In this module we will discuss algorithms. These lecture notes are accompanied by Maple worksheet Mat294Algorithms.mw.
Chapter 1

Introduction

In this module we will discuss basic algorithmic techniques. Consider the following problem.

**Problem 1** *Give an algorithm which finds string TATA in a sequence of nucleotides S.*

For example if $S$ is

$$ACCCGTACTACTACTTTACACACTTTACTTATACCGAACTCT$$

we should identify TATA as starting in the 35th position:

$$ACCCGTACTACTACTTTACACACTTTACTTATACCGAACTCT.$$  

Note that our problem is not to find 35 but to give a method which will work for any string $S$. Such a method is called an *algorithm*. Imagine a very long string $S$ (say 10,000 characters). How will you find the substring TATA in it? Maybe you can examine a substring of 4 characters at a time to see if it is the desired one but certainly it will be hard to immediately notice if $S$ contains TATA. Computers are not much different in this task and we will assume that we can compare two characters in one step to see if they are equal or not. Under this assumption the easiest thing to do is to start scanning $S$ from left to right and move a window of length four one character at a time to see if TATA appears in it. In our example, in the first step we compare A (first character of $S$) with T (first character of TATA). They are different and so we move to the next character of $S$, which is C and compare it with T (first character of TATA). Different, we move by one. In step 6, we compare T (sixth character of $S$) with T (first character of TATA) and see that they match. Then we
look at the seventh character of $S$ which is $A$ and compare it with $A$ (the second character in TATA) and they match. As a result we compare the eighth character of $S$ which is $C$ with $T$ and they do not match. We move the window. We look at the seventh character of $S$ and compare it with $T$ (the first character of TATA). Clearly if we continue in this way we will find a location of the string TATA.

In general, for an algorithm to be eligible to solve a problem it must have a few properties:

- It must be definite. It does not leave any room for doubt of what to do and when.
- It must be general. If we change string $S$ to something else, the method should still work.
- It must be finite. Our time is limited.
- It must be correct. It should do what it claims it does.

To check if an algorithm satisfies the above requirements (especially the last one), the algorithms are almost always described using what is called a pseudo-code. You can think about the pseudo-code as an intermediate step between an informal description and an implementation of the algorithm in a programming language. On one hand pseudo-code gives a very precise and transparent description of the method, on the other hand it ignores unnecessary details of an implementation. We will use a subset of the following pseudo-code language.

Pseudo-code keywords

- Assignment statement: $x := x + y$. Set $x$ to be equal to $x + y$.
- Comparison condition: $x = y$ (or $x \neq y$). The value is true or false. It is true if $x = y$ (it is true if $x \neq y$).
- Block: begin ... end
- Condition statement: if condition then ... else ...
- ”for” loop: for $i := m$ to $n$ ... Iterate with index $i$ from $m$ to $n$ increasing $i$ by one in each iteration.
- ”while” loop: while condition do ... Execute the loop as long as the condition is satisfied.
• return statement: **return** \( a \). Procedure returns value \( a \) and ends its execution.

**Example 1** Consider the following procedure.

**Algorithm Add** *(Positive integer: \( n \))*

1. result := 0
2. for \( i := 1 \) to \( n \) do
3. \hspace{1em} result := result + \( i \)
4. return result

The algorithm adds integers from 1 to \( n \). In the second step we have the "for" loop in which we iterate with variable \( i \) starting with \( i = 1 \) and ending when \( i = n \). Each iteration "start" in the step 3 (first step inside the loop) and "ends" in step three (last step inside the loop). In the third step we assign to the variable result the value of result plus \( i \). Note this is not an equality (we do not claim that result = result + \( i \)), it is an assignment. Suppose that \( n = 5 \) and we call \( \text{Add}(5) \). First step, we set result = 0 and go to the "for" loop. There will be 5 iterations of step 3. In the first, \( i = 1 \) and so in the third step we set result = 0 + 1 = 1. In the second \( i = 2 \) and so result = 1 + 2 = 3. In the third, \( i = 3 \) and so result = 3 + 3 = 6. In the fourth, \( i = 4 \) and we have result = 6 + 4 = 10. In the fifth, result = 10 + 5 = 15. Then we leave the "for" loop and move to the fourth step where we return 15.

In the next example we will give the simple string searching algorithm for searching the substring TATA.

**Example 2** We will assume that a string and the length of it are given as arguments.

**Algorithm SimpleSearch** *(String \( S_1S_2 \ldots S_k \))*

1. for \( i := 1 \) to \( k - 3 \) do
2. \hspace{1em} if \( S_i = 'T' \) then
3. \hspace{1em} if \( S_{i+1} = 'A' \) then
4. \hspace{1em} if \( S_{i+2} = 'T' \) then
5. \hspace{1em} if \( S_{i+3} = 'A' \) then return \( i \)
6. return 'Not found'
First of all let us note that nested "if" statements can be combined into one: \( \text{if } S_i = T' \text{ and } S_{i+1} = A' \text{ and } S_{i+2} = T' \text{ and } S_{i+3} = A' \text{ then return } i \) (This is precisely what we are doing in our MAPLE implementation). It is however more transparent for our purposes to assume that an algorithm can compare one character at a time.

Note that finding a substring in a string is an important problem in computer science and there are methods like Rabin-Karp algorithm or Knuth-Morris-Pratt algorithm which are faster than our naive approach. Consult [CLR92] for more details.
Chapter 2

Basic Alignment

In this section, we will discuss the most basic algorithm for the alignment problem. In this problem we are given two strings \( S \) and \( T \) of not necessarily the same length and we want to measure if they are similar or not. Similarity (or lack of it) is displayed by finding an alignment of two strings and computing the so-called score of an alignment. Alignment consists of insertions of spaces into \( S \) and \( T \) so that new strings (i.e. strings after insertion) have the same length. New strings are typically written one below another so that corresponding positions can be easily compared. For example an alignment of GACAATT and ACTTT can look like

\[
\begin{align*}
G & \ A & C & A & A & T & T & - \\
- & A & C & - & - & T & T & T \\
\end{align*}
\]

We will assume the following basic scoring model. If corresponding characters are the same (match) then we add \( \alpha \) to the score of an alignment, if two corresponding characters are different (mismatch) then we add \( -\beta \) to the score, and if a character is aligned to the space (space) we add \( -\gamma \) to the score. The score of the alignment is then the sum of scores of pairs of corresponding characters. In our example, the score of the alignment is \( 4 \cdot \alpha - 4 \cdot \gamma \).

**Example 3** Find the score of the following alignment with \( \alpha = 3 \), \( \beta = 2 \) and \( \gamma = 1.5 \).

\[
\begin{align*}
G & \ A & T & G & A & - & T & T & - \\
- & A & C & - & A & C & T & T & T \\
\end{align*}
\]

The score is \( 4\alpha - \beta - 4\gamma = 4 \).

Of course interpretation of the numerical value of the score is a serious problem and must be approached with care (usually the score is compared with the score obtained by aligning two completely random strings of same length).
In the **optimal alignment problem** we want to find an alignment that maximizes the score. It is possible that more than one alignment is optimal, in which case we are happy to find just one of them (The algorithm can be easily modified to give all optimal alignments). This is not an easy problem! Solution is divided into two parts. First of all we want to find the score of an optimal alignment. Second, we want to be able to spell out the alignment which gives that score. The method that we will apply is an important general technique called the **dynamic programming** algorithm. Suppose two strings to be aligned are \( s_1s_2 \cdots s_n \) and \( t_1t_2 \cdots t_m \) (for example ACCGT with \( n = 5 \) and GAC with \( m = 3 \)). We will introduce an auxiliary score function \( f \) with two arguments. Let \( f(i, j) \) be the score of an optimal alignment of \( s_1s_2 \cdots s_i \) with \( t_1t_2 \cdots t_j \). Now, we try to invent a formula for \( f(i, j) \) that can be used to find \( f(n, m) \) — the score of an optimal alignment of both strings. First note that we allow \( i \) and \( j \) to be equal to zero (this corresponds to empty string) and we set \( f(0, j) = -\gamma \cdot j \) as the only way to align \( t_1 \cdots t_j \) with an empty string is to have each of \( t_k \)'s aligned to the space. For the same reason \( f(i, 0) = \gamma \cdot i \). General formula for \( f(i, j) \) is a bit more complicated:

\[
f(i, j) = \max\{f(i - 1, j - 1) + \text{score}(s_i, t_j), f(i - 1, j) - \gamma, f(i, j - 1) - \gamma\} \tag{2.1}
\]

where \( \text{score}(s_i, t_j) = \alpha \) if \( s_i = t_j \) and is equal to \( -\beta \) if \( s_i \neq t_j \). Let us see if the formula makes sense. How can we obtain an alignment of \( s_1s_2 \cdots s_i \) with \( t_1t_2 \cdots t_j \)? We look at the last position in the alignment. There are really only three possibilities: (1) \( s_i \) is aligned to \( t_j \) which gives the score of \( \text{score}(s_i, t_j) \) in this last position (2) \( s_i \) is aligned to the space which gives the score of \( -\gamma \) (3) the space is aligned to \( t_j \) which gives the score of \( -\gamma \). Since we want to find an alignment of the highest score we want to select the option from the above three that maximizes the score (this is the reason for the maximum in the formula). In the case of (1), what precedes the alignment of \( s_i \) with \( t_j \) is an alignment of \( s_1 \cdots s_{i-1} \) with \( t_1 \cdots t_{j-1} \) of the highest score given by \( f(i - 1, j - 1) \). In the case of (2), what precedes \( s_i \) aligned to the space is an optimal alignment of \( s_1 \cdots s_{i-1} \) with \( t_1 \cdots t_{j-1} \) which has the score of \( f(i - 1, j - 1) \). Case (3) is similar to (2).

Can we use this formula? Notice that to find the value of \( f(i, j) \) we only need values of \( f(l, k) \) with either \( l < i \) or \( k < j \) and so we can find \( f(0, 0), f(0, 1), f(1, 0) \) to obtain \( f(1, 1) \). Then \( f(2, 0), f(0, 2) \) to obtain \( f(2, 1) \) and \( f(1, 2) \). Then \( f(2, 2) \) and so on.

**Example 4** Find the score of an optimal alignment of ACCC with CTC if \( \alpha = 3 \), \( \beta = 3 \), and \( \gamma = 2 \). We have \( n = 4 \), \( m = 3 \). Also \( f(0, 0) = 0 \), \( f(1, 0) = -2 \), \( f(2, 0) = -4 \), \( f(3, 0) = -6 \), \( f(4, 0) = -8 \) and \( f(0, 1) = -2 \), \( f(0, 2) = -4 \), \( f(0, 3) = -6 \). Now we continue with computations.
• \( f(1, 1) = \max\{0 - 3, -2 - 2, -2 - 2\} = -3 \text{ as the score} (A, C) = -3. \)

• \( f(1, 2) = \max\{f(0, 1) - 3, f(0, 2) - 2, f(1, 1) - 2\} = \max\{-2 - 3, -4 - 2, -3 - 2\} = -5 \)

• \( f(1, 3) = \max\{f(0, 2) - 3, f(0, 3) - 2, f(1, 2) - 2\} = \max\{-4 - 3, -6 - 2, -5 - 2\} = -7 \)

• \( f(2, 1) = \max\{f(1, 0) + 3, f(1, 1) - 2, f(2, 0) - 2\} = \max\{-2 + 3, -3 - 2, -4 - 2\} = 1 \)

• \( f(2, 2) = \max\{f(1, 1) - 3, f(1, 2) - 2, f(2, 1) - 2\} = \max\{-3 - 3, -5 - 2, 1 - 2\} = -1 \)

• \( f(2, 3) = \max\{f(1, 2) + 3, f(1, 3) - 2, f(2, 2) - 2\} = \max\{-5 + 3, -7 - 2, -5 - 2\} = -2 \)

• \( f(3, 1) = \max\{f(2, 0) + 3, f(2, 1) - 2, f(3, 0) - 2\} = \max\{-4 + 3, 1 - 2, -6 - 2\} = -1 \)

• \( f(3, 2) = \max\{f(2, 1) - 3, f(2, 2) - 2, f(3, 1) - 2\} = \max\{1 - 3, -1 - 2, -1 - 2\} = -2 \)

• \( f(3, 3) = \max\{f(2, 2) + 3, f(3, 2) - 2, f(2, 3) - 2\} = \max\{-1 + 3, -2 - 2, -2 - 2\} = 2 \)

• \( f(4, 1) = \max\{f(3, 0) + 3, f(3, 1) - 2, f(4, 0) - 2\} = \max\{-6 + 3, -1 - 2, -8 - 2\} = -3 \)

• \( f(4, 2) = \max\{f(3, 1) - 3, f(3, 2) - 2, f(4, 1) - 2\} = \max\{-1 - 3, -2 - 2, -3 - 2\} = -4 \)

• \( f(4, 3) = \max\{f(3, 2) + 3, f(3, 3) - 2, f(4, 2) - 2\} = \max\{-2 + 3, 2 - 2, -4 - 2\} = 1 \)

Optimal score is 1. It is easy to guess an optimal alignment in such a small example:

\[
\begin{array}{cccc}
A & C & C & C \\
\text{–} & C & T & C
\end{array}
\]

which has the score of \( 2\alpha - \beta - \gamma = 1 \).
We can now formally describe the algorithm. There are two main parts of it. In the first part, we use (2.1) and the method described above to find an optimal score. In order to find an alignment of the score \( f(n, m) \) we will need one more table. This will be the pointer table were each cell contains a pointer to the cell which gave the maximum value in (2.1). For programming purposes, we will follow the convention: 0-no pointer, 1-cell to the left of the current one gives the maximum, 2-cell above the current one gives the maximum, 3-diagonal gives the maximum. Ties are resolved arbitrarily. This encoding is used in the second part of the algorithm (which we will not describe in the pseudo-code but include in the Maple worksheet) to backtrack the computations and spell out an optimal alignment.

**Algorithm** `ALIGN` (Scores: \( \alpha = 3, \beta = 2, \gamma = 1.5 \), String 1: \( s_1 \ldots s_m \), String 2: \( t_1 \ldots t_n \))

1. for \( i := 0 \) to \( n \) do

2. \[ f[i, 0] := -\gamma * i, \quad \text{pointer}[i, 0] := 2 \]

3. for \( j := 0 \) to \( m \) do

4. \[ f[0, j] := -\gamma * j, \quad \text{pointer}[0, j] := 1 \]

5. \( \text{pointer}[0, 0] := 0 \)

6. for \( i := 1 \) to \( n \) do

7. for \( j := 1 \) to \( m \) do

8. if \( t_i = s_j \) then \( \text{score} := \alpha \) else \( \text{score} := -\beta \)

9. if \( f(i - 1, j - 1) + \text{score} \geq f(i - 1, j) - \gamma \) and \( f(i - 1, j - 1) + \text{score} \geq f(i, j - 1) - \gamma \) then

   • \( f(i, j) := f(i - 1, j - 1) + \text{score} \)

   • \( \text{pointer}[i, j] := 3 \)

10. else if \( f(i - 1, j) - \gamma \geq f(i, j - 1) - \gamma \) then

    • \( f(i, j) := f(i - 1, j) - \gamma \)

    • \( \text{pointer}[i, j] := 2 \)

11. else \( f(i, j) := f(i, j - 1) - \gamma, \quad \text{pointer}[i, j] := 1 \)
12. return $f(n, m)$

Note that in steps 9, 10, 11, we simply find a maximum in an elaborate way. First (in step 9) check if $f(i-1, j-1) + \text{score}(s_i, t_j)$ is larger than or equal to the other two numbers, else (if it is smaller than at least one of these numbers) try $f(i-1, j) - \gamma$ (note that $\gamma$ on both sides is irrelevant and could be deleted from the pseudo-code), if it is not the case then maximum is at $f(i, j) := f(i, j - 1) - \gamma$. Pointer table is of no use in the algorithm $\text{ALIGN}$, but as indicated before, it is useful to find an alignment in the trace-back. We do not give the pseudo-code for the trace-back but the method is really very basic: Start at $(n, m)$ cell in the table. If $\text{pointer}[n, m]$ is 3 then align $s_m$ with $t_n$ and move to the diagonal cell ($(n-1,m-1)$). If the pointer is 2 then align $s_m$ with the space character and move to the cell above ($(n-1,m)$). If it is 1 then align the space character with $t_n$ and move to the left cell ($(n,m-1)$). Continue moving in this way until you reach 0 pointer (i.e. cell $(0,0)$). This gives alignment in the reverse order. There are many ways of implementing this strategy. Our Maple implementation is not a classical one as we wanted to avoid recursion.

Example 5 We will find an optimal global alignment of ACCGT and ATCG with scores 4 for match, -3 for mismatch, and with penalty of $-2$ for a space. Computations and pointers are given in the table in Figure 2.1. For example, look at the row labeled with 1 and column labeled with 2. To find the value of this entry we look at the maximum $\max\{f(0, 1) - 3, f(1, 1) - 2, f(0, 2) - 2\} = \max\{-5, 2 - 6\} = 2$ and the maximum value comes from the entry to the left (we mark it by drawing an arrow to the left cell; the value of the pointer in $\text{ALIGN}$ is 1). We use the trace-back to find the alignment:

\[
\begin{array}{cccc}
A & C & C & G & T \\
A & T & C & G & -
\end{array}
\]

The score of the alignment is

\[3 \cdot 4 - 3 - 2 = 7.\]

Note that organizing computations in the table as it is done in the previous example is a very useful way of thinking about the global alignment algorithm. Indeed, to find the entry in the $i$th row and $j$th column. We look at: (1) diagonal entry and modify it by adding the score of characters in the $i$th row and $j$th column, (2) entry to the left and modify it by subtracting $\gamma$ and (3) entry above and modify it by subtracting $\gamma$. Then we find maximum of (1), (2), (3) and keep a pointer to the entry which yields maximum.
Figure 2.1: Global alignment

![Global Alignment Table]

Figure 2.2: Global alignment algorithm

![Global Alignment Algorithm Diagram]
Bibliography


