1 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.

1.1 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.[1]

  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)[2] which simply counts the number of these operations required to transform x into y.[3] 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.

1.2 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:

[1] 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.

[2] 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".

[3] 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.

1.3 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.[4]

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.

1.4 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 $x_1 x_2 \dots x_i = x_{1\dots i}$ and $y_1 y_2 \dots y_j = y_{1\dots j}$ 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 $y_{1\dots j-2}$.

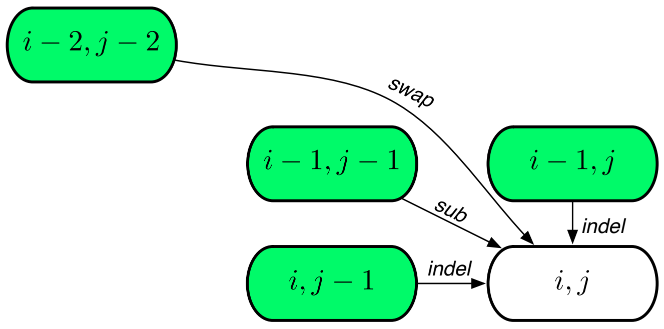
Let $\text{cost}(i, j)$ be the minimum cost of aligning $x_{1\dots i}$ and $y_{1\dots j}$, where we define as a base case $\text{cost}(0, 0) = 0$.

[4]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 $0 \le i \le n_x$ (length of x) and $0 \le j \le n_y$ (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.

1.5 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[5] 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,

[5]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 $1 \le j \le n_y$, and the costs in the 0th column are $S(i,0)=i$ for $1 \le i \le n_x$.

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 $x'=\texttt{S}$ with $y'=\texttt{A}$. 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 $\boxed{\texttt{S}} \rightarrow \boxed{\texttt{A}}$. Now substitute $\boxed{\texttt{T}}$ for $\boxed{\texttt{P}}$ Cost $= S(1,1) + 1 = 2$
- (Delete) Previously $\boxed{\texttt{S}} \rightarrow \boxed{\texttt{AP}}$. Now delete $\boxed{\texttt{T}}$. Cost $= S(1,2) + 1 = 3$
- (Insert) Previously $\boxed{\texttt{ST}} \rightarrow \boxed{\texttt{A}}$. Now insert $\boxed{\texttt{P}}$. Cost $= S(2,1) + 1 = 3$
- (Swap) Previously $\boxed{\_} \rightarrow \boxed{\_}$. 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 $\boxed{\texttt{S}} \rightarrow \boxed{\texttt{AP}}$. Now substitute $\boxed{\texttt{T}}$ for $\boxed{\texttt{E}}$. Cost $= S(1,2) + 1 = 3$
- (Delete) Previously $\boxed{\texttt{S}} \rightarrow \boxed{\texttt{APE}}$. Now delete $\boxed{\texttt{T}}$. Cost $= S(1,3) + 1 = 4$
- (Insert) Previously $\boxed{\texttt{ST}} \rightarrow \boxed{\texttt{AP}}$. Now insert $\boxed{\texttt{E}}$. Cost $= S(2,2) + 1 = 3$
- (Swap) Previously $\boxed{\_} \rightarrow \boxed{\texttt{A}}$. 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 $\boxed{\texttt{ST}} \rightarrow \boxed{\texttt{A}}$. Now substitute $\boxed{\texttt{E}}$ for $\boxed{\texttt{P}}$. Cost $= S(2,1) + 1 = 3$
- (Delete) Previously $\boxed{\texttt{ST}} \rightarrow \boxed{\texttt{AP}}$. Now delete $\boxed{\texttt{E}}$. Cost $= S(2,2) + 1 = 3$
- (Insert) Previously $\boxed{\texttt{STE}} \rightarrow \boxed{\texttt{A}}$. Now insert $\boxed{\texttt{P}}$. Cost $= S(3,1) + 1 = 4$
- (Swap) Previously $\boxed{\texttt{S}} \rightarrow \boxed{\_}$. 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 $\boxed{\texttt{ST}} \to \boxed{\texttt{AP}}$. Now substitute $\boxed{\texttt{E}}$ for $\boxed{\texttt{E}}$. Cost $= S(2,2) + 0 = 2$
- (Delete) Previously $\boxed{\texttt{ST}} \to \boxed{\texttt{APE}}$. Now delete $\boxed{\texttt{E}}$ (from x). Cost $= S(2,3) + 1 = 4$
- (Insert) Previously $\boxed{\texttt{STE}} \to \boxed{\texttt{AP}}$. Now insert $\boxed{\texttt{E}}$ (into y). Cost $= S(3,2) + 1 = 4$
- (Swap) Previously $\boxed{\texttt{S}} \to \boxed{\texttt{A}}$. Now transpose $\boxed{\texttt{TE}}$ and sub for $\boxed{\texttt{PE}}$. 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 $\boxed{\texttt{STE}} \to \boxed{\texttt{A}}$. Now substitute $\boxed{\texttt{P}}$ for $\boxed{\texttt{P}}$. Cost $= S(3,1) + 0 = 3$
- (Delete) Previously $\boxed{\texttt{STE}} \to \boxed{\texttt{AP}}$. Now delete $\boxed{\texttt{P}}$ (from x). Cost $= S(3,2) + 1 = 4$
- (Insert) Previously $\boxed{\texttt{STEP}} \to \boxed{\texttt{A}}$. Now insert $\boxed{\texttt{P}}$ (into y). Cost $= S(4,1) + 1 = 5$
- (Swap) Previously $\boxed{\texttt{ST}} \to \boxed{\texttt{\_}}$. Now transpose $\boxed{\texttt{EP}}$ and sub for $\boxed{\texttt{AP}}$. 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 $\boxed{\texttt{STE}} \to \boxed{\texttt{AP}}$. Now substitute $\boxed{\texttt{P}}$ for $\boxed{\texttt{E}}$. Cost $= S(3,2) + 1 = 4$
- (Delete) Previously $\boxed{\texttt{STE}} \to \boxed{\texttt{APE}}$. Now delete $\boxed{\texttt{P}}$ (from x). Cost $= S(3,3) + 1 = 3$
- (Insert) Previously $\boxed{\texttt{STEP}} \to \boxed{\texttt{AP}}$. Now insert $\boxed{\texttt{E}}$ (into y). Cost $= S(4,2) + 1 = 4$
- (Swap) Previously $\boxed{\texttt{ST}} \to \boxed{\texttt{A}}$. Now transpose $\boxed{\texttt{EP}}$ and sub for $\boxed{\texttt{PE}}$. 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

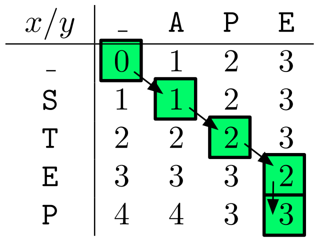
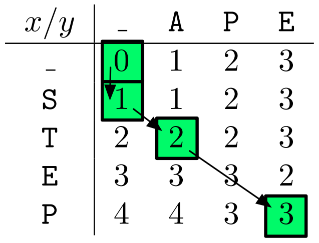
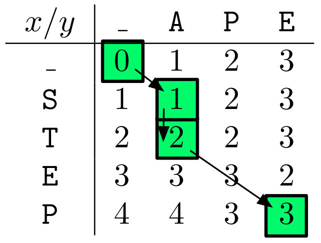
x/y	_	A	P	E
_	0	1	2	3
S	1	1	2	3
T	2	2	2	3
E	3	3	3	2
P	4	4		

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*.

1.6 A large worked example

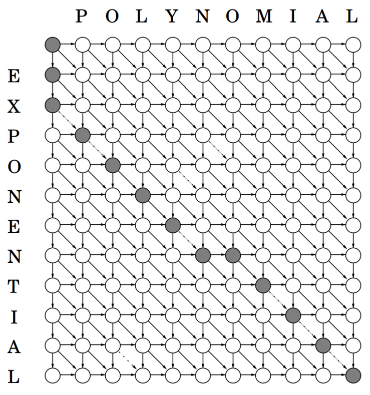
Consider aligning the strings $x =$ `EXPONENTIAL` and $y =$ `POLYNOMIAL`.[6] The full matrix S of costs is shown below, which is produced by starting at $i, j = 0$ and applying $\text{cost}(i, j)$ as given above iteratively to each element. (Or, by starting at $i = n_x$ and $j = n_y$ and making the recursive 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

[6]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.

1.7 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. □

1.8 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 cost(i, j) given above.

Assuming that each call to 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 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.

2 On your own

1. Read Chapter 15