3 Dynamic Programming

In this chapter, we study an algorithmic technique known as dynamic programming (DP). The idea is to solve a sequence of increasingly larger subproblems by using previously computed solutions until we've solved the original problem. For all of these problems, there are two parts of a solution: the optimum *value* (e.g., the *length* of a subsequence), or the actual solution itself (e.g., a subsequence of an array). Technically, the algorithms only compute the value of the optimal solution, but as we'll see, recovering the solution itself is often relatively straightforward.

The proofs of correctness for all the algorithms in this chapter simply rely on the correctness of a recurrence. The recurrence tells us the structure of an optimal solution in terms of smaller subproblems. Once the recurrence is established, the only thing that remains is to solve the subproblems (using the recurrence) in an order such that when we solve one subproblem, we have already solved the required smaller subproblems. Thus, for clarity of presentation, we omit the formal "theorem-proof" approach in favor of a pre-algorithm analysis that establishes the recurrence.

3.1 Longest Increasing Subsequence

Problem statement. Let $A[1, \ldots, n]$ denote an array of integers. Our goal is to find the longest increasing subsequence (LIS) in A. For example, the LIS of $[4, 1, 5, 2, 3, 0]$ is $[1, 2, 3]$, and the LIS of $[3, 2, 1]$ is $[3]$ (or $[2]$, or $[1]$).

Analysis. Let $\mathsf{OPT}[i]$ denote the length of the longest increasing subsequence of A that must end at i, so $\mathsf{OPT}[1] = 1$. For $i > 1$, $\mathsf{OPT}[i]$ corresponds to appending $A[i]$ to the longest subsequence ending before i, but the last element of this subsequence must be less than $A[i]$. In other words, we have

$$\mathsf{OPT}[i] = 1 + \max_{\substack{1 \le j < i \\ A[j] < A[i]}} \mathsf{OPT}[j]. \tag{1}$$

(If none of the elements before i are smaller than $A[i]$, then $\mathsf{OPT}[i] = 1$.) The optimal solution OPT must end somewhere, so $\mathsf{OPT} = \max_i \mathsf{OPT}[i]$.

Algorithm. In the algorithm, we maintain an array $d[1, \ldots, n]$, and compute $d[i]$ according to Eq. (1). Computing $d[i]$ requires making $i - 1$ comparisons, so the overall running time is $O(n^2)$. Since we don't know where the optimal solution ends, we return $\max_i d[i]$.

Algorithm 5 Longest Increasing Subsequence

```
1: Initialize an array d[1, ..., n] containing all 1's.
2: for i = 2, ..., n do
3:     for j = 1, ..., i − 1 do
4:         if A[j] < A[i] and d[i] < 1 + d[j] then
5:             Set d[i] = 1 + d[j].          ▷ append A[i] to the LIS ending at j
6: return max_i d[i]
```

To obtain the actual longest subsequence, and not just its length, we can maintain an array p that tracks pointers. In particular, if $d[i]$ is set to $1 + d[j]$, then we set $p[i] = j$, indicating that the LIS ending at i includes j immediately before i. By following these pointers, we can recover the subsequence itself in $O(n)$ time.

3.2 The Knapsack Problem

Problem statement. Suppose there are n items $\{1, 2, \ldots, n\}$, where each item i has integer weight $w_i > 0$ and value $v_i > 0$. We have a knapsack of capacity B. Our goal is to select a subset of items with maximum total value, subject to the constraint that the total weight of selected items is at most B.[1]

Analysis. Let $\mathsf{OPT}[i][j]$ denote the optimal value for the subproblem in which we could only select from items $\{1, \ldots, i\}$ and the knapsack only has capacity j. Notice that $\mathsf{OPT}[i][j]$ either contains item i, or it doesn't (and "chooses" whichever one is more profitable). If it contains item i, then its value is v_i plus the optimal value obtained from items $\{1, \ldots, i-1\}$ with capacity $j - w_i$. Otherwise, its value is the optimal value obtained from items $\{1, \ldots, i-1\}$ with capacity j. Thus, the recurrence is

$$\mathsf{OPT}[i][j] = \max(v_i + \mathsf{OPT}[i][j - w_i], \mathsf{OPT}[i-1][j]) \tag{2}$$

and $\mathsf{OPT} = \mathsf{OPT}[n][B]$ denotes the value of the optimal solution to the original problem.

Algorithm. The algorithm fills out an array $d[1, \ldots, n][1, \ldots, B]$ according to Eq. (2). Notice that each row depends on the previous row, and we return $d[n][B]$ as our solution. (We implicitly assume $d[i][j] = 0$ if $j \leq 0$.) The array has nB entries, and computing each one takes $O(1)$ time, so the total running time is $O(nB)$.

Algorithm 6 Knapsack

```
Initialize an n × B array d containing all 0's.
for j = 1, ..., n do                                   ▷ initialize the first row
    if w_1 ≤ j then set d[1][j] = v_1.
for i = 2, ..., n do
    for j = 1, ..., B do
        Set d[i][j] = max(v_i + d[i − 1][j − w_i], d[i − 1][j]).   ▷ follow (2)
return d[n][B]
```

By tracing back through the array d starting at $d[n][B]$, we can construct the optimal subset of items. At any point, if $d[i][j] = d[i-1][j]$, then we can exclude item i from the solution. Otherwise, we must have $d[i][j] = v_i + d[i-1][j - w_i]$, and this means we include item i in the solution.

[1] Recall that we solved the fractional version of this problem in Sec. 2.2.

Remark. Despite having a running time of $O(nB)$, Algorithm 6 does *not* run in polynomial time. This is because the input length is proportional to $\log B$ rather than B. In other words, if B is doubled then the running time doubles, but the input length only increases by 1 bit.

3.3 Minimum Edit Distance

Problem statement. We are given two strings $A[1, \ldots, m]$ and $B[1, \ldots, n]$. There are three valid operations we can perform on a: insert a character, delete a character, and replace one character for another. Our goal is to convert A into B (or *match* them) using a minimum number of operations (known as the *edit distance*), where substitutions are free if (and only if) the two characters match.

Analysis. Let $\mathsf{OPT}[i][j]$ denote the edit distance between $A[1, \ldots, i]$ and $B[1, \ldots, j]$. Notice that $\mathsf{OPT}[i][j]$ must modify $A[1, \ldots, i]$ such that its last character is $B[j]$, so it must make one of the following three choices:

1. Insert $B[j]$ after $A[1, \ldots, i]$ for a cost of 1; then match $A[1, \ldots, i]$ with $B[1, \ldots, j-1]$.
2. Delete $A[i]$ for a cost of 1; then must match $A[1, \ldots, i-1]$ and $B[1, \ldots, j]$.
3. Replace $A[i]$ with $B[j]$ for a cost of 0 (if $A[i] = B[j]$) or 1 (otherwise); then match $A[1, \ldots, i-1]$ and $B[1, \ldots, j-1]$.

Thus, the subproblems satisfy the following recurrence:

$$\mathsf{OPT}[i, j] = \min \begin{cases} 1 + \mathsf{OPT}[i][j-1] \\ 1 + \mathsf{OPT}[i-1][j] \\ \delta(i, j) + \mathsf{OPT}[i-1][j-1] \end{cases} \tag{3}$$

where $\delta(i, j) = 0$ if $A[i] = B[j]$ and 1 otherwise. For the base cases, we have $\mathsf{OPT}[i][0] = i$ (i.e., delete all i characters of A to obtain the empty string) and $\mathsf{OPT}[0][j] = j$ (i.e., insert all j characters of B to obtain B).

Algorithm. The algorithm fills out an array $d[0, \ldots, m][0, \ldots, n]$ according to (3) and returns $d[m][n]$. Each of the $O(mn)$ entries takes $O(1)$ time to compute, so the total running time is $O(mn)$. To recover the optimal sequence of operations, we can use the same technique of tracing through the table, as we've seen in the previous sections.

3.4 Independent Set on Trees

Problem statement. Let $T = (V, E)$ be a tree on n vertices, where each vertex u has weight $w(u) > 0$. An *independent set* is a subset of vertices such that no two vertices in the subset share an edge. Our goal is to find an independent set S that maximizes $\sum_{u \in S} w(u)$; such as set is known as a *maximum independent set* (MIS).

Analysis. First, we establish some notation. We turn T into a rooted tree by arbitrarily selecting some vertex r as the root. For any vertex u, let $T(u)$ denote the subtree rooted at u (so $T = T(r)$), and let $C(u)$ denote the set of children of u. Also let $\mathsf{OPT}_{\mathsf{in}}[u], \mathsf{OPT}_{\mathsf{out}}[u]$ denote the MIS of $T(u)$ that includes/excludes u, respectively.

If u is a leaf, then $T(u)$ contains u and no edges, so $\mathsf{OPT}_{\mathsf{in}}[u] = w(u)$ and $\mathsf{OPT}_{\mathsf{out}}[u] = 0$. Now suppose u is not a leaf. The MIS corresponding to $\mathsf{OPT}_{\mathsf{in}}[u]$ must include u, so it cannot include any vertices in $C(u)$. This means

$$\mathsf{OPT}_{\mathsf{in}}[u] = w(u) + \sum_{v \in C(u)} \mathsf{OPT}_{\mathsf{out}}[v]. \tag{4}$$

On the other hand, the MIS corresponding to $\mathsf{OPT}_{\mathsf{out}}[u]$ cannot include u, but this means it is free to include/exclude every vertex in $C(u)$, which implies

$$\mathsf{OPT}_{\mathsf{out}}[u] = \sum_{v \in C(u)} \max(\mathsf{OPT}_{\mathsf{in}}[v], \mathsf{OPT}_{\mathsf{out}}[v]). \tag{5}$$

In the end, the maximum independent set of $T = T(r)$ either contains r or it doesn't, so we return $\max(\mathsf{OPT}_{\mathsf{in}}[r], \mathsf{OPT}_{\mathsf{out}}[r])$.

Algorithm. The algorithm maintains two arrays $d_{\mathsf{in}}[1, \ldots, n]$ and $d_{\mathsf{out}}[1, \ldots, n]$, calculated according to Eq. (4) and Eq. (5), respectively. When calculating these two values at a vertex u, we need to ensure that we have already calculated both values for every child of u. One way to do this is to run a DFS (see Sec. 1.2) starting at the root r, and process V in increasing post values. For clarity of presentation, we assume that the vertices are already ordered in this way (or some other that allows us to follow the recurrences).

Algorithm 7 MIS on a Tree

```
Initialize d_in, d_out as two arrays of length n containing all 0's.
for each u ∈ V do
    Let C(u) denote the set of children of u.
    if C(u) is empty then                                   ▷ u is a leaf
        Set d_in[u] = w(u) and d_out[u] = 0.
    else                                                    ▷ u has at least one child
        Set d_in[u] = w(u) + Σ_{v∈C(u)} d_out[v].
        Set d_out[u] = Σ_{v∈C(u)} max(d_in[v], d_out[v]).
return max(d_in[r], d_out[r])
```

In Algorithm 7, the time it takes to process each vertex u is proportional to the number of children of u. Thus, the overall running time is proportional to

$$\sum_{u \in V} |C(u)| = n - 1 = O(n).$$

Note that the sum holds because every vertex v, except for the root, has exactly one parent, so v appears in exactly one set of children. To recover the MIS of T itself, we can maintain

pointers in Algorithm 7. If $d_{\mathsf{in}}[r] > d_{\mathsf{out}}[r]$, then we include r in the solution, exclude all of its children, and recurse on its grandchildren. Otherwise, we exclude r from the solution and recurse on its children.

3.5 Exercises

1. Consider the knapsack problem from Sec. 3.2. Show that the following two greedy algorithms do not always return an optimal solution: (1) sort the items by non-increasing v_i and then iterate through the list, adding every item that fits (2) sort the items by non-increasing v_i/w_i (i.e., "value per weight") and then iterate through the list, adding every item that fits.

2. Suppose, in the knapsack problem, there is an unlimited quantity of every item. Give an $O(nB)$-time algorithm for this problem.

3. Let $A[1, \ldots, n]$ be an array of integers. Give an $O(n)$-time algorithm that finds a contiguous subarray whose sum is maximized.

4. Let $A[1, \ldots, n]$ be an array where $A[i]$ denotes the price of a stock on day i. A *trade* consists of buying the stock on some day i and selling it on some day $j > i$ for a profit of $A[j] - A[i]$. Give an $O(n)$-time algorithm that finds the trade that maximizes profit.

5. Consider the same setting as the previous exercise, but now you are allowed to make at most k trades, where $k < n/2$. Give an $O(nk)$-time algorithm for this problem. (Hint: First design an $O(n^2k)$-time algorithm, then reduce its running time.) (Bonus: what happens when $k \geq n/2$?)

6. Consider the same setup as Sec. 2.1, but now each interval i has weight $w(i) > 0$. Assume that the intervals are sorted such that $t(1) \leq t(2) \leq \cdots \leq t(n)$. You may also assume that you are given an array p where $p[j]$ is that largest i such that $t(i) < s(j)$ ($p[j] = 0$ if no such i exists.) Give an $O(n)$-time algorithm that finds a subset of compatible events whose total weight is maximized.

7. Suppose you are given a function f whose input can be any string and output is an integer. You are also given a string A of length n. Your goal is to partition A into substrings $(s_1, s_2, \ldots, s_k)$ (where $1 \leq k \leq n$) such that, $\sum_{i=1}^{k} f(s_i)$ is maximized. (You get to choose the value of k.) Give an $O(n^2)$-time algorithm for this problem.

8. Consider the same setup as Sec. 3.3, but now we want to find the longest common subsequence (LCS) of A and B. (Recall that a subsequence of a string is not necessarily contiguous.) Give an $O(mn)$-time algorithm for this problem.

9. A string is *palindromic* if it is equal to its reverse (e.g., "abcba"). Give an $O(n^2)$-time algorithm that finds the longest palindromic subsequence of a string of length n. Additionally, give an $O(n^2)$-time algorithm that finds the longest palindromic substring (i.e., contiguous subsequence), also of a string of length n.

10. Suppose we are given a rod of length n, and we can cut it into pieces of integer length. For each $i \in \{1, \ldots, n\}$, we are told the profit $p_i > 0$ we would earn from selling a piece of length i. Give an $O(n^2)$-time algorithm that returns the lengths we should cut to maximize our total profit.

11. Suppose there are an unlimited number of coins in integer denominations $x_1, x_2, \ldots, x_n$. We are given a value v, and our goal is to select a minimum subset of coins such that their total value is v. Give an $O(nv)$-time algorithm for this problem. (Recall from Sec. 2.5 Exercise 6 that the greedy algorithm is not always optimal.)

12. Give an $O(n)$-time algorithm for the MIS on Trees problem (Sec. 3.4) that only uses one DP table (rather than two).

13. A *matching* in a graph is a subset of edges such that no two edges share an endpoint. Let $T = (V, E)$ be a tree where each edge e has weight $w(e) > 0$. Give an $O(n)$-time algorithm that returns a maximum matching in T. (Recall that in Sec. 2.5 Exercise 8, we solved the unweighted version of this problem.)

14. Let $T = (V, E)$ be a tree where each edge e has weight $w(e) \geq 0$, and let $k \geq 2$ be an integer. Give an $O(nk^2)$-time algorithm that finds a connected subgraph of T containing at lesat k edges whose total weight is minimized. (Hint: First solve the problem when T is a binary tree.)