4 Dynamic Programming

In this chapter, we study an algorithmic technique known as dynamic programming (DP) that often works when greedy doesn't. The idea is to solve a sequence of increasingly larger subproblems by using solutions to smaller, overlapping subproblems. Technically, the algorithms in this chapter only compute the *value* of an optimal solution (rather than the optimal solution itself) but as we'll see, recovering the optimal solution itself is usually a straightforward extension.[1]

Furthermore, instead of the usual "theorem, proof" analysis, we'll *start* by analyzing a recurrence that drives the entire algorithm. If we can convince ourselves that the recurrence is correct, then it's usually relatively straightforward to see that the algorithm is correct as well. Analyzing the running time of a DP algorithm is usually not super difficult either.

4.1 Sum of Array

Problem statement. Let A be an array of n integers. Our goal is to find the sum of elements in A.

Analysis. For any $i \in \{1, \ldots, n\}$, consider the subproblem of finding the sum of elements in $A[1\,{:}\,i]$. We'll store the solution to this subproblem in an array $\mathsf{OPT}[i]$; notice that $\mathsf{OPT}[1] = A[1]$ and for $i \geq 2$, $\mathsf{OPT}[i]$ satisfies the following recurrence:

$$\mathsf{OPT}[i] = \mathsf{OPT}[i-1] + A[i].$$

Our algorithm will use this recurrence to compute $\mathsf{OPT}[1]$ through $\mathsf{OPT}[n]$ and return $\mathsf{OPT}[n]$ at the end.

Algorithm. Populate an array $d[1, \ldots, n]$, according to the recurrence for OPT above. That is, initialize $d[1] = A[1]$ and for $i \geq 2$, set $d[i] = d[i-1] + A[i]$. At the end, return $d[n]$. Computing each $d[i]$ takes $O(1)$ time, and since d has length n, the total running time is $O(n)$.

Algorithm 10 Sum of Array

Input: Array A of n integers.

```
d ← A
for i ← 2, ..., n do
    d[i] ← d[i − 1] + A[i]
return d[n]
```

4.2 Longest Increasing Subsequence

Problem statement. Let A be an array of n 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 $[1]$).

[1] It's the same process as using BFS to find shortest paths rather than just distances.

Analysis. For any index i, let $\mathsf{OPT}[i]$ denote the length of the longest increasing subsequence of A that must end with $A[i]$, so $\mathsf{OPT}[1] = 1$. Intuitively, to calculate $\mathsf{OPT}[i]$ for $i \geq 2$, we should find the "best" entry in A to "come from" before jumping to $A[i]$. We're allowed to jump from $A[j]$ to $A[i]$ if $j < i$ and $A[j] < A[i]$, and we want the subsequence ending with $A[j]$ to be as long as possible. So more formally, for any $i \geq 2$, $\mathsf{OPT}[i]$ satisfies the following recurrence:

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

(If the max is taken over the empty set, that means there are no numbers in $A[1:i-1]$ smaller than $A[i]$, so $\mathsf{OPT}[i] = 1$.) We don't know where the optimal solution OPT ends, but it must end somewhere, and the recurrence allows us to calculate the best way to end at every position. So $\mathsf{OPT} = \max_i \mathsf{OPT}[i]$.

Algorithm. We maintain an array $d[1, \ldots, n]$, and compute $d[i]$ according to Eq. (1). When computing $d[i]$ we scan $A[1:i-1]$, so the overall running time is $O(n^2)$. Once we have $d[i] = \mathsf{OPT}[i]$ for all i, we return $\max(d)$.

Algorithm 11 Longest Increasing Subsequence

Input: Array A of n integers.

```
d[1, ..., n] ← [1, ..., 1]
for i ← 2, ..., n do
    for j ← 1, ..., i − 1 do
        if A[j] < A[i] and d[i] < 1 + d[j] then
            d[i] ← 1 + d[j]                    ▷ append A[i] to the LIS ending at j
return max(d)
```

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.

4.3 Knapsack

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.[2]

Analysis. For any $i \in \{1, \ldots, n\}$ and $j \in \{0, 1, \ldots, B\}$, 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 the subset of items S_{ij} corresponding to $\mathsf{OPT}[i][j]$ either contains item i, or it doesn't (depending on which decision leads to a more profitable

[2]Recall that we solved the fractional version of this problem in Sec. 3.2.

outcome). If $w_i > j$, then S_{ij} must ignore i, so $\mathsf{OPT}[i][j] = \mathsf{OPT}[i-1][j]$. Otherwise, $\mathsf{OPT}[i][j]$ is either $v_i + \mathsf{OPT}[i-1][j-w_i]$ (meaning S_{ij} includes i) or $\mathsf{OPT}[i-1][j]$ (meaning S_{ij} ignores i). This yields the following recurrence:

$$\mathsf{OPT}[i][j] = \begin{cases} \mathsf{OPT}[i-1][j] & \text{if } w_i > j \\ \max(v_i + \mathsf{OPT}[i-1][j-w_i], \mathsf{OPT}[i-1][j]) & \text{otherwise.} \end{cases} \tag{2}$$

The value of the optimal solution is $\mathsf{OPT}[n][B]$.

Algorithm. The algorithm fills out a 2-dimensional array $d[1, \ldots, n][0, \ldots, B]$ according to Eq. (2). Notice that the values in each row depend on those in the previous row. We return $d[n][B]$ as the value of an optimal solution. The array has $n(B+1)$ entries, and computing each one takes $O(1)$ time, so the total running time is $O(nB)$.

Algorithm 12 Knapsack

Input: n items with weights w and values v, capacity B

```
d[1,...,n][0,...,B] ← [0,...,0][0,...,0]
for j ← 1,...,B do                                  ▷ populate the first row
    if w_1 ≤ j then d[1][j] ← v_1
for i ← 2,...,n do
    for j ← 1,...,B do
        if w_i > j then
            d[i][j] ← d[i-1][j]                     ▷ we must ignore i
        else
            d[i][j] ← max(v_i + d[i-1][j-w_i], d[i-1][j])   ▷ we can include i
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.

Remark. Despite its polynomial-like running time of $O(nB)$, Algorithm 12 does *not* run in polynomial time. This is because the input length of the number B is not B, but rather $\log B$. For example, specifying the number 512 only requires 9 bits, not 512.

4.4 Edit Distance

Problem statement. Let $A[1, \ldots, m]$ and $B[1, \ldots, n]$ be two strings. There are three valid operations we can perform on A: insert a character, delete a character, and replace one character with another. (A replacement is equivalent to an insertion followed by a deletion, but it only costs a single operation.) Our goal is to convert A into B using a minimum number of operations as possible (known as the *edit distance*).

Analysis. Let $\mathsf{OPT}[i][j]$ denote the edit distance between $A[1,\ldots,i]$ and $B[1,\ldots,j]$. Notice that $\mathsf{OPT}[i][j]$ corresponds to a procedure that edits $A[1,\ldots,i]$ 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 turn $A[1,\ldots,i]$ into $B[1,\ldots,j-1]$.
2. Delete $A[i]$ for a cost of 1; then turn $A[1,\ldots,i-1]$ into $B[1,\ldots,j]$.
3. Replace $A[i]$ with $B[j]$ for a cost of 0 (if $A[i] = B[j]$) or 1 (otherwise); then turn $A[1,\ldots,i-1]$ into $B[1,\ldots,j-1]$.

Thus, $\mathsf{OPT}[i][j]$ satisfies 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). The overall edit distance is $\mathsf{OPT}[m][n]$.

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 pointers through the table, as we've already described multiple times.

4.5 Independent Set in Trees

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

Analysis. Let's first establish some notation. We can 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 $\mathsf{OPT}_{\mathsf{in}}[u]$ includes u, so it cannot include any $v \in C(u)$, but it can include descendants of these v. 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 $\mathsf{OPT}_{\mathsf{out}}[u]$ excludes u, which means it is free to either include or exclude each $v \in C(u)$, 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 $d_{\mathsf{in}}[u]$ and $d_{\mathsf{out}}[u]$, we need to ensure that we have already calculated $d_{\mathsf{in}}[v]$ and $d_{\mathsf{out}}[v]$ for every $v \in C(u)$. One way to do this is to run a DFS (see Sec. 2.2) 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 way that allows us to follow the recurrences).

Algorithm 13 MIS in a Tree

Input: Tree $T = (V, E)$, positive vertex weights w

$d_{\mathsf{in}} \leftarrow [0 \ldots, 0], d_{\mathsf{out}} \leftarrow [0, \ldots, 0]$
for each $u \in V$ **do**
  if $C(u)$ is empty **then** ▷ u is a leaf
    $d_{\mathsf{in}}[u] \leftarrow w(u)$, $d_{\mathsf{out}}[u] \leftarrow 0$
  else ▷ u has at least one child
    $d_{\mathsf{in}}[u] \leftarrow w(u) + \sum_{v \in C(u)} d_{\mathsf{out}}[v]$
    $d_{\mathsf{out}}[u] \leftarrow \sum_{v \in C(u)} \max(d_{\mathsf{in}}[v], d_{\mathsf{out}}[v])$
return $\max(d_{\mathsf{in}}[r], d_{\mathsf{out}}[r])$

In Algorithm 13, 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 13 and follow them using DFS. If $d_{\mathsf{in}}[r] > d_{\mathsf{out}}[r]$, then we include r in the solution, exclude all of its children, and move on to its grandchildren. Otherwise, we exclude r from the solution and check its children.

4.6 Exercises

4.1. Prove or disprove each of the following:

(a) The values in any LIS DP table are non-decreasing.

(b) The values in any row of a Knapsack DP table are non-decreasing.

(c) The values in any column of a Knapsack DP table are non-decreasing (from top to bottom).

(d) For any instance of the Knapsack problem, the value of the optimal fractional solution is at least the value of the optimal integral solution.

(e) The value in the last row and last column of an Edit Distance DP table is always be the largest in the table.

4.2. Run the Edit Distance DP on the following: $A = [a, b, b, c]$, $B = [b, b, a, b, c]$.

4.3. Consider the Knapsack problem. 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.

4.4. Give an $O(nB)$-time algorithm for the Knapsack problem, assuming that there is an unlimited supply of every item.

4.5. Prove or disprove: It is possible that one row of an Edit Distance DP table is $[2, 1, 2, 2]$.

4.6. 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.

4.7. 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 a trade that maximizes profit.

4.8. 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 $k \geq n/2$?)

4.9. Consider the same setup as Sec. 3.1, 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.

4.10. Suppose we're given a function f whose input can be any string and output is a positive number. We're also given a string A of length n. Our 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.

(We get to choose the value of k.) Give an $O(n^2)$-time algorithm for this problem, assuming that calling f always takes $O(1)$ time.

4.11. Consider the same setup as Sec. 4.4, 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.

4.12. 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.

4.13. 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 given 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.

4.14. 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 that the greedy algorithm is not always optimal.)

4.15. Give an $O(n)$-time algorithm for the MIS on Trees problem (Sec. 4.5) that only uses one DP table (rather than two).

4.16. A subset F of edges in a graph is called a *matching* if no two edges in F 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.

4.17. 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.)