the egg never breaks, then $f = n + 1$.

You seek to find the critical floor f using an n-story building. The only operation you can perform is to drop an egg off some floor and see what happens. You start out with k eggs, and seek to drop eggs as few times as possible. Broken eggs cannot be reused. Let $E(k, n)$ be the minimum number of egg droppings that will always suffice.

(a) Show that $E(1, n) = n$.

(b) Show that $E(k, n) = \Theta(n^{\frac{1}{k}})$.

(c) Find a recurrence for $E(k, n)$. What is the running time of the dynamic program to find $E(k, n)$?

Graph Problems

8-16. *[4]* Consider a city whose streets are defined by an $X \times Y$ grid. We are interested in walking from the upper left-hand corner of the grid to the lower right-hand corner.

Unfortunately, the city has bad neighborhoods, whose intersections we do not want to walk in. We are given an $X \times Y$ matrix BAD, where *BAD[i,j] = "yes"* if and only if the intersection between streets i and j is in a neighborhood to avoid.

(a) Give an example of the contents of BAD such that there is no path across the grid avoiding bad neighborhoods.

(b) Give an $O(XY)$ algorithm to find a path across the grid that avoids bad neighborhoods.

(c) Give an $O(XY)$ algorithm to find the *shortest* path across the grid that avoids bad neighborhoods. You may assume that all blocks are of equal length. For partial credit, give an $O(X^2Y^2)$ algorithm.

8-17. *[5]* Consider the same situation as the previous problem. We have a city whose streets are defined by an $X \times Y$ grid. We are interested in walking from the upper left-hand corner of the grid to the lower right-hand corner. We are given an $X \times Y$ matrix BAD, where $BAD[i,j]$ = *"yes"* if and only if the intersection between streets i and j is somewhere we want to avoid.

If there were no bad neighborhoods to contend with, the shortest path across the grid would have length $(X-1)+(Y-1)$ blocks, and indeed there would be many such paths across the grid. Each path would consist of only rightward and downward moves.

Give an algorithm that takes the array BAD and returns the *number* of safe paths of length $X+Y-2$. For full credit, your algorithm must run in $O(XY)$.

Design Problems

8-18. *[4]* Consider the problem of storing n books on shelves in a library. The order of the books is fixed by the cataloging system and so cannot be rearranged. Therefore, we can speak of a book b_i, where $1 \leq i \leq n$, that has a thickness t_i and height h_i. The length of each bookshelf at this library is L.

Suppose all the books have the same height h (i.e. , $h = h_i = h_j$ for all i, j) and the shelves are all separated by a distance of greater than h, so any book fits on any shelf. The greedy algorithm would fill the first shelf with as many books as we can until we get the smallest i such that b_i does not fit, and then repeat with subsequent shelves. Show that the greedy algorithm always finds the optimal shelf placement, and analyze its time complexity.

8-19. *[6]* This is a generalization of the previous problem. Now consider the case where the height of the books is not constant, but we have the freedom to adjust the height of each shelf to that of the tallest book on the shelf. Thus the cost of a particular layout is the sum of the heights of the largest book on each shelf.

- Give an example to show that the greedy algorithm of stuffing each shelf as full as possible does not always give the minimum overall height.
- Give an algorithm for this problem, and analyze its time complexity. Hint: use dynamic programming.

8-20. *[5]* We wish to compute the laziest way to dial given n-digit number on a standard push-button telephone using two fingers. We assume that the two fingers start out on the * and # keys, and that the effort required to move a finger from one button to another is proportional to the Euclidean distance between them. Design an algorithm that computes the method of dialing that involves moving your fingers the smallest amount of total distance, where k is the number of distinct keys on the keypad ($k = 16$ for standard telephones). Try to use $O(nk^3)$ time.

8-21. *[6]* Given an array of n real numbers, consider the problem of finding the maximum sum in any contiguous subvector of the input. For example, in the array

$$\{31, -41, 59, 26, -53, 58, 97, -93, -23, 84\}$$

the maximum is achieved by summing the third through seventh elements, where $59+26+(-53)+58+97 = 187$. When all numbers are positive, the entire array is the answer, while when all numbers are negative, the empty array maximizes the total at 0.

- Give a simple, clear, and correct $\Theta(n^2)$-time algorithm to find the maximum contiguous subvector.
- Now give a $\Theta(n)$-time dynamic programming algorithm for this problem. To get partial credit, you may instead give a *correct* $O(n \log n)$ divide-and-conquer algorithm.

8-22. *[7]* Consider the problem of examining a string $x = x_1 x_2 \ldots x_n$ from an alphabet of k symbols, and a multiplication table over this alphabet. Decide whether or not it is possible to parenthesize x in such a way that the value of the resulting expression is a, where a belongs to the alphabet. The multiplication table is neither commutative or associative, so the order of multiplication matters.

	a	b	c
a	a	c	c
b	a	a	b
c	c	c	c

For example, consider the above multiplication table and the string *bbbba*. Parenthesizing it $(b(bb))(ba)$ gives a, but $((((bb)b)b)a)$ gives c.
Give an algorithm, with time polynomial in n and k, to decide whether such a parenthesization exists for a given string, multiplication table, and goal element.

8-23. *[6]* Let α and β be constants. Assume that it costs α to go left in a tree, and β to go right. Devise an algorithm that builds a tree with optimal worst case cost, given keys $k_1, \ldots, k_n$ and the probabilities that each will be searched $p_1, \ldots, p_n$.

Interview Problems

8-24. *[5]* Given a set of coin denominators, find the minimum number of coins to make a certain amount of change.

8-25. *[5]* You are given an array of n numbers, each of which may be positive, negative, or zero. Give an efficient algorithm to identify the index positions i and j to the maximum sum of the ith through jth numbers.

8-26. *[7]* Observe that when you cut a character out of a magazine, the character on the reverse side of the page is also removed. Give an algorithm to determine whether you can generate a given string by pasting cutouts from a given magazine. Assume that you are given a function that will identify the character and its position on the reverse side of the page for any given character position.

Programming Challenges

These programming challenge problems with robot judging are available at *http://www.programming-challenges.com* or *http://online-judge.uva.es*.

8-1. "Is Bigger Smarter?" – Programming Challenges 111101, UVA Judge 10131.

8-2. "Weights and Measures" – Programming Challenges 111103, UVA Judge 10154.

8-3. "Unidirectional TSP" – Programming Challenges 111104, UVA Judge 116.

8-4. "Cutting Sticks" – Programming Challenges 111105, UVA Judge 10003.

8-5. "Ferry Loading" – Programming Challenges 111106, UVA Judge 10261.

9

Intractable Problems and Approximation Algorithms

We now introduce techniques for proving that *no* efficient algorithm exists for a given problem. The practical reader is probably squirming at the notion of proving anything, and will be particularly alarmed at the idea of investing time to prove that something does not exist. Why are you better off knowing that something you don't know how to do in fact can't be done at all?

The truth is that the theory of NP-completeness is an immensely useful tool for the algorithm designer, even though all it provides are negative results. The theory of NP-completeness enables the algorithm designer to focus her efforts more productively, by revealing that the search for an efficient algorithm for this particular problem is doomed to failure. When one *fails* to show a problem is hard, that suggests there may well be an efficient algorithm to solve it. Two of the war stories in this book described happy results springing from bogus claims of hardness.

The theory of NP-completeness also enables us to identify what properties make a particular problem hard. This provides direction for us to model it in different ways or exploit more benevolent characteristics of the problem. Developing a sense for which problems are hard is an important skill for algorithm designers, and only comes from hands-on experience with proving hardness.

The fundamental concept we will use is that of *reductions* between pairs of problems, showing that the problems are really equivalent. We illustrate this idea through a series of reductions, each of which either yields an efficient algorithm or an argument that no such algorithm can exist. We also provide brief introductions to (1) the complexity-theoretic aspects of NP-completeness, one of the most fundamental notions in Computer Science, and (2) the theory of approximation algorithms, which leads to heuristics that probably return something *close* to the optimal solution.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_9,

9.1 Problems and Reductions

We have encountered several problems in this book for which we couldn't find any efficient algorithm. The theory of NP-completeness provides the tools needed to show that all these problems are on some level really the same problem.

The key idea to demonstrating the hardness of a problem is that of a *reduction*, or translation, between two problems. The following allegory of NP-completeness may help explain the idea. A bunch of kids take turns fighting each other in the schoolyard to prove how "tough" they are. Adam beats up Bill, who then beats up Chuck. So who if any among them is "tough?" The truth is that there is no way to know without an external standard. If I told you that Chuck was in fact Chuck Norris, certified tough guy, you have to be impressed—both Adam and Bill are at least as tough as he is. On the other hand, suppose I tell you it is a kindergarten school yard. No one would call me tough, but even I can take out Adam. This proves that none of the three of them should be called be tough. In this allegory, each fight represents a reduction. Chuck Norris takes on the role of satisfiability—a certifiably hard problem. The part of an inefficient algorithm with a possible shot at redemption is played by me.

Reductions are operations that convert one problem into another. To describe them, we must be somewhat rigorous in our definitions. An algorithmic *problem* is a general question, with parameters for input and conditions on what constitutes a satisfactory answer or solution. An *instance* is a problem with the input parameters specified. The difference can be made clear by an example:

Problem: The Traveling Salesman Problem (TSP)
Input: A weighted graph G.
Output: Which tour $\{v_1, v_2, ..., v_n\}$ minimizes $\sum_{i=1}^{n-1} d[v_i, v_{i+1}] + d[v_n, v_1]$?

Any weighted graph defines an instance of TSP. Each particular *problem* has at least one minimum cost tour. The general traveling salesman *instance* asks for an algorithm to find the optimal tour for all possible instances.

9.1.1 The Key Idea

Now consider two algorithmic problems, called *Bandersnatch* and *Bo-billy*. Suppose that I gave you the following reduction/algorithm to solve the *Bandersnatch* problem:

Bandersnatch(G)
    Translate the input G to an instance Y of the Bo-billy problem.
    Call the subroutine Bo-billy on Y to solve this instance.
    Return the answer of Bo-billy(Y) as the answer to Bandersnatch(G).

This algorithm will *correctly* solve the Bandersnatch problem provided that the translation to Bo-billy always preserves the correctness of the answer. In other words, the translation has the property that for any instance of G,

$$\text{Bandersnatch}(G) = \text{Bo-billy}(Y)$$

A translation of instances from one type of problem to instances of another such that the answers are preserved is what is meant by a *reduction*.

Now suppose this reduction translates G to Y in $O(P(n))$ time. There are two possible implications:

- *If* my Bo-billy subroutine ran in $O(P'(n))$, this means I could solve the Bandersnatch problem in $O(P(n) + P'(n))$ by spending the time to translate the problem and then the time to execute the Bo-Billy subroutine.
- *If* I know that $\Omega(P'(n))$ is a lower bound on computing Bandersnatch, meaning there definitely exists no faster way to solve it, then $\Omega(P'(n) - P(n))$ *must* be a lower bound to compute Bo-billy. Why? If I could solve Bo-billy any faster, then I could violate my lower bound by solving Bandersnatch using the above reduction. This implies that there can be no way to solve Bo-billy any faster than claimed.

This first argument is Steve demonstrating the weakness of the entire schoolyard with a quick right to Adam's chin. The second highlights the Chuck Norris approach we will use to prove that problems are hard. Essentially, this reduction shows that Bo-billy is no easier than Bandersnatch. Therefore, if Bandersnatch is hard this means Bo-billy must also be hard.

We will illustrate this point by giving several problem reductions in this chapter.

> *Take-Home Lesson:* Reductions are a way to show that two problems are essentially identical. A fast algorithm (or the lack of one) for one of the problems implies a fast algorithm (or the lack of one) for the other.

9.1.2 Decision Problems

Reductions translate between problems so that their answers are identical in every problem instance. Problems differ in the *range* or *type* of possible answers. The traveling salesman problem returns a permutation of vertices as the answer, while other types of problems return numbers as answers, perhaps restricted to positive numbers or integers.

The simplest interesting class of problems have answers restricted to true and false. These are called *decision problems*. It proves convenient to reduce/translate answers between decision problems because both only allow true and false as possible answers.

Fortunately, most interesting optimization problems can be phrased as decision problems that capture the essence of the computation. For example, the traveling salesman decision problem could be defined as:

Problem: The Traveling Salesman Decision Problem
Input: A weighted graph G and integer k.
Output: Does there exist a TSP tour with cost $\leq k$?

The decision version captures the heart of the traveling salesman problem, in that if you had a fast algorithm for the decision problem, you could use it to do a binary search with different values of k and quickly hone in on the optimal solution. With a little more cleverness, you could reconstruct the actual tour permutation using a fast solution to the decision problem.

From now on we will generally talk about decision problems, because it proves easier and still captures the power of the theory.

9.2 Reductions for Algorithms

An engineer and an algorist are sitting in a kitchen. The algorist asks the engineer to boil some water, so the engineer gets up, picks up the kettle from the counter top, adds water from the sink, brings it to the burner, turns on the burner, waits for the whistling sound, and turns off the burner. Sometime later, the engineer asks the algorist to boil more water. She gets up, takes the kettle from the burner, moves it over to the counter top, and sits down. "Done." she says, "I have *reduced* the task to a solved problem."

This boiling water reduction illustrates an honorable way to generate new algorithms from old. If we can translate the input for a problem we *want to solve* into input for a problem we *know how to solve*, we can compose the translation and the solution into an algorithm for our problem.

In this section, we look at several reductions that lead to efficient algorithms. To solve problem a, we translate/reduce the a instance to an instance of b, then solve this instance using an efficient algorithm for problem b. The overall running time is the time needed to perform the reduction plus that solve the b instance.

9.2.1 Closest Pair

The *closest pair* problem asks to find the pair of numbers within a set that have the smallest difference between them. We can make it a decision problem by asking if this value is less than some threshold:

Input: A set S of n numbers, and threshold t.
Output: Is there a pair $s_i, s_j \in S$ such that $|s_i - s_j| \leq t$?

The closest pair is a simple application of sorting, since the closest pair must be neighbors after sorting. This gives the following algorithm:

CloseEnoughPair(S,t)
    Sort S.
    Is $\min_{1 \leq i < n} |s_i - s_{i+1}| \leq t$?

There are several things to note about this simple reduction.

1. The decision version captured what is interesting about the problem, meaning it is no easier than finding the actual closest pair.
2. The complexity of this algorithm depends upon the complexity of sorting. Use an $O(n \log n)$ algorithm to sort, and it takes $O(n \log n + n)$ to find the closest pair.
3. This reduction and the fact that there is an $\Omega(n \log n)$ lower bound on sorting *does not* prove that a close-enough pair must take $\Omega(n \log n)$ time in the worst case. Perhaps this is just a slow algorithm for a close-enough pair, and there is a faster one lurking somewhere?
4. On the other hand, *if* we knew that a close-enough pair required $\Omega(n \log n)$ time to solve in the worst case, this reduction would suffice to prove that sorting couldn't be solved any faster than $\Omega(n \log n)$ because that would imply a faster algorithm for a close-enough pair.

9.2.2 Longest Increasing Subsequence

In Chapter 8, we demonstrated how dynamic programming can be used to solve a variety of problems, including string edit distance (Section 8.2 (page 280)) and longest increasing subsequence (Section 8.3 (page 289)). To review,

Problem: Edit Distance
Input: Integer or character sequences S and T; penalty costs for each insertion (c_{ins}), deletion (c_{del}), and substitution (c_{del}).
Output: What is the minimum cost sequence of operations to transform S to T?

Problem: Longest Increasing Subsequence
Input: An integer or character sequence S.
Output: What is the longest sequence of integer positions $\{p_1, \ldots, p_m\}$ such that $p_i < p_{i+1}$ and $S_{p_i} < S_{p_{i+1}}$?

In fact, longest increasing subsequence (LIS) can be solved as a special case of edit distance:

```
Longest Increasing Subsequence(S)
    T = Sort(S)
    c_ins = c_del = 1
    c_sub = ∞
    Return (|S| - EditDistance(S,T,c_ins,c_del,c_del)/2)
```

Why does this work? By constructing the second sequence T as the elements of S sorted in increasing order, we ensure that any common subsequence must be an

increasing subsequence. If we are never allowed to do any substitutions (because $c_{sub} = \infty$), the optimal alignment of two sequences finds the longest common subsequence between them and removes everything else. Thus, transforming $\{3, 1, 2\}$ to $\{1, 2, 3\}$ costs two, namely inserting and deleting the unmatched 3. The length of S minus half this cost gives the length of the LIS.

What are the implications of this reduction? The reduction takes $O(n \log n)$ time. Because edit distance takes time $O(|S| \cdot |T|)$, this gives a quadratic algorithm to find the longest increasing subsequence of S, which is the same complexity as the the algorithm presented in Section 8.3 (page 289). In fact, there exists a faster $O(n \log n)$ algorithm for LIS using clever data structures, while edit distance is known to be quadratic in the worst case. Here, our reduction gives us a simple but not optimal polynomial-time algorithm.

9.2.3 Least Common Multiple

The *least common multiple* and *greatest common divisor* problems arise often in working with integers. We say b *divides* a ($b|a$) if there exists an integer d that $a = bd$. Then:

Problem: Least Common Multiple (lcm)
Input: Two integers x and y.
Output: Return the smallest integer m such that m is a multiple of x and m is also a multiple of y.

Problem: Greatest Common Divisor (gcd)
Input: Two integers x and y.
Output: Return the largest integer d such that d divides x and d divides y.

For example, $lcm(24, 36) = 72$ and $gcd(24, 36) = 12$. Both problems can be solved easily after reducing x and y to their prime factorizations, but no efficient algorithm is known for factoring integers (see Section 13.8 (page 420)). Fortunately, Euclid's algorithm gives an efficient way to solve greatest common divisor without factoring. It is a recursive algorithm that rests on two observations. First,

> if $b|a$, then $gcd(a, b) = b$.

This should be pretty clear. if b divides a, then $a = bk$ for some integer k, and thus $gcd(bk, b) = b$. Second,

> If $a = bt + r$ for integers t and r, then $gcd(a, b) = gcd(b, r)$.

Since $x \cdot y$ is a multiple of both x and y, $lcm(x, y) \leq xy$. The only way that there can be a smaller common multiple is if there is some nontrivial factor shared between x and y. This observation, coupled with Euclid's algorithm, provides an efficient way to compute least common multiple, namely

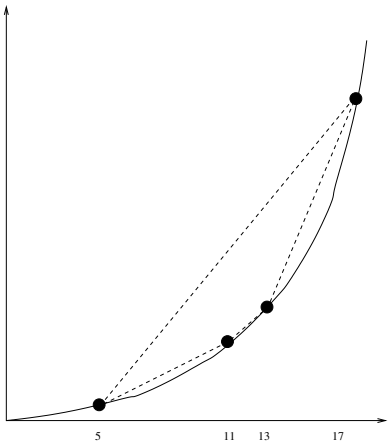

Figure 9.1: Reducing convex hull to sorting by mapping points to a parabola

LeastCommonMultiple(x,y)
Return $(xy/gcd(x, y))$.

This reduction gives us a nice way to reuse Euclid's efforts on another problem.

9.2.4 Convex Hull (*)

Our final example of a reduction from an "easy" problem (i.e. , one that can be solved in polynomial time) goes from finding convex hulls to sorting numbers. A polygon is *convex* if the straight line segment drawn between any two points inside the polygon P must lie completely within the polygon. This is the case when P contains no notches or *concavities*, so convex polygons are nicely shaped. The convex hull provides a very useful way to provide structure to a point set. Applications are presented in Section 17.2 (page 568).

Problem: Convex Hull
Input: A set S of n points in the plane.
Output: Find the smallest convex polygon containing all the points of S.

We now show how to transform from sorting to convex hull. This means we must translate each number to a point. We do so by mapping x to (x, x^2). Why? It means each integer is mapped to a point on the parabola $y = x^2$. Since this parabola is convex, every point must be on the convex hull. Furthermore, since neighboring points on the convex hull have neighboring x values, the convex hull returns the points sorted by the x-coordinate—i.e., the original numbers. Creating and reading off the points takes $O(n)$ time:

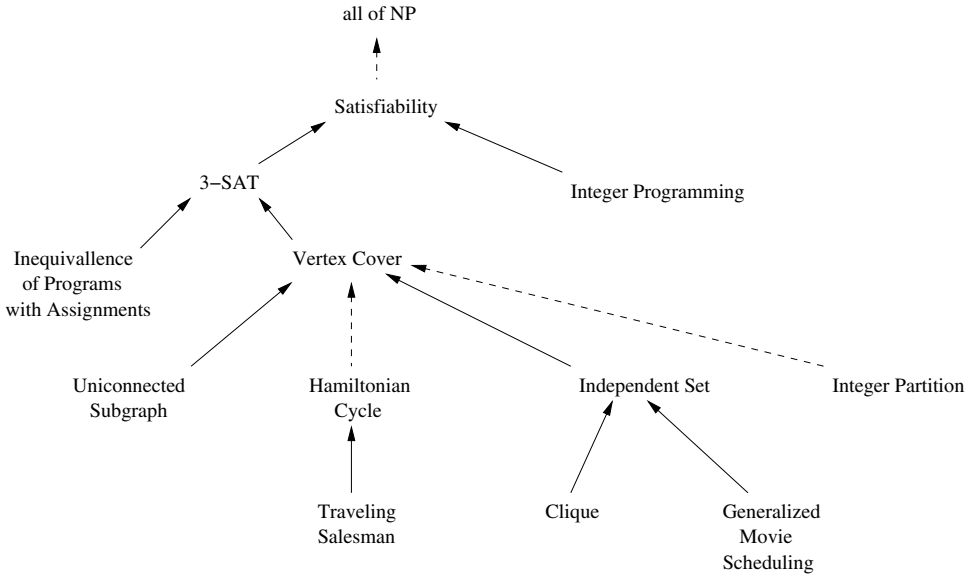

Figure 9.2: A portion of the reduction tree for NP-complete problems. Solid lines denote the reductions presented in this chapter

---

Sort(S)
    For each $i \in S$, create point (i, i^2).
    Call subroutine convex-hull on this point set.
    From the leftmost point in the hull,
        read off the points from left to right.

What does this mean? Recall the sorting lower bound of $\Omega(n \lg n)$. If we could compute convex hull in better than $n \lg n$, this reduction implies that we could sort faster than $\Omega(n \lg n)$, which violates our lower bound. Thus, convex hull must take $\Omega(n \lg n)$ as well! Observe that any $O(n \lg n)$ convex hull algorithm also gives us a complicated but correct $O(n \lg n)$ sorting algorithm as well.

9.3 Elementary Hardness Reductions

The reductions in the previous section demonstrate transformations between pairs of problems for which efficient algorithms exist. However, we are mainly concerned with using reductions to prove hardness, by showing that a fast algorithm for *Bandersnatch* would imply one that cannot exist for *Bo-billy*.

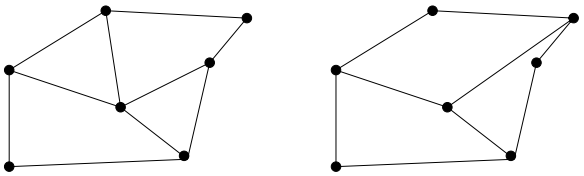

Figure 9.3: Graphs with (l) and without (r) Hamiltonian cycles

For now, I want you to trust me when I say that *Hamiltonian cycle* and *vertex cover* are hard problems. The entire picture (presented in Figure 9.2) will become clear by the end of the chapter.

9.3.1 Hamiltonian Cycle

The Hamiltonian cycle problem is one of the most famous in graph theory. It seeks a tour that visits each vertex of a given graph exactly once. It has a long history and many applications, as discussed in Section 16.5. Formally, it is defined as:

Problem: Hamiltonian Cycle
Input: An unweighted graph G.
Output: Does there exist a simple tour that visits each vertex of G without repetition?

Hamiltonian cycle has some obvious similarity to the traveling salesman problem. Both problems seek a tour that visits each vertex exactly once. There are also differences between the two problems. TSP works on weighted graphs, while Hamiltonian cycle works on unweighted graphs. The following reduction from Hamiltonian cycle to traveling salesman shows that the similarities are greater than the differences:

HamiltonianCycle($G = (V, E)$)
    Construct a complete weighted graph $G' = (V', E')$ where $V' = V$.
    $n = |V|$
    for $i = 1$ to n do
        for $j = 1$ to n do
            if $(i, j) \in E$ then $w(i, j) = 1$ else $w(i, j) = 2$
    Return the answer to Traveling-Salesman-Decision-Problem(G', n).

The actual reduction is quite simple, with the translation from unweighted to weighted graph easily performed in $O(n^2)$ time. Further, this translation is designed to ensure that the answers of the two problems will be identical. If the graph G has a Hamiltonian cycle $\{v_1, \ldots, v_n\}$, then this exact same tour will correspond to n edges in E', each with weight 1. This gives a TSP tour in G' of weight exactly

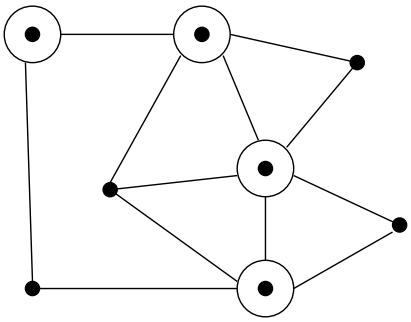

Figure 9.4: Circled vertices form a vertex cover, and the others form an independent set

n. If G does not have a Hamiltonian cycle, then there can be no such TSP tour in G' because the only way to get a tour of cost n in G would be to use only edges of weight 1, which implies a Hamiltonian cycle in G.

This reduction is both efficient and truth preserving. A fast algorithm for TSP would imply a fast algorithm for Hamiltonian cycle, while a hardness proof for Hamiltonian cycle would imply that TSP is hard. Since the latter is the case, this reduction shows that TSP is hard, at least as hard as the Hamiltonian cycle.

9.3.2 Independent Set and Vertex Cover

The vertex cover problem, discussed more thoroughly in Section 16.3 (page 530), asks for a small set of vertices that contacts each edge in a graph. More formally:

Problem: Vertex Cover
Input: A graph $G = (V, E)$ and integer $k \leq |V|$.
Output: Is there a subset S of at most k vertices such that every $e \in E$ contains at least one vertex in S?

It is trivial to find *a* vertex cover of a graph, namely the cover that consists of all the vertices. More tricky is to cover the edges using as small a set of vertices as possible. For the graph in Figure 9.4, four of the eight vertices are sufficient to cover.

A set of vertices S of graph G is *independent* if there are no edges (x, y) where both $x \in S$ and $y \in S$. This means there are no edges between any two vertices in independent set. As discussed in Section 16.2 (page 528), independent set arises in facility location problems. The maximum independent set decision problem is formally defined:

Problem: Independent Set
Input: A graph G and integer $k \leq |V|$.
Output: Does there exist an independent set of k vertices in G?

Both vertex cover and independent set are problems that revolve around finding special subsets of vertices: the first with representatives of every edge, the second with no edges. If S is the vertex cover of G, the remaining vertices $S - V$ must form an independent set, for if an edge had both vertices in $S - V$, then S could not have been a vertex cover. This gives us a reduction between the two problems:

VertexCover(G, k)
    $G' = G$
    $k' = |V| - k$
    Return the answer to IndependentSet(G', k')

Again, a simple reduction shows that the two problems are identical. Notice how this translation occurs without any knowledge of the answer. We transform the *input*, not the solution. This reduction shows that the hardness of vertex cover implies that independent set must also be hard. It is easy to reverse the roles of the two problems in this particular reduction, thus proving that both problems are equally hard.

Stop and Think: Hardness of General Movie Scheduling

Problem: Prove that the *general* movie scheduling problem is NP-complete, with a reduction from independent set.

Problem: General Movie Scheduling Decision Problem
Input: A set I of n *sets of* intervals on the line, integer k.
Output: Can a subset of at least k mutually nonoverlapping interval *sets* which can be selected from I?

Solution: Recall the movie scheduling problem, discussed in Section 1.2 (page 9). Each possible movie project came with a single time interval during which filming took place. We sought the largest possible subset of movie projects such that no two conflicting projects (i.e. , both requiring the actor at the same time) were selected.

The general problem allows movie projects to have discontinuous schedules. For example, Project A running from January-March and May-June does not intersect Project B running in April and August, but *does* collide with Project C running from June-July.

If we are going to prove general movie scheduling hard from independent set, what is Bandersnatch and what is Bo-billy? We need to show how to translate *all* independent set problems into instances of movie scheduling—i.e. , sets of disjointed line intervals.

What is the correspondence between the two problems? Both problems involve selecting the largest subsets possible—of vertices and movies, respectively. This

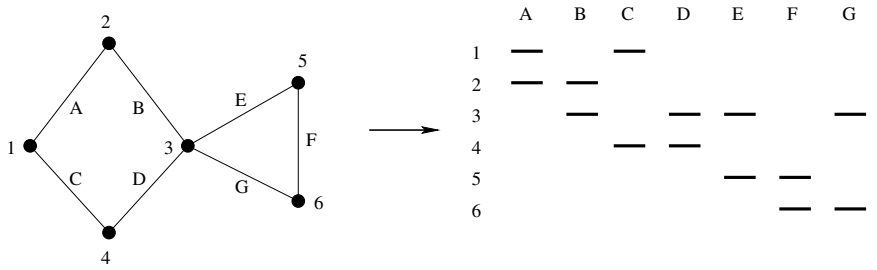

Figure 9.5: Reduction from independent set to generalized movie scheduling, with numbered vertices and lettered edges

---

suggests we must translate vertices into movies. Further, both require the selected elements not to interfere, by sharing an edge or overlapping an interval, respectively.

IndependentSet(G, k)
    $I = \emptyset$
    For the ith edge (x, y), $1 \leq i \leq m$
        Add interval $[i, i + 0.5]$ for movie X to I
        Add interval $[i, i + 0.5]$ for movie y to I
    Return the answer to GeneralMovieScheduling(I, k)

My construction is as follows. Create an interval on the line for each of the m edges of the graph. The movie associated with each vertex will contain the intervals for the edges adjacent with it, as shown in Figure 9.5.

Each pair of vertices sharing an edge (forbidden to be in independent set) defines a pair of movies sharing a time interval (forbidden to be in the actor's schedule). Thus, the largest satisfying subsets for both problems are the same, and a fast algorithm for solving general movie scheduling gives us a fast algorithm for solving independent set. Thus, general movie scheduling must be hard as hard as independent set, and hence NP-complete. ∎

9.3.3 Clique

A social clique is a group of mutual friends who all hang around together. A graph-theoretic *clique* is a complete subgraph where each vertex pair has an edge between them. Cliques are the densest possible subgraphs:

Problem: Maximum Clique
Input: A graph $G = (V, E)$ and integer $k \leq |V|$.
Output: Does the graph contain a clique of k vertices; i.e. , is there a subset $S \subset V$, where $|S| \leq k$, such that every pair of vertices in S defines an edge of G?

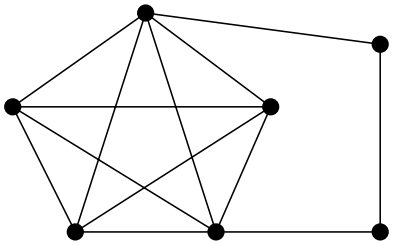

Figure 9.6: A small graph with a five-vertex clique

The graph in Figure 9.6 contains a clique of five vertices. Within the friendship graph, we would expect to see large cliques corresponding to workplaces, neighborhoods, religious organizations, and schools. Applications of cliques are further discussed in Section 16.1 (page 525).

In the independent set problem, we looked for a subset S with no edges between two vertices of S. This contrasts with clique, where we insist that there always be an edge between two vertices. A reduction between these problems follows by reversing the roles of edges and nonedges—an operation known as *complementing* the graph:

IndependentSet(G, k)
    Construct a graph $G' = (V', E')$ where $V' = V$, and
        For all (i, j) not in E, add (i, j) to E'
    Return the answer to Clique(G', k)

These last two reductions provide a chain linking of three different problems. The hardness of clique is implied by the hardness of independent set, which is implied by the hardness of vertex cover. By constructing reductions in a chain, we link together pairs of problems in implications of hardness. Our work is done as soon as all these chains begin with a single problem that is accepted as hard. Satisfiability is the problem that will serve as the first link in this chain.

9.4 Satisfiability

To demonstrate the hardness of all problems using reductions, we must start with a single problem that is absolutely, certifiably, undeniably hard. The mother of all NP-complete problems is a logic problem named *satisfiability*:

Problem: Satisfiability
Input: A set of Boolean variables V and a set of clauses C over V.
Output: Does there exist a satisfying truth assignment for C—i.e. , a way to set the variables $v_1, \ldots, v_n$ true or false so that each clause contains at least one true literal?