8

Dynamic Programming

The most challenging algorithmic problems involve optimization, where we seek to find a solution that maximizes or minimizes some function. Traveling salesman is a classic optimization problem, where we seek the tour visiting all vertices of a graph at minimum total cost. But as shown in Chapter 1, it is easy to propose "algorithms" solving TSP that generate reasonable-looking solutions but did not *always* produce the minimum cost tour.

Algorithms for optimization problems require proof that they always return the best possible solution. Greedy algorithms that make the best local decision at each step are typically efficient but usually do not guarantee global optimality. Exhaustive search algorithms that try all possibilities and select the best always produce the optimum result, but usually at a prohibitive cost in terms of time complexity.

Dynamic programming combines the best of both worlds. It gives us a way to design custom algorithms that systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing (thus providing efficiency). By storing the *consequences* of all possible decisions and using this information in a systematic way, the total amount of work is minimized.

Once you understand it, dynamic programming is probably the easiest algorithm design technique to apply in practice. In fact, I find that dynamic programming algorithms are often easier to reinvent than to try to look up in a book. That said, *until* you understand dynamic programming, it seems like magic. You must figure out the trick before you can use it.

Dynamic programming is a technique for efficiently implementing a recursive algorithm by storing partial results. The trick is seeing whether the naive recursive algorithm computes the same subproblems over and over and over again. If so, storing the answer for each subproblems in a table to look up instead of recompute

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_8,

can lead to an efficient algorithm. Start with a recursive algorithm or definition. Only once we have a correct recursive algorithm do we worry about speeding it up by using a results matrix.

Dynamic programming is generally the right method for optimization problems on combinatorial objects that have an inherent *left to right* order among components. Left-to-right objects includes: character strings, rooted trees, polygons, and integer sequences. Dynamic programming is best learned by carefully studying examples until things start to click. We present three war stories where dynamic programming played the decisive role to demonstrate its utility in practice.

8.1 Caching vs. Computation

Dynamic programming is essentially a tradeoff of space for time. Repeatedly recomputing a given quantity is harmless unless the time spent doing so becomes a drag on performance. Then we are better off storing the results of the initial computation and looking them up instead of recomputing them again.

The tradeoff between space and time exploited in dynamic programming is best illustrated when evaluating recurrence relations such as the Fibonacci numbers. We look at three different programs for computing them below.

8.1.1 Fibonacci Numbers by Recursion

The Fibonacci numbers were originally defined by the Italian mathematician Fibonacci in the thirteenth century to model the growth of rabbit populations. Rabbits breed, well, like rabbits. Fibonacci surmised that the number of pairs of rabbits born in a given year is equal to the number of pairs of rabbits born in each of the two previous years, starting from one pair of rabbits in the first year. To count the number of rabbits born in the nth year, he defined the recurrence relation:

$$F_n = F_{n-1} + F_{n-2}$$

with basis cases $F_0 = 0$ and $F_1 = 1$. Thus, $F_2 = 1$, $F_3 = 2$, and the series continues $\{3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots\}$. As it turns out, Fibonacci's formula didn't do a very good job of counting rabbits, but it does have a host of interesting properties.

Since they are defined by a recursive formula, it is easy to write a recursive program to compute the nth Fibonacci number. A recursive function algorithm written in C looks like this:

```
long fib_r(int n)
{
        if (n == 0) return(0);
        if (n == 1) return(1);

        return(fib_r(n-1) + fib_r(n-2));
}
```

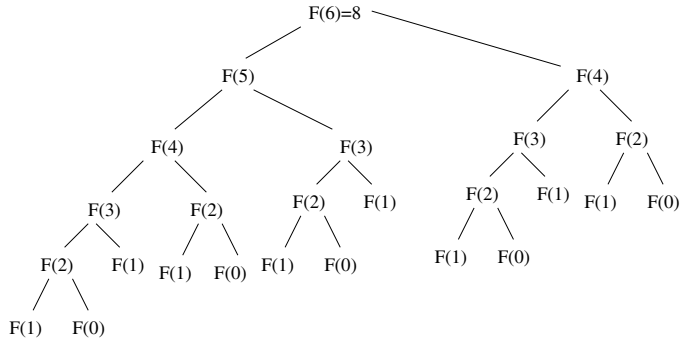

Figure 8.1: The computation tree for computing Fibonacci numbers recursively

The course of execution for this recursive algorithm is illustrated by its *recursion tree*, as illustrated in Figure 8.1. This tree is evaluated in a depth-first fashion, as are all recursive algorithms. I encourage you to trace this example by hand to refresh your knowledge of recursion.

Note that $F(4)$ is computed on both sides of the recursion tree, and $F(2)$ is computed no less than five times in this small example. The weight of all this redundancy becomes clear when you run the program. It took more than 7 minutes for my program to compute the first 45 Fibonacci numbers. You could probably do it faster by hand using the right algorithm.

How much time does this algorithm take to compute $F(n)$? Since $F_{n+1}/F_n \approx \phi = (1+\sqrt{5})/2 \approx 1.61803$, this means that $F_n > 1.6^n$. Since our recursion tree has only 0 and 1 as leaves, summing up to such a large number means we must have at least 1.6^n leaves or procedure calls! This humble little program takes exponential time to run!

8.1.2 Fibonacci Numbers by Caching

In fact, we can do much better. We can explicitly store (or *cache*) the results of each Fibonacci computation $F(k)$ in a table data structure indexed by the parameter k. The key to avoiding recomputation is to explicitly check for the value before trying to compute it:

```
#define MAXN    45       /* largest interesting n */
#define UNKNOWN   -1     /* contents denote an empty cell */
long f[MAXN+1];          /* array for caching computed fib values */
```

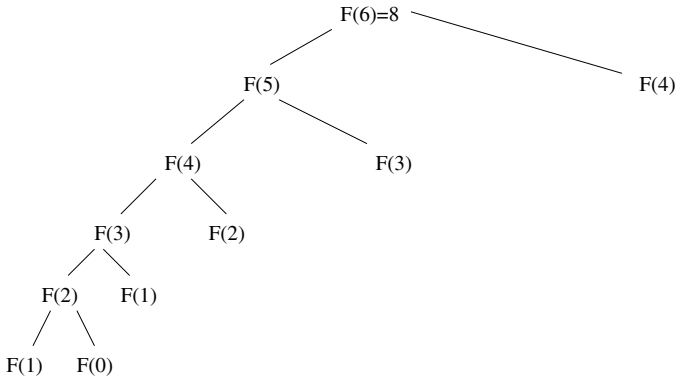

Figure 8.2: The Fibonacci computation tree when caching values

```
long fib_c(int n)
{
        if (f[n] == UNKNOWN)
                f[n] = fib_c(n-1) + fib_c(n-2);

        return(f[n]);
}


long fib_c_driver(int n)
{
        int i;                  /* counter */

        f[0] = 0;
        f[1] = 1;
        for (i=2; i<=n; i++)  f[i] = UNKNOWN;

        return(fib_c(n));
}
```

To compute $F(n)$, we call `fib_c_driver(n)`. This initializes our cache to the two values we initially know ($F(0)$ and $F(1)$) as well as the `UNKNOWN` flag for all the rest we don't. It then calls a look-before-crossing-the-street version of the recursive algorithm.

This cached version runs instantly up to the largest value that can fit in a long integer. The new recursion tree (Figure 8.2) explains why. There is no meaningful branching, because only the left-side calls do computation. The right-side calls find what they are looking for in the cache and immediately return.

What is the running time of this algorithm? The recursion tree provides more of a clue than the code. In fact, it computes $F(n)$ in linear time (in other words, $O(n)$ time) because the recursive function `fib_c(k)` is called exactly twice for each value $0 \leq k \leq n$.

This general method of explicitly caching results from recursive calls to avoid recomputation provides a simple way to get *most* of the benefits of full dynamic programming, so it is worth a more careful look. In principle, such caching can be employed on any recursive algorithm. However, storing partial results would have done absolutely no good for such recursive algorithms as *quicksort*, *backtracking*, and *depth-first search* because all the recursive calls made in these algorithms have distinct *parameter values*. It doesn't pay to store something you will never refer to again.

Caching makes sense only when the space of distinct parameter values is modest enough that we can afford the cost of storage. Since the argument to the recursive function `fib_c(k)` is an integer between 0 and n, there are only $O(n)$ values to cache. A linear amount of space for an exponential amount of time is an excellent tradeoff. But as we shall see, we can do even better by eliminating the recursion completely.

> *Take-Home Lesson:* Explicit caching of the results of recursive calls provides *most* of the benefits of dynamic programming, including usually the same running time as the more elegant full solution. If you prefer doing extra programming to more subtle thinking, you can stop here.

8.1.3 Fibonacci Numbers by Dynamic Programming

We can calculate F_n in linear time more easily by explicitly specifying the order of evaluation of the recurrence relation:

```
long fib_dp(int n)
{
        int i;              /* counter */
        long f[MAXN+1];     /* array to cache computed fib values */

        f[0] = 0;
        f[1] = 1;
        for (i=2; i<=n; i++)  f[i] = f[i-1]+f[i-2];

        return(f[n]);
}
```

We have removed all recursive calls! We evaluate the Fibonacci numbers from smallest to biggest and store all the results, so we know that we have F_{i-1} and F_{i-2} ready whenever we need to compute F_i. The linearity of this algorithm should

be apparent. Each of the n values is computed as the simple sum of two integers in total $O(n)$ time and space.

More careful study shows that we do not need to store all the intermediate values for the entire period of execution. Because the recurrence depends on two arguments, we only need to retain the last two values we have seen:

```
long fib_ultimate(int n)
{
        int i;                  /* counter */
        long back2=0, back1=1;  /* last two values of f[n] */
        long next;              /* placeholder for sum */

        if (n == 0) return (0);

        for (i=2; i<n; i++) {
                next = back1+back2;
                back2 = back1;
                back1 = next;
        }
        return(back1+back2);
}
```

This analysis reduces the storage demands to constant space with no asymptotic degradation in running time.

8.1.4 Binomial Coefficients

We now show how to compute the *binomial coefficients* as another illustration of how to eliminate recursion by specifying the order of evaluation. The binomial coefficients are the most important class of counting numbers, where $\binom{n}{k}$ counts the number of ways to choose k things out of n possibilities.

How do you compute the binomial coefficients? First, $\binom{n}{k} = n!/((n-k)!k!)$, so in principle you can compute them straight from factorials. However, this method has a serious drawback. Intermediate calculations can easily cause arithmetic overflow, even when the final coefficient fits comfortably within an integer.

A more stable way to compute binomial coefficients is using the recurrence relation implicit in the construction of Pascal's triangle:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
```

m / n	0	1	2	3	4	5
0	A					
1	B	G				
2	C	1	H			
3	D	2	3	I		
4	E	4	5	6	J	
5	F	7	8	9	10	K

m / n	0	1	2	3	4	5
0	1					
1	1	1				
2	1	1	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	**5**	1

Figure 8.3: Evaluation order for `binomial_coefficient` at $M[5,4]$ (l). Initialization conditions A-K, recurrence evaluations 1-10. Matrix contents after evaluation (r)

Each number is the sum of the two numbers directly above it. The recurrence relation implicit in this is that

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Why does this work? Consider whether the nth element appears in one of the $\binom{n}{k}$ subsets of k elements. If so, we can complete the subset by picking $k-1$ other items from the other $n-1$. If not, we must pick all k items from the remaining $n-1$. There is no overlap between these cases, and all possibilities are included, so the sum counts all k subsets.

No recurrence is complete without basis cases. What binomial coefficient values do we know without computing them? The left term of the sum eventually drives us down to $\binom{n-k}{0}$. How many ways are there to choose 0 things from a set? Exactly one, the empty set. If this is not convincing, then it is equally good to accept that $\binom{m}{1} = m$ as the basis case. The right term of the sum runs us up to $\binom{k}{k}$. How many ways are there to choose k things from a k-element set? Exactly one—the complete set. Together, these basis cases and the recurrence define the binomial coefficients on all interesting values.

The best way to evaluate such a recurrence is to build a table of possible values up to the size that you are interested in:

Figure 8.3 demonstrates a proper evaluation order for the recurrence. The initialized cells are marked from A-K, denoting the order in which they were assigned values. Each remaining cell is assigned the sum of the cell directly above it and the cell immediately above and to the left. The triangle of cells marked 1 to 10 denote the evaluation order in computing $\binom{5}{4} = 5$ using the code below:

```
long binomial_coefficient(n,m)
int n,m;                          /* computer n choose m */
{
        int i,j;                  /* counters */
        long bc[MAXN][MAXN];      /* table of binomial coefficients */

        for (i=0; i<=n; i++) bc[i][0] = 1;

        for (j=0; j<=n; j++) bc[j][j] = 1;

        for (i=1; i<=n; i++)
                for (j=1; j<i; j++)
                        bc[i][j] = bc[i-1][j-1] + bc[i-1][j];

        return( bc[n][m] );
}
```

Study this function carefully to see how we did it. The rest of this chapter will focus more on formulating and analyzing the appropriate recurrence than the mechanics of table manipulation demonstrated here.

8.2 Approximate String Matching

Searching for patterns in text strings is a problem of unquestionable importance. Section 3.7.2 (page 91) presented algorithms for *exact* string matching—finding where the pattern string P occurs as a substring of the text string T. Life is often not that simple. Words in either the text or pattern can be mispelled (sic), robbing us of exact similarity. Evolutionary changes in genomic sequences or language usage imply that we often search with archaic patterns in mind: "Thou shalt not kill" morphs over time into "You should not murder."

How can we search for the substring closest to a given pattern to compensate for spelling errors? To deal with inexact string matching, we must first define a cost function telling us how far apart two strings are—i.e., a distance measure between pairs of strings. A reasonable distance measure reflects the number of *changes* that must be made to convert one string to another. There are three natural types of changes:

- *Substitution* – Replace a single character from pattern P with a different character in text T, such as changing "shot" to "spot."
- *Insertion* – Insert a single character into pattern P to help it match text T, such as changing "ago" to "agog."
- *Deletion* – Delete a single character from pattern P to help it match text T, such as changing "hour" to "our."

Properly posing the question of string similarity requires us to set the cost of each of these string transform operations. Assigning each operation an equal cost of 1 defines the *edit distance* between two strings. Approximate string matching arises in many applications, as discussed in Section 18.4 (page 631).

Approximate string matching seems like a difficult problem, because we must decide exactly where to delete and insert (potentially) many characters in pattern and text. But let us think about the problem in reverse. What information would we like to have to make the final decision? What can happen to the last character in the matching for each string?

8.2.1 Edit Distance by Recursion

We can define a recursive algorithm using the observation that the last character in the string must either be matched, substituted, inserted, or deleted. Chopping off the characters involved in this last edit operation leaves a pair of smaller strings. Let i and j be the last character of the relevant prefix of P and T, respectively. There are three pairs of shorter strings after the last operation, corresponding to the strings after a match/substitution, insertion, or deletion. *If* we knew the cost of editing these three pairs of smaller strings, we could decide which option leads to the best solution and choose that option accordingly. We *can* learn this cost through the magic of recursion.

More precisely, let $D[i, j]$ be the minimum number of differences between $P_1, P_2, \ldots, P_i$ and the segment of T ending at j. $D[i, j]$ is the *minimum* of the three possible ways to extend smaller strings:

- If $(P_i = T_j)$, then $D[i-1, j-1]$, else $D[i-1, j-1]+1$. This means we either match or substitute the ith and jth characters, depending upon whether the tail characters are the same.

- $D[i-1, j]+1$. This means that there is an extra character in the pattern to account for, so we do not advance the text pointer and pay the cost of an insertion.

- $D[i, j-1]+1$. This means that there is an extra character in the text to remove, so we do not advance the pattern pointer and pay the cost of a deletion.

```
#define MATCH       0       /* enumerated type symbol for match */
#define INSERT      1       /* enumerated type symbol for insert */
#define DELETE      2       /* enumerated type symbol for delete */
```

```
int string_compare(char *s, char *t, int i, int j)
{
        int k;                  /* counter */
        int opt[3];             /* cost of the three options */
        int lowest_cost;        /* lowest cost */

        if (i == 0) return(j * indel(' '));
        if (j == 0) return(i * indel(' '));

        opt[MATCH] = string_compare(s,t,i-1,j-1) + match(s[i],t[j]);
        opt[INSERT] = string_compare(s,t,i,j-1) + indel(t[j]);
        opt[DELETE] = string_compare(s,t,i-1,j) + indel(s[i]);

        lowest_cost = opt[MATCH];
        for (k=INSERT; k<=DELETE; k++)
                if (opt[k] < lowest_cost) lowest_cost = opt[k];

        return( lowest_cost );
}
```

This program is absolutely correct—convince yourself. It also turns out to be impossibly slow. Running on my computer, the computation takes several seconds to compare two 11-character strings, and disappears into Never-Never Land on anything longer.

Why is the algorithm so slow? It takes exponential time because it recomputes values again and again and again. At every position in the string, the recursion branches three ways, meaning it grows at a rate of at least 3^n—indeed, even faster since most of the calls reduce only one of the two indices, not both of them.

8.2.2 Edit Distance by Dynamic Programming

So, how can we make this algorithm practical? The important observation is that most of these recursive calls are computing things that have been previously computed. How do we know? There can only be $|P| \cdot |T|$ possible unique recursive calls, since there are only that many distinct (i, j) pairs to serve as the argument parameters of recursive calls. By storing the values for each of these (i, j) pairs in a table, we just look them up as needed and avoid recomputing them.

A table-based, dynamic programming implementation of this algorithm is given below. The table is a two-dimensional matrix m where each of the $|P| \cdot |T|$ cells contains the cost of the optimal solution to a subproblem, as well as a parent pointer explaining how we got to this location:

```
typedef struct {
        int cost;               /* cost of reaching this cell */
        int parent;             /* parent cell */
} cell;

cell m[MAXLEN+1][MAXLEN+1];     /* dynamic programming table */
```

Our dynamic programming implementation has three differences from the recursive version. First, it gets its intermediate values using table lookup instead of recursive calls. Second, it updates the `parent` field of each cell, which will enable us to reconstruct the edit sequence later. Third, it is implemented using a more general `goal_cell()` function instead of just returning `m[|P|][|T|].cost`. This will enable us to apply this routine to a wider class of problems.

```
int string_compare(char *s, char *t)
{
        int i,j,k;              /* counters */
        int opt[3];             /* cost of the three options */

        for (i=0; i<MAXLEN; i++) {
                row_init(i);
                column_init(i);
        }

        for (i=1; i<strlen(s); i++) {
                for (j=1; j<strlen(t); j++) {
                        opt[MATCH] = m[i-1][j-1].cost + match(s[i],t[j]);
                        opt[INSERT] = m[i][j-1].cost + indel(t[j]);
                        opt[DELETE] = m[i-1][j].cost + indel(s[i]);

                        m[i][j].cost = opt[MATCH];
                        m[i][j].parent = MATCH;
                        for (k=INSERT; k<=DELETE; k++)
                                if (opt[k] < m[i][j].cost) {
                                        m[i][j].cost = opt[k];
                                        m[i][j].parent = k;
                                }
                }
        }
        goal_cell(s,t,&i,&j);
        return( m[i][j].cost );
}
```

	T		y	o	u	-	s	h	o	u	l	d	-	n	o	t
P	pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
:		**0**	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t:	1	**1**	1	2	3	4	5	6	7	8	9	10	11	12	13	13
h:	2	2	**2**	2	3	4	5	5	6	7	8	9	10	11	12	13
o:	3	3	3	**2**	3	4	5	6	5	6	7	8	9	10	11	12
u:	4	4	4	3	**2**	3	4	5	6	5	6	7	8	9	10	11
-:	5	5	5	4	3	**2**	3	4	5	6	6	7	7	8	9	10
s:	6	6	6	5	4	3	**2**	3	4	5	6	7	8	8	9	10
h:	7	7	7	6	5	4	3	**2**	**3**	4	5	6	7	8	9	10
a:	8	8	8	7	6	5	4	3	3	**4**	5	6	7	8	9	10
l:	9	9	9	8	7	6	5	4	4	4	**4**	5	6	7	8	9
t:	10	10	10	9	8	7	6	5	5	5	5	**5**	6	7	8	8
-:	11	11	11	10	9	8	7	6	6	6	6	6	**5**	6	7	8
n:	12	12	12	11	10	9	8	7	7	7	7	7	6	**5**	6	7
o:	13	13	13	12	11	10	9	8	7	8	8	8	7	6	**5**	6
t:	14	14	14	13	12	11	10	9	8	8	9	9	8	7	6	**5**

Figure 8.4: Example of a dynamic programming matrix for editing distance computation, with the optimal alignment path highlighted in bold

Be aware that we adhere to somewhat unusual string and index conventions in the routine above. In particular, we assume that each string has been padded with an initial blank character, so the first real character of string `s` sits in `s[1]`. Why did we do this? It enables us to keep the matrix `m` indices in sync with those of the strings for clarity. Recall that we must dedicate the zeroth row and column of `m` to store the boundary values matching the empty prefix. Alternatively, we could have left the input strings intact and just adjusted the indices accordingly.

To determine the value of cell (i, j), we need three values sitting and waiting for us—namely, the cells $(i-1, j-1)$, $(i, j-1)$, and $(i-1, j)$. Any evaluation order with this property will do, including the row-major order used in this program.[1]

As an example, we show the cost matrices for turning $p =$ "thou shalt not" into $t =$ "you should not" in five moves in Figure 8.4.

8.2.3 Reconstructing the Path

The string comparison function returns the cost of the optimal alignment, but not the alignment itself. Knowing you can convert "thou shalt not" to "you should not" in only five moves is dandy, but what is the sequence of editing operations that does it?

The possible solutions to a given dynamic programming problem are described by paths through the dynamic programming matrix, starting from the initial configuration (the pair of empty strings $(0, 0)$) down to the final goal state (the pair

[1]Suppose we create a graph with a vertex for every matrix cell, and a directed edge (x, y), when the value of cell x is needed to compute the value of cell y. Any topological sort on the resulting DAG (why must it be a DAG?) defines an acceptable evaluation order.

	T		y	o	u	-	s	h	o	u	l	d	-	n	o	t
P	pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	0	**-1**	1	1	1	1	1	1	1	1	1	1	1	1	1	1
t:	1	**2**	0	0	0	0	0	0	0	0	0	0	0	0	0	0
h:	2	2	**0**	0	0	0	0	0	1	1	1	1	1	1	1	1
o:	3	2	0	**0**	0	0	0	0	0	1	1	1	1	1	0	1
u:	4	2	0	2	**0**	1	1	1	1	0	1	1	1	1	1	1
-:	5	2	0	2	2	**0**	1	1	1	1	0	0	0	1	1	1
s:	6	2	0	2	2	2	**0**	1	1	1	1	0	0	0	0	0
h:	7	2	0	2	2	2	2	**0**	**1**	1	1	1	1	1	0	0
a:	8	2	0	2	2	2	2	2	0	**0**	0	0	0	0	0	0
l:	9	2	0	2	2	2	2	2	0	0	**0**	1	1	1	1	1
t:	10	2	0	2	2	2	2	2	0	0	0	**0**	0	0	0	0
-:	11	2	0	2	2	0	2	2	0	0	0	0	**0**	1	1	1
n:	12	2	0	2	2	2	2	2	0	0	0	0	2	**0**	1	1
o:	13	2	0	0	2	2	2	2	0	0	0	0	2	2	**0**	1
t:	14	2	0	2	2	2	2	2	2	0	0	0	2	2	2	**0**

Figure 8.5: Parent matrix for edit distance computation, with the optimal alignment path highlighted in bold

of full strings ($|P|, |T|$)). The key to building the solution is to reconstruct the decisions made at every step along the optimal path that leads to the goal state. These decisions have been recorded in the `parent` field of each array cell.

Reconstructing these decisions is done by walking backward from the goal state, following the `parent` pointer back to an earlier cell. We repeat this process until we arrive back at the initial cell. The `parent` field for `m[i,j]` tells us whether the operation at (i, j) was `MATCH`, `INSERT`, or `DELETE`. Tracing back through the parent matrix in Figure 8.5 yields the edit sequence `DSMMMMMISMSMMMM` from "thou-shalt-not" to "you-should-not"—meaning delete the first "t", replace the "h" with "y", match the next five characters before inserting an "o", replace "a" with "u", and finally replace the "t" with a "d."

Walking backward reconstructs the solution in reverse order. However, clever use of recursion can do the reversing for us:

```
reconstruct_path(char *s, char *t, int i, int j)
{
        if (m[i][j].parent == -1) return;

        if (m[i][j].parent == MATCH) {
                reconstruct_path(s,t,i-1,j-1);
                match_out(s, t, i, j);
                return;
        }
        if (m[i][j].parent == INSERT) {
                reconstruct_path(s,t,i,j-1);
                insert_out(t,j);
```

```
                return;
        }
        if (m[i][j].parent == DELETE) {
                reconstruct_path(s,t,i-1,j);
                delete_out(s,i);
                return;
        }
}
```

For many problems, including edit distance, the tour can be reconstructed from the cost matrix without explicitly retaining the last-move pointer array. The trick is working backward from the costs of the three possible ancestor cells and corresponding string characters to reconstruct the move that took you to the current cell at the given cost.

8.2.4 Varieties of Edit Distance

The `string_compare` and path reconstruction routines reference several functions that we have not yet defined. These fall into four categories:

- *Table Initialization* – The functions `row_init` and `column_init` initialize the zeroth row and column of the dynamic programming table, respectively. For the string edit distance problem, cells $(i, 0)$ and $(0, i)$ correspond to matching length-i strings against the empty string. This requires exactly i insertions/deletions, so the definition of these functions is clear:

```
row_init(int i)
{
 m[0][i].cost = i;
 if (i>0)
     m[0][i].parent =  INSERT;
 else
     m[0][i].parent = -1;
}
```

```
column_init(int i)
{
 m[i][0].cost = i;
 if (i>0)
     m[i][0].parent = DELETE;
 else
     m[i][0].parent = -1;
}
```

- *Penalty Costs* – The functions `match(c,d)` and `indel(c)` present the costs for transforming character c to d and inserting/deleting character c. For standard edit distance, `match` should cost nothing if the characters are identical, and 1 otherwise; while `indel` returns 1 regardless of what the argument is. But application-specific cost functions can be employed, perhaps more forgiving of replacements located near each other on standard keyboard layouts or characters that sound or look similar.

```
int match(char c, char d)
{
  if (c == d) return(0);
  else return(1);
}
```

```
int indel(char c)
{
  return(1);
}
```

- *Goal Cell Identification* – The function `goal_cell` returns the indices of the cell marking the endpoint of the solution. For edit distance, this is defined by the length of the two input strings. However, other applications we will soon encounter do not have fixed goal locations.

```
goal_cell(char *s, char *t, int *i, int *j)
{
  *i = strlen(s) - 1;
  *j = strlen(t) - 1;
}
```

- *Traceback Actions* – The functions `match_out`, `insert_out`, and `delete_out` perform the appropriate actions for each edit operation during traceback. For edit distance, this might mean printing out the name of the operation or character involved, as determined by the needs of the application.

```
insert_out(char *t, int j)
{
        printf("I");
}

delete_out(char *s, int i)
{
        printf("D");
}
```

```
match_out(char *s, char *t,
          int i, int j)
{
    if (s[i]==t[j]) printf("M");
    else printf("S");
}
```

All of these functions are quite simple for edit distance computation. However, we must confess it is difficult to get the boundary conditions and index manipulations correctly. Although dynamic programming algorithms are easy to design once you understand the technique, getting the details right requires carefully thinking and thorough testing.

This may seem to be a lot of infrastructure to develop for such a simple algorithm. However, several important problems can now be solved as special cases of edit distance using only minor changes to some of these stub functions:

- *Substring Matching* – Suppose that we want to find where a short pattern P best occurs within a long text T—say, searching for "Skiena" in all its misspellings (Skienna, Skena, Skina, ...) within a long file. Plugging this

search into our original edit distance function will achieve little sensitivity, since the vast majority of any edit cost will consist of deleting that which is not "Skiena" from the body of the text. Indeed, matching any scattered $\cdots S \cdots k \cdots i \cdots e \cdots n \cdots a$ and deleting the rest yields an optimal solution.

We want an edit distance search where the cost of starting the match is independent of the position in the text, so that a match in the middle is not prejudiced against. Now the goal state is not necessarily at the end of both strings, but the cheapest place to match the entire pattern somewhere in the text. Modifying these two functions gives us the correct solution:

```
row_init(int i)
{
        m[0][i].cost = 0;       /* note change */
        m[0][i].parent = -1;    /* note change */
}

goal_cell(char *s, char *t, int *i, int *j)
{
        int k;                  /* counter */

        *i = strlen(s) - 1;
        *j = 0;
        for (k=1; k<strlen(t); k++)
                if (m[*i][k].cost < m[*i][*j].cost) *j = k;
}
```

- *Longest Common Subsequence* – Perhaps we are interested in finding the longest scattered string of characters included within both strings. Indeed, this problem will be discussed in Section 18.8. The *longest common subsequence* (LCS) between "democrat" and "republican" is *eca*.

  A common subsequence is defined by all the identical-character matches in an edit trace. To maximize the number of such matches, we must prevent substitution of nonidentical characters. With substitution forbidden, the only way to get rid of the noncommon subsequence is through insertion and deletion. The minimum cost alignment has the fewest such "in-dels", so it must preserve the longest common substring. We get the alignment we want by changing the match-cost function to make substitutions expensive:

```
int match(char c, char d)
{
        if (c == d) return(0);
        else return(MAXLEN);
}
```

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-

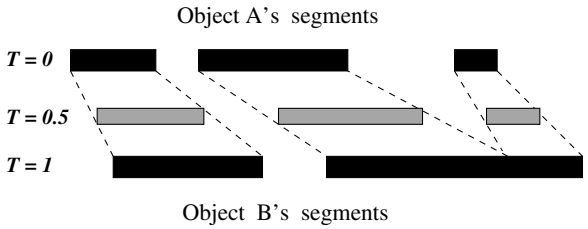

Figure 8.6: A successful alignment of two lines of pixels

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

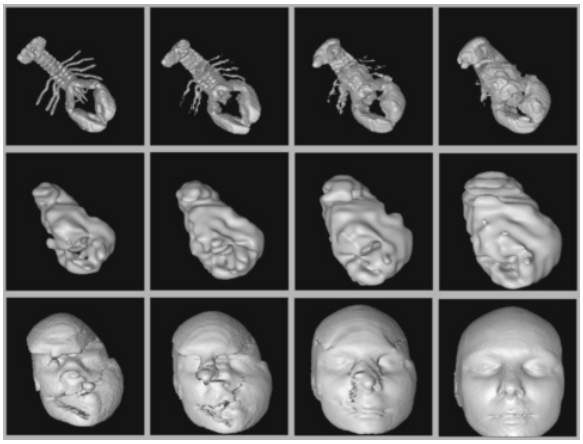

Figure 8.7: Morphing a lobster into a head via dynamic programming

$O(n^2)$ 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*

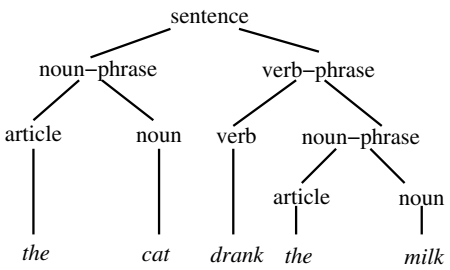

Figure 8.9: A context-free grammar (l) with an associated parse tree (r)

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

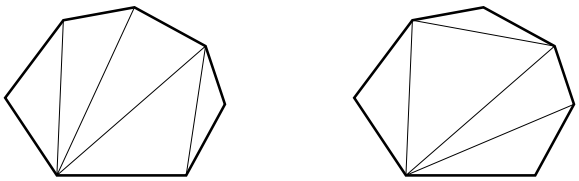

Figure 8.10: Two different triangulations of a given convex seven-gon

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 $T[i,i+1] = 0$.

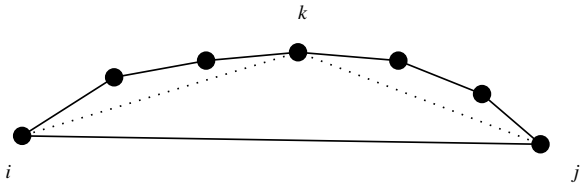

Figure 8.11: Selecting the vertex k to pair with an edge (i, j) of the polygon

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.

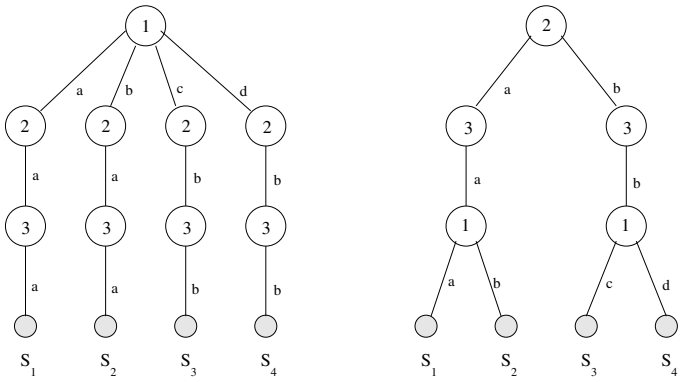

Figure 8.12: Two different tries for the same set of rule heads.

"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.

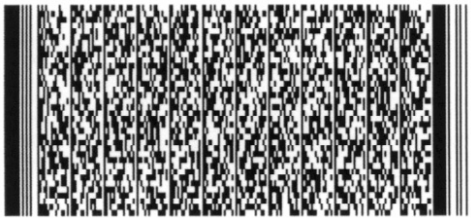

Figure 8.13: A two-dimensional bar-code label of the Gettysburg Address using PDF-417.

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.

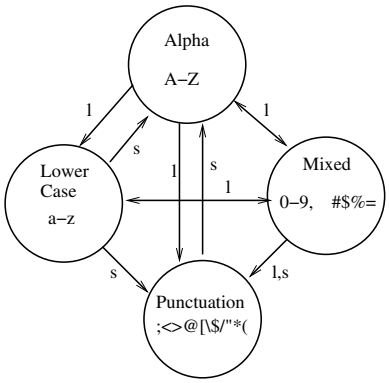

Figure 8.14: Mode switching in PDF-417

"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.