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.

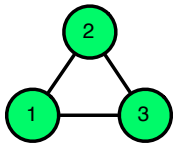
1.1.1 Directed acyclic graphs and paths

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

Acknowledgement: Adopted from Prof. Aaron Clauset's lecture notes.

[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.

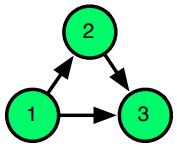
of edges is $m = |E|$.



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

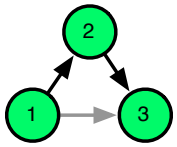
In an *undirected* graph, edges are *undirected*, meaning that the edge (i, j) and the edge (j, i) both represent the same connection between nodes i and j.

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 $E = \{(1 \to 2), (1 \to 3), (2 \to 3)\}$

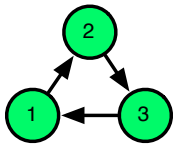
A *path* is a sequence of edges $\sigma = [\sigma_1, \sigma_2, \ldots, \sigma_\ell]$ such that $\forall_k \sigma_k \in E$ and each consecutive pair of edges has the form $\sigma_k, \sigma_{k+1} = (i, j), (a, b)$ where $j = a$. That is, the endpoint of σ_a is the origin of σ_{a+1}.



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

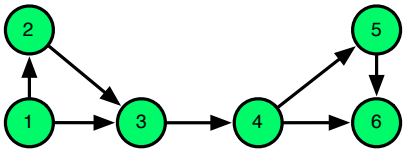
Given a DAG $G = (V, E)$, and a pair of nodes i, j, how many paths are there from i to j?

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 benefits: $\{(2, 3), (3, 4), (4, 5), (5, 8), (9, 10)\}$, 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	benefit b_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 benefit b_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 benefit 19 and total weight of 18.

A second greedy approach ("greedy2" above) would be first sort items in decreasing order of benefit-to-weight ratio b_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 benefit 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 benefit 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 benefit b_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 benefit b_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:

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

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 $B(k, w) = B(k-1, w-w_k) + b_k$, 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 Aligning sequences with dynamic programming

Suppose we can represent some user-input as a sequence of symbols x, where $x_i \in \Sigma$ with Σ denoting the input alphabet. Our task is to identify a best match of that input to a library of sequences $Y = \{y\}$. However, users are prone to input errors, which means that x may not be an exact match to any of those in our reference set. Thus, in order to correctly identify the user's input, we must first identify the $y \in Y$ is the best match for the input x.

Examples of this type of problem are more common than we might imagine. For instance, search engines often try to correct misspelled words in the query string; in speech recognition, the string is a sequence of values representing the recorded sound wave, which must be matched to a known word; and in forensic analysis, the input string is a DNA sequence, which may differ from those in our reference set by some number of nucleic acid mutations, insertions or deletions.

In each case, we aim to *align* a pair of sequences so that we find the elements in each that correspond exactly to each other, while ignoring the elements between these aligned parts. Here, we will focus on what is called a *global alignment* in which we aim to align the entire two sequences. In order to define an algorithm for finding such an alignment, we must also define a set of *edit operations* $E \in \mathcal{E}$ and a cost for each $c(E) \geq 0$.

The problem of sequence alignment is to *find a minimal-cost set of edit operations that transforms the sequence x into the sequence y.* We will solve this problem using a dynamic programming algorithm.

2.1 Edit operations and costs

Before we can write down an algorithm, we must define the set of edit operations $\mathcal{E}$ we may use. Here, we will utilize three operations beyond the default "no-op" operation, which leaves a letter unchanged.

- *Substitution* (*sub*): replace a letter x_i with some other letter in the alphabet Σ, at the same position as x_i.

  For instance, "so" and "do" are two strings that differ by a single substitution edit, and which are commonly misspelled for each other on a keyboard because s and d are next to each other.

- *Insertion* and *Deletion* (*indel*): insert some letter from the alphabet Σ into x, shifting all subsequent letters one position later in the string; or, delete x_i from x, shifting all subsequent letters one position earlier in the string. Note that an insertion operation into one string is equivalent to a deletion operation in the other string.

  For instance, "grande" and "grand" are two strings that differ by a single *indel* operation.

- *Transposition* (*swap*): take two consecutive letters x_i, x_{i+1} and exchange their positions, and then substitute them into the aligned positions in y.[6]

  For instance, both "their" / "thier" and "teh" / "the" are pairs of strings that differ by a single transposition.

Given these operations, we must now also choose a cost function $c(E)$. There are several choices for this function, but here we choose the "edit distance" function (technically called the Damerau-Levenshtein Distance)[7] which simply counts the number of these operations required to transform x into y.[8] The one wrinkle is that transposition is actually three operations: one *swap*, followed by two *subs*, for a total cost of 3, while any single *sub* or *indel* costs 1.

2.2 An example

To illustrate how to compute the cost of a particular alignment, consider aligning the two strings $x =$ `THEIR` and $y =$ `THERE`.

Alignment 1: Substitute the last two characters, for a total cost of 2 *sub* operations:

```
THEIR
|||ss
THERE
```

Alignment 2: Insert and delete so that the `R` lines up, for a total cost of 2 *indel* operations:

```
THEIR-
|||d|i
THE-RE
```

where "`-`" denotes a "gap" character, implying an insertion on the opposing string.

Alignment 3: At worst, delete the entire first string, and insert the entire second string, for a total cost of 10 *indel* operations:

[6] Generalizations exist that allow letters to be transposed more than one, or to allow longer substrings to be transposed, but these algorithms are more complicated.

[7] Supposedly, these types of "edits" represent a large fraction, possibly 80% or more, of all human misspellings, with the remaining presumably being confusion over which word to use in the first place, e.g., "their" versus "they're".

[8] Other cost structures are certainly possible, depending on the application. For instance, a transposition might be less costly than an insertion, etc. Furthermore, cost may depend on the letters being changed, perhaps reflecting the probability of the error. For instance, adjacent letters on a QWERTY keyboard may have lower costs for substitution or transposition than letters far apart.

```
THEIR-----
dddddiiiii
-----THERE
```

Clearly, the first two alignments are cheaper than the third alignment, and under the edit-distance cost function, either of those would be an acceptable alignment.

2.3 When can we apply dynamic programming?

Recall that in dynamic programming, we will assemble the solution to a larger problem by utilizing the exact solutions to a smaller problem contained within our larger problem. In general, the relationship a problem and its subproblems defines a recursive structure that we can use to build the full solution in a "bottom-up" fashion.[9]

A general requirement for dynamic programming is that there cannot be a cycle among subproblem dependencies, such that solving some problem A requires eventually solving some B that requires solving A. Thus, dynamic programming can be applied only if the space of subproblems can be organized into a directed acyclic graph (a "DAG"), in which each subproblem is a vertex and an arc $i \to j$ represents that solving j requires solving i first.

2.4 Dynamic programming solution

The ordered substructure in sequence alignment comes from the additive cost of making additional edit operations, as we move from left-to-right through the sequences. That is, the cost of aligning two subsequences $x_1x_2 \dots x_i = x_{1\dots i}$ and $y_1y_2 \dots y_j = y_{1\dots j}$ is the cost of the edit operation for x_i and y_j plus the cost of aligning the subproblem that got us to needing to align x_i and y_j.

There are only three ways we could have gotten to needing to align x_i and y_j:

- the last op was *sub*, and we paid the cost of aligning $x_{1\dots i-1}$ and $y_{1\dots j-1}$,
- the last op was *indel*, and we paid the cost of aligning either $x_{1\dots i}$ and $y_{1\dots j-1}$ or aligning $x_{1\dots i-1}$ and $y_{1\dots j}$, or
- the last op was *swap*, and we paid the cost of aligning $x_{1\dots i-2}$ and $y_{1\dots j-2}$.

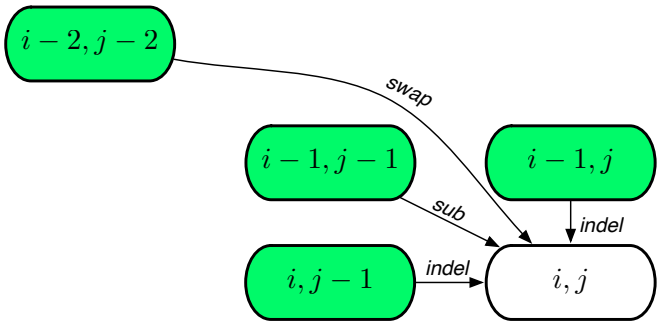
Let $\text{cost}(i, j)$ be the minimum cost of aligning $x_{1\dots i}$ and $y_{1\dots j}$, where we define as a base case $\text{cost}(0, 0) = 0$.

[9] There are additional requirements for dynamic programming to produce a polynomial-time algorithm: the number of subproblems must be polynomial in size and the recursive function must run in polynomial time.

Thus, recursive structure of the subproblems we identified above implies that $\text{cost}(i, j)$ may be computed recursively as

$$\text{cost}(i,j) = \min \begin{cases} \text{cost}(i-2, j-2) + c(swap) \\ \text{cost}(i-1, j-1) + c(sub) \\ \text{cost}(i-1, j) + c(indel) \\ \text{cost}(i, j-1) + c(indel) \end{cases}$$

where we define $c(sub) = 0$ if $x_i = y_j$, i.e., a "no-op." This function is equivalent to this DAG template:



which represents the relationship between subproblems.

By memoizing the solutions (costs) to the subproblems for $0 \leq i \leq n_x$ (length of x) and $0 \leq j \leq n_y$ (length of y), storing them in a 2-dimensional array $S[i, j] = \text{cost}(i, j)$, we can recursively compute the minimum cost of aligning x and y.

2.5 A small and fully worked example

Before tackling a large example, let us exhaustively do a small one. Consider aligning $x =$ `STEP` and $y =$ `APE`.

We begin by writing out the cost matrix[10] S, and filling in the base case for aligning two empty strings, which has $\text{cost}(0, 0) = 0$.

We may now immediately fill in the values for the 0th column and 0th row, which correspond to the cost of aligning an empty string with x (column 0) or with y (row 0). In each of these cases,

[10] For convenience, we will assume this matrix is 0-indexed, meaning that the first element in a row or a column is the 0th element.

the alignment consists of inserting each character in the target string into the empty string, and thus the costs in the 0th row are $S(0, j) = j$ for $1 \leq j \leq n_y$, and the costs in the 0th column are $S(i, 0) = i$ for $1 \leq i \leq n_x$.

x/y	_	A	P	E
_	0			
S				
T				
E				
P				

base case

x/y	_	A	P	E
_	0	1	2	3
S	1			
T	2			
E	3			
P	4			

empty strings aligned

x/y	_	A	P	E
_	0	1	2	3
S	1	1	2	3
T	2	2		
E	3	3		
P	4	4		

first character aligned

At the next step, we set $i = 1$ and $j = 1$ and align $x' =$ S with $y' =$ A. There are three subproblems to consider (the fourth subproblem, corresponding to *swap*, isn't allowed yet):

- (Sub) We previously aligned $\boxed{\_}$ from x with $\boxed{\_}$ from y, for cost $S(0, 0) = 0$.
  Now we substitute $\boxed{\texttt{S}}$ for $\boxed{\texttt{A}}$, which costs $c(sub) = 1$.
  Cost = 1.
- (Delete) We previously aligned $\boxed{\_}$ from x with $\boxed{\texttt{A}}$ from y, for cost $S(0, 1) = 1$.
  Now we delete $\boxed{\texttt{S}}$, which costs $c(indel) = 1$.
  Cost = 2.
- (Insert) We previously aligned $\boxed{\texttt{S}}$ from x with $\boxed{\_}$ from y, for cost $S(1, 0) = 1$.
  Now we insert $\boxed{\texttt{A}}$, which costs $c(indel) = 1$.
  Cost = 2.

The minimum of these choices is uniquely the first one, then thus we record $S(1, 1) = 1$.

Next we consider $i = 1$ and $j = \{2, 3\}$, in which we align S with {AP, APE}. Although we could write down the three subproblems for each of these, we may also simply recognize that S appears in neither of these strings, and thus the minimum cost for each alignment will be the cost of deleting S and inserting $y_{1\ldots j}$ for $j = 2, 3$. Thus, we may record $S(1, j) = j$ for $j = \{2, 3\}$.

The same fact is true for $i = \{2, 3, 4\}$ and $j = 1$, in which we align {ST, STE, STEP} with A. Thus, we may record $S(i, 1) = i$ for $i = \{2, 3, 4\}$. What now remains is to align the remaining cases of substrings. We will treat each of the 6 cases, one at a time.

Set $i, j = 2$ and align `ST` with `AP`. There are four subproblems to consider:

- (Sub) Previously $\boxed{\texttt{S}} \to \boxed{\texttt{A}}$. Now substitute $\boxed{\texttt{T}}$ for $\boxed{\texttt{P}}$ Cost $= S(1,1) + 1 = 2$
- (Delete) Previously $\boxed{\texttt{S}} \to \boxed{\texttt{AP}}$. Now delete $\boxed{\texttt{T}}$. Cost $= S(1,2) + 1 = 3$
- (Insert) Previously $\boxed{\texttt{ST}} \to \boxed{\texttt{A}}$. Now insert $\boxed{\texttt{P}}$. Cost $= S(2,1) + 1 = 3$
- (Swap) Previously $\boxed{\_} \to \boxed{\_}$. Now transpose $\boxed{\texttt{ST}}$ and sub for $\boxed{\texttt{AP}}$ Cost $= S(0,0) + 3 = 3$

Thus, we record $S(2,2) = 2$.

Now setting $i = 2$ and $j = 3$, we align `ST` with `APE`:

- (Sub) Previously $\boxed{\texttt{S}} \to \boxed{\texttt{AP}}$. Now substitute $\boxed{\texttt{T}}$ for $\boxed{\texttt{E}}$. Cost $= S(1,2) + 1 = 3$
- (Delete) Previously $\boxed{\texttt{S}} \to \boxed{\texttt{APE}}$. Now delete $\boxed{\texttt{T}}$. Cost $= S(1,3) + 1 = 4$
- (Insert) Previously $\boxed{\texttt{ST}} \to \boxed{\texttt{AP}}$. Now insert $\boxed{\texttt{E}}$. Cost $= S(2,2) + 1 = 3$
- (Swap) Previously $\boxed{\_} \to \boxed{\texttt{A}}$. Now transpose $\boxed{\texttt{ST}}$ and sub for $\boxed{\texttt{PE}}$. Cost $= S(0,1) + 3 = 4$

Thus, we record $S(2,3) = 3$, which represents the cost of either of these subalignments:

```
S-T      ST-
sis      ssi
APE      APE
```

Now we set $i = 3$ and $j = 2$, in which we align `STE` with `AP`. Again, there are four subproblems to consider:

- (Sub) Previously $\boxed{\texttt{ST}} \to \boxed{\texttt{A}}$. Now substitute $\boxed{\texttt{E}}$ for $\boxed{\texttt{P}}$. Cost $= S(2,1) + 1 = 3$
- (Delete) Previously $\boxed{\texttt{ST}} \to \boxed{\texttt{AP}}$. Now delete $\boxed{\texttt{E}}$. Cost $= S(2,2) + 1 = 3$
- (Insert) Previously $\boxed{\texttt{STE}} \to \boxed{\texttt{A}}$. Now insert $\boxed{\texttt{P}}$. Cost $= S(3,1) + 1 = 4$
- (Swap) Previously $\boxed{\texttt{S}} \to \boxed{\_}$. Now transpose $\boxed{\texttt{TE}}$ and sub for $\boxed{\texttt{AP}}$. Cost $= S(1,0) + 3 = 4$

Thus, we record $S(3,2) = 3$.

Now setting $j = 3$ and aligning `STE` with `APE`, we have:

- (Sub) Previously `ST` → `AP`. Now substitute `E` for `E`. Cost $= S(2,2) + 0 = 2$
- (Delete) Previously `ST` → `APE`. Now delete `E` (from x). Cost $= S(2,3) + 1 = 4$
- (Insert) Previously `STE` → `AP`. Now insert `E` (into y). Cost $= S(3,2) + 1 = 4$
- (Swap) Previously `S` → `A`. Now transpose `TE` and sub for `PE`. Cost $= S(1,1) + 3 = 4$

Thus, we record $S(3,3) = 2$.

Penultimately, we consider $i = 4$ and $j = 2$ and align `STEP` with `AP`:

- (Sub) Previously `STE` → `A`. Now substitute `P` for `P`. Cost $= S(3,1) + 0 = 3$
- (Delete) Previously `STE` → `AP`. Now delete `P` (from x). Cost $= S(3,2) + 1 = 4$
- (Insert) Previously `STEP` → `A`. Now insert `P` (into y). Cost $= S(4,1) + 1 = 5$
- (Swap) Previously `ST` → `_`. Now transpose `EP` and sub for `AP`. Cost $= S(2,0) + 3 = 5$

Thus, we record $S(4,2) = 3$.

And finally, we set $i = 4$ and $j = 3$ and align `STEP` with `APE`:

- (Sub) Previously `STE` → `AP`. Now substitute `P` for `E`. Cost $= S(3,2) + 1 = 4$
- (Delete) Previously `STE` → `APE`. Now delete `P` (from x). Cost $= S(3,3) + 1 = 3$
- (Insert) Previously `STEP` → `AP`. Now insert `E` (into y). Cost $= S(4,2) + 1 = 4$
- (Swap) Previously `ST` → `A`. Now transpose `EP` and sub for `PE`. Cost $= S(2,1) + 1 = 3$

Thus, we record $S(4,3) = 3$, which gives the final minimum cost for aligning `STEP` with `APE`, via any of these alignments:

```
STEP    STEP    STEP
ss|d    dstt    sdtt
APE-    -APE    A-PE
```

Here are the completed cost matrices:

x/y	_	A	P	E
_	0	1	2	3
S	1	1	2	3
T	2	2	2	3
E	3	3		
P	4	4		

align ST with y

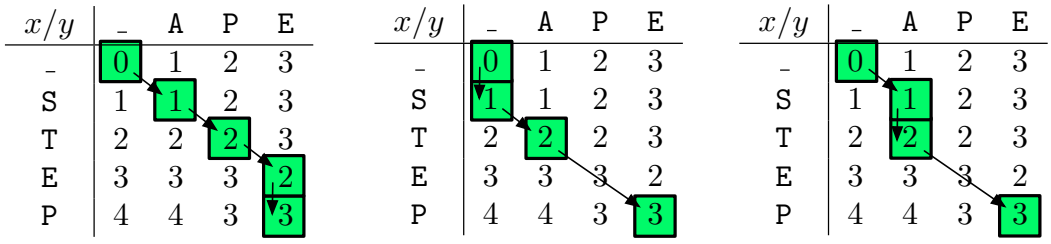
x/y	_	A	P	E
_	0	1	2	3
S	1	1	2	3
T	2	2	2	3
E	3	3	3	2
P	4	4		

align STE with y

x/y	_	A	P	E
_	0	1	2	3
S	1	1	2	3
T	2	2	2	3
E	3	3	3	2
P	4	4	3	3

align STEP with y

To extract the 3 minimum-cost alignments given above, we examine the sequences of choices we made to arrive at $S(4,3) = 3$. Specifically, there are three paths from $S(0,0)$ that all reach $S(4,3)$, and each of these paths corresponds to a minimum-cost alignment. Left- or down- moves represent *indel* operations, single-diagonal moves are a *sub*, and double-diagonal moves are a *swap*.



2.6 A large worked example

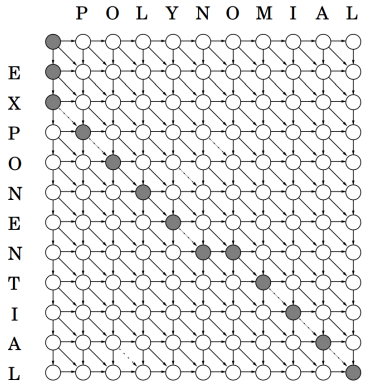
Consider aligning the strings $x =$ EXPONENTIAL and $y =$ POLYNOMIAL.[11] The full matrix S of costs is shown below, which is produced by starting at $i, j = 0$ and applying $\text{cost}(i,j)$ as given above iteratively to each element. (Or, by starting at $i = n_x$ and $j = n_y$ and making the recursive calls.) Let us focus on a small piece of the overall calculation: aligning EXP and POLY. The cost is given by

$$\begin{aligned}\text{cost}(3,4) &= \min\{\text{cost}(1,2)+3, \text{cost}(2,3)+1, \text{cost}(2,4)+1, \text{cost}(3,3)+1\} \\ &= \min\{5,4,5,4\} \\ &= 4\end{aligned}$$

The overall minimum cost of 6 is in the bottom-right corner of S. Note, however, that our cost matrix does not contain corresponding alignment. Given the completed matrix, we may extract

[11] This example is taken from Dasgupta, Papadimitriou and Vazirani's excellent book *Algorithms* (2006).

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6



the corresponding alignment by starting in the bottom-right corner and finding the minimum cost path backwards through the DAG to $S(0,0)$. The right-hand figure above shows this path, whose corresponding alignment is

```
--POLYNOMIAL
ii||ss|ds|||
EXPONEN-TIAL
```

for a cost of 3 *indel*s and 3 *sub*s, or 6 overall.

2.7 Correctness

We now prove that this algorithm is correct, i.e., finds a minimum-cost alignment. As usual with recursive functions, we provide a proof-by-induction, on the cost of aligning the leading substrings of x and y.

Claim: Any alignment of strings x and y that satisfies the cost(i, j) function, is a minimal cost alignment.

Proof: First, we dispense with the base case of aligning two strings of length 0. The cost here must be 0 because there are no letters to align and there can be no edit operations. Thus, cost$(0, 0) = 0$.

Now, assume that we have calculated a minimum cost alignment on $x_{1\ldots k}$ and $y_{1\ldots \ell}$, for $k < i$ and $\ell < j$. There are only four possible previous subalignments to consider, each of which corresponds to the last edit operation used:

- Transpose: First, we swap x_{i-1} and x_i, and we then substitute them for y_{j-1} and y_j respectively. These three edits together cost $c(swap)$, by definition.

The remaining cost is from aligning $x_{1\ldots i-2}$ with $y_{1\ldots j-2}$, whose minimum cost is $\text{cost}(i-2, j-2)$. Therefore, the minimum cost ending with a *swap* is $\text{cost}(i-2, j-2) + c(swap)$.

- Substitute: We substitute the value at x_i for the value at y_j. This costs $c(sub)$ by definition.

  The remaining cost is from aligning $x_{1\ldots i-1}$ with $y_{1\ldots j-1}$, whose minimum cost is $\text{cost}(i-1, j-1)$. Therefore, the minimum cost ending with a *sub* is $\text{cost}(i-1, j-1) + c(sub)$.

- Delete in x and Insert in y: We add a gap character after y_j to match x_i. This costs $c(indel)$ by definition.

  The remaining cost is from aligning $x_{1\ldots i-1}$ with $y_{1\ldots j}$, whose minimum cost is $\text{cost}(i-1, j)$. Therefore, the minimum cost ending with a *sub* is $\text{cost}(i-1, j) + c(indel)$.

- Insert in x and Delete in y: We add a gap character after x_i to match y_j. This costs $c(indel)$ by definition.

  The remaining cost is from aligning $x_{1\ldots i}$ with $y_{1\ldots j-1}$, whose minimum cost is $\text{cost}(i, j-1)$. Therefore, the minimum cost ending with a *sub* is $\text{cost}(i, j-1) + c(indel)$.

Because $\text{cost}(i, j)$ is defined as the minimum cost over the four possibilities, and because these are the only paths to aligning substrings i, j, the recursion relation must give the minimal cost for aligning i, j. □

2.8 Pseudocode and running time

Although a recursive algorithm that carries out the work of filling in the matrix S is easy to define, an iterative algorithm is almost as easy to write down. Much like the iterative algorithm for the 0-1 Knapsack problem, the iterative sequence alignment algorithm begins at the base case and fills in the elements in each column, and then repeats this for each row. Furthermore, without using asymptotically more space than S, we may also construct the alignment itself in parallel with filling in S. The algorithm below is a simple generalization of the one originally given by Needleman and Wunsch in 1970.

```
input: x with length nx and y with length ny
initialize S, of dimensions nx+1 by ny+1
initialize p, of dimensions nx+1 by ny+1

S[0,0] = 0
p[0,0] = NULL
for i = 0 to nx                        // consider all letters of x
   for j = 0 to ny                     //  consider all letters of y
      if i>0 or j>0                    //    skip the base case
         S[i,j] = cost(i,j)            //     minimum cost up to xi and yj
```

```
        p[i,j] = argmin of cost(i,j)  //     record the branch did we took
      end
   end
end
return S[nx,ny] and path starting from p[nx,ny]
```

where we have used the definition of $\text{cost}(i, j)$ given above.

Assuming that each call to $\text{cost}(i, j)$ takes constant time, and we carry out $(n_x + 1) \times (n_y + 1) - 1 = O(n_x n_y) = O(n^2)$ of them, then the running time is $O(n^2)$. (Note that x and y are treated symmetrically, and we may simply adopt the convention of naming the longer length to be n.) The space requirement is given by the size of S and p, which are also $O(n^2)$.

There are more space-efficient versions of this algorithm. For instance, notice that $\text{cost}(i, j)$ only ever refers to elements at most two rows up or two columns left of the current problem parameters. Thus, we may calculate the final solution by only storing three rows of S. (Do you see why we need three entire rows, rather than a 3×3 submatrix with $S(i, j)$ as the bottom-right element?) Now, the space requirement is only $O(n)$, but we must also give up the matrix p which means we lose the record of the optimal alignment. In 1975, Hirschberg gave a clever divide-and-conquer algorithm that solves both problems.

3 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.

4 On your own

1. Read Chapter 15