



# Chapter 10

## Dynamic Programming

The most challenging algorithmic problems involve optimization, where we seek to find a solution that maximizes or minimizes an objective 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 TSP “algorithms” that generate reasonable-looking solutions but do 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 intermediate 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.

*After* 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. That said, *until* you understand dynamic programming, it seems like magic. You have to figure out the trick before you can use it.

Dynamic programming is a technique for efficiently implementing a recursive algorithm by storing partial results. It requires seeing that a naive recursive algorithm computes the same subproblems over and over and over again. In such a situation, storing the answer for each subproblem in a table to look up instead of recompute can lead to an efficient algorithm. Dynamic programming starts with a recursive algorithm or definition. Only after we have a correct recursive algorithm can we worry about speeding it up by using a results matrix.

Dynamic programming is generally the right method for optimization prob-

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

## 10.1 Caching vs. Computation

Dynamic programming is essentially a tradeoff of space for time. Repeatedly computing a given quantity can become a drag on performance. If so, we are better off storing the results of the initial computation and looking them up instead of recomputing them.

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.

### 10.1.1 Fibonacci Numbers by Recursion

The Fibonacci numbers were 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 month is equal to the number of pairs of rabbits born in each of the two previous months, starting from one pair of rabbits at the start. Thus, the number of rabbits born in the  $n$ th month is defined by 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, ... As it turns out, Fibonacci's formula didn't do a great job of counting rabbits, but it does have a host of interesting properties and applications.

That they are defined by a recursive formula makes it easy to write a recursive program to compute the  $n$ th Fibonacci number. A recursive function 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));
}
```

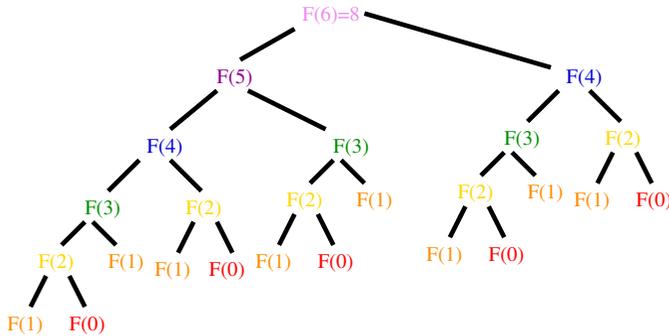


Figure 10.1: The recursion tree for computing Fibonacci numbers.

The course of execution for this recursive algorithm is illustrated by its *recursion tree*, as illustrated in Figure 10.1. 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 4 minutes and 40 seconds for this program to compute  $F(50)$  on my laptop. You might well do it faster by hand using the algorithm below.

How much time does the recursive algorithm take to compute  $F(n)$ ? Since  $F_{n+1}/F_n \approx \phi = (1 + \sqrt{5})/2 \approx 1.61803$ , this means that  $F_n > 1.6^n$  for sufficiently large  $n$ . Since our recursion tree has only 0 and 1 as leaves, summing them up to get 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!

### 10.1.2 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  $k$ —a technique also known as *memoization*. The key to implement the recursive algorithm efficiently is to explicitly check whether we already know a particular value before trying to compute it:

```
#define MAXN    92          /* largest n for which F(n) fits in a long */
#define UNKNOWN -1        /* contents denote an empty cell */
long f[MAXN+1];          /* array for caching fib values */
```

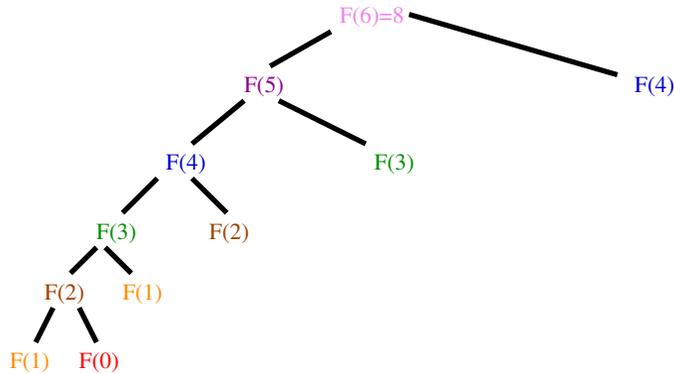


Figure 10.2: The recursion tree for computing Fibonacci numbers with caching.

```

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 that 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 10.2) 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 looking at 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 at most twice for each value  $0 \leq k \leq n - 1$ .

This general method of explicitly caching (or *tabling*) results from recursive calls to avoid recomputation provides a simple way to get *most* of the benefits of full dynamic programming. It is thus worth a 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 use once and 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, usually including the same running time as the more elegant full solution. If you prefer doing extra programming to more subtle thinking, I guess you can stop here.

### 10.1.3 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 for caching values */

    f[0] = 0;
    f[1] = 1;

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

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

Observe that 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 is now obvious. Each of the  $n$  values is simply computed as the sum of two integers, in  $O(n)$  total 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.

### 10.1.4 Binomial Coefficients

We now show how to compute *binomial coefficients* as another example 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 binomial coefficients? First,  $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ , 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:

$$\begin{array}{cccccc}
 & & & & & & 1 \\
 & & & & & & & 1 & & 1 \\
 & & & & & & & 1 & & 2 & & 1 \\
 & & & & & & & 1 & & 3 & & 3 & & 1 \\
 & & & & & & & 1 & & 4 & & 6 & & 4 & & 1 \\
 & & & & & & & 1 & & 5 & & 10 & & 10 & & 5 & & 1
 \end{array}$$

$n / k$	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

$n / k$	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

Figure 10.3: Evaluation order for `binomial_coefficient` at  $M[5, 4]$  (left). The initialization conditions are labeled A–K and recurrence evaluations labeled 1–10. The matrix contents after evaluation are shown on the right.

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

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

Why does this work? Consider whether the  $n$ th element appears in one of the  $\binom{n}{k}$  subsets having  $k$  elements. If it does, we can complete the subset by picking  $k-1$  other items from the remaining  $n-1$ . If it does 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  $k$ -element 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  $\binom{m}{0}$ . How many ways are there to choose zero things from a set? Exactly one, the empty set. If this is not convincing, then it is equally good to accept  $\binom{m}{1} = m$  as the basis case. The right term of the sum runs us up to  $\binom{m}{m}$ . How many ways are there to choose  $m$  things from a  $m$ -element set? Exactly one—the complete set. Together, these basis cases and the recurrence define the binomial coefficients on all interesting values.

Figure 10.3 demonstrates a proper evaluation order for the recurrence. The initialized cells are marked 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 1–10 denote the evaluation order in computing  $\binom{5}{4} = 5$  using the following code:

```
long binomial_coefficient(int n, int k) {
    int i, j;                /* counters */
    long bc[MAXN+1][MAXN+1]; /* binomial coefficient table */

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

```

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

for (i = 2; i <= n; i++) {
    for (j = 1; j < i; j++) {
        bc[i][j] = bc[i-1][j-1] + bc[i-1][j];
    }
}

return(bc[n][k]);
}

```

Study this function carefully to make sure you 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.

## 10.2 Approximate String Matching

Searching for patterns in text strings is a problem of unquestionable importance. Back in Section 6.7 (page 188) I presented algorithms for *exact* string matching—finding where the pattern string  $P$  occurs as a substring of the text string  $T$ . But life is often not that simple. Words in either the text or pattern can be misspelled (sic), robbing us of exact similarity. Evolutionary changes in genomic sequences or language usage mean 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. 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 in pattern  $P$  with a different character, 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 such transform operation. Assigning each operation an equal cost of 1 defines the *edit distance* between two strings. Approximate string matching arises in many applications, as detailed in Section 21.4 (page 688).

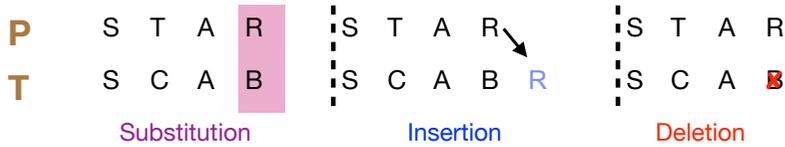


Figure 10.4: In a single string edit operation, the last character must be either matched/substituted, inserted, or deleted.

Approximate string matching seems like a difficult problem, because we must decide exactly where to best perform a complicated sequence of insert/delete operations in pattern and text. To solve it, let's think about the problem in reverse. What information would we need to select the final operation correctly? What can happen to the last character in the matching for each string?

### 10.2.1 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. There is no other possible choice, as shown in Figure 10.4. Chopping off the characters involved in this last edit operation leaves a pair of smaller strings. Let  $i$  and  $j$  be the indices of 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 the substrings  $P_1P_2 \dots P_i$  and  $T_1T_2 \dots T_j$ .  $D[i, j]$  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  $i$ th and  $j$ th characters, depending upon whether these tail characters are the same. More generally, the cost of a single character substitution can be returned by a function  $match(P_i, T_j)$ .
- $D[i, j - 1] + 1$ . This means that there is an extra character in the text to account for, so we do not advance the pattern pointer and we pay the cost of an insertion. More generally, the cost of a single character insertion can be returned by a function  $indel(T_j)$ .
- $D[i - 1, j] + 1$ . This means that there is an extra character in the pattern to remove, so we do not advance the text pointer and we pay the cost of a deletion. More generally, the cost of a single character deletion can be returned by a function  $indel(P_i)$ .

```

#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_r(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) {       /* indel is the cost of an insertion or deletion */
        return(j * indel(' '));
    }

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

    /* match is the cost of a match/substitution */

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

### 10.2.2 Edit Distance by Dynamic Programming

So, how can we make this algorithm practical? The important observation is that most of these recursive calls compute 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 the recursive calls. By storing the values for each of these  $(i, j)$  pairs in a table, we can 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 field 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.

Be aware that we adhere to special string and index conventions in the routine below. 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 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 adjusted the indices accordingly.

```
int string_compare(char *s, char *t, cell m[MAXLEN+1][MAXLEN+1]) {
    int i, j, k;      /* counters */
    int opt[3];       /* cost of the three options */

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

```

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);
}

```

To determine the value of cell  $(i, j)$ , we need to have three values sitting and waiting for us in matrix  $m$ —namely, the cells  $m(i-1, j-1)$ ,  $m(i, j-1)$ , and  $m(i-1, j)$ . Any evaluation order with this property will do, including the row-major order used in this program.<sup>1</sup> The two nested loops do in fact evaluate  $m$  for every pair of string prefixes, one row at a time. Recall that the strings are padded such that  $s[1]$  and  $t[1]$  hold the first character of each input string, so the lengths (`strlen`) of the padded strings are one character greater than those of the input strings.

As an example, we show the cost matrix for turning  $P =$  “thou shalt” into  $T =$  “you should” in five moves in Figure 10.5. I encourage you to evaluate this example matrix by hand, to nail down exactly how dynamic programming works.

### 10.2.3 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” to “you should” 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

<sup>1</sup>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.

	<i>T</i>		y	o	u	-	s	h	o	u	l	d
<i>P</i>	pos	0	1	2	3	4	5	6	7	8	9	10
:		<u>0</u>	1	2	3	4	5	6	7	8	9	10
t:	1	<u>1</u>	1	2	3	4	5	6	7	8	9	10
h:	2	2	<u>2</u>	2	3	4	5	5	6	7	8	9
o:	3	3	3	<u>2</u>	3	4	5	6	5	6	7	8
u:	4	4	4	3	<u>2</u>	3	4	5	6	5	6	7
-:	5	5	5	4	3	<u>2</u>	3	4	5	6	6	7
s:	6	6	6	5	4	3	<u>2</u>	3	4	5	6	7
h:	7	7	7	6	5	4	3	<u>2</u>	<u>3</u>	4	5	6
a:	8	8	8	7	6	5	4	3	3	<u>4</u>	5	6
l:	9	9	9	8	7	6	5	4	4	4	<u>4</u>	5
t:	10	10	10	9	8	7	6	5	5	5	5	<u>5</u>

Figure 10.5: Example of a dynamic programming matrix for editing distance computation, with the underlined entries appearing on the optimal alignment path. Blue values denote insertions, green values deletions, and red values match/substitution.

initial configuration (the pair of empty strings  $(0,0)$ ) down to the final goal state (the pair of full strings  $(|P|,|T|)$ ). The key to building the solution is reconstructing 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, analogous to how we reconstructed the path found by BFS or Dijkstra’s algorithm. 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 10.6 yields the edit sequence `DSMMMMISMS` from “`thou_shalt`” to “`you_should`”—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:

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

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

		y o u - s h o u l d										
<i>P</i>	<i>T</i> pos	0	1	2	3	4	5	6	7	8	9	10
	0	<u>-1</u>	1	1	1	1	1	1	1	1	1	1
t:	1	<u>2</u>	0	0	0	0	0	0	0	0	0	0
h:	2	2	<u>0</u>	0	0	0	0	0	1	1	1	1
o:	3	2	0	<u>0</u>	0	0	0	0	0	1	1	1
u:	4	2	0	2	<u>0</u>	1	1	1	1	0	1	1
-:	5	2	0	2	2	<u>0</u>	1	1	1	1	0	0
s:	6	2	0	2	2	2	<u>0</u>	1	1	1	1	0
h:	7	2	0	2	2	2	2	<u>0</u>	<u>1</u>	1	1	1
a:	8	2	0	2	2	2	2	2	0	<u>0</u>	0	0
l:	9	2	0	2	2	2	2	2	0	0	<u>0</u>	1
t:	10	2	0	2	2	2	2	2	0	0	0	<u>0</u>

Figure 10.6: Parent matrix for edit distance computation, with the optimal alignment path underlined to highlight. Again, blue values denote insertions, green values deletions, and red values match/substitution.

```

    match_out(s, t, i, j);
    return;
}

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

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

```

For many problems, including edit distance, the solution can be reconstructed from the cost matrix without explicitly retaining the last-move array. In edit distance, 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. But it is cleaner and easier to explicitly store the moves.

### 10.2.4 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)                column_init(int i)
{                               {
    m[0][i].cost = i;          m[i][0].cost = i;
    if (i>0)                   if (i>0)
        m[0][i].parent = INSERT;    m[i][0].parent = DELETE;
    else                         else
        m[0][i].parent = -1;      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 0 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 with substitution more forgiving for characters located near each other on standard keyboard layouts or those that sound or look similar.

```
int match(char c, char d)      int indel(char c)
{                               {
    if (c == d) return(0);      return(1);
    else 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 always defined by the length of the two input strings. However, other applications we will soon encounter do not have fixed goal locations.

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

```

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 correct. Although dynamic programming algorithms are easy to design once you understand the technique, getting the details right requires clear thinking and thorough testing.

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

- *Substring matching* – Suppose 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 all that is not “Skiena” from the body of the text. Indeed, matching any scattered ... $S$ ... $k$ ... $i$ ... $e$ ... $n$ ... $a$ ... and deleting the rest will yield 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 we are not prejudiced against a match that starts in the middle of the text. 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:

```

void row_init(int i, cell m[MAXLEN+1][MAXLEN+1]) {
    m[0][i].cost = 0;           /* NOTE CHANGE */
    m[0][i].parent = -1;        /* NOTE CHANGE */
}

```

```

void 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, without changing their relative order. Indeed, this problem will be discussed in Section 21.8. Do Democrats and Republicans have anything in common? Certainly! The *longest common subsequence* (LCS) between “democrats” and “republicans” is *ecas*.

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 non-identical characters. With substitution forbidden, the only way to get rid of the non-common subsequence will be 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);
    }
    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  $i$ th 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 maximum monotone 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: 123456789. 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 simply by 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 to complete the computation. Thus,  $O(m)$  space is sufficient to evaluate the recurrence without changing the time complexity. This is good, but 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 the same  $O(nm)$  time but only  $O(m)$  space. It is discussed in Section 21.4 (page 688).

### 10.3 Longest Increasing Subsequence

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

1. Formulate the answer you want 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 evaluation order 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 Section 10.2.4 (page 323), 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  $S$  is of length 5: for example, (2,3,5,6,8). In fact, there are eight of this length (can you enumerate them?). There are four 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 easily devise a linear-time algorithm. 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 design a recurrence relation for the length of the longest sequence. To find the right recurrence, ask what information about the first  $n - 1$  elements of  $S = (s_1, \dots, s_n)$  would enable you to find the answer for the entire sequence:

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

Unfortunately, this length  $L$  is not enough information to complete the full solution. Suppose I told you that the longest increasing sequence in  $(s_1, s_2, \dots, s_{n-1})$  was of length 5 and that  $s_n = 8$ . Will the length of the longest increasing subsequence of  $S$  be 5 or 6? It depends on whether the length-5 sequence ended with a value  $< 8$ .

- 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 ending at *every* possible value  $s_i$ .

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  $s_n$  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  $L_i$ :

$$L_i = 1 + \max_{\substack{0 \leq j < i \\ s_j < s_i}} L_j,$$

$$L_0 = 0$$

These values define the length of the longest increasing sequence ending at each sequence element. The length of the longest increasing subsequence of  $S$  is given by  $L = \max_{1 \leq i \leq n} L_i$ , since the winning sequence must end somewhere. Here is the table associated with our previous example:

Index $i$	1	2	3	4	5	6	7	8	9
Sequence $s_i$	2	4	3	5	1	7	6	9	8
Length $L_i$	1	2	2	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 a 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 back 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 the  $i - 1 \leq n$  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.

## 10.4 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 (now Zebra), the world’s leading manufacturer of bar code scanning equipment. 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, pack of gum, or can of soup 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 to only store a single ID number in a label. Thus, any application of supermarket bar codes must have a database mapping (say) 11141-47011 to a particular brand and size of soy sauce. The holy grail of the bar code world had 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 10.7. Although you may be more familiar with QR codes, PDF-417 is now a well accepted standard. Indeed, the back of every New York State drivers license contains the criminal record of its owner, elegantly rendered in PDF-417.

“How much data can you fit in a typical 1-inch label?” I asked him.

“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.

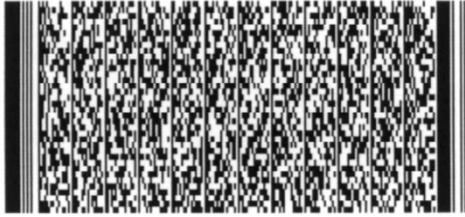


Figure 10.7: A two-dimensional barcode label of the Gettysburg Address using PDF-417.

“Interesting. You should use some data compression technique to maximize the amount of text you can store in a label.” See Section 21.5 (page 693) for a discussion of standard data compression algorithms.

“We do incorporate a data compaction method,” he explained. “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 5 bits as long as we stay within a mode. To switch modes, we issue a mode switch command first (taking an extra 5 bits) and then code for the new character.”

“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 10.8.

“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. We can *shift* into a new mode just for the next character, perhaps to produce a punctuation mark. Or we can *latch* permanently into a different mode, if we are at the start of a run of several characters from there, like a phone number.”

“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 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 starts in a given mode and consists of a sequence of intermixed character codes and mode shift/latch operations. From any given position in the text, we can either output the next character code (assuming it is 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. For a given position/mode

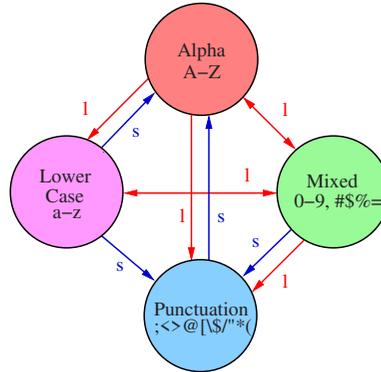


Figure 10.8: Mode switching in PDF-417.

pair, we would have been interested in the cheapest way of getting there, over all possible encodings. . . .

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 fill a matrix  $M[i, m]$  with the cost of the cheapest encoding 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 at each position.”

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  $k$  that minimizes  $M[n, k]$ . 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 gave 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. Of course, an 8% *average* improvement meant that it did much better than that on certain labels, and it never did worse than the original encoder. While our encoder took slightly longer to run than the greedy encoder, this was not significant, because the bottleneck would be the time needed to print the label.

Our observed impact of replacing a heuristic solution with the global optimum is probably typical of most applications. Unless you really botch up your

heuristic, you should get a decent solution. Replacing it with an optimal result, however, usually gives a modest but noticeable improvement, which can have pleasing consequences for your application.

## 10.5 Unordered Partition or Subset Sum

The *knapsack* or *subset sum* problem asks whether there exists a subset  $S'$  of an input multiset of  $n$  positive integers  $S = \{s_1, \dots, s_n\}$  whose elements add up to a given target  $k$ . Think of a backpacker trying to completely fill a knapsack of capacity  $k$  with possible selections from set  $S$ . Applications of this important problem are discussed in greater detail in Section 16.10.

Dynamic programming works best on linearly ordered items, so we can consider them from left to right. The ordering of items in  $S$  from  $s_1$  to  $s_n$  provides such an arrangement. To formulate a recurrence relation, we need to determine what information we need on items  $s_1$  to  $s_{n-1}$  in order to decide what to do about  $s_n$ .

Here is the idea. Either the  $n$ th integer  $s_n$  is part of a subset adding up to  $k$ , or it is not. If it is, then there must be a way to make a subset of the first  $n - 1$  elements of  $S$  adding up to  $k - s_n$ , so the last element can finish the job. If not, there may well be a solution that does not use  $s_n$ . Together this defines the recurrence:

$$T_{n,k} = T_{n-1,k} \vee T_{n-1,k-s_n}$$

This gives an  $O(nk)$  algorithm to decide whether target  $k$  is realizable:

```
bool sum[MAXN+1][MAXSUM+1];    /* table of realizable sums */
int parent[MAXN+1][MAXSUM+1]; /* table of parent pointers */

bool subset_sum(int s[], int n, int k) {
    int i, j;                    /* counters */

    sum[0][0] = true;
    parent[0][0] = NIL;

    for (i = 1; i <= k; i++) {
        sum[0][i] = false;
        parent[0][i] = NIL;
    }

    for (i = 1; i <= n; i++) { /* build table */
        for (j = 0; j <= k; j++) {
            sum[i][j] = sum[i-1][j];
            parent[i][j] = NIL;

            if ((j >= s[i-1]) && (sum[i-1][j-s[i-1]]==true)) {
```

```

        sum[i][j] = true;
        parent[i][j] = j-s[i-1];
    }
}

return(sum[n][k]);
}

```

The `parent` table encodes the actual subset of numbers totaling to  $k$ . An appropriate subset exists whenever `sum[n][k]==true`, but it does not use  $s_n$  as an element when `parent[n][k]==NIL`. Instead, we walk up the matrix until we find an interesting parent, and follow the corresponding pointer:

```

void report_subset(int n, int k) {
    if (k == 0) {
        return;
    }

    if (parent[n][k] == NIL) {
        report_subset(n-1,k);
    }
    else {
        report_subset(n-1,parent[n][k]);
        printf(" %d ",k-parent[n][k]);
    }
}

```

Below is an example showing the `sum` table for input set  $S = \{1, 2, 4, 8\}$  and target  $k = 11$ . The true in the lower right corner signals that the sum is realizable. Because  $S$  here represents all the powers of twos, and every target integer can be written in binary, the entire bottom row consists of trues:

$i$	$s_i$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	T	F	F	F	F	F	F	F	F	F	F	F
1	1	T	T	F	F	F	F	F	F	F	F	F	F
2	2	T	T	T	T	F	F	F	F	F	F	F	F
3	4	T	T	T	T	T	T	T	T	F	F	F	F
4	8	T	T	T	T	T	T	T	T	T	T	T	T

Below is the corresponding `parents` array, encoding the solution  $1 + 2 + 8 = 11$ . The 3 in the lower right corner reflects that  $11 - 8 = 3$ . The red bolded cells represent those encountered on the walk back to recover the solution.

$i$	$s_i$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	<b>-1</b>	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	1	-1	<b>0</b>	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	2	-1	-1	0	<b>1</b>	-1	-1	-1	-1	-1	-1	-1	-1
3	4	-1	-1	-1	<b>-1</b>	0	1	2	3	-1	-1	-1	-1
4	8	-1	-1	-1	-1	-1	-1	-1	-1	0	1	2	<b>3</b>

The alert reader might wonder how we can have an  $O(nk)$  algorithm for subset sum when subset sum is an NP-complete problem? Isn't this polynomial in  $n$  and  $k$ ? Did we just prove that  $P = NP$ ?

Unfortunately, no. Note that the target number  $k$  can be specified using  $O(\log k)$  bits, meaning that this algorithm runs in time exponential in the *size* of the input, which is  $O(n \log k)$ . This is the same reason why factoring integer  $N$  by explicitly testing all  $\sqrt{N}$  candidates for smallest factor is not polynomial, because the running time is exponential in the  $O(\log N)$  bits of the input.

Another way to see the problem is to consider what happens to the algorithm when we take a specific problem instance and multiply each integer by 1,000,000. Such a transform would not have affected the running time of sorting or minimum spanning tree, or any other algorithm we have seen so far in this book. But it would slow down our dynamic programming algorithm by a factor of 1,000,000, and require a million times as much space for storing the table. The range of the numbers matters in the subset sum problem, which becomes hard for large integers.

## 10.6 War Story: The Balance of Power

One of the many (presumably too many) uncharitable suspicions I hold is that most electrical engineering (EE) students today would not know how to build a radio. The reason for this is that the EE students I encounter study electrical and *computer* engineering, focusing on computer architecture and embedded systems that involve as much software as hardware. When a natural disaster comes, these guys are not going to be very concerned about restoring the operation of my favorite AM radio station.

Thus, it was a relief when an EE professor and his students came to me with an honest EE problem, about optimizing the performance of the power grid.

“Alternating current (AC) power systems transmit electricity on each of three different phases. Call them  $A$ ,  $B$ , and  $C$ . The system works best when the loads on each phase are roughly equal.” he explained.

“I guess loads are the machines needing power, right?” I asked insightfully.

“Yeah, think of every house on the street as being a load. Each house will get assigned one of the three phases as its source of power.”

“Presumably they connect every third house  $A$ ,  $B$ ,  $C$ ,  $A$ ,  $B$ ,  $C$  as they wire up the street to balance the load.”

“Something like that,” the EE professor confirmed. “But not all houses use the same amount of power, and it is even worse in industrial areas. One

company might just turn on the lights when another runs an arc furnace. After we measure the loads people are actually using, we would like to move some to different phases to balance the loads.”

Now I saw the algorithmic problem. “So given a set of numbers representing the various loads, you want to assign them phases  $A$ ,  $B$ , and  $C$  so the load is balanced as well as possible, right?”

“Yeah. Can you give me a fast algorithm to do this?,” he asked.

This seemed clear enough to me. It smelled like an integer partition problem, namely the subset sum problem of the previous section where the target  $k = (\sum_{i=1}^n s_n)/2$ . The most balanced possible partition occurs when the sum of elements in the selected subset (here  $k$ ) equals the sum of the elements left behind (here  $\sum_{i=1}^n s_n - k$ ).

The generalization of the problem to partition into three subsets instead of two was straightforward, but it wasn’t going to get any easier to solve. Adding a single new item  $s_{n+1} = k$  and asking for a partitioning of  $S$  into three equal weight subsets requires solving an integer partition on the original elements.

I broke the bad news gently. “Integer partition is an NP-complete problem, and three-phase balancing is just as hard as it is. There is no polynomial-time algorithm for your problem.”

They got up and started to leave. But then I remembered the dynamic programming algorithm for subset sum described in Section 10.5 (page 329). Why couldn’t this be extended to three phases? Indeed, define the function  $C[n, A, B]$  for a given set of loads  $S$ , where  $C[n, w_A, w_B]$  is true if there is a way to partition the first  $n$  loads of  $S$  such that the weight on phase  $A$  is  $w_A$  and the weight on phase  $B$  is  $w_B$ . Note that there is no need to explicitly keep track of the weight on phase  $C$ , because  $w_C = \sum_{i=1}^n s_i - w_A - w_B$ . Then we get the following recurrence, defined by which subset we put the  $n$ th load on:

$$C[n, w_A, w_B] = C[n-1, w_A - s_n, w_B] \vee C[n-1, w_A, w_B - s_n] \vee C[n-1, w_A, w_B]$$

This took constant time per cell to update, but there were  $nk^2$  cells to update, where  $k$  is the maximum amount of power we are willing to consider on any single phase. Thus, we could optimally balance the phases in  $O(nk^2)$  time.

This pleased them immensely, and they set to work to implement the algorithm. But I had one question before they went off, which I purposely directed to one of the computer engineering students. “Why is it that AC power has three phases?”

“Uh, maybe impedance matching and, uh, complex numbers?” he fumphered. His advisor shot him a dirty look, as I felt the warm glow of reassurance.

But that computer engineering student could code, and that was what mattered here. He quickly implemented the dynamic programming algorithm and performed experiments on representative problems, reported in [WSR13].

Our dynamic programming algorithm always produced at least as good a solution as several heuristics, and usually better. This is no surprise, since we always produced an optimal solution and they didn’t. Our dynamic program had a running time that grew quadratically in the range of the loads, which

could be a problem, but binning the loads by (say)  $\lfloor s_i/10 \rfloor$  would reduce the running time by a factor of 100 and produce solutions that were still pretty good for the original problem.

Dynamic programming really proved its worth when our electrical engineers got interested in more ambitious objective functions. It is not a cost-free operation to change which phase a load is on, and so they wanted to find a relatively balanced load assignment which minimized the number of changes required to achieve it. This is essentially the same recurrence, storing the cheapest cost to realize each state instead of just a flag indicating that you could reach it:

$$C[n, w_A, w_B] = \min(C[n-1, w_A - s_n, w_B] + 1, \\ C[n-1, w_A, w_B - s_n] + 1, \\ C[n-1, w_A, w_B])$$

They then got greedy, and wanted the lowest cost solution that never got seriously unbalanced at any point on the line. A globally balanced solution might choose to fill the total load on  $A$  before any loads on  $B$  or  $C$ , and that this would be bad. But the same recurrence above *still* does the job, provided we set  $C[n, w_A, w_B] = \infty$  whenever the loads at this state are deemed too unbalanced to be desirable.

That is the power of dynamic programming. Once you can reduce your state space to a small enough size, you can optimize just about anything. Just walk through each possible state and score it appropriately.

## 10.7 The Ordered 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,

$$100 \ 100 \ 100 \mid 100 \ 100 \ 100 \mid 100 \ 100 \ 100$$

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:

$$100 \ 200 \ 300 \mid 400 \ 500 \ 600 \mid 700 \ 800 \ 900$$

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

$$100 \ 200 \ 300 \ 400 \ 500 \mid 600 \ 700 \mid 800 \ 900$$

where the largest job is only 1,700 pages.

In general, we have the following problem:

**Problem:** Integer Partition without Rearrangement

**Input:** An arrangement  $S$  of non-negative numbers  $s_1, \dots, s_n$  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 *ordered partition* problem arises often in parallel processing. We seek to balance the work done across processors to minimize the total elapsed running time. The bottleneck in this computation will be the processor assigned the most work. Indeed, the war story of Section 5.8 (page 161) revolves around a botched solution to the very problem discussed here.

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 by computing the average weight of a partition,  $\sum_{i=1}^n s_i/k$ , and then trying 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  $k$ th partition starts right after the  $(k-1)$ st divider. Where can we place this last divider? Between the  $i$ th and  $(i+1)$ st elements for some  $i$ , where  $1 \leq i \leq n$ . What is the cost after this insertion? The total cost will be the larger of two quantities:

- the cost of the last partition  $\sum_{j=i+1}^n s_j$ , and
- the cost of the largest partition formed to the left of the last divider.

What is the size of this left partition? To minimize our total, we must use the  $k-2$  remaining dividers to partition the elements  $s_1, \dots, s_i$  as equally as possible. *This is a smaller instance of the same problem, and hence can be solved recursively!*

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

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

We also need to specify the boundary conditions of the recurrence relation. These boundary conditions resolve 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

$M$					$D$			
$s$	$k$				$s$	$k$		
1	1	2	3		1	1	2	3
1	1	1	1		1	-	-	-
1	2	1	1		1	-	1	1
1	3	2	1		1	-	1	2
1	4	2	2		1	-	2	2
1	5	3	2		1	-	2	3
1	6	3	2		1	-	3	4
1	7	4	3		1	-	3	4
1	8	4	3		1	-	4	5
1	9	5	3		1	-	4	6

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

Figure 10.9: Dynamic programming matrices  $M$  and  $D$  for two instances of the ordered partition problem. Partitioning  $(1, 1, 1, 1, 1, 1, 1, 1, 1)$  into  $((1, 1, 1), (1, 1, 1), (1, 1, 1))$  (left) and  $(1, 2, 3, 4, 5, 6, 7, 8, 9)$  into  $((1, 2, 3, 4, 5), (6, 7), (8, 9))$  (right). Prefix sum entries appear in red and the optimal solution divider positions in blue.

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:

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

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

How long does it take to compute this when we store the partial results? There are a total of  $k \cdot n$  cells in the table. How much time does it take to compute the values of  $M[n', k']$  for  $1 \leq n' \leq n$ ,  $1 \leq k' \leq k$ ? Calculating this quantity using the general recurrence involves finding the minimum of  $n'$  quantities, each of which is the larger of two numbers: a table lookup and the sum of at most  $n'$  elements (taking  $O(n')$  time). 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 following implementation:

```
void partition(int s[], int n, int k) {
    int p[MAXN+1];           /* prefix sums array */
    int m[MAXN+1][MAXK+1];  /* DP table for values */
    int d[MAXN+1][MAXK+1];  /* DP table for dividers */
    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  $n$  prefix sums  $p_i = \sum_{k=1}^i s_k$ , since  $\sum_{k=i}^j s_k = p_j - p_{i-1}$ . This enables us to evaluate the recurrence in linear time per cell, yielding an  $O(kn^2)$  algorithm. These prefix sums also appear as the initialization values for  $k = 1$ , and are shown in the dynamic programming matrices of Figure 10.9.

By studying the recurrence relation and the dynamic programming matrices of these two examples, 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. But for most applications, we need 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 used to achieve this value. We reconstruct the path used to get the optimal solution by working backwards from  $D[n, k]$ , and add a divider at each specified position. This backwards walking is best achieved by a recursive subroutine:

```

void 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);
    }
}

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

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

```

## 10.8 Parsing Context-Free Grammars

Compilers identify whether a particular program is a legal expression in a particular programming language, and reward you with syntax errors if it is not. This requires a precise description of the language syntax, typically given by a *context-free grammar*, as shown in Figure 10.10(1). 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 sequence  $S$  as per 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 10.10(right) presents 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, more recently a friend easily explained it to me over lunch. The difference is that I understand dynamic programming much better now than when I was a student.

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

Further, we assume that the definitions of each rule are in *Chomsky normal form*, like the example of Figure 10.10. This means that the right sides of every rule consists of either (a) exactly two nonterminals, for example,  $X \rightarrow YZ$ , or

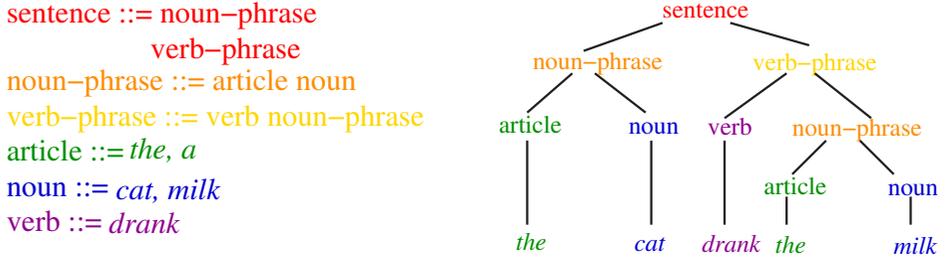


Figure 10.10: A context-free grammar (on left) with an associated parse tree (right)

(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  $S$  using a context-free grammar where each interesting rule produces 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,  $S_1 \cdots S_i$ , must be *generated* by nonterminal  $Y$ , and the right part ( $S_{i+1} \cdots S_n$ ) generated by  $Z$ .

This suggests a dynamic programming algorithm, where we keep track of all nonterminals generated by each contiguous subsequence of  $S$ . Define  $M[i, j, X]$  to be a Boolean function that is true iff subsequence  $S_i \cdots S_j$  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, for  $i < j$  we have

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

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

The 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  $n(n+1)/2$  subsequences defined by  $(i, j)$  pairs with  $i \geq j$ . Multiplying this by the number of nonterminals, which is finite because the grammar was defined to be of constant size, has no impact on the Big Oh. Evaluating  $M[i, j, X]$  requires testing all intermediate values  $k$  where  $i \leq k < j$ , 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 sequence  $S$ , find the smallest number of character substitutions you must make to  $S$  so that the resulting sequence is accepted by  $G$ .

---

*Solution:* This problem seemed extremely difficult when I first encountered it. But on reflection, it is just a very general version of edit distance, addressed naturally by dynamic programming. Parsing first sounded difficult, 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 subsequence  $S_i \cdots S_j$  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 to the left of the breaking point and some to the right, but all we care about is minimizing the sum. In other words, for  $i < j$  we have

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

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

*Take-Home Lesson:* For optimization problems 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.

**10.9 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 problem, 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  $G = (V, E)$ , 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 simply by 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 critical 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 of weight  $n - 1$ . Finding such *Hamiltonian paths* (if they exist) is an important graph problem, discussed in Section 19.5 (page 598).

### 10.9.1 When is Dynamic Programming Correct?

Dynamic programming algorithms are only as correct as the recurrence relations they are based on. Suppose we define  $LP[i, j]$  to be the length of the longest simple path from  $i$  to  $j$ . Note that the longest simple path from  $i$  to  $j$  has 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  $(x, j)$ :

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

This 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, then adding the edge  $(x, j)$  will create a cycle. To prevent this, we must define a recursive function that explicitly remembers where we have been. Perhaps we could define  $LP'[i, j, k]$  to denote 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.

The 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 stuck in an infinite loop as soon as we try to do anything.

Dynamic programming can be applied to any problem that obeys the *principle of optimality*. Roughly stated, this means that partial solutions can be optimally extended given 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 the sequence of operations that 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 their cost. Such would be the case with a special 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.

### 10.9.2 When is Dynamic Programming Efficient?

The running time of any dynamic programming algorithm is a function of two things: (1) the number of partial solutions we must keep track of, and (2) how long it takes 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 possible stopping *places* in the input. This is because the combinatorial objects being worked on (typically strings and numerical sequences) 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 likely have an exponential number of possible partial solutions. Suppose the state of our partial longest simple path solution is the entire path  $P$  taken from the start to end vertex. Thus,  $LP[i, j, P_{ij}]$  denotes the cost of longest simple path from  $i$  to  $j$ , where  $P_{ij}$  is the sequence of intermediate vertices between  $i$  and  $j$  on this path. The following recurrence relation works correctly to compute this, where  $P + x$  denotes appending  $x$  to the end of  $P$ :

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

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

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

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

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 can be used to solve TSPs for up to thirty vertices or more, where  $n = 20$  would be impossible using the  $O(n!)$  algorithm. Still, dynamic programming proves 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.

## 10.10 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.

$$\begin{aligned} p(a, a, a) &:= h(a); \\ p(b, a, a) &:= h(a) * h(b); \\ p(c, b, b) &:= h(b) + h(c); \\ p(d, b, b) &:= h(d) + h(b); \end{aligned}$$

“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 15.3 (page 448). 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  $i$ th node of the path corresponding to the string's  $i$ th character.

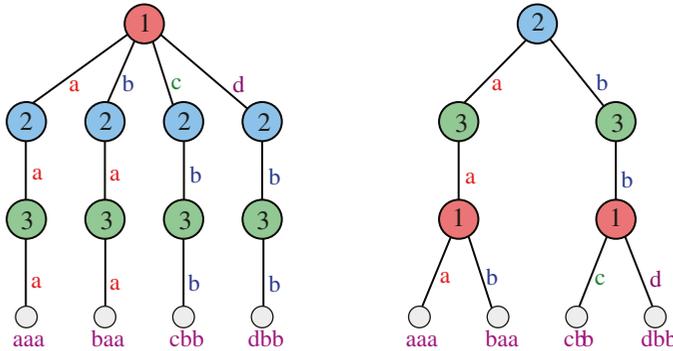


Figure 10.11: Two different tries for the same set of Prolog rule heads, where the trie on the right has four less edges.

“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 10.11. 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) on both sides, we could 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. 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 that picks 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-to-right ordering constraint. After all, if a given set of  $n$  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 my colleague 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 possible trie!” It was a pleasure watching his frown change to a smile as the realization took hold. An efficient algorithm to compute what you need is infinitely better than a proof saying you can’t do it!

My recurrence looked something like this. Suppose that we are given  $n$  ordered rule heads  $s_1, \dots, s_n$ , each with  $m$  arguments. Probing at the  $p$ th position,  $1 \leq p \leq m$ , partitions the rule heads into runs  $R_1, \dots, R_r$ , where each rule in a given run  $R_x = s_i, \dots, s_j$  has the same character value as  $s_i[p]$ . 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  $p$ :

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

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—that is, 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 with arbitrary rule orderings, something we put in the paper to make it complete.

*Take-Home Lesson:* The global optimum (found perhaps 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.

## Chapter Notes

Bellman [Bel58] is credited with inventing 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 Khanna et al. [KMS97].

Techniques such as dynamic programming and backtracking can be used to generate worst-case efficient (although still non-polynomial) algorithms for many NP-complete problems. See Downey and Fellows [DF12] and Woeginger [Woe03] for nice surveys of such techniques.

More details about the war stories in this chapter are available in published papers. See Dawson et al. [DRR<sup>+</sup>95] for more on the Prolog trie minimization problem. Our algorithm for phase-balancing power loads from Section 10.6 (page 331) is reported in Wang et al. [WSR13]. Two-dimensional bar codes, presented in Section 10.4 (page 326), 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). See [You67]. The generalization of parsing to edit distance is due to Aho and Peterson [AP72].

## 10.11 Exercises

### Elementary Recurrences

- 10-1. [3] Up to  $k$  steps in a single bound! A child is running up a staircase with  $n$  steps and can hop between 1 and  $k$  steps at a time. Design an algorithm to count how many possible ways the child can run up the stairs, as a function of  $n$  and  $k$ . What is the running time of your algorithm?
- 10-2. [3] Imagine you are a professional thief who plans to rob houses along a street of  $n$  homes. You know the loot at house  $i$  is worth  $m_i$ , for  $1 \leq i \leq n$ , but you cannot rob neighboring houses because their connected security systems will automatically contact the police if two adjacent houses are broken into. Give an efficient algorithm to determine the maximum amount of money you can steal without alerting the police.
- 10-3. [5] Basketball games are a sequence of 2-point shots, 3-point shots, and 1-point free throws. Give an algorithm that computes how many possible mixes (1s,2s,3s) of scoring add up to a given  $n$ . For  $n = 5$  there are four possible solutions: (5, 0, 0), (2, 0, 1), (1, 2, 0), and (0, 1, 1).
- 10-4. [5] Basketball games are a sequence of 2-point shots, 3-point shots, and 1-point free throws. Give an algorithm that computes how many possible scoring sequences add up to a given  $n$ . For  $n = 5$  there are thirteen possible sequences, including 1-2-1-1, 3-2, and 1-1-1-1-1.
- 10-5. [5] Given an  $s \times t$  grid filled with non-negative numbers, find a path from top left to bottom right that minimizes the sum of all numbers along its path. You can only move either down or right at any point in time.

- (a) Give a solution based on Dijkstra's algorithm. What is its time complexity as a function of  $s$  and  $t$ ?
- (b) Give a solution based on dynamic programming. What is its time complexity as a function of  $s$  and  $t$ ?

### Edit Distance

- 10-6. [3] Typists often make transposition errors exchanging neighboring characters, such as typing "setve" for "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.
- 10-7. [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 *cchocohilaptse* 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.)
- 10-8. [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.
- 10-9. [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  $P$ .
- 10-10. [5] Suppose you are given  $n$  poker chips stacked in two stacks, where the edges of all chips can be seen. Each chip is one of three colors. A turn consists of choosing a color and removing all chips of that color from the tops of the stacks. The goal is to minimize the number of turns until the chips are gone.  
For example, consider the stacks (*RRGG, GBBB*). Playing red, green, and then blue suffices to clear the stacks in three moves. Give an  $O(n^2)$  dynamic programming algorithm to find the best strategy for a given pair of chip piles.

## Greedy Algorithms

- 10-11. [4] Let  $P_1, P_2, \dots, P_n$  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  $D < \sum_{i=1}^n s_i$
- Does a greedy algorithm that selects programs in order of non-decreasing  $s_i$  maximize the number of programs held on the disk? Prove or give a counter-example.
  - Does a greedy algorithm that selects programs in order of non-increasing  $s_i$  use as much of the capacity of the disk as possible? Prove or give a counter-example.
- 10-12. [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  $\{d_1, \dots, d_k\}$  units. We seek an algorithm to make change of  $n$  units using the minimum number of this country's coins.
- 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  $\{1, 6, 10\}$ .
  - Give an efficient algorithm that correctly determines the minimum number of coins needed to make change of  $n$  units using denominations  $\{d_1, \dots, d_k\}$ . Analyze its running time.
- 10-13. [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  $\{d_1, \dots, d_k\}$  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  $\{1, 6, 10\}$ ,  $C(5) = 1$ ,  $C(6) = 2$ ,  $C(10) = 3$ , and  $C(12) = 4$ .
- How many ways are there to make change of 20 units from  $\{1, 6, 10\}$ ?
  - 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.)
- 10-14. [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

- 10-15. [3] You are given a rod of length  $n$  inches and a table of prices obtainable for rod-pieces of size  $n$  or smaller. Give an efficient algorithm to find the maximum value obtainable by cutting up the rod and selling the pieces. For example, if  $n = 8$  and the values of different pieces are:

length	1	2	3	4	5	6	7	8
price	1	5	8	9	10	17	17	20

then the maximum obtainable value is 22, by cutting into pieces of lengths 2 and 6.

- 10-16. [5] Your boss has written an arithmetic expression of  $n$  terms to compute your annual bonus, but permits you to parenthesize it however you wish. Give an efficient algorithm to design the parenthesization to maximize the value. For the expression:

$$6 + 2 \times 0 - 4$$

there exist parenthesizations with values ranging from  $-32$  to  $2$ .

- 10-17. [5] Given a positive integer  $n$ , find an efficient algorithm to compute the smallest number of perfect squares (e.g. 1, 4, 9, 16, ...) that sum to  $n$ . What is the running time of your algorithm?
- 10-18. [5] Given an array  $A$  of  $n$  integers, find an efficient algorithm to compute the largest sum of a continuous run. For  $A = [-3, 2, 7, -3, 4, -2, 0, 1]$ , the largest such sum is 10, from the second through fifth positions.
- 10-19. [5] Two drivers have to divide up  $m$  suitcases between them, where the weight of the  $i$ th suitcase is  $w_i$ . Give an efficient algorithm to divide up the loads so the two drivers carry equal weight, if possible.
- 10-20. [6] The *knapsack problem* is as follows: given a set of integers  $S = \{s_1, s_2, \dots, s_n\}$ , 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 dynamic programming algorithm for knapsack that runs in  $O(nT)$  time.

- 10-21. [6] The *integer partition* takes a set of positive integers  $S = \{s_1, \dots, s_n\}$  and seeks a subset  $I \subset S$  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.

- 10-22. [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.
- 10-23. [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-to-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 units. If the breaks are made in right-to-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 units.

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.

10-24. [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.

10-25. [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 plays first and so has an advantage. The champion plays white in the first game. The champ 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 get  $i$  points (which may be a multiple of  $1/2$ ).
- (b) Based on your recurrence, give a dynamic programming algorithm to calculate the champion's probability of retaining the title.
- (c) Analyze its running time for an  $n$  game match.

10-26. [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  $f$ th 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  $n$ -floor 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 make as few drops as possible. Broken eggs cannot be reused. Let  $E(k, n)$  be the minimum number of egg drops 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

- 10-27. [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  $bad[i,j] = \text{“yes”}$  iff 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.
- 10-28. [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  $bad[i,j] = \text{“yes”}$  iff 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  $O(XY)$ .
- 10-29. [5] You seek to create a stack out of  $n$  boxes, where box  $i$  has width  $w_i$ , height  $h_i$ , and depth  $d_i$ . The boxes cannot be rotated, and can only be stacked on top of one another when each box in the stack is strictly larger than the box above it in width, height, and depth. Give an efficient algorithm to construct the tallest possible stack, where the height is the sum of the heights of each box in the stack.

## Design Problems

- 10-30. [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.,  $h = h_i$  for all  $i$ ) and the shelves are all separated by a distance 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 book placement that uses the minimum number of shelves, and analyze its time complexity.
- 10-31. [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. Here 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.)

10-32. [5] Consider a linear keyboard of lowercase letters and numbers, where the leftmost 26 keys are the letters A–Z in order, followed by the digits 0–9 in order, followed by the 30 punctuation characters in a prescribed order, and ended on a blank. Assume you start with your left index finger on the “A” and your right index finger on the blank.

Give a dynamic programming algorithm that finds the most efficient way to type a given text of length  $n$ , in terms of minimizing total movement of the fingers involved. For the text  $ABABABAB \dots ABAB$ , this would involve shifting both fingers all the way to the left side of the keyboard. Analyze the complexity of your algorithm as a function of  $n$  and  $k$ , the number of keys on the keyboard.

10-33. [5] You have come back from the future with an array  $G$ , where  $G[i]$  tells you the price of Google stock  $i$  days from now, for  $1 \leq i \leq n$ . You seek to use this information to maximize your profit, but are only permitted to complete at most one transaction (i.e. either buy one or sell one share of the stock) per day. Design an efficient algorithm to construct the buy–sell sequence to maximize your profit. Note that you cannot sell a share unless you currently own one.

10-34. [8] You are given a string of  $n$  characters  $S = s_1 \dots s_n$ , which you believe to be a compressed text document in which all spaces have been removed, like `itwasthebestoftimes`.

(a) You seek to reconstruct the document using a dictionary, which is available in the form of a Boolean function  $dict(w)$ , where  $dict(w)$  is true iff string  $w$  is a valid word in the language. Give an  $O(n^2)$  algorithm to determine whether string  $S$  can be reconstituted as a sequence of valid words, assuming calls to  $dict(w)$  take unit time.

(b) Now assume you are given the dictionary as a set of  $m$  words each of length at most  $l$ . Give an efficient algorithm to determine whether string  $S$  can be reconstituted as a sequence of valid words, and its running time.

10-35. [8] Consider the following two-player game, where you seek to get the biggest score. You start with an  $n$ -digit integer  $N$ . With each move, you get to take either the first digit or the last digit from what is left of  $N$ , and add that to your score, with your opponent then doing the same thing to the now smaller number. You continue taking turns removing digits until none are left. Give an efficient algorithm that finds the best possible score that the first player can get for a given digit string  $N$ , assuming the second player is as smart as can be.

10-36. [6] Given an array of  $n$  real numbers, consider the problem of finding the maximum sum in any contiguous subarray 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 and clear  $\Theta(n^2)$ -time algorithm to find the maximum contiguous subarray.
- 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.

10-37. [7] Consider the problem of examining a string  $x = x_1x_2 \dots x_n$  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 symbol.

10-38. [6] Let  $\alpha$  and  $\beta$  be constants. Assume that it costs  $\alpha$  to go left in a binary search tree, and  $\beta$  to go right. Devise an algorithm that builds a tree with optimal expected query cost, given keys  $k_1, \dots, k_n$  and the probabilities that each will be searched  $p_1, \dots, p_n$ .

### Interview Problems

- 10-39. [5] Given a set of coin denominations, find the minimum number of coins to make a certain amount of change.
- 10-40. [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 obtain the maximum sum of the  $i$ th through  $j$ th numbers.
- 10-41. [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.

### LeetCode

10-1. <https://leetcode.com/problems/binary-tree-cameras/>

10-2. <https://leetcode.com/problems/edit-distance/>

10-3. <https://leetcode.com/problems/maximum-product-of-splitted-binary-tree/>

**HackerRank**

10-1. <https://www.hackerrank.com/challenges/ctci-recursive-staircase/>

10-2. <https://www.hackerrank.com/challenges/coin-change/>

10-3. <https://www.hackerrank.com/challenges/longest-increasing-subsequent/>

**Programming Challenges**

These programming challenge problems with robot judging are available at <https://onlinejudge.org>:

10-1. “Is Bigger Smarter?”—Chapter 11, problem 10131.

10-2. “Weights and Measures”—Chapter 11, problem 10154.

10-3. “Unidirectional TSP”—Chapter 11, problem 116.

10-4. “Cutting Sticks”—Chapter 11, problem 10003.

10-5. “Ferry Loading”—Chapter 11, problem 10261.