11. Dynamic Programming

Optimization is all about making intelligent decisions. In the preceding chapter, we saw how to build up solutions with a sequence of "greedy" decisions focusing only on the short-term improvement of a partial solution at each step. For many problems, however, being near-sighted like this is detrimental, and we need to consider longer-term consequences of our decisions in order to reach an optimal solution. This leads to a powerful albeit confusingly-named technique called *dynamic programming* (DP), which we study in detail in this chapter.

As a motivating example to explain the high-level idea behind DP, suppose we go to the post office to mail a package that requires C cents in postage. We find a vending machine that sells three types of stamps: 49-cent stamps, 34-cent stamps, and 1-cent stamps[1]. Our goal is to purchase the smallest number of stamps whose total value is exactly C. One's first instinct might be to consider the natural greedy algorithm for this problem: repeatedly select the largest stamp whose value doesn't exceed our remaining postage. However, the short-sighted nature of this approach leads to trouble down the road. For example, if $C = 68$, the greedy algorithm first chooses a 49-cent stamp, but then is stuck using nineteen 1-cent stamps, while the optimal solution uses just two 34-cent stamps. It therefore seems that we need to look farther ahead when making decisions.

How do we make a more informed decision about which stamp to purchase first? According to Figure 11.1, after making this first decision, we are left with a smaller subproblem of the same form as the original problem[2]. If we already knew the optimal solutions to the three subproblems reachable from the initial decision, the decision becomes easy, since we now understand its long-term ramifications — we would simply choose the first stamp in order to land on the subproblem that requires the minimum number of remaining stamps. This is a crucial insight behind DP: large problems often become much easier to solve after first computing solutions to smaller subproblems embedded within them.

To solve the postage problem with DP, we decompose it into a series of subproblems of the same form. We first compute the minimum number of stamps required to

[1] At the time of writing of this chapter, 49 cents is the current US rate for first-class mail, and 34 cents is the current rate for postcards.

[2] In the previous chapter, we called this the *optimal substructure* property. Optimal substructure is perhaps the most important property we need a problem to satisfy for it to allow a DP solution.

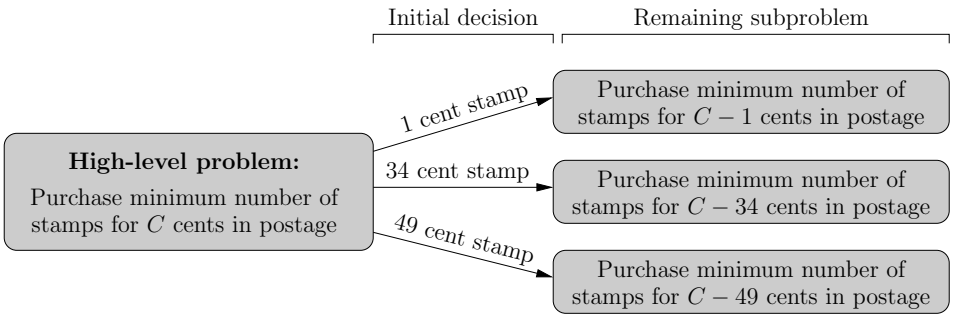


FIGURE 11.1: The solution of the postage problem viewed an initial decision plus the solution of a smaller left-over subproblem of the same form.

make 1 cent in postage, then the minimum number of stamps required to make 2 cents in postage, and so on. Letting $M(c)$ denote the minimum number of stamps required to make exactly c cents in postage, our subproblems are therefore the computation of $M(1)$, $M(2)$, and so on, ending with $M(C)$, the problem we originally wanted to solve. From Figure 11.1, we can deduce that

$$M(c) = \underbrace{1}_{\text{first stamp}} + \underbrace{\min\{M(c-1), M(c-34), M(c-49)\}}_{\text{stamps to solve left-over subproblem}}$$

Our entire DP algorithm now consists of applying this formula to compute $M(c)$ for $c = 1$ up to C, treating $M(c) = 0$ for $c = 0$ and $M(c) = +\infty$ for $c < 0$ as base cases[3]. By solving subproblems in this order, the formula above allows us to compute each $M(c)$ in just constant time, since we have conveniently already computed the solutions of all the smaller subproblems on which it depends. Our total running time is therefore $\Theta(C)$.

Many algorithmic techniques solve large problems by breaking them into smaller subproblems whose solutions are then recombined to solve the larger problem. For example, a "divide and conquer" algorithm might break a problem of size n into two subproblems of size $n/2$, and an "incremental construction" algorithm might recursively solve a subproblem of size $n - 1$ and then augment it with one final element. DP belongs to this same family of approaches, only specialized to solve optimization problems. Unfortunately, the name "dynamic programming" does little to convey the essence of the technique. Historically, the term "programming" refers not to computer programming but to "mathematical programming", meaning the formulation and solution of optimization problems. Its closest synonym in this context is probably "planning". The "dynamic" part of the name refers to the fact that the technique was originally introduced as a means of solving dynamic multi-stage planning problems involving decision-making over time. We have tried to convey the spirit of this idea in our motivation above.

DP is a powerful, highly versatile technique that applies to a wide range of problems in practice. Although it is based on a simple premise, DP can be a relatively difficult

[3]Defining $M(c) = +\infty$ for $c < 0$ ensures that it will never be optimal to choose a solution that lands on a subproblem with $c < 0$, since this represents an infeasible situation.

technique for the introductory student to master. Therefore, like the last chapter, this chapter is also built from a series of examples that highlight the common designs and applications of DP algorithms.

11.1 Example: Shortest Paths in Acyclic Graphs

As our first major example, we consider the well-known *single-source shortest path problem*. Its input is a directed graph with n nodes and m directed edges, each with an associated cost, where we would like to find a shortest (i.e., minimum-cost) path from a designated source node s to every other node in the graph. An example is pictured in Figure 11.2(a). The shortest path problem is an extremely important and well-studied problem in algorithmic graph theory, one to which we will devote an entire chapter later on. For now, we consider the special case of computing shortest paths through a *directed acyclic graph* (DAG). We can solve the single-source shortest path problem in a DAG in $\Theta(m+n)$ time (i.e., linear time). This is the best running time possible, given that we need to at least examine all of the nodes and edges in the graph. Many practical problems involve computation of shortest paths through a DAG; the following is a prototypical example:

Problem 205 (Project Scheduling). DAGs often appear in scheduling problems with precedence constraints. Suppose we have n jobs to schedule, each with associated processing times $p_1 \dots p_n$, and in addition we are given a list of m precedence constraints among these jobs. Each precedence constraint is a pair (i, j) specifying that job i must be completed before we can start working on job j. We can work on multiple jobs in parallel, but not if they are related directly or indirectly via precedence constraints. Please show how to use acyclic shortest paths to compute the minimum possible *makespan* (the latest completion time over all the jobs), as well as for each job, the following two quantities: its *earliest start time* (the earliest time we may possibly expect to be able to work on the job), and its *latest start time* (the latest possible time we could start working on the job without adversely affecting the optimal makespan). [Solution]

Our linear-time shortest path algorithm for DAGs can actually be cleverly adapted to compute shortest paths in a general directed graph. When we study shortest paths in Chapter **??**, we will see how this leads to the *Bellman-Ford* algorithm, which computes single-source shortest paths in $O(mn)$ time.

11.1.1 Building up Optimal Solutions to Subproblems

We now highlight the main steps involved in the design of a DP algorithm, while we build a solution for the acyclic shortest path problem.

(i) **Decompose the problem into a sequence of increasingly larger subproblems, all having the same structure**. For some problems, this step is quite straightforward; for others, determining the right subproblem decomposition can be quite challenging. For acyclic shortest paths and many other problems involving a DAG, a natural subproblem decomposition arises from its *topological ordering*. Recall from Chapter 3 that we can *topologically sort*

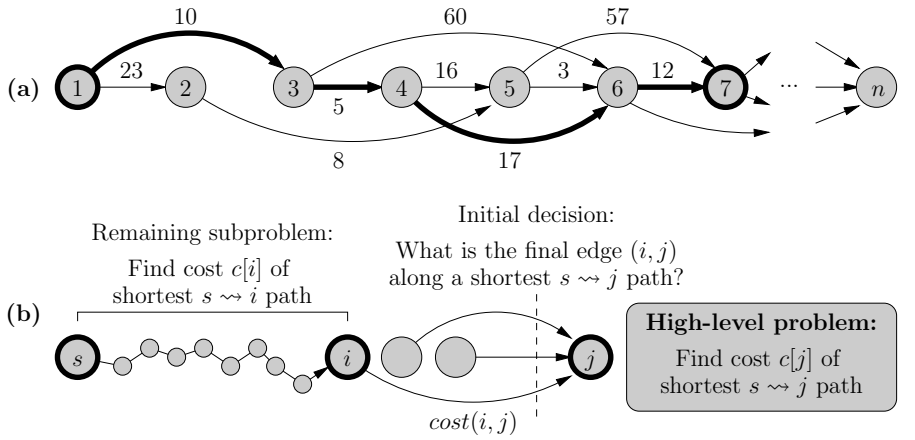


FIGURE 11.2: Part (a) shows a directed acyclic graph (DAG) and its associated edge costs. The nodes of the graph are arranged in a topological ordering, and the shortest path from the source node $s = 1$ to node 7 is shaded. Part (b) explains how we decompose the problem of finding a shortest $s \rightsquigarrow j$ path into that of finding a shortest $s \rightsquigarrow i$ path plus one final edge (i, j).

the nodes of a DAG in linear time, ordering them in a line so that all edges point from left to right. Figure 11.2(a) shows an example. We assume the source node s is the leftmost node in the ordering (node 1), since nodes to the left of s would be unreachable from s and can be ignored. Letting $c[j]$ denote the shortest path cost from s to node j (after topologically sorting as a preprocessing step), our DP algorithm will compute $c[1], c[2], \ldots, c[n]$ as its sequence of subproblem solutions. Subproblems are often of this form (i.e., "what is the best solution up to and ending at position j?" for $j = 1, 2, \ldots$), corresponding to solutions of increasingly-larger "prefixes" of the full problem.

(ii) **Express the solution of a large subproblem recursively in terms of solutions to smaller subproblems**. A shortest $s \rightsquigarrow j$ path arrives at j along some final edge (i, j), where $i < j$ is an earlier node in our topological ordering. As shown in Figure 11.2(b), we can therefore compute $c[j]$ by looking for the best such incoming edge — the one minimizing the cost of the edge (i, j) itself (denoted by $cost(i, j)$) plus the cost of the optimally solving the left-over subproblem of finding a shortest path from s to i:

$$c[j] = \min_{\text{incoming edges } (i,j)} \{c[i] + cost(i, j)\}.$$

Note that this formula only depends on solutions to smaller subproblems we will already have computed — that is, values $c[i]$ for $i < j$.

These two steps represent 90% of the difficult work in developing our algorithm. We now only need to specify appropriate base cases (here, $c[1] = 0$) and to loop over all subproblems in the appropriate order to solve them. According to Figure 11.3(a), this involves nothing more than wrapping the formula above in a "for" loop. The final implementation is quite simple, and this is typical for DP algorithms; the


```
     0.  Initialization:  c[1] ← 0, c[2...n] ← +∞, b[1...n] ← nil

     1.  For j ← 2 to n:
     2.     For all incoming edges (i,j) of node j:
(a)  3.       If c[i] + cost(i,j) < c[j]:
     4.         c[j] ← c[i] + cost(i,j)
     5.         b[j] ← i

     1.  For i ← 1 to n − 1:
     2.     For all outgoing edges (i,j) of node i:
(b)  3.       If c[i] + cost(i,j) < c[j]:
     4.         c[j] ← c[i] + cost(i,j)
     5.         b[j] ← i
```

FIGURE 11.3: Computing shortest paths through a DAG, assuming it has already been topologically sorted. In (a), we compute $c[j]$ for $j = 1 \dots c[n]$ in sequence, while (b) is quite similar but uses "pushing" instead of "pulling". Both approaches also compute backpointers $b[1 \dots n]$.

challenge lies not in the implementation, but in the initial *formulation* of the solution in steps (i) and (ii). Likewise, running time analysis is usually quite straightforward. Here, our algorithm runs in $\Theta(m+n)$ time since it does a constant amount of work for each node and edge. If desired, correctness can be established formally using induction on subproblem size. Here, we would argue that the formula from step (ii) correctly computes $c[j]$ given the assumption (by induction) that $c[i]$ has already been correctly computed for all $i < j$.

Pulling Versus Pushing. In order to compute $c[j]$ using the formula from (ii), each node in the DAG needs to know its incoming edges. Unfortunately, directed graphs are often stored by having nodes keep track of their outgoing, rather than incoming edges. Figure 11.3(b) shows how we can adapt our solution in this case. Since each subproblem knows which larger problems are consumers of its solution, it "pushes" its optimal solution forward to these larger problems, instead of "pulling" solutions from smaller subproblems. The value of $c[j]$ is no longer computed by processing the explicit formula from (ii) when we visit node j, but rather it is computed implicitly during the course of visiting the earlier subproblems upon which $c[j]$ depends. By the time our algorithm reaches node j, the value $c[j]$ will already be computed, so all of the work spent on this subproblem will be invested in pushing its value forward to larger subproblems.

Whether "pulling" or "pushing" is appropriate depends on problem structure, for example whether subproblems have access to backward links to smaller subproblems or forward links to larger subproblems (in fact, these two approaches are sometimes called *backward* and *forward* DP). When both methods apply, they generally give the same running time.

Reconstructing a Solution via Traceback. As discussed so far, our DP algorithm computes the cost of the shortest path to every node, but not the actual paths themselves. This is a common trait of DP algorithms — we end up with the *value* of the optimal solution, but not its *structure* (e.g., the individual elements

we selected to be part of the solution). Fortunately, it is usually easy to find the structure as well, by storing for each subproblem not only the value of its optimal solution, but also a *backpointer* to the previous subproblem(s) on which its value is based. In our example, we compute the shortest path cost $c[j]$ by minimizing over j's incoming edges, so the pseudocode in Figure 11.3 computes a backpointer $b[j]$ to the node i corresponding to which edge (i, j) was best among these. After the algorithm terminates, we can then follow the sequence of backpointers from any node j back to the source to trace out the shortest $s \rightsquigarrow j$ path in reverse. This technique works for nearly any DP algorithm. By following the so-called *traceback path* of backpointers starting from the final subproblem, we can reconstruct (in reverse) the chain of decisions we used to optimally solve that problem. These decisions may tell us, for example, the set of elements that were included in the solution.

Now that you are familiar with the basics, here are some simple problems you can use to practice formulating DP solutions:

Problem 206 (Maximum Value Subarray). Given an array $A[1 \ldots n]$ of numbers (both positive and negative), we would like to find a subarray $A[i \ldots j]$ whose sum is maximized. We introduced this problem back in Section 2.3, where we showed that the obvious "brute force" solution to this problem runs in $\Theta(n^3)$ time, although we can reduce this to $\Theta(n^2)$ or even $\Theta(n \log n)$ with a small amount of extra cleverness. Show how to solve it in only $\Theta(n)$ time using DP. [Solution]

Problem 207 (Longest Increasing Subsequence). Given an array $A[1 \ldots n]$ of numbers, a *subsequence* of A is just a subset of its elements $A[i_1], A[i_2], \ldots, A[i_k]$ where $i_1 < i_2 < \ldots < i_k$. Note that the elements in a subsequence do not need to form a contiguous block (in contrast to a *subarray*). We say a subsequence is increasing if $A[i_1] < A[i_2] < \ldots < A[i_k]$. Give an $O(n^2)$ DP algorithm that finds a longest increasing subsequence of A — that is, a subsequence containing a maximum number of elements. We will improve this running time to $O(n \log n)$ later in the book in problem **??** by leveraging a duality relationship between this problem and the problem of partitioning a sequence into a minimum number of disjoint non-increasing subsequences. [Solution]

Problem 208 (Interval Covering and Packing). Let us revisit the interval packing and covering problems from Section 10.5 in the previous chapter. As input, we are given a set of n intervals, each one being a subinterval of $[0, 1]$ on the number line. A *packing* of intervals is a subset of intervals that are all mutually disjoint from each-other, and a *covering* of intervals is a set of intervals whose union is $[0, 1]$.

(a) If our intervals have values $v_1 \ldots v_n$, give an $O(n^2)$ DP algorithm that computes a maximum-value interval packing. Can you improve its running time to $O(n \log n)$? [Solution]

(b) If our intervals have costs $c_1 \ldots c_n$, give an $O(n^2)$ DP algorithm that computes a minimum-cost interval cover. Can you improve it to $O(n \log n)$ time? [Solution]

11.1.2 Dynamic Programs as Computations on DAGs

We can gain some additional insight into the problems we have studied thus far by showing how they share quite a bit of common structure. In fact, the reason we chose shortest acyclic path as our first main example problem is that this problem is somewhat "canonical" in that all of the problems we have seen so far can be viewed as special cases of it (or of the equivalent longest path problem in a DAG).

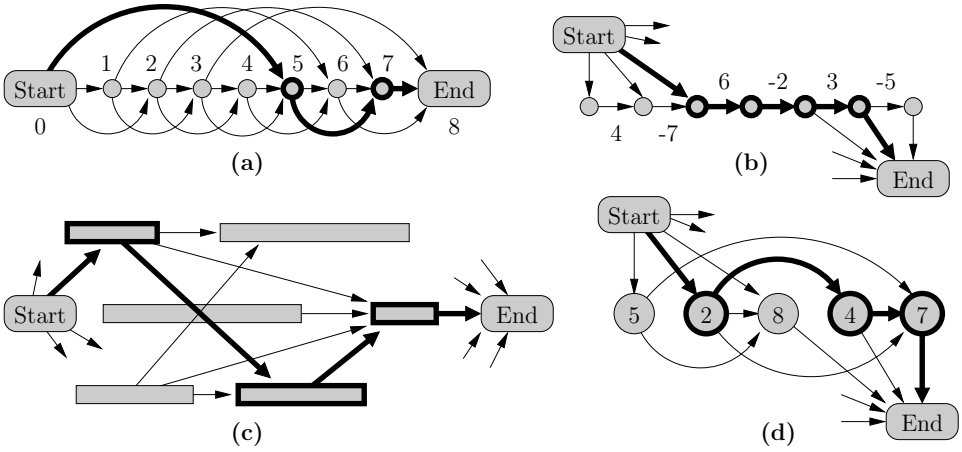


FIGURE 11.4: Representing the (a) postage problem, (b) maximum subarray problem, (c) maximum interval packing problem, and (d) longest increasing subsequence problem as a shortest or longest path problem in a DAG. To reduce clutter, we have abbreviated some of the edges out of the "start" node and into the "end" node in (b), (c), and (d). See also Figure 12.4(b) for a picture of an interval covering problem as a shortest path problem in a DAG.

Consequently, the DP algorithms for solving these problems can be viewed as just specializations of the generic algorithm we devised earlier to solve the shortest path problem in a DAG.

- Figure 11.4(a) shows our earlier postage problem expressed as a shortest path problem in a DAG. Here, we want to make $C = 8$ cents worth of postage using a combination of 1-cent, 2-cent, and 5-cent stamps, and we can view this as computing a shortest path through a DAG whose nodes represent the amount of postage we have accumulated. Edges each have unit cost, and represent the use of 1-cent stamps (horizontal edges), 2-cent stamps (lower edges), and 5-cent stamps (upper edges).

- Figure 11.4(b) shows how to express a maximum value subarray problem over the sequence $A = 4, -7, 6, -2, 3, -5$ as a longest path problem through a DAG. Edge values incident to the "dummy" start and end nodes are zero. The value of a path from start to end corresponds to the sum of a subarray.

- In Figure 11.4(c), we see how to represent the maximum interval packing problem as a longest path problem in a DAG whose nodes represent intervals, with an edge joining interval i with some later interval j if i and j are disjoint. Here, a longest path from a dummy start node (with edges outgoing to all intervals) to a dummy end node (with edges incoming from all intervals) corresponds to a maximum interval packing.

- Figure 11.4(d) shows the longest increasing subsequence problem on the sequence $A = 5, 2, 8, 4, 7$ similarly represented as a longest path problem, where there is an edge from $A[i]$ to $A[j]$ if $i < j$ and $A[i] < A[j]$.

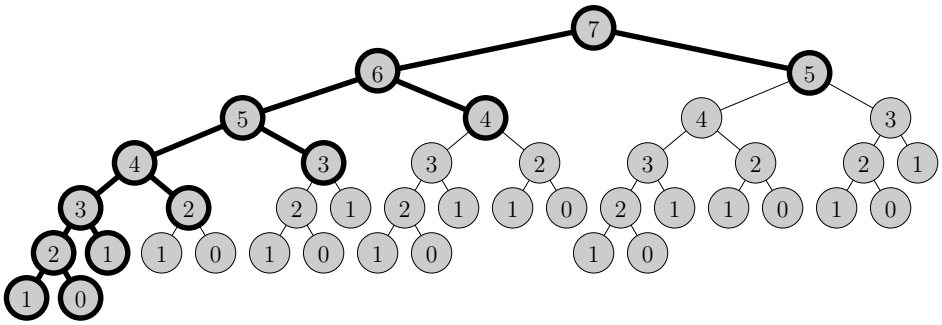


FIGURE 11.5: The exponential-sized tree of all recursive calls generated when computing the Fibonacci number F_7 using the straight recursive `FibRec` function from Figure 11.6(a). By employing memoization, the function `FibMemo` from Figure 11.6(b) reduces the running time dramatically by visiting only the subproblems shown in bold.

If you are a visual thinker, you may want to continue trying to visualize DP algorithms as computations on DAGs of subproblems, even for the more sophisticated examples later in the chapter that do not map directly to shortest or longest path problems. Nodes correspond to subproblems, and each node has incoming edges from the subproblems on which its value depends. The graph is acyclic since edges only go from smaller to larger subproblems. You might even want to view this sort of DAG as an abstract computing machine, since each node represents the computation of a simple function based on values it receives from its incoming edges. As we will see moving forward, subproblems can often also be naturally arranged in a sequence, grid, or tree, giving other nice options for visual layout.

11.1.3 Top-Down Versus Bottom-Up Computation

Since a DP formulation tells us a recursive formula for solving large subproblems, we might be tempted to use nothing more than straight recursion as our solution. As a simple example, the nth Fibonacci number is recursively defined as $F_n = F_{n-1} + F_{n-2}$, so we could use the code in Figure 11.6(a) to compute F_n in a purely recursive manner. Although this will ultimately give a correct answer, it will take an exceptionally long time to do so. The tree of recursively-generated subproblems (Figure 11.2) grows at an exponential rate, having $\Theta(\phi^n)$ nodes.

The drawback of straight recursion is that we end up solving the same subproblems over and over. To remedy this, we employ a technique known as *memoization*. Once we have solved a subproblem, we store its solution in a table (or in keeping with the name of the technique, we make a "memo" of its solution). If that subproblem is ever generated again, we can return its solution immediately without further recursion. Although memoization only adds a small amount of code to our implementation (Figure 11.6(b)), its running time benefits are substantial, as computing F_n now requires only $\Theta(n)$ time (ignoring for a moment the issue of word size, since the number of digits in F_n grows at an exponential rate as n increases).

(a)

```
FibRec(j):
1.  If j = 0:  Return 0
2.  If j = 1:  Return 1
3.  Return FibRec(j − 1) + FibRec(j − 2)
```

(b)

```
Initialization:  f[0] ← 0, f[1] ← 1, f[2...n] ← nil
FibMemo(j):
1.  If f[j] ≠ nil:  Return f[j]
2.  f[j] ←FibMemo(j − 1) + FibMemo(j − 2)
3.  Return f[j]
```

(c)

```
1.  f[0] ← 0, f[1] ← 1
2.  For j = 2 to n:
3.      f[j] ← f[j − 1] + f[j − 2]
```

FIGURE 11.6: Computing the nth Fibonacci number (a) via straightforward recursion (in exponential time), and (b) using top-down recursion plus "memoization" (in linear time). In (c), we see the more standard DP approach that solves subproblems in a bottom-up fashion.

In contrast to the top-down recursive approach above, most DP algorithms are traditionally implemented in a "bottom-up" fashion, looping over all possible subproblems from smallest to largest, as in Figure 11.6(c). However, the "bottom-up" and "top-down with memoization" approaches are generally equivalent in terms of worst-case running time, since both ultimately involve solving the same collection of subproblems, each at most once. In our example here, both run in $\Theta(n)$ time. In fact, it is possible that for some problems, a top-down approach *could* run faster if it manages to only visit only a small subset of all possible subproblems during its recursive expansion (e.g., suppose we had defined $F_n = F_{n-61} + F_{n-83}$). However, since the bottom-up approach is "leaner", involving only loops and no recursive function overhead, it may still win in practice. Both methods can be useful to keep in your algorithmic toolbox.

From our discussion above, we have now accumulated sufficient insight to characterize the types of problems for which DP is an effective solution technique:

- The problem must satisfy the optimal substructure property, allowing us to express the solution of large subproblems recursively in terms of solutions of smaller subproblems,
- The total number of distinct subproblems should be rather small, and
- A top-down recursive algorithm would end up solving the same subproblems multiple times, making straightforward recursion inadvisable.

Problem 209 (More Practice Problems). The only way to become an expert at DP is by solving as many problems as possible. Here are a few more examples of simple problems to work through for practice.

(a) During a town festival, n local businesses are planning to give away free products and food, but only at certain times of the day. In particular, business i will be giving away

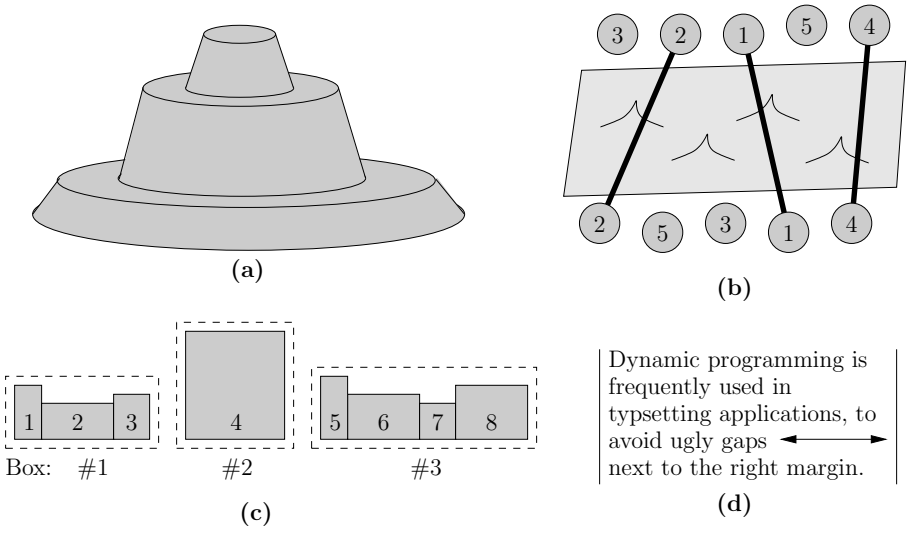


FIGURE 11.7: Illustrations for problems: (a) stacking conical frustra, (b) building bridges, (c) box packing, and (d) typesetting.

free items only at time T_i. We know, for every pair of businesses (i, j), the amount of time $t_{ij} > 0$ it takes to walk from i to j (you may assume the t_{ij}'s satisfy the triangle inequality, so the fastest way to walk from i to j is the direct route that takes t_{ij} units of time). Please design an $O(n^2)$ DP algorithm that computes the maximum number of businesses we can successfully visit. We start at business 1 (not necessarily the business with minimum T_i) at time 0. [Solution]

(b) A sequence of n crates with sizes $s_1 \ldots s_n$ arrives via truck at a train station and must be packed onto a series of rail cars. Each rail car holds a minimum total size of b units and a maximum total size of B units. Rail cars must be loaded according to the order in which crates arrive. For example, crates $1 \ldots 4$ might be placed on the first rail car, after which that car departs and then crates 5 and 6 are loaded onto the next car, and so on. Using DP, please determine if it is possible to pack the crates at all, and if so, what is the minimum number of rail cars we need. Try to achieve a running time of $O(n^2)$. For an extra challenge, can you improve the running time to $O(n \log n)$ or even $O(n)$? [Solution]

(c) Given a numeric sequence $A_1 \ldots A_n$, please describe an $O(n^2)$ algorithm for computing a monotonically nondecreasing sequence $B_1 \ldots B_n$ minimizing the distance between A and B, defined as $\sum_i |A_i - B_i|$. As a hint, refer back to problem 60. [Solution]

(d) A *conical frustum* is a horizontal slice of a cone. You are given a set of n conical frustra, specified by the radii of their bases $R_1 \ldots R_n$, the smaller radii of their tops $r_1 \ldots r_n$, and their heights $h_1 \ldots h_n$. Using a subset of these frustra, you would like to build a tower that is as tall as possible. Of course, in the interest of stability, you can only stack frustum i on top of frustum j if $R_i \leq r_j$, as shown in Figure 11.7(a). Please design an $O(n \log n)$ dynamic programming algorithm that determines the height of the tallest possible tower. [Solution]

(e) Consider the diagram shown in Figure 11.7(b). You are told the x-coordinates of n cities along the northern bank of a river, and also of n cities along the southern bank. The cities along each bank are numbered from $1 \ldots n$, and you would like to build bridges to connect equally-numbered pairs of cities with bridges. However, you are

not allowed to build two bridges that cross each-other. Give an $O(n^2)$ DP algorithm that determines the maximum number of bridges you can build. [Solution]

(f) You are responsible for packaging a shipment of n rectangular objects into boxes. The objects will arrive in a particular order $1 \dots n$ on a conveyor belt, and you know the heights $h_1 \dots h_n$ and widths $w_1 \dots w_n$ of all the objects in advance. You must split the sequence of objects into *contiguous* groups that are then to be packed into boxes. As shown in Figure 11.7(a), each group is arranged from left to right and then packed into a rectangular box whose dimensions are as small as possible. You are given a per-unit-length cost A of box material (so a set of boxes having total perimeter P would cost AP in total), and you are also given a per-unit-area penalty B for each unit of empty space we pack. Subject to these costs, please give an $O(n^2)$ DP algorithms for computing a minimum-cost box packing. [Solution]

(g) DP is frequently used in typesetting applications, to avoid ugly gaps next to the right margin, as shown in Figure 11.7(d). Suppose you are told the widths $w_1 \dots w_n$ of n sequential words that you would like to typeset on a page of width W. Depending on how you choose to insert line breaks, there will be some amount of space left over on each line between the last word on the line and the right-hand margin — we call these spaces *gaps*. Please devise an $O(n^2)$ DP algorithm that minimizes the sum of squared lengths of all gaps except the gap on the last line of text. [Solution]

(h) You are managing a large project consisting of n tasks. Each task i has an associated value v_i you receive for completing the task and a time t_i marking the exact moment at which the task must be started, if you choose to perform the task at all. Unfortunately, The duration of each task is not known with certainty until after the task is completed: task i takes either d_i or d'_i units of time to complete, each with probability $1/2$ (and also independent of the duration of all other tasks). As time progresses, you must decide for each task i (when we reach time t_i and find that no other task is currently pending) whether you will skip the task or perform the task. If you elect to perform task i, no other tasks can be performed until i is completed. Please give an $O(n \log n)$ time dynamic programming algorithm that computes a scheduling policy of maximum expected value. [Solution]

(i) You are again managing a large project with n tasks. Each task i has an associated value v_i you receive for completing the task, a known duration d_i, and a time t_i at which the task must be started. It may not be possible to perform all the tasks, so you would like to perform a subset of tasks of maximum total value. Furthermore, you can direct your workers to "rush" up to two tasks (a rushed task takes only half the duration it normally would require), but since rushing a task requires significant effort you cannot perform these two rushed tasks in a row. Please show how to compute an optimal schedule in $O(n \log n)$ time using dynamic programming. [Solution]

11.2 Example: The Knapsack Problem

So far, all of our DP problems have been "one dimensional", involving a single linear chain of subproblems. These are among the simplest DP problems, and the best suited for introducing the technique. We now move from one dimension to two, introducing a common type of DP algorithm that involves filling in a two-dimensional table of subproblem solutions row by row.

Our next example is a classic problem in discrete optimization called the *knapsack problem*. Its input consists of n items each having sizes $s_1 \dots s_n$ and values $v_1 \dots v_n$, and the goal is to compute a maximum-value collection of items that can be packed

into a knapsack of some specified capacity C. We focus on the *integer knapsack problem*, where C and all item sizes are integers; values can still be non-integral.

In problem 199 in the preceding chapter, we showed how to solve the *fractional* knapsack problem (where fractional quantities of items may be placed in the knapsack) in $\Theta(n)$ time using greedy techniques. However, when we require whole items to be placed in the knapsack in their entirety, the problem becomes much more challenging — NP-hard, in fact. Consequently, the algorithms we develop will have only *pseudo-polynomial* running times[4] like $O(nC)$. Although these algorithms are technically not "efficient", they are widely used in practice for problems in which C is not too large.

A natural one-dimensional subproblem decomposition would involve solving for the best way to fill smaller knapsacks of capacity 1, 2, and so on, ending with our final capacity of C. However, this doesn't quite work for the following reason: suppose our first decision involves inserting item 7 into the knapsack. To complete our solution, we would like to solve the remaining subproblem "What is the maximum value one can pack into a capacity $C - s_7$ knapsack without using item 7?". Unfortunately, all we know is the solution to the subproblem "What is the maximum value one can pack into a capacity $C - s_7$ knapsack?". In other words, our subproblems don't contain enough information to remember which items we have already used. To address this, we use a formulation involving a two-dimensional space of subproblems:

> $V[i, j]$: The maximum value one can pack into a knapsack of capacity j using only some subset of items $1 \dots i$.
>
> $$V[i,j] = \max\{\underbrace{V[i-1,j]}_{\text{(a)}}, \underbrace{V[i-1, j-s_i] + v_i}_{\text{(b)}}\}.$$
>
> Base cases: $V[i = 0, j] = V[i, j = 0] = 0$, $V[i, j < 0] = -\infty$.

Each subproblem $V[i, j]$ is solved in $O(1)$ time by deciding between the best possible value we can obtain (a) if we do not include item i, or (b) if we do. Filling in the table of all subproblem solutions row-by-row for $i = 1 \dots n$ and $j = 1 \dots C$ takes $\Theta(nC)$ time, after which the optimal solution to the overall problem resides in $V[n, C]$. A traceback path starting from this location reveals the set of items selected for inclusion in the optimal solution. [More detailed elaboration on this solution]

The following problems give us more practice solving similar "two-dimensional" DP problems where we fill in a table of subproblem solutions one row at a time.

Problem 210 (Additional Knapsack Variants). This problem highlights some common variants of the knapsack problem.

(a) **Multiple Copies of Items Allowed.** If multiple copies of each item can be placed in the knapsack, please show how this problem can by solved with a simple "one-dimensional" formulation leading to an $O(nC)$ DP algorithm. [Solution]

(b) **High Multiplicity Knapsack.** Suppose we specify along with each item i a count k_i governing the maximum number of copies of item i one can include in a solution. For a challenge, please give an $O(nC)$ DP algorithm for this variant. [Solution]

[4] Recall from Section 1.4.6 that an $O(nC)$ running time is pseudo-polynomial since it depends directly on C; a dependence on $\log C$ would be necessary for a (weakly-)polynomial running time.

(c) **Integer Values.** Please give a fast DP algorithm for solving a 0/1 knapsack problem in which item values rather than sizes are integers. [Solution]

Problem 211 (Subset Sum). Given a set $S = \{a_1 \dots a_n\}$ of n positive integers and a target value T, the NP-hard *subset sum problem* involves finding a subset of S whose elements sum to T, if one exists. Give a DP algorithm that solves a slightly more general problem: find a subset of S whose elements sum to a value as close to T as possible. Your algorithm should run in $O(nC)$ time, where $C = \sum_i a_i$. [Solution]

Problem 212 (Edit Distance and Sequence Alignment). In this problem we investigate several algorithmic approaches for "aligning" two similar strings to measure their similarity or dissimilarity. Algorithms for these problems and their relatives have been studied extensively in the literature. For example, in bioinformatics they are used to measure similarity between related DNA sequences. A related technique called *dynamic time warping* is used for aligning general time series data, often to discover patterns occurring at different rates (e.g., motion capture data of an individual walking versus running, or speech signal data of the same sound but spoken at different speeds). Related results appear also in problems **??** and **??**.

(a) **Edit Distance.** A common way to measure dissimilarity between two objects is to compute the cost for transforming one into the other. The minimum cost of transforming from string A to string B is known as the *edit distance* between A and B (sometimes also *Levenshtein distance*, named after the first author to introduce the concept). While transforming A into B, we are allowed to use the following operations: *insertion* of a new character into A (at a cost of C_i), *deletion* of a character from A (at a cost of C_d), and *replacement* of a character in A with another character (at a cost of C_r). Given these costs, design an $O(mn)$ dynamic programming algorithm that computes the edit distance between A (length m) and B (length n). [Solution]

(b) **Maximum-Similarity Alignment.** Let us define a character-to-character similarity function $f(c_1, c_2)$ that expresses the similarity of two characters c_1 and c_2. Another way to compare two strings A and B, we first insert spaces (i.e., "empty" characters) into various locations in both strings so that the two have equal length, such that at no common location do we find a space in both strings. This is known as an *alignment* of A and B. We can compute the value of this alignment by summing up the similarity of each pair of corresponding characters across the two strings (our similarity function f also specifies the value of aligning a character against a space — often this value is zero). Our goal is to find an alignment with maximum similarity. For example, if we have strings A = "SIMILAR" and B = "ALIGNMENT", then if characters nearby in the alphabet are treated as being very similar, the optimal alignment might be:

```
-SI-MILA-R
ALIGN-MENT
```

Give an $O(mn)$ DP algorithm that computes an alignment of maximum similarity between strings of length m and n. Show also how the edit distance problem is nothing more than a special case of this problem. [Solution]

(c) **Pattern Matching with up to k Differences.** Suppose you want to find every offset within a larger text at which a specified pattern approximately matches, meaning here that the edit distance is at most k (with unit costs for insertion, deletion, and modification used in the edit distance calculation). For a challenge, please show how to extend the solution to problem 152 to solve this problem in $O(kn)$ time, where k is provided as input. Just as in problem 152, please extend your algorithm to report the offsets achieving minimum edit distance also in $O(kn)$ time, where k is the minimum edit distance over all offsets, and as such is not known to you in advance. As a result, this gives an $O(kn)$ algorithm for computing the unit-cost edit distance k between two strings of combined length n. [Solution]

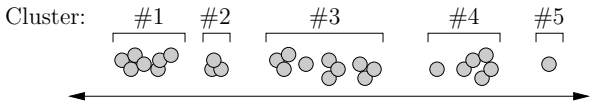


FIGURE 11.8: Partitioning a one-dimensional set of points into clusters.

Problem 213 (Common Subsequences and Supersequences). Here are a few more well-studied string problems that typically involve DP solutions.

(a) In the *longest common subsequence* problem, you are given two strings of lengths m and n, and you wish to find the longest possible string that is a subsequence of each of them (recall that a subsequence does not need to be contiguous). Use DP to solve this problem in $O(mn)$ time. [Solution]

(b) The *shortest common supersequence* problem gives you two strings of lengths m and n, and asks for the shortest string that contains each of them as a subsequence. Use dynamic programming to solve this problem in $O(mn)$ time. [Solution]

(c) The longest common subsequence of two strings A and B of length m and n can be computed in $O(mn)$ time using the solution to part (a). However, if A and B share very few characters in common, one can solve the problem much faster. For each character c in A, let $r_A(c)$ denote the number of occurrences of c in A, let $r_B(c)$ be the number of occurrences of c in B, and let $r = \sum_c r_A(c) r_B(c)$. See if you can give a clever reduction from the longest common subsequence problem to a longest increasing subsequence problem (problem 207) over a sequence of length r. Using the fast algorithm for longest increasing subsequence we develop later in problem **??**, we can therefore solve the longest common sequence problem in just $O(r \log r)$ time. In the worst case ($r = mn$), this is inferior to our previous dynamic programming approach; however, for smaller values of r, it can be much faster. [Solution]

Problem 214 (One-Dimensional Clustering). We study clustering and other data analysis problems extensively in Chapter **??**. Whereas most clustering problems are quite challenging in higher dimensions, they are often approachable with DP in one dimension. Consider dividing points $x_1 \ldots x_n$ into a specified number of k clusters, as in Figure 11.8. The *k-cluster* problem asks us to do this by minimizing the largest cluster *diameter* (its maximum point minus its minimum point). Other objectives ask us to compute not only a partition into clusters, but also to designate a *center* point for each cluster, which may not necessarily be one of the n points in our input. Clustering problems with objectives of this type include the *k-means* problem (minimize the sum of squared distances between each of our points and its respective cluster center), the *k-medians* problem (minimize the sum of distances between each point and its cluster center), and the *k-center* problem (minimize the largest distance from any point to its cluster center). It is not too difficult to show that the k-center and k-cluster problems are equivalent in one dimension.

(a) For each of the clustering problems above (k-cluster, k-means, k-medians, and k-center), give an $O(n^2 k)$ DP algorithm that generates an optimal solution. Does it affect the complexity of your algorithm if you also provide a lower bound and an upper bound on the number points allowed in a cluster? [Solution]

(b) If you are feeling ambitious, give an $O(nk)$ solution to the k-center problem. [Solution]

(c) Consider the k-center problem in which k of the input points are designated as "anchor" points that must end up belonging to different clusters. Subject to this new constraint, show how to compute an optimal solution in $\Theta(n)$ time. [Solution]

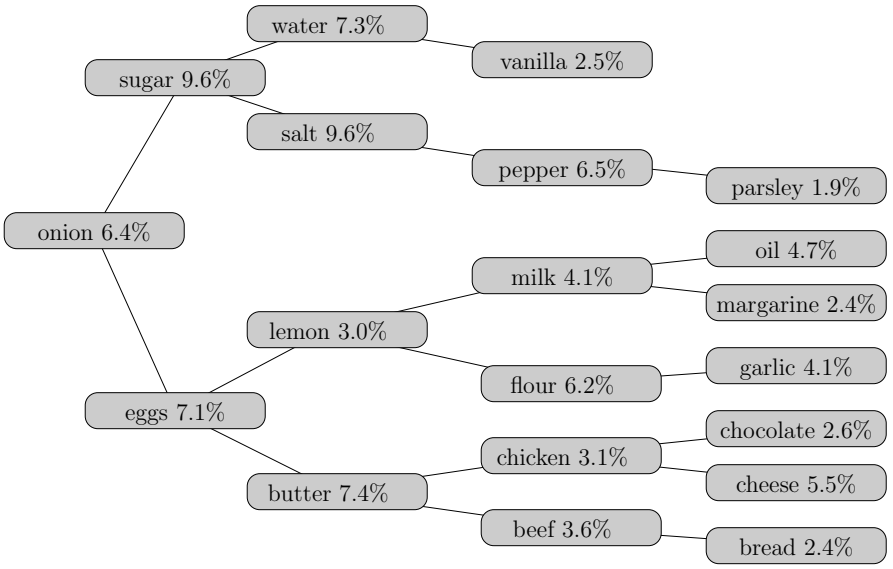


FIGURE 11.9: An optimal binary search tree (drawn sideways, so it fits more easily on the page) built from a collection of the most common ingredients extracted from a large set of recipes. Based on the relative frequencies of the ingredients, the average lookup in this tree will examine only 3.44 nodes, even though two thirds of the elements require either 4 or 5 node examinations to locate.

11.3 Example: Optimal Binary Search Trees

A balanced binary search tree (BST) is a natural choice for storing a dictionary of n elements, allowing fast lookup via the *find* operation in just $O(\log n)$ time. However, if we are also given access probabilities $p_1 \ldots p_n$ for the elements, we can do even better: the *optimal BST* problem asks us to tune the shape of our BST to minimize the expected search depth for a randomly-chosen element (element i being chosen with probability p_i). As an example, Figure 11.9 gives the optimal BST on a set of common cooking ingredients.

To compute the optimal BST on a set of elements ordered $1 \ldots n$, the natural first decision is to pick which element r goes at the root, after which the left and right subtrees can be recursively constructed from elements $1 \ldots r-1$ and $r+1 \ldots n$. These smaller subproblems are themselves instances of the optimal BST problem, leading us to the following DP formulation. [Detailed derivation]

> $D[i,j]$: Expected search depth for an optimal BST constructed from only elements $i \ldots j$, normalized by multiplying by $p_{ij} = p_i + \ldots + p_j$.
>
> $$D[i,j] = p_{ij} + \min_{r=i \ldots j} \{D[i, r-1] + D[r+1, j]\}.$$
>
> Base case: $D[i, i-1] = 0$.

This $O(n^3)$ DP algorithm represents a different structure than we have seen before, a structure that is common to many other problems. To begin with, our $n \times n$ table

of subproblem solutions D isn't filled in row by row, but diagonal by diagonal. We first compute $D[i,i]$ for all i, then $D[i,i+1]$ for all i, then $D[i,i+2]$, and so on, until we eventually solve the top-level problem $D[1,n]$ involving all the elements $1 \ldots n$. Furthermore, our subproblems decomposition is more "divide and conquer" than "incremental construction" in flavor, as we are decomposing a problem into two large pieces in each recursive step, instead of repeatedly adding one element at a time to build up incrementally larger solutions to prefixes of our problem. You may hear this sometimes called "nested parenthesis" DP, as motivated by the following problem involving multiplication of a chain of matrices $A_1 A_2 \ldots A_n$, since the top-level decision involves parenthesizing the expression into two high-level blocks $(A_1 \ldots A_r)$ and $(A_{r+1} \ldots A_n)$ that are then multiplied together. Problems amenable to this type of decomposition tend to be those involving objects with a similar "balanced parenthesis" structure, such as the large class of combinatorially-equivalent structures introduced in Section 2.6.

Problem 215 (Matrix Chain Multiplication). Many applications require the multiplication of sequences of matrices. Suppose you are given a sequence of n matrices $A_1 \ldots A_n$, and you would like to compute their product $A_1 A_2 \ldots A_n$. Since matrix multiplication is associative, we can parenthesize such an expression as we see fit, and the parenthesization we choose can have a significant impact on the amount of time it takes to evaluate the expression. For example, suppose we have four matrices $A_1(1 \times 100)$, $A_2(100 \times 100)$, $A_3(100 \times 100)$, and $A_4(100 \times 1)$. If we compute the product $(A_1 A_2)(A_3 A_4)$ using the straightforward algorithm for matrix multiplication, this requires 20100 individual multiplications. However, if we instead compute $(A_1(A_2 A_3))A_4$, this requires in excess of 1 million multiplications! Devise an $O(n^3)$ DP algorithm that optimally parenthesizes our matrix expression so as to minimize the number of individual multiplications required for its evaluation. [Solution]

Problem 216 (Parsing with a Context-Free Grammar). A *context-free* grammar is a set of rules that can be applied to generate various types of strings. For example, consider the set of all strings consisting of one or more 0s followed by the same number of 1s. A simple context-free grammar that generates this set consists of the two rules (also called *productions*) $S \to 01$ and $S \to 0S1$. Here, the symbols 0 and 1 are known as *terminal* symbols since they may exist in our final string generated by the grammar, and the symbol S is called a *nonterminal* symbol since it cannot be present in our final string. To generate a string using our grammar, we start with the initial string S and then apply a sequence of productions in some order until we obtain a string containing only terminal symbols, at which point we stop. Any time we have a string containing nonterminal symbols, we must continue to apply productions (each of which replaces a single nonterminal symbol with a string containing terminal and nonterminal symbols) until we finally get rid of all the nonterminals. Context-free grammars are quite powerful in their ability to describe many different classes of strings — for example valid arithmetic expressions, computer programs, and so on. They are called "context free" because each production (e.g., $S \to 0S1$) has only a single nonterminal symbol on its left-hand side, so it can be applied to any instance of S in our string regardless of its surrounding context. It turns out that any context-free grammar can be converted fairly easily into a simple canonical form known as *Chomsky normal form* (CNF), where each production transforms a single nonterminal symbol into either two nonterminal symbols (e.g., $S \to AB$, with uppercase letters denoting nonterminal symbols, as is the usual convention), or a single terminal symbol (e.g., $S \to 0$). Given a context-free grammar G with k productions written in CNF form, please describe a simple $O(n^3 k)$ DP algorithm for deciding whether or not G can generate a given string of length n. [Solution]

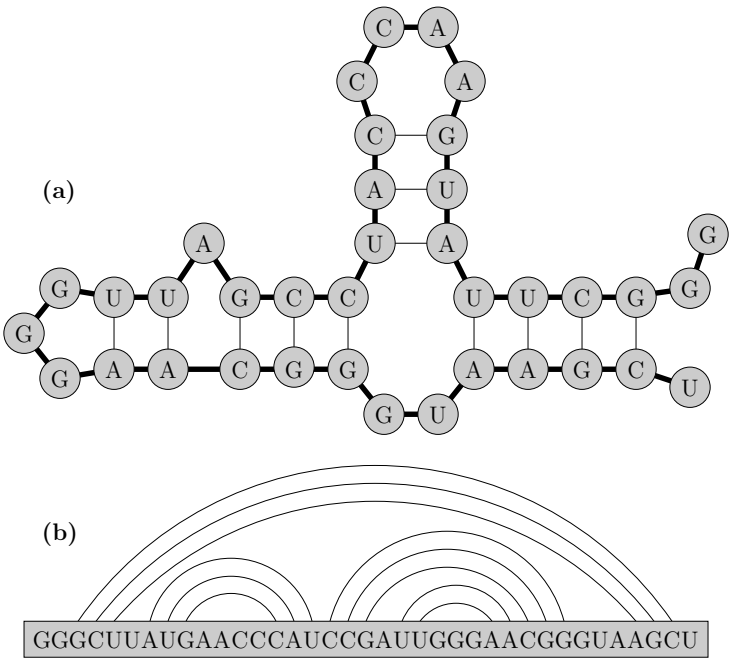


FIGURE 11.10: A pseudoknot-free folding of an RNA sequence, shown (a) in terms of its high-level molecular structure, and (b) as a sequence, with arcs drawn between bonded pairs of bases.

Problem 217 (Prediction of RNA Folding). In biology, RNA molecules are long polymers (chains) consisting of four types of bases (small molecular units that can hook together into a chain): Adenine (A), Guanine (G), Cytosine (C), and Uracil (U). The structure of these bases is such that A and U tend to stick together, as do G and C. These affinities (and many other factors) cause RNA molecules to fold into certain shapes. Since folded RNA molecules have been found to carry out many crucial biological functions (catalyzing reactions, etc.), an important problem in computational biology is the prediction of the folding structure of an RNA molecule based on its sequence of bases. In this problem, we concern ourselves only with the *secondary structure* of an RNA folding; that is, what parts of the RNA chain bond to what other parts, and not the geometric aspects of how this folding is realized in three dimensions. Figure 11.10(a) illustrates a folding of a short RNA molecule, which we can also depict as in Figure 11.10(b) by drawing the chain of bases and indicating which are linked. We call a folding *pseudoknot-free* if the pairing of bases is properly nested as in our example (i.e., no lines cross). It is not too difficult to see that pseudoknot-free foldings look like trees. Many RNA sequences are known to have pseudoknot-free foldings, so we will focus on this case since it is computationally much simpler. An RNA molecule tends to fold into a configuration that minimizes its *free energy*. Every A-U or C-G bond formed has a stabilizing effect and lowers the free energy. On the other hand, if these bonds bend the structure so it has sharp "hairpin" loops, this can also raise the free energy. Many different techniques have been proposed for modeling the free energy of a particular folding. Let us be as simple as possible and try to compute a pseudoknot-free folding that maximizes the number of A-U and C-G pairings. Please give an $O(n^3)$ DP algorithm that solves this problem. [Solution]

11.4 DP and Approximation Algorithms

Just like greedy methods, DP also plays an important role in many approximation algorithms. In this section, we will see in particular how DP helps in building a polynomial-time approximation scheme (PTAS) for several prominent problems; recall from Section 1.7.2 that a PTAS delivers a solution whose value differs from that of an optimal solution by only $(1 \pm \varepsilon)$, for any constant $\varepsilon > 0$.

Example: Approximating the Knapsack Problem. In the previous chapter (problem 199) we developed a simple greedy 1/2-approximation algorithm for the knapsack problem that runs in $\Theta(n)$ time. We now improve on this by developing an $O(\frac{n}{\varepsilon} \log n)$ time algorithm that finds a set of items fitting in the knapsack whose value is at least $(1 - \varepsilon)OPT$, where OPT denotes the value of an optimal solution.

We start by running our greedy 1/2-approximation algorithm to obtain a solution value V that is a good initial estimate of OPT, since $OPT/2 \leq V \leq OPT$. We then "discretize" the values of our n items by rounding each one down to the next-lowest multiple of $\varepsilon V/n$. Since an optimal solution contains at most n items, this rounding step decreases the value of an optimal solution by at most $\varepsilon V \leq \varepsilon OPT$, so by optimally solving the resulting discretized problem, we will find a solution whose value is at least $(1 - \varepsilon)OPT$, our desired approximation guarantee. Note that the guarantee holds even when we then "un-round" our values back to their original settings, since this can only increase the value of our solution.

To solve the discretized problem, we use the DP algorithm from problem 210(c), which optimally solves the knapsack problem with integer values. Thanks to our discretization step, our item values are now effectively integers in a small range, since they are all multiples of $\varepsilon V/n$ and at most $OPT \leq 2V$. If we re-scale everything for convenience so that $\varepsilon V/n = 1$, then our values become integers in the range $0 \ldots 2n/\varepsilon$. This gives $\Theta(n/\varepsilon)$ subproblems in our DP formulation, each solved in $\Theta(n)$ time, for a total running time of $\Theta(n^2/\varepsilon)$. We will shortly improve this to $O(\frac{n}{\varepsilon} \log n)$ using additional tricks.

Problem 218 (A PTAS for Deadline-Constrained Scheduling). Suppose we are given n jobs, where each job j has a processing time p_j, a deadline $d_j \geq p_j$, and a value v_j. We would like to non-preemptively schedule a maximum-value subset of jobs on a single machine such that deadline constraints are respected. This problem is NP-hard since if all deadlines are equal to a common value C, it is nothing more than the knapsack problem. In problem 163, we showed how a greedy algorithm optimally solves the special case where all values are 1. For the general case, show how to construct a PTAS by modifying our preceding PTAS for the knapsack problem. [Solution]

Treating Large and Small Elements Separately. Greedy algorithms are usually effective for "fine-grained" problem instances — involving continuous quantities or many small elements. Conversely, DP algorithms are generally more effective on "coarse-grained" instances involving only large elements. We illustrate both of these phenomena in the following two problems, then show how to combine these two regimes together to build approximation algorithms where small elements are processed greedily and large elements are handled by DP.

Problem 219 (A Greedy PTAS for Problems with Small Elements). For problem instances involving only small elements, greedy algorithms often provide a PTAS.

(a) Suppose all items in a capacity-C knapsack problem have size at most εC. Show that the natural greedy algorithm provides a PTAS in this case. [Solution]

(b) The minimum-makespan scheduling problem is introduced in problem 203. Show how the greedy algorithm from part (a) of that problem is a PTAS if it is known that the processing time of each job is at most εOPT, where OPT denotes the optimal makespan. [Solution]

(c) The bin packing problem is introduced in problem 201. A natural greedy algorithm for this problem is to consider items in any order, and for each one in sequence, to place it into any bin in which it fits, opening a new bin only when necessary. If all item sizes are at most ε, then this algorithm is not technically a PTAS according to our exact definition above since it does not necessarily use at most $(1+\varepsilon)OPT$ bins[5]. However, one can show something very close: please prove that the greedy algorithm uses at most $(1+2\varepsilon)OPT+1$ bins as long as $\varepsilon < 1/2$. [Solution]

Problem 220 (Using DP for Problems with Large Elements). Consider the three problems in problem 219 above: knapsack, minimum-makespan scheduling, and bin packing. For problem instances involving only suitably large elements, we can develop either an optimal solution or PTAS for each problem using DP.

(a) Consider the knapsack problem in which every item has size at least εC. Please show how to optimally solve this problem in polynomial time with DP, assuming ε is a constant. As a hint, how many items can fit in the knapsack? [Solution]

(b) Consider the minimum-makespan scheduling problem in which each job has a processing time of at least εOPT, where OPT denotes the optimal makespan. Construct a PTAS for this problem using DP. As a hint, round all processing times up to the nearest multiple of $\varepsilon^2 OPT$ and then show that there are only polynomially-many distinct configurations of jobs that can be scheduled on a single machine. [Solution]

(c) Consider the bin packing problem in which there are a only a constant number of distinct item sizes, and show that this problem can be solved in polynomial time using DP. As before, consider how many configurations of items can fit in a single bin. [Solution]

(d) Consider the bin packing problem in which all item sizes are at least ε. Construct a PTAS for this problem using the following scheme: sort the items by size and partition this sorted list into blocks each of which (except possibly the last) contains $n\varepsilon^2$ items. Construct a new problem instance by rounding down the size of each item to that of the smallest item in its block. This instance will have only a constant number of distinct item sizes, so we can solve it optimally using the DP algorithm from the previous part. The only challenging aspect that remains is proving that that this solution will use at most $(1+\varepsilon)OPT$ bins, where OPT denotes the number of bins required in an optimal solution for the original problem instance. As a hint, it may help to make a comparison to yet another problem instance in which we round the items in each block *up* to the largest item size in the block. [Solution]

[5] You may recall from back in Section 1.7.2 that it is NP-hard to approximate the bin packing problem to within a factor better than 3/2. While this does rule out the possibility of a PTAS as defined as a $(1+\varepsilon)$-approximation, it still leaves open the possibility of achieving an approximation bound similar to that of a PTAS but with a small additive term. This type of approximation result is sometimes known as an *asymptotic* PTAS, since as our problem instances (and hence also OPT) grow larger and larger, the additive term becomes less significant and our approximation bound begins to behave more like a purely multiplicative bound.

By carefully merging these approaches, we can build approximation algorithms that use a combination of DP (for large elements) and greedy methods (for small elements) to achieve strong approximation results for general instances. For example, we can obtain a PTAS for all instances of the minimum-makespan scheduling problem [Complete algorithm] and an asymptotic PTAS for all instances of the bin packing problem [Complete algorithm].

Problem 221 (Approximating the Knapsack Problem, Revisited). Recall that we recently developed a PTAS for the knapsack problem running in $O(n^2/\varepsilon)$ time.

(a) Can you show how to improve the running time of our knapsack PTAS to $O(\frac{n}{\varepsilon}\log n)$? As a hint, treat separately items of large value (say, greater than εV, where V is the value given by our $\Theta(n)$-time greedy 1/2-approximation algorithm), and those of small value. [Solution]

(b) Consider the variant of the knapsack problem where we are allowed to place arbitrarily many copies of each item in the knapsack. Why does our former PTAS not work for this variant, and how can we develop an alternate PTAS that does work? [Solution]

11.5 Advanced Tricks

We have now seen three main classes of DP algorithms. In Section 11.1 we studied "one-dimensional" problems, where we solve a sequentially solve a single linear chain of subproblems. In Section 11.2 we studied "two-dimensional" problems, where we fill in a table of subproblem solutions one row at a time. Finally, Section 11.3 studies two-dimensional problems with "balanced parenthesis" structure, where subproblem solutions are filled in one diagonal at a time. Although there are several other common dynamic program "templates" (e.g., dynamic programs in trees that solve a subproblem for each subtree during a post-order traversal, as in problem 228), the three above are probably the most common. In this section, we highlight several elegant general tricks for improving the time or space usage of DP algorithms belonging to these three classes.

11.5.1 Reducing Space Usage

Consider the two-dimensional "row-by-row" problems we studied in Section 11.2 (e.g., knapsack, subset sum, edit distance, maximum similarity alignment, longest common subsequence, one-dimensional clustering). For all of these problems, we can save space while filling in our table of subproblem solutions by noting that each row depends only on subproblems in the preceding row. To give an example, consider the knapsack problem with n items and capacity C, which we can solve in $\Theta(nC)$ time by filling an $n \times C$ table of subproblem solutions one row at a time. Since we only need to maintain the current and previous row as we fill the table, we can obtain the *value* of an optimal solution to this problem using only $\Theta(C)$ total space. Unfortunately, if we do not retain the entire $n \times C$ table of subproblem solutions and their associated backpointers, we seemingly lose the ability to recover the *structure* of this optimal solution, since we no longer have a usable traceback path. Our first advanced trick is a clever divide and conquer method that allows us to recover the

structure of an optimal solution while still only using space proportional to the size of a single row in our table, without harming the asymptotic running time of our DP algorithm. [Full details]

11.5.2 Speedups via Quadrangle/Monge Inequalities

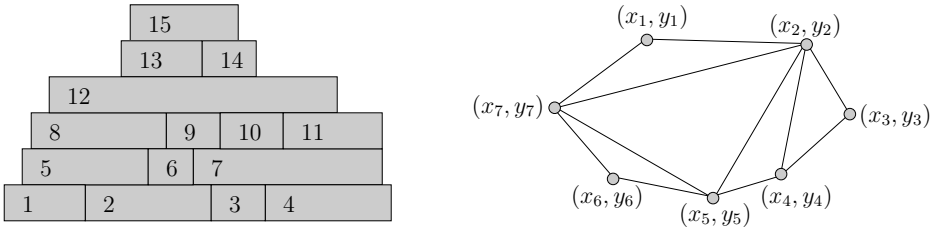
Consider now the "balanced parenthesis" problems we studied in Section 11.3 (e.g., optimal binary search trees, matrix chain multiplication). Let A denote our two-dimensional table of subproblem solutions, so for example with optimal binary search tree construction, $A[i,j]$ is the cost of building an optimal binary search tree on a range of elements $i \ldots j$, by selecting some root k in this range and then recursively building optimal binary search trees on subranges $i \ldots k-1$ and $k+1 \ldots j$. This problem and many other DP problems satisfy useful *monotonicity* properties, allowing us to achieve dramatic speedups in running time. For this example, if we determine the optimal root k for range $i \ldots j$, then after increasing either i or j, the optimal root for our new subproblem k' tracks in a monotonic fashion: $k' \geq k$. If we are filling in a diagonal of our DP subproblem table, say by solving ranges $i \ldots i+99, i+1 \ldots i+100, i+2 \ldots i+101$, we would normally need to loop over 100 prospective roots when solving each range. Thanks to monotonicity, however, we need only one linear scan through the n possible roots during this entire process, ultimately saving a factor of n in our final running time.

The structural property often enabling this sort of speedup is the *quadrangle* inequality (also known as the *Monge* property), which states that $A[i,j] + A[i',j'] \leq A[i',j] + A[i,j']$ for all $i \leq i'$ and $j \leq j'$. This formula may look slightly imposing, but it is really nothing more than a four-point generalization of the triangle inequality based on the fact that the combined length of the two diagonals in any quadrilateral can be no shorter than the combined length of two opposite sides[6]. An alternative, slightly simpler characterization for quadrangle/Monge matrices is that they must satisfy $A[i,j] + A[i+1,j+1] \leq A[i+1,j] + A[i,j+1]$ for all (i,j). As it turns out, the subproblem tables for many "balanced parenthesis" dynamic programs satisfy the quadrangle/Monge property; this is true for optimal binary search trees, matrix chain multiplication, and optimal polygon triangulation (problem 223). As shown by Knuth and Yao, this property enables speedup by a factor of n using monotonicity as described above, improving running time from $O(n^3)$ to $O(n^2)$ for all three of these example problems. [Full details]

11.5.3 Speedups via Monotone Matrix Searching

Here, we discuss another class of DP-solvable problems we can speed up by exploiting monotonicity. In Section 11.1, we showed how many problems are equivalent to acyclic shortest path computation, more specifically the calculation of a shortest path from node 1 to node n along a topological ordering of nodes numbered $1 \ldots n$. We can abstractly specify any such problem with a matrix C whose (i,j) entry gives the cost of an edge from node i to node j. Since our acyclic graph only contains edges (i,j) with $i < j$, we set $C[i,j] = +\infty$ for $i \geq j$. Let $M(i)$ denote

[6]For more geometric intuition on the quadrangle inequality, please consult problem **??** at the end of Chapter **??**.



(a) **(b)**

FIGURE 11.11: An example of (a) a stable stack of blocks, and (b) a triangulation of a convex polygon.

the index of the minimum entry in row i of C (in the event of a tie, we take the rightmost minimum entry). We say C is *monotone* if $M(1) \leq M(2) \leq \ldots \leq M(n)$, and we say C is *totally monotone* if every submatrix of C is monotone (actually, it suffices simply to prove that every 2×2 submatrix of C is monotone). It is also fairly easy to prove that C is totally monotone if it satisfies the quadrangle/Monge properties above. The problem of computing $M(1) \ldots M(n)$ in a totally monotone matrices arises in several situations (e.g., see problem **??**). As a result, substantial research was devoted to this problem in the 1980s, culminating in a simple and elegant divide-and-conquer algorithm known as the SMAWK algorithm (named for its authors: Shor, Moran, Aggarwal, Wilber, and Klawe), which is remarkably capable of computing $M(1) \ldots M(n)$ in only $\Theta(n)$ time, despite the $n \times n$ size of our matrix. [Details of the SMAWK algorithm]

Many one-dimensional DP problems (for example, problem 209(g) on optimal typesetting) have a cost matrix C that is totally monotone. For such problems, calculation of $M(1) \ldots M(n)$ in linear time can be used to solve a batch of n subproblems in just $\Theta(n)$ time. Still more DP problems can be decomposed into an "interleaved" set of one-dimensional problems, each with totally monotone cost matrices [Example: sequence alignment problems with gaps]. Our final trick is a somewhat sophisticated algorithm, sometimes called the LARSCH algorithm (named after its authors, Larmore and Schieber), that makes clever use of the SMAWK algorithm to deliver a factor of n speedup for these problems [Full details]. We can also leverage this approach to save a factor of n in the running time of certain two-dimensional row-by-row problems (e.g., one-dimensional clustering) for which the computation of each row of subproblem solutions is equivalent to a one-dimensional problem with a totally monotone cost matrix. [Details]

11.6 Additional Problems

Problem 222 (Block Stacking). You are given a set of n unit-height blocks with widths $w_1 \ldots w_n$. Your goal is to stack these blocks to the maximum possible height. As shown in Figure 11.11(a), the blocks must be stacked in order from block 1 to block n, and for stability each level of the stack must be no wider than the level upon which it rests. Show how to solve this problem efficiently using dynamic programming. [Solution]

Problem 223 (Optimal Convex Polygon Triangulation). A convex polygon P is specified by giving the coordinates $(x_1, y_1) \ldots (x_n, y_n)$ of its n vertices in clockwise order. As shown in Figure 11.11(b), a *triangulation* of P is a collection of line segments that partitions P into a set of triangles, where each segment runs between two of the vertices of P. We wish to compute a triangulation of P that minimizes the combined lengths of these added segments. Show how to compute such an optimal triangulation in $O(n^3)$ time. [Solution]

Problem 224 (The Transportation Problem with Only Two Sources). Suppose we have n factories that produce goods to be shipped to a collection of m warehouses. The ith factory supplies s_i (an integer) units of goods, and the jth warehouse has a demand for d_i units of goods. For each pair (i, j), we have a transportation constraint that says we can ship at most u_{ij} units of goods from factory i to warehouse j, and each unit of goods shipped from i to j costs c_{ij} units. We would like to determine how many units of goods to ship from each factory to each warehouse so that (i) all demand is satisfied — d_j units arrive at each warehouse j, (ii) at most s_i units are shipped out of each factory i, (iii) at most u_{ij} units are shipped from factory i to warehouse j. Subject to these constraints, we would like to incur the least amount of cost possible. In Chapters **??** and **??**, we will see how to solve this problem (called the *transportation problem*) in polynomial time using network flow techniques. Here, we consider only the special case with $n = 2$ factories, since this is a nice DP problem (recall also that there is a simple greedy solution for the restricted case with no upper capacities u_{ij} and exact supply/demand constraints, as we saw in problem 197). Please show how to solve this problem in $O(mS^2)$ time, where $S = s_1 + s_2$ is the total amount of supply at both factories. [Solution]

Problem 225 (Longest Arithmetic Progression). Given a set S of n numbers, we would like to find the longest arithmetic progression (either increasing or decreasing) that is a subset of S. For example, in the set $\{4, 2, 5, 7, 3, 12, -8, 17, 20, 21\}$, the longest arithmetic progression is the sequence 2, 7, 12, 17. See if you can devise an $O(n^3)$, or better still, $O(n^2)$, DP algorithm for this problem. [Solution]

Problem 226 (Shortest Bitonic Tour). This is a "classic" DP problem. Suppose you are given a list of n cities ordered from west to east, and you are told an $n \times n$ matrix of distances between every pair of cities. You would like to construct a path of minimum total length that starts at the eastmost city, travels west until it reaches the westmost city, then travels east again until it returns to the eastmost city. In the process, the path must visit every city exactly once except the eastmost city, which it visits twice. Please solve this problem in $O(n^2)$ time using DP. Later, in problem **??**, we consider the more general problem in which you can make k passes back and forth rather than just 1. [Solution]

Problem 227 (Disk Scheduling). Suppose we need to read n blocks of data that are spread out across a disk. For the purposes of this problem, let us model a disk as having a single read/write head that moves back and forth across a linear array of blocks. For simplicity, we adopt the slightly unrealistic assumption that the amount of time required to move the read/write head from block i to block j is proportional to $|i - j|$ (the distance between the two blocks). Once the read/write head reaches one of our n input blocks, it reads it instantly, so the only time-consuming operation is the movement of the head. Given the locations of our n blocks as well as the starting location of the read/write head, devise an $O(n^2)$ dynamic programming algorithm that minimizes the total time required to read all the blocks. [Solution]

Problem 228 (DP Algorithms on Trees). It is often the case that special cases of NP-hard problems can be solved in polynomial time. In this problem we investigate special cases of NP-hard graph problems that are easily solved on trees via DP. See Section **??** for a more complete discussion of these problems.

(a) A subset of nodes in a graph is an *independent set* if there is no edge connecting any

pair of these nodes, and a *node cover* (also commonly called a *vertex cover*) if every edge has at least one endpoint in the set. If we associate values with the nodes of a graph, the maximum-value independent set problem asks for an independent set of maximum total value. If nodes have associated costs, we can similarly ask for a minimum-cost node cover. Both problems are NP-hard in general graphs. Please show how to solve them both in $\Theta(n)$ time on an n-node tree using DP. [Solution]

(b) A subset of nodes in a graph is a *dominating set* if it covers all the nodes in the graph. That is, every node not in the set must at least have one of its neighbors in the set. If we associate costs with the nodes in a graph, we can consider the *minimum-cost dominating set problem*, which is NP-hard on general graphs. Please show how to solve it in $\Theta(n)$ time on an n-node tree using DP. [Solution]

(c) A *distance-k dominating set* is a subset of the nodes of a graph such that every node not in the set has at least one node in the set within distance k of it. Stated differently, every node we select for our set has the ability to cover, or "dominate", every node within distance k, and we wish to cover every node in the entire graph. For a challenge, please give an $O(kn)$ DP algorithm for the minimum-cost distance-k dominating set problem in a tree. [Solution]

(d) As a challenge, see if you can design a $\Theta(n)$ "greedy" algorithm for the *minimum-cardinality* distance-k dominating set problem in an n-node tree. [Solution]

(e) Suppose we associate a numeric "weight" with every node in a tree, some of which may potentially be negative. For a specified value of k, we would like to find a subtree induced by a selection of exactly k leaves (that is, a tree consisting of just those leaves plus any internal nodes that are necessary to connect them all together) of maximum total weight. Please show how to solve this problem in $O(nk^2)$ time. For a challenge, can you refine your analysis of the same algorithm to show that it runs in $O(n^2)$ time, even if you want the answer for every value of k? [Solution]

Problem 229 (Asymmetric Binary Search). Suppose we would like to find the value of some unknown quantity X whose value is known to live in the range $1 \ldots n$. Our only means of learning about X's value is to make repeated guesses, where we can tell if each successive guess is too high or too low. For each guess that is too high, we incur a penalty of P_{high}, and for each guess that is too low, we incur a penalty of P_{low}. Devise an $O(n^2)$ DP algorithm that determines a guessing strategy that minimizes the maximum possible penalty we might incur. If P_{high} and P_{low} are integers of magnitude at most B, please show an alternate formulation achieving a running time of $O(B \log n)$. [Solution]

Problem 230 (Scheduling with Release Times and Deadlines). Please use DP to give an optimal offline algorithm for problem 166 (i.e., where we know the release times of all jobs in advance). [Solution]

