1 Dynamic programming

Dynamic programming is a general strategy for solving problems, much like divide and conquer is a general strategy. And, like divide and conquer, dynamic programming[1] builds up a final solution by combining elements of intermediate solutions.

Recall that divide and conquer (i) partitions a problem into subproblems (with the same properties as the original problem), (ii) recursively solves them, and then (iii) combines their solutions to form a solution of the original problem.

In contrast, a dynamic programming solution (i) defines the value of an optimal solution in terms of overlapping subproblems (with the same properties as the original problem; this is the *optimal substructure* property again), (ii) computes the optimal value by recursively solving its subproblems while (iii) storing results of solved subproblems for later use, and finally (iv) reconstructing the optimal solution from the stored information.

Not every problem can be solved using dynamic programming. For instance, sorting a list of numbers cannot be expressed in terms of overlapping subproblems (i.e., sorting numbers does not have optimal substructure), since every subarray could contain different values in different orders. But, many problems do have overlapping subproblems, and in a dynamic programming approach, once we have solved a subproblem once, we can store or *memoize* its result for future use. In this way, as the algorithm progresses, it can simply look up the answer to a previously solved subproblem rather than resolving it from scratch, and this can lead to dramatic speedups in running time over "brute force" approaches that repeatedly resolve the same subproblems.

1.1 Counting paths in a DAG

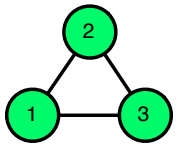
As a first illustration of the dynamic programming approach, we will consider counting the number of paths in a directed acyclic graph (also called a "DAG"). We'll spend much more time later in the class on graph algorithms, and so we'll defer most of the terminology and concepts until later. Here's what we need to understand how to use dynamic programming to count the number of paths between some pair of nodes i and j in a DAG.

1.1.1 Directed acyclic graphs and paths

A *graph* G is defined as a pair of sets $G = (V, E)$, where V is a set of *vertices* or *nodes* and $E : V \times V$ is a set of pairwise *edges* or *arcs*. Often, we say that the number of nodes is $n = |V|$ and the number

[1] The name "dynamic programming" comes from a time when "programming" meant tabulation rather than writing computer code. A more modern and interpretable name would be something like "dynamic tabulation," but it's hard to change a name this late in the game.

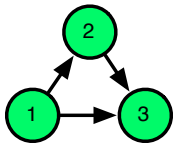
of edges is $m = |E|$.



An undirected graph representing a triangle: $V = \{1, 2, 3\}$ and $E = \{(1, 2), (1, 3), (2, 3)\}$

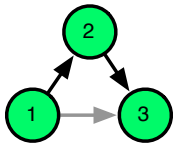
In an *undirected* graph, edges are *undirected*, meaning that the edge (i, j) and the edge (j, i) both represent the same connection between nodes i and j.

In a *directed* graph, the edge set E may contain both (i, j) and (j, i), which are sometimes denoted $(i \to j)$ and $(j \to i)$ to illustrate their directionality.



A DAG representing a triangle: $V = \{1, 2, 3\}$ and $E = \{(1 \to 2), (1 \to 3), (2 \to 3)\}$

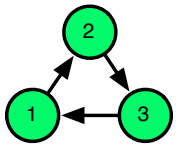
A *path* is a sequence of edges $\sigma = [\sigma_1, \sigma_2, \ldots, \sigma_\ell]$ such that $\forall_k \sigma_k \in E$ and each consecutive pair of edges has the form $\sigma_k, \sigma_{k+1} = (i, j), (a, b)$ where $j = a$. That is, the endpoint of σ_a is the origin of σ_{a+1}.



A path from node 1 to node 3 on the DAG triangle: $\sigma = [(1 \to 2), (2 \to 3)]$

How many paths are there from node 1 to node 3? There are two: $[(1 \to 2), (2 \to 3)]$ and $[(1, 3)]$. Notably, to be a well-defined or non-trivial path, we require that $|\sigma| > 0$, i.e., that the path includes at least one edge.

A *cycle* is a path σ such that there exists some $\sigma_x, \sigma_y = (i, j), (a, b)$ where $i = b$. That is, the origin of σ_a is the endpoint of σ_b.



A cycle from node 1 to node 1 on a triangle: $\sigma = [(1 \to 2), (2 \to 3), (3 \to 1)]$

A *directed acyclic graph* (DAG) is a direct graph with no cycles.

1.1.2 Counting paths

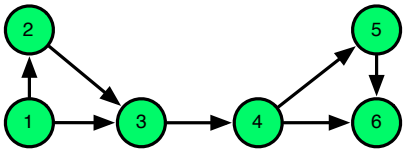
Given a DAG $G = (V, E)$, and a pair of nodes i, j, how many paths are there from i to j?

We can solve this problem using dynamic programming. Suppose we knew the number of paths from i to j was X, and let the set $s(j)$ denote the set of nodes x such that $(x \to j) \in E$, i.e., x is a neighbor of j. Each of the X paths to j must pass through some particular neighbor of j, i.e., each path pass though a node $x \in s(j)$. Thus, the total number of paths X must equal the number of paths from i to $x_1 \in s(j)$ plus the number of paths from i to $x_2 \in s(j)$, etc. That is, X is the sum of the number of paths that start at i and terminate at some node x that neighbors j.

Mathematically, we can say that $X_{i,j}$ counts the number of paths from i to j, and, if we let ℓ index the nodes $s(j)$ that point to j, we can define $X_{i,j}$ recursively:

$$X_{i,j} = \sum_{\ell \in s(j)} X_{i,\ell} \ . \tag{1}$$

Now, we can recursively compute the number of paths $X_{i,j}$, and store the intermediate values for later use in a table to make things faster. Consider the following DAG as a concrete example, where we want to count the number of paths from node 1 to node 6:



Count the paths from node 1 to node 6

Memoizing path counts: Because counting paths on a DAG has optimal substructure, we can memoize the results of subproblems and use them to compute the solution to larger problems. Memoization is thus a way to trade space—the space required to store the subproblem solutions—for time, which we save by not having to recompute the subproblem solutions each time we break a problem into its subproblems. Crucially, memoization only helps if specific subproblems reoccur across the recursion tree. If they do not, then we save nothing by remembering their solutions.

For path counting, memoization requires only a simple 1-dimensional table (a.k.a., an array) of the number of paths $X_{1,i}$. We will use the contents of this array to build up the optimal value of $X_{1,6}$. The Eq. (1) gives the optimal value for any $X_{1,i}$ in terms of optimal answers to subproblems, which are stored in other elements in the table. As the algorithm runs, the contents of the array are filled in, as follows:

node i	6	5	4	3	2	1
$X_{1,i}$	$X_{1,5} + X_{1,4}$	$X_{1,4}$	$X_{1,3}$	$X_{1,2} + 1$	1	0
$X_{1,i}$	$X_{1,5} + X_{1,4}$	$X_{1,4}$	$X_{1,3}$	2	1	0
$X_{1,i}$	$X_{1,5} + X_{1,4}$	$X_{1,4}$	2	2	1	0
$X_{1,i}$	$X_{1,5} + X_{1,4}$	2	2	2	1	0
$X_{1,i}$	4	2	2	2	1	0

The base case occurs for all nodes that are directly connected to node 1. Hence, the number of paths from 1 to 6 in this simple example is 4.

1.2 The 0-1 Knapsack problem

Another problem that is amenable to the dynamic programming approach is the 0-1 Knapsack problem. In this problem, you are faced with a set of n *indivisible* items S, where each item $i \in S$ has both a *value* v_i and a weight w_i. Your task is to select a subset of items $T \subseteq S$ such that the total value of the items $\sum_{i \in T} v_i$ is maximized and the total weight of the items does not exceed a threshold W, i.e., $\sum_{i \in T} w_i \leq W$.

A *brute force* approach to solving this problem would consider all 2^n possible choices, in which for each of the n items, we either try to include it or exclude it, and then evaluate whether this particular set of choices satisfies our weight limit W and maximizes the total value. This approach is infeasible for any reasonable value of n, and besides, there is a much more efficient approach.[2]

A simple example: Suppose we are given a capacity of $W = 20$, and five items with the following weights and values: $\{(2, 3), (3, 4), (4, 5), (5, 8), (9, 10)\}$, which we can arrange in a table like this,

[2] If items are infinitely divisible, as in piles of gold or silver dust, then a greedy algorithm is optimal: take as much of the most valuable item as will fit in the bag, and then recurse with the next most valuable item.

where the two right-most columns give two different greedy solutions in binary (a 1 indicates the item is taken, and a 0 that the item is not):

item i	weight w_i	value v_i	greedy1	greedy2	optimal
1	3	4	0	1	1
2	4	7	0	1	1
3	5	5	0	0	1
4	8	8	1	0	1
5	10	11	1	1	0

A first greedy approach ("greedy1" above) would be to first sort items in decreasing order of their value v_i, iterate down the list, and add each item so long as doing so does not cause our total weight to exceed our capacity. This approach takes items 5 and 4 (in that order), for total value 19 and total weight of 18.

A second greedy approach ("greedy2" above) would be first sort items in decreasing order of value-to-weight ratio v_i/w_i, and then again proceed down the list, filling up the bag until no remaining item is small enough to fit. This approach takes items 2, 1 and 5 (in that order), for total value 22 and total weight of 17, a better solution.

But the optimal choice ("optimal" above) is to take items 1, 2, 3 and 4, for total value 24 and total weight of 20. To be a correct solution for the 0-1 Knapsack problem, an algorithm must succeed on all inputs. The example input above is a proof by counter-example showing that both greedy approaches are not correct. Dynamic programming, however, is a correct algorithm for 0-1 Knapsack.

1.3 The algorithm

To develop a dynamic programming solution, we need to understand how to exploit the substructure to build up an optimal solution. That is, how can we break a current problem down into simple decisions that combine the optimal solutions to subproblems to produce an optimal solution for the current problem? Or, suppose we have an optimal solution to a 0-1 Knapsack problem with n items and W capacity. How could we extend this solution to be an optimal solution for a problem with $n+1$ items or with $W+1$ capacity?

This question presents a key insight: every choice of $k \in [0, n]$ and capacity $w \in [0, W]$ is a subproblem for parameters n and W. The parameter k lets us vary how many fewer items we consider, and the parameter w lets us vary how much smaller capacity bag to consider.[3] (Crucially, both param-

[3]The values of $k = 0$ and $w = 0$ represent base cases. If there are no items, then the optimal value, regardless of w, is 0. Similarly, if the bag has no capacity, the optimal value, regardless of items, is 0.

eters are necessary. At home exercise: prove that an approach using only k will fail on some inputs.)

Suppose we have already computed the optimal solution for some particular value of $k-1$ and for all values $0 \leq w \leq W$. That is, for every choice of a smaller bag than W, we know the optimal value that can be obtained. Now consider adding one more item, with weight w_k and value v_k, meaning we have k items total to consider. There are three possibilities:

- (No room) If $w_k > w$, then our bag is too small to include the kth item, regardless of what subproblem we might build off of.
- (Take it) If our bag is large enough, i.e., $w_k \leq w$, the optimal value is this item's value v_k plus the optimal value for the subproblem on k items and weight $w - w_k$, i.e., a bag with just enough extra capacity to hold item k.
- (Leave it) If our bag is large enough, i.e., $w_k \leq w$, the optimal solution on $k-1$ items for capacity w.

We can take these three possibilities and write them down as a simple recursive expression that gives the optimal solution for k items and capacity w, which we denote $B(k, w)$, in terms of optimal solutions to smaller problems:

$$B(k,w) = \begin{cases} B(k-1,w) & \text{if } w_k > w \\ \max(\ B(k-1,w)\ ,\ B(k-1,w-w_k) + v_k\ ) & \text{otherwise} \end{cases}$$

The upper branch governs the "No room" condition, while the lower branch covers "Take it" and "Leave it" conditions. Crucially, because we want the optimal value for $B(k, w)$, we take the larger value of the two possibilities where we could take the item.

As with the path counting problem, the recursive formula immediately implies a memoization scheme by which to store the optimal solutions for the (k, w) subproblems. Here, we use a table B with n rows and $W + 1$ columns. Because our formula for $B(k, w)$ only ever refers to subproblems on the $k-1$ row, we may fill in this table one row at a time. Here is pseudocode:

```
for w = 0 to W { B[0,w] = 0 }  // base case: no items
for k = 1 to n { B[k,0] = 0 }  // base case: no capacity
for k = 1 to n {
   for w = 0 to W {
      if w[k] <= w {                        // item could fit
         take   = b[k] + B[ k-1, w-w[k] ]  // optimal if we take
         leave  = B[ k-1, w ]              // optimal if we leave
         B[k,w] = max( take , leave )      // optimal either way
      else { B[k,w] = B[ k-1, w ] }        // optimal if item couldn't fit
}}}
```

1.3.1 Running time

The running time of the 0-1 Knapsack problem is straightforward. Allocating the B matrix requires time $\Theta(n\,W)$, and the two initialization loops take $\Theta(W)$ and $\Theta(n)$ time. Every element in the table is filled by the double loop. The inner part of the loop takes $\Theta(1)$ time, because it consists only of atomic operations (maximum takes constant time—do you see why?). Hence, the double loop takes $\Theta(n\,W)$ time, and the algorithm as a whole takes $\Theta(n\,W)$ time.[4]

1.3.2 Correctness

The correctness of the 0-1 Knapsack algorithm follows a proof by strong induction on the contents of B. (Left as an exercise. Also left as an exercise: running the algorithm on the small example given above.)

1.4 Items from the table

When the Knapsack algorithm completes, the optimal value is stored in the $B(n, W)$ entry of the table. This does not tell us which items were chosen, however. But this information is contained *inside* the table B, and we can recover it by using a "trace back" algorithm, which reconstructs a sequence of take it / leave it decisions that are consistent with the optimal value.[5]

The idea is straightforward. For a particular choice of k and w, there were only two possibilities by which we could have obtained the value $B(k, w)$. First, if $B(k, w) = B(k-1, w-w_k) + v_k$, then item k is in the optimal set and the next subproblem to consider is $k-1$ and $w - w_k$; otherwise, $B(k, w) = B(k-1, w)$, implying that k is not in the optimal set, and the next subproblem to consider is $k-1$ and w. Or, in pseudocode:

```
for k = 1 to n { solution[k] = 0 }        // binary array: is in optimal set?
w = W                                     // starting capacity
for k=n down to 1 {
         taken = b[k] + B[ k-1, w-w[k] ]  // optimal value if we took k
         if B[k,w] == taken {             // did we take k?
            solution[k] = 1               // add k to optimal set
            w = w-w[k]                    // update capacity size
}}
```

The result is a binary array `solution` of length n, which contains a 1 anywhere there was an item that was "taken" in order to construct a solution with value $B(n, W)$. This algorithm runs in $\Theta(n)$ time, because it only examines one element per row of the table B.

[4] We call this algorithm a *pseudo-polynomial* time algorithm because W is not a function of n, but rather is part of the input itself. If we increase W, we increase the running time, even though the number of items n, which is the "length" of the input, has not changed.

[5] There could be multiple sets of choices of items that produce the same optimal value, and we only need one.

2 Other common dynamic programming problems

There are many other commonly encountered dynamic programming problems.

In the Longest Common Subsequence (LCS) problem, we are given two strings x and y, each of which is a sequence of symbols drawn from some alphabet Σ. Given x and y, LCS asks what is the length of the longest common subsequence in both strings. This problem is a special case of the Optimal String Alignment problem.

The LaTeX typesetting system uses dynamic programming to layout text and equations on a page. In this problem, a string x is given as input, and the algorithm must choose where to insert line breaks and how much whitespace to insert between words. The optimal solution for the first ℓ lines is simply the optimal solution for the first $\ell - 1$ lines plus the optimal solution for the ℓth line, and hence this approach to typesetting also has optimal substructure.

3 On your own

1. Read Chapter 15