The Education Column

by

Juraj Hromkovič

Department of Computer Science
ETH Zürich
Universitätstrasse 6, 8092 Zürich, Switzerland
juraj.hromkovic@inf.ethz.ch

An Elementary Approach Towards Teaching Dynamic Programming

Hans-Joachim Böckenhauer
Department of Computer Science, ETH Zürich
hjb@inf.ethz.ch

Tobias Kohn
University of Cambridge
tk534@cam.ac.uk

Dennis Komm
Pädagogische Hochschule Graubünden, Chur
Department of Computer Science, ETH Zurich
dennis.komm@inf.ethz.ch

Giovanni Serafini
Department of Computer Science, ETH Zürich
giovanni.serafini@inf.ethz.ch

Abstract

Dynamic programming has its firm place in the toolbox of a computer scientist. Its educational value, however, goes far beyond the border of our discipline. The example of dynamic programming illustrates how such an important algorithm design principle, and more generally algorithmic thinking skills, can be applied to solve problems in other fields. We describe our experiences with an approach towards teaching dynamic programming without a formal introduction to recursion, which allowed us to successfully introduce it to first-semester students of natural sciences with almost no background in computer science.

1 Introduction

For many non-majors in disciplines related to mathematics, natural sciences and engineering, computer science is a mandatory part of the university

curriculum. The objective of such a course is typically to introduce the students to those aspects of algorithmic thinking that may be needed for further studies and possibly for research in their fields. In addition to teaching the basics of programming in a language such as Python or Matlab, we feel it is a necessity to also convey a basic understanding of algorithmic design principles, with particular focus on the efficiency of algorithms.

Efficiency of algorithms is one of the most fundamental pillars of computer science. For many problems, a naive approach becomes quickly infeasible, and stronger methods are needed – even though such stronger methods might not always exist. Since our classes address non-majors, a discussion of these principles is most fruitful when based on actual problems and examples, if possible even taken from a field familiar to the students.

As an important algorithmic design principle, we teach *dynamic programming*, which is the method of breaking down complex problems into smaller problems, which are easier solved, and then lead to a solution to the original complex problem. Typically, dynamic programming is strongly related to recursion. However, due to our constraints, recursion is not formally covered in the introductory programming classes. Accordingly, we were not able to build a discussion of dynamic programming on this major programming principle.

In this paper, we present our approach to include dynamic programming into the introductory programming course for natural science students. After covering the basics of both Python and Matlab [6], we discuss dynamic programming in class, based on the alignment problem for DNA sequences.

1.1 Dynamic Programming and Recursion

Dynamic programming can serve as an example for a widely useful technique that illustrates the power of clever algorithm design. In many standard textbooks providing an introduction to algorithm design, dynamic programming is discussed after the principle of recursion and motivated by analyzing the drawbacks of a recursive solution to certain problems such as computing the Fibonacci numbers [14] or some scheduling problem [11]. The central task in dynamic programming, namely building solutions for larger subproblems from those for smaller ones, is often explicitly coined in terms of recursion [5]. For a detailed discussion of how to teach dynamic programming via recursion, see also the paper by Forišek [8].

Inspired by the classical textbooks by Aho et al. [1, 2], we chose another, more elementary approach, and centered our lecture around constructing and filling out the dynamic programming table. We found that analyzing these tables on the one hand enables an intuitive approach to analyze the time

complexity of an algorithm and on the other hand is well-suited for training in a matrix-based programming language such as Matlab.

1.2 The DNA Alignment Problem

Since it offered the largest common ground for our students, we motivated dynamic programming with examples from biology. Here, we describe one such approach, which deals with the following scenario.

One of the most basic tasks in computational biology is to measure the similarity of biological molecules such as DNA or proteins, e.g., for the purpose of measuring the evolutionary distance between two given individuals or for handling error-prone data. DNA molecules are formed by long chains of four different small building blocks, the so-called *nucleotides*. The genetic information carried by a DNA molecule is fully determined by the sequence of these nucleotides; thus, it is convenient to encode a DNA molecule by a string over a four-letter alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, where the letters stand for the four nucleotides *Adenin*, *Cytosin*, *Guanin*, and *Thymin*, respectively. This reduces the task of comparing molecules to the much easier task of comparing long strings. For a deeper insight into the biological background and the area of string algorithms, we refer to the textbooks by Gusfield [9] or Böckenhauer and Bongartz [3, 4].

In our lectures, we discuss only one very basic string comparison task. It is easy to argue, while only using simple arguments, that a straightforward brute-force approach towards solving this problem is infeasible. Consequently, this example provides a rather impressive opportunity to point out the power of dynamic programming by again using rather simple and straightforward arguments. One of the main points raised is to compare exponential and polynomial time complexity.

1.3 Classroom Experience

We taught the following material at various occasions, the largest being a first-semester course of students of biology, pharmacology, environmental sciences, health sciences and technology, agriculture, geology, and nutritional sciences. The class typically lasts 90 minutes and takes place at one of the largest halls at our university, with the audience usually consisting of several hundred students. The chosen teaching method is direct instruction, enriched by carefully planned questions that are expected to increase participation and initiate discussion.

Other occasions at which the material was presented include a course on computational biology at a community college, and various high school classes.

Although age and prior knowledge of the students may differ across the groups, the didactic approach, the content as well as the learning objectives are generally the same.

1.4 Didactical Aspects

Since computer science is yet just one among several concurring optional subjects at high school in our country, we are not allowed to assume any pertinent prior knowledge in concepts and methods of programming. A survey in the first week of the lecture confirms that the majority of the students is used to office applications and social media, but lacks even a minimal scientific background in computer science when entering university.

When presenting this teaching unit on dynamic programming, our students are assumed to already understand the basics of programming in Matlab, including variables, conditional execution, and simple loops, but no recursion. Furthermore, they have themselves designed simple algorithms of around ten lines of code, and carried out some hands-on analyses of their algorithms with respect to both correctness and running time.

In general terms, the teaching unit aims at introducing the students to the notion of an algorithm design technique as a generalized method to approach the solution of a given problem. We expect the students to become aware that algorithm design techniques are special instances of problem solving strategies [15], and that dynamic programming is only one among several such paradigms [11]. Accordingly, we want our students to first search for adequate design techniques when facing new computational problems in their field of study or research subject.

The specific goals of our teaching materials and our lecture are the following. 1. Analyzing the time complexity of algorithms, explaining the drawbacks of brute-force strategies and the need for clever algorithm design. 2. Introducing dynamic programming as an example of a clever algorithm design technique. 3. Modeling real-world problems, e.g., from biological applications, as computer science tasks. 4. Training programming skills, especially dealing with (multidimensional) arrays in Matlab or Python.

2 Designing the Algorithm

In the following, we describe the setting and our approach towards teaching dynamic programming in an elementary way.

2.1 Modeling: The DNA Alignment Problem

We motivate the task as described above while usually giving a little more context and, e.g., background information on how the DNA is obtained, etc. At first, of course, we have to think of an accurate way to express how similar two given strings are; in particular, simply taking the *Hamming distance* of two strings, say, `ACGTACGTACGTACGT` and `CGTACGTACGTACGTA`, suggests that they are not similar at all, although this probably does not reflect reality. This is a good opportunity to talk about interdisciplinary approaches and the process of modeling a biological problem in terms of computer science. The two most important events that can happen to a DNA sequence are point mutations, leading to the change of a single letter of the corresponding string, and insertions or deletions of short subsequences. One of the most straightforward, yet very meaningful approaches to define a distance measure based on these two events is to consider the so-called *edit distance*, which is due to Levenshtein [12], and which can be explained easily to students, even without any background in computer science. Here, we allow the operation of inserting *spaces* into the two strings in order to *align* them; every inserted space yields a penalty of 1. Furthermore, whenever two different letters are at the same position, this is called a *mismatch*, which also results in a penalty of 1. The sum of all the penalties yields the cost of the given *alignment*, and the cost of a best (i.e., cheapest) such alignment is called the edit distance of the two strings.

Consider, e.g., the two strings $s =$ `GACGATTATG` and $t =$ `GATCGAATAG` over $\{$`A`, `C`, `G`, `T`$\}$. There are different ways to align them, e.g.,

```
GA-CGATTATG    GAC-GATTATG         GACGAT----TA-TG
GATCGAATA-G ,  GATCGAATAG- , and   ---GATCGAATAG-- ,
```

where we marked positions that cause penalties by a gray background. The three alignments result in penalties of 3, 5, and 10, respectively. Here, the first alignment yields the best result of the three; the resulting cost is 3, and it is easy to see that there is no better way of aligning s and t with respect to this distance measure. As a result, s and t have edit distance 3.

2.2 A Brute-Force Approach

In what follows, the task is to find a best alignment for given s and t. After explaining the problem like this in class, we start by analyzing the simplest approach possible, namely to try out all possibilities and pick a best among them. It is easy to explain, on an intuitive level, why this means trying out exponentially many possibilities: Note that an alignment of two strings

is unambiguously defined by the positions where gaps are inserted into the two strings. Counting the number of alignments that have to be considered thus means counting the number of possibilities to insert the gaps. Suppose both strings have the same number n of letters, i.e., $s = s_1 s_2 \dots s_n$ and $t = t_1 t_2 \dots t_n$, for $s_i, t_i \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ with $1 \leq i \leq n$. For the resulting string, every position i can be aligned as

$$\begin{matrix} s_i \\ t_i \end{matrix}, \quad \begin{matrix} s_i\texttt{-} \\ \texttt{-}t_i \end{matrix}, \text{ or } \quad \begin{matrix} \texttt{-}s_i \\ t_i\texttt{-} \end{matrix},$$

which leads to 3^n different alignments. Of course, this only gives a very loose lower bound on the total number of possible alignments; there are many other possibilities to align s and t. Without any formal introduction to running time analysis or polynomial-time complexities, we can argue that n is usually a string of some thousands of characters and therefore it is not possible to find a best alignment by this brute-force approach; hence, we need to come up with something more clever.

2.3 The Dynamic Programming Idea

This is the point where we introduce the concept of dynamic programming as an indispensable part of the algorithmic toolbox. The general idea of computing the solution to a given instance from the solutions for smaller (sub-)instances is intuitively very plausible and easy to understand even with very little previous knowledge. Nevertheless, finding the right set of smaller instances to solve is not trivial in this case and nicely illustrates the work of an algorithm designer.

In the case of the alignment problem with respect to the edit distance, this creative work was first done by Needleman and Wunsch [13]. The idea is to compute the edit distance for all pairs of prefixes of the two given strings, including the empty prefix. This might be surprising at first glance since it seems to require much extra work. Why is it necessary to compute a quadratic number of optimal alignments instead of only one? From the point of view of an experienced computer scientist, this question is easiest investigated in terms of a recursive procedure. We reduce the task of computing the alignment of two given strings to the task of computing it recursively for three different prefix pairs of strictly smaller total length.

This basic idea behind the algorithm can be illustrated as follows. Again, let us consider two strings s and t with n letters each. Now consider the last letters s_n and t_n. There are three options to align those: (1) we insert a space beneath s_n inducing a penalty of 1; (2) we insert a space above t_n again

leading to an additional cost of 1; or (3) we write s_n above t_n, which causes no penalty if $s_n = t_n$, and a penalty of 1 otherwise. Observe that there is always an optimal alignment in which there are no two spaces beneath each other; thus, this option is ignored.

As an example, consider the two strings $s =$ `ATG` and $t =$ `TAG`, and in particular the third position. The three possibilities are

$$\begin{array}{r|l} \texttt{AT} & \texttt{G} \\ \texttt{TAG} & \texttt{-} \end{array}, \quad \begin{array}{r|l} \texttt{ATG} & \texttt{-} \\ \texttt{TA} & \texttt{G} \end{array}, \text{ and } \quad \begin{array}{r|l} \texttt{AT} & \texttt{G} \\ \texttt{TA} & \texttt{G} \end{array}.$$

The penalties are 1, 1, and 0, respectively. Hence, the value of an optimal alignment for s and t can be computed recursively from the three alignments for the remaining string pairs after cutting off the last column of the alignment, i.e., from the penalties for aligning `AT` and `TAG`, aligning `ATG` and `TA`, and aligning `AT` and `TA`, respectively.

However, the main idea behind the dynamic programming approach is to perform this recursive computation not via recursive function calls, but by using a bottom-up approach of filling a table of penalty values. Since the actual computation is only concerned with filling this table, we can explain the algorithm without explicitly mentioning the concept of recursion. Cutting off the last column of any possible alignment of the given strings s and t can be motivated as follows. Assume we had a possibility to compute the alignments for the three smaller instances; then we could also solve our actual task. In this case, the only difficulty that is left is to find some sufficiently easy small cases to start with, which will be the alignments of some string with the empty string. In the following, we describe the procedure in more detail.

2.4 Computing the Penalty Table

The common approach is to use a table P (for "penalties") in order to both illustrate and implement this idea. To be more general, we allow that s and t have different lengths; say, $s = s_1 \dots s_m$ and $t = t_1 \dots t_n$. In this case, P has a size of $(m + 1) \times (n + 1)$, and the cell (i, j) of P with $0 \le i \le m$ and $0 \le j \le n$ contains the smallest penalty for aligning the first i letters of s with the first j letters of t; we denote this value by $P(i, j)$.

The special cases of the first row and column (with index 0) need to be explained carefully. These represent aligning some prefix of one of the two strings with the empty string ε that contains zero letters. Aligning any prefix of s or t with ε results in

$$\begin{array}{c} s_1 s_2 \dots s_i \\ \texttt{-}\ \texttt{-}\quad\ \texttt{-} \end{array} \quad \text{or} \quad \begin{array}{c} \texttt{-}\ \texttt{-}\quad\ \texttt{-} \\ t_1 t_2 \dots t_j \end{array},$$

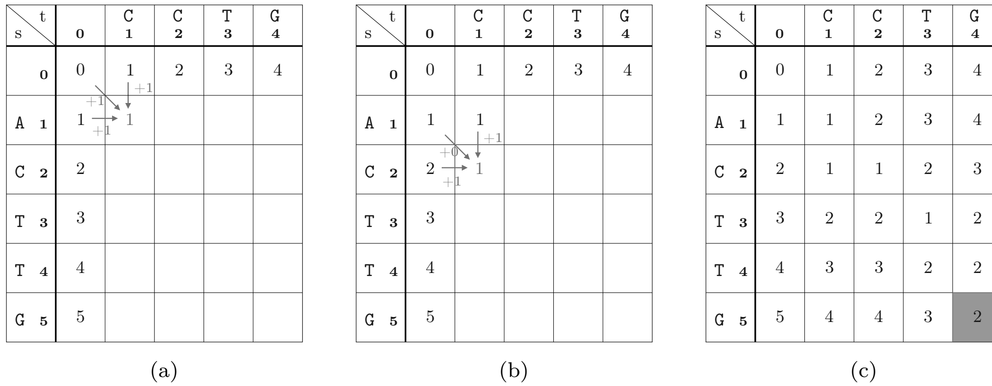


Figure 1: Filling out the penalty matrix

with penalties i and j, respectively. The concept of empty strings is in general unknown to the target audience and might appear a little confusing to the students at first glance. However, from this example, where it provides a very easy and elegant way to define the base cases for our bottom-up dynamic programming approach, the students can usually be easily convinced that empty strings are a very helpful concept.

We now demonstrate how to fill out P cell by cell using the two strings $s =$ `ACTTG` and $t =$ `CCTG` as an example. We start with the first row and column, which get a penalty equal to the length of the string that is aligned with ε; see Figure 1a. Next, we fill out cell $(1, 1)$, which contains the minimum penalty to align the two first letters of s and t. As described above, there are three possibilities for the last column of this alignment; again, see Figure 1a.

1. In this last column, we can insert a space beneath the first letter s_1 of s, i.e., into t. This way, we obtain a penalty of 1, the first letter of s is written down, but no letter of t. This leaves us with the subproblem of aligning the empty prefix ε of s with the prefix t_1 of t, yielding

   $$\begin{array}{c|c} \varepsilon & \texttt{A} \\ \texttt{C} & \texttt{-} \end{array},$$

   and the penalty to align ε with $t_1 =$ `C` can already be found in P in cell $(0, 1)$, i.e., one row above the cell we are about to fill out. We already know that this penalty is 1, and thus we get a total penalty of 2 in this case.

2. Conversely, we can insert a space into s in the last column of the

alignment, which yields

$$\begin{array}{c|c} \texttt{A} & \texttt{-} \\ \varepsilon & \texttt{C} \end{array} .$$

This again causes a penalty of 1 plus an additional penalty of 1 to align s_1 with ε, which is already written down in $(1, 0)$. Hence, we get a penalty of 2 also in this case.

3. Finally, we can create a mismatch, i.e., we let the last column of the alignment contain the letters $s_1 = \texttt{A}$ and $t_1 = \texttt{C}$, resulting in

$$\begin{array}{c|c} \varepsilon & \texttt{A} \\ \varepsilon & \texttt{C} \end{array} .$$

This mismatch also causes a penalty of 1, and it leaves us with the subproblem of aligning ε with ε, which causes no penalty, as can be looked up in cell $(0, 0)$. The total penalty in this case is therefore 1.

In all three cases, the penalty for the candidate solution can be easily computed from the already known penalty values in the table. The smallest cost to align s_1 and t_1, (i.e., the value in $(1, 1)$) is then given by the minimum of the three values just described, i.e.,

$$P(1, 1) = \min\{P(0, 1) + 1, P(1, 0) + 1, P(0, 0) + 1\} = 1 .$$

This procedure can then be repeated for computing the remaining cells in the table. Let us consider the cell $(2, 1)$ of P, i.e., aligning the first two letters of s with the first one of t. We again have three cases; see Figure 1b.

1. Inserting a space into t in the last alignment column gives

$$\begin{array}{c|c} \texttt{A} & \texttt{C} \\ \texttt{C} & \texttt{-} \end{array} ,$$

and the penalty to align s_1 and t_1 can be found in cell $(1, 1)$.

2. Inserting a space into s in the last column yields

$$\begin{array}{c|c} \texttt{AC} & \texttt{-} \\ \varepsilon & \texttt{C} \end{array} ,$$

and the penalty to align $s_1 s_2$ with ε can be found in cell $(2, 0)$.

3. Aligning the two last letters, i.e., s_2 and t_1, with each other gives

$$\begin{array}{c|c} \texttt{A} & \texttt{C} \\ \varepsilon & \texttt{C} \end{array} ,$$

   and the penalty to align s_1 with ε is written in cell $(1, 0)$.

Since the last option does not cause a mismatch, we obtain

$$P(2, 1) = \min\{P(1, 1) + 1, P(2, 0) + 1, P(1, 0)\} = 1 ,$$

and we observe that we look at the same relative cells as before, namely that one row above, that one column to the left, and the one above left. Generalizing this strategy, we get

$$P(i, j) = \min\{P(i - 1, j) + 1, P(i, j - 1) + 1, P(i - 1, j - 1) + p_{ij}\} ,$$

where p_{ij} is 1 if and only if a mismatch is created in the last option, and 0 otherwise. The cell (m, n) finally contains the cost of a best alignment of s and t, i.e., the edit distance of the two input strings.

At this point, especially in smaller classes, we ask the students to try to fill out the rest of the table for themselves, which takes around five to ten minutes. The complete table is shown in Figure 1c. This way, they quickly discover themselves that the work carried out is rather repetitive. It becomes obvious that a computer can be easily told to do this instead.

3 Implementation of the Algorithm

Now that the high-level description of the algorithm is produced, we can start with the practical part.

3.1 Implementation in Matlab

Another nice thing about reducing the problem to filling out a table is that it can be implemented rather easily in Matlab. One of the reasons is that Matlab handles 2-dimensional arrays without much syntactical overhead. We can essentially follow the above high-level description and translate the algorithm into code right away.

As noted above, the students know at this point conditional execution and loops; not much more is needed. The only unfortunate inconvenience is that, in Matlab, vector indices start with 1 instead of 0. Algorithm 1 shows our implementation, which initializes the input strings s and t and the penalty

```
1.  s = 'ACTTG';                          5.  P = zeros(m+1,n+1);
2.  t = 'CCTG';                           6.  for j=1:n+1
3.  m = length(s);                        7.    P(1,j) = j-1;
4.  n = length(t);                        8.  end
                                          9.  for i=1:m+1
                                          10.   P(i,1) = i-1;
                                          11. end

12. for i=2:m+1                                          %    For every row
13.   for j=2:n+1                                        % and every column
14.     if s(i-1) == t(j-1)
15.       pij=0;                                              % No mismatch
16.     else
17.       pij=1;                                                 % Mismatch
18.     end
19.     x = [P(i-1,j)+1, P(i,j-1)+1, P(i-1,j-1)+pij];
20.     [smin,imin] = min(x);     %                         Compute minimum
21.     P(i,j) = smin;            % and store it in penalty matrix
22.
23.   end
24. end
```

Algorithm 1: Computing the penalty matrix P

matrix P in lines 1 to 11, and fills out the latter in lines 12 to 24. This block follows exactly our description. In lines 14 to 18, we compute whether a match or mismatch happens if the two letters are written beneath each other; the minimum of the three values is computed in lines 19 to 20, and stored in P in line 21.

3.2 Backtracing

After the implementation shown in Algorithm 1 is understood, we can discuss in class that the output of the algorithm indeed gives the smallest cost to align the two input strings, but does not tell us what the actual alignment looks like. In this context, it is very comfortable that Matlab's minimum function returns a two-component vector, namely the minimum and the index from which the minimum stems. We save these two values as `smin` and `imin` in line 20 of Algorithm 1, but only use the first one so far.

For every entry of P, we now just compare the minimum of three values,

which correspond to three adjacent cells. At least one of those gives the minimum, and in turn corresponds to one of the three options we have already described.

1. Inserting a space into t; this corresponds to the cell in the row above and the first entry of the vector `x` in line 19 of Algorithm 1.
2. Inserting a space into s, which corresponds to the column to the left and thus the second entry of `x`.
3. Writing the two letters underneath each other, which corresponds to the third entry of `x`.

We now save which of the three values actually is the minimum; of course, this is not necessarily unique, and ties are broken in favor of the smaller index. This is the index of the entry of `x` with minimum value; we have stored this value as `imin` in line 20 of Algorithm 1, and we now additionally save it in the corresponding cell of a matrix B (for "backtracing"). Hence, we add the line

```
22.     B(i,j) = imin;
```

to Algorithm 1. B has the same size as P, and it stores one possible optimal alignment in the following way: The cell (i, j) of B contains a 1 if and only if a space was inserted into t, a 2 if and only if a space was inserted into s, and a 0 if and only if both letters were written underneath each other, with respect to this particular alignment. Again, the first row and column need to be handled in a special manner. In the first column, all entries are 1 since there is no column to the left, and spaces can only be inserted into t; likewise, all entries in the first row are 2. The cell in the upper-left corner is assigned value 0. The backtracing matrix for our sample strings $s =$ `ACTTG` and $t =$ `CCTG` is shown in Figure 2a, where arrows indicate how the numbers are interpreted. In this example, the algorithm computes the alignment

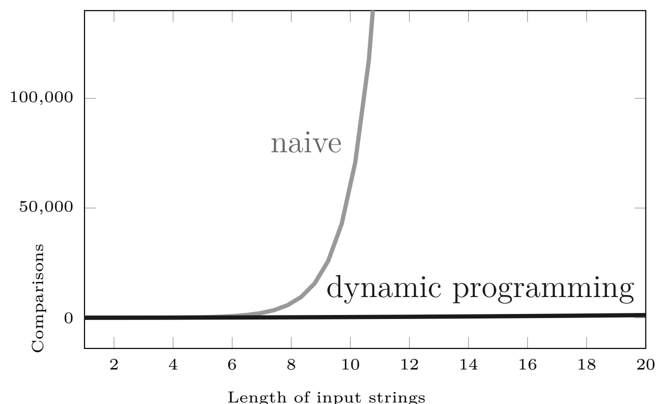
```
ACTTG
-CCTG,
```

which indeed contains one space and one mismatch and thus has a penalty of 2 as computed in Figure 1c.

Furthermore, we create a 2-dimensional vector `result`, whose first row corresponds to the aligned string s and whose second row corresponds to the aligned string t, by marking spaces with "`-`." We fill `result` from the right to the left as shown in Algorithm 2: We start in cell (m, n) of B; recall that this is cell (`m+1`, `n+1`) in Matlab, thus the variables `i` and `j` are initialized as `m+1`

s \ t	0	C 1	C 2	T 3	G 4
0	0	2	2	2	2
A 1	1	0	0	0	0
C 2	1	0	0	2	2
T 3	1	1	0	0	2
T 4	1	1	0	0	0
G 5	1	1	0	1	0

(a)



(b)

Figure 2: The backtracing matrix and comparison of time complexities

and `n+1`, respectively. If we find a 1 in this cell, we write the last letter of `s` in the first line of `result` and a space in the second line of `result`, since this means that a space was inserted into `t`. Likewise, if there is a 2, we write a space in the first row and the last letter of `t` in the second row of `result`. Last, if there is a 0, we write the last letters of `s` and `t` at the first and second row of `result`, respectively. According to the entry, we continue in the indicated cell and repeat the procedure until we arrive at cell $(0, 0)$. Once we reach this cell, `result` contains a representation of an optimal alignment of `s` and `t` (preceded by some empty cells).

4 Analysis of the Time Complexity

The correctness of the algorithm of Needleman and Wunsch follows from our discussion at the beginning. We do not formally prove it in class. Analyzing the time complexity of the algorithm is again straightforward, since its main work is to fill out the tables P and B of size $(m+1) \times (n+1)$ each. Computing the value of any cell only involves a constant number of comparisons and elementary arithmetic operations, and we therefore get a running time in $O(m \cdot n)$ for this part. Computing the actual alignment as shown in Algorithm 2 can be done in time $O(\max\{m, n\})$. Note that we are not using the O-notation in our lecture since the students are not familiar with it. Instead, we just count the comparisons and assignments in the Matlab code. It follows that dynamic programming for the alignment problem induces an exponential improvement over the brute-force approach discussed at the beginning. It

```
1. result = cell(2,m+n);
2. i = m+1;
3. j = n+1;
4. pos = m+n;
5.  B = zeros(m+1,n+1);
6.
7.  for j=1:n+1
8.    back(1,j) = 2;
9.  end
10. for i=1:m+1
11.   back(i,1) = 1;
12. end

13. while (i > 1 || j > 1)          % While we are not in cell (1,1)
14.    if B(i,j) == 1                               % Step from above
15.      result{1,pos} = s(i-1);
16.      result{2,pos} = '-';
17.      i = i-1;                             % Continue in row above
18.    else if B(i,j) == 2                           % Step from left
19.      result{1,pos} = '-';
20.      result{2,pos} = t(j-1);
21.      j = j-1;                   % Continue in column to the left
22.      else                                       % Diagonal step
23.        result{1,pos} = s(i-1);
24.        result{2,pos} = t(j-1);
25.        i = i-1;                           %  Continue in row above
26.        j = j-1;                          % and column to the left
27.      end
28.    end
29.    pos = pos-1;
30. end
```

Algorithm 2: Computing the alignment using the backtracing matrix B

is therefore a good candidate to discuss polynomial versus exponential time. To illustrate the impact of the difference, we usually show a graph as in Figure 2b.

5 Exercises and Extensions

The presented alignment algorithm is very robust in the sense that it can be used to solve also many variants of the basic problem. In the lecture, we discuss two of these extensions if time permits. Implementing these extensions in Matlab can serve as a good exercise for the students.

First, the algorithm can be easily adapted to more complex distance functions, for example choosing different penalties for different mismatching pairs of letters. This is very helpful, e.g., when comparing protein sequences which are comprised of 20 different amino acids some of which are chemically more similar than others.

Second, the results of the alignment algorithm are biologically meaningful only in the case when both strings are of approximately the same size. Consider the strings $s = \texttt{TAAGGT}$ und $t = \texttt{AGTTTATAGCCTGGT}$. An optimal alignment according to the edit distance measure is

```
----TA-A----GGT
AGTTTATAGCCTGGT
```

with a penalty of 9. However, from a biological point of view, the following alignment is better motivated, although it induces a penalty of 10, because the smaller string is aligned to a compact region of the longer string:

```
----TA-AGG-T---
AGTTTATAGCCTGGT .
```

To adapt the algorithm such that it finds the second alignment rather than the first one, we have to make sure that gap symbols at the beginning and the end of the shorter string do not cause any penalty. This can be done (if, without loss of generality, s is the shorter string) by initializing the first row of P with all zeros, and taking as a result not necessarily the value in the lower-right corner cell of the table, but the minimum value in the last row.

6 Conclusion

In this paper, we described our approach towards teaching dynamic programming in an elementary way to students without a strong background in computer science. Our general goal is to teach algorithmic thinking as early as possible [10], and the described way allows the students to realize one of the most essential principles of algorithm design without much syntactical overhead and especially without a formal introduction to recursion.

The presented alignment algorithm is of large practical importance as it lies at the heart of all implementations that are actually used for sequence comparisons in biology (although a lot of heuristic rules are added in practice to reduce the running time from quadratic to linear while keeping the error small). Especially for our students from biology and related sciences, it is highly motivating to see how algorithmic thinking can help to solve basic problems from their field.

Implementing this algorithm is also well-suited with respect to the goal of teaching Matlab programming since handling the dynamic programming table fits very well into Matlab's easy syntax for matrices.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[3] Hans-Joachim Böckenhauer and Dirk Bongartz. *Algorithmische Grundlagen der Bioinformatik.* Vieweg+Teubner, 2003.

[4] Hans-Joachim Böckenhauer and Dirk Bongartz. *Algorithmic Aspects of Bioinformatics.* Springer, 2007.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms,* 3rd edition. MIT Press, 2009.

[6] Markus Dahinden, Lukas Fässler, Dennis Komm, and David Sichau. Einführung in die Programmierung mit Python und Matlab. *Begleitunterlagen zum Onlinekurs und zur Vorlesung*, 2015.

[7] Benedict Du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2:57–73, 1986.

[8] Michal Forišek. Towards a better way to teach dynamic programming. *Olympiads in Informatics*, 9:45–55, 2015.

[9] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

[10] Juraj Hromkovič, Tobias Kohn, Dennis Komm, and Giovanni Serafini: Algorithmic thinking from the start. *Bulletin of the EATCS* 121, The Education Column, 2017.

[11] Jon Kleinberg and Éva Tardos. *Algorithm Design.* Addison Wesley, 2006.

[12] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In: *Doklady Akademii Nauk SSSR* 163(4):845–848, 1965.

[13] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48(3):443–53, 1970.

[14] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen,* 5th edition. Spektrum Akademischer Verlag, 2012.

[15] Jeanne E. Ormrod. *Human Learning* 5th edition. Pearson/Merrill Prentice Hall, 2008.