Introduction to Algorithms for Biologists

A one–week course given at the
South African National Bioinformatics Institute (SANBI)
Cape Town, 2–8 July, 2002

Zsuzsanna Lipták
Institute for Theoretical Computer Science, ETH Zurich
zsuzsa@inf.ethz.ch

Contents

1 Introduction

This course was given in July 2002 to bioinformatics masters students with a background in biology. It assumes no prior mathematical or computer science knowledge and tries to concentrate on aspects that are important to biologists starting work in bioinformatics.

The students' programming experience consisted mainly of one intensive Perl course. They did have a lot of experience in using standard bioinformatics tools such as BLAST. The course was given during one week, with 3–hour lectures in mornings, and a 1–2 hour lecture in the afternoons. The rest of the day was spent solving problems. In addition to the contents of this document, we included a brief introduction to graph theory. The course was completed by one "recap day" during which the concepts encountered were reinforced and remaining questions answered. In addition, each student was given a project to present. The students were assessed in a 3–hour written exam.

Each of the following sections is followed by a list of problems. More challenging problems are marked with a star. The students were required to solve all non–star problems. The student projects are listed at the end of this document. Literature used in preparation for this course can be found in the references section. In particular, I borrowed many of the problems from these references, and acknowledge the reference where the problem is stated.

I thank the students Zayed Albertyn, Grant Carelse, Elana Ernstoff, Estienne Swart, Victoria Nembaware, as well as Tzu–Ming Chern and Liza Groenewald for their enthusiastic participation in the course. Further thanks go to Cathal Seoighe for many useful discussions during my preparation, to Win Hide for his support, and to Uli Wagner and Ingo Schurr who read a draft version of these notes and made many helpful comments.

2 What is an Algorithm?

Algorithms. The word *algorithm* derives from the name of the Arab mathematician Al–Khwarizmi who lived in Baghdad from the end of the 8th until the middle of the 9th century. He wrote an influential book on algebra that is considered the first collection of algorithms: It contains "recipes" of how to solve algebraic problems encountered in real life (e.g. in lawsuits, inheritance, land measurements). And this is exactly what an algorithm is: A finite set of instructions on how to solve a given type of problems. The description can be written in any language, e.g. English, Swahili, C++, Perl.

Examples. A recipe for chicken tikka. A description of how to change your tyres. An explanation of how to solve quadratic equations in one variable (i.e., of the type $x^2+px=q$). A Perl script[1] that will convert a text–file into an `html`–file. A C++ program that will generate all prime numbers[2] in ascending order.

Inputs and Outputs. Most algorithms have inputs and outputs, but this is not necessary. In the above examples, the recipe's input are the ingredients, its output is the chicken tikka. The second algorithm can be viewed as having no input or output. Alternatively, one could say that the algorithm's input is a car with a flat tyre, and its output is one with four good tyres. The third's input is an equation, e.g. $x^2-6=10$, and its output are the (possibly several) numbers satisfying the equation, here 4 and -4. The Perl script's input is the text–file, its output the `html`–file. The C++ program has no input; its output is an (infinite) list of numbers.

Finiteness. The word *finite* is important in the definition of an algorithm: The description has to be finite, i.e., the number of characters used for the description has to be a positive integer. This does not mean, however, that each execution of the algorithm will stop after finitely many steps. For example, the C++ program mentioned above has infinite output: If not stopped, it will run forever, since there are infinitely many prime numbers to be generated. However, the program itself will be finite, namely something along the lines of

[1] a program written in the programming language Perl

[2] A positive integer p is called a *prime number* if it has exactly two divisors: 1 and itself. The first seven prime numbers are 2, 3, 5, 7, 11, 13, 17.

Example 2.1.

Specification:	Generate all prime numbers in ascending order.
Step 1:	Look at the first positive integer. (This is 1.)
Step 2:	Test whether it is a prime. If so, output it.
Step 3:	Increase the number you are looking at by 1.
Step 4:	Go back to step 2.

Of course it wouldn't look like this because it would be written in C++ and not in every–day English. But the C++ program would have the same type of instructions, just written in a different syntax. In addition, we would have to specify in detail how the prime number testing is to be done.

Example 2.2. The widely used database search algorithm BLAST is another example of an algorithm. It is given a (biological) sequence as input, does a search against a sequence database, outputting a list of sequences that are judged to be similar to the input sequence. BLAST uses many different ideas and heuristics; a simplified version of its underlying idea is as follows:

Specification:	Find sequences similar to query sequence in a database.
Input:	A sequence (called the *query*).
Step 1:	Break the query up into small pieces and find exact matches of these in the database. Each of the matches will be called a *hit*.
Step 2:	Try and extend each hit such that you get a good alignment with the query.
Output:	The list of the 10 best scoring sequences from Step 2.

The first step is called *exact matching*, while the second step is *inexact* or *approximate matching*. In actual fact, BLAST doesn't really look for exact matches but for near exact matches using hashing, but the idea is similar. We will look at the problem of exact matching in detail and use it to introduce algorithms analysis.

3 Exact Matching

3.1 Strings

Definition 3.1 (Strings)**.** Given a finite set Σ, a *string* over Σ is an ordered succession of elements from Σ, i.e., of the form $s = s(1)s(2)\dots s(n)$, where for $i = 1, \dots, n$, $s(i) \in \Sigma$, and n can be any positive integer, called the *length*

of s and denoted by $|s|$.[3] We refer to Σ as the *alphabet* and to its elements as *characters* or *letters*[4]. $s(i)$ is the i'th character of the string s.

Example 3.1. $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}, s = \mathrm{AACTAG}$. Then, $|s| = 6$, and $s(1) = \mathrm{A}, s(2) = \mathrm{A}, s(3) = \mathrm{G}, s(4) = \mathrm{A}, s(5) = \mathrm{A}, s(6) = \mathrm{G}$.

Example 3.2. The three major types of biological sequences, DNA–strings, RNA–strings, and (primary structure of) proteins, are strings over the alphabets $\{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$, $\{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{U}\}$, and the 20–letter alphabet of the 20 amino acid residues, respectively.

Definition 3.2 (Substrings)**.** Given two strings t and s, where $|s| = n$. Then t is a *substring* of s if there are two positions i, j, $1 \leq i \leq j \leq n$, such that $t = s(i, j)$, where $s(i, j)$ denotes the string stretching from the i'th position of s to its j'th position. Thus, if $|t| = m$, then we require that $t(1) = s(i), t(2) = s(i + 1), \ldots, t(m) = s(j)$. Substrings of s beginning at the first position of s are called *prefixes* of s, and substrings that end at its last position are called *suffixes* of s. Finally, *proper* substrings, prefixes, and suffixes of s are those that are not equal to s itself.

Definition 3.3 (Subsequences)**.** Given two strings t and s, where $|t| = m$, then t is a subsequence of s if there are positions $1 \leq i_1 < i_2 < \ldots < i_m \leq n$ such that $t(1) = s(i_1), t(2) = s(i_2), \ldots, t(m) = s(i_m)$.

Example 3.3. Let $s = \mathrm{AACTAG}$. Then CTA is a substring of s, where the two positions required by the definition are $i = 3, j = 5$. CAG is a subsequence of s, where $i_1 = 3, i_2 = 5, i_3 = 6$. However, it is not a substring of s, since no two positions can be found such that it will match without leaving any letters of s out. AA is a substring of s, with $i = 1$ and $j = 2$. As a subsequence, AA has three occurrences in s: $i_1 = 1, i_2 = 2$; $i_1 = 1, i_2 = 5$; and $i_1 = 2, i_2 = 5$. AACT is a prefix of s, AG is a suffix of s. The full list of prefixes of s is: A, AA, AAC, AACT, AACTA, AACTAG, suffixes: G, AG, TAG, CTAG, ACTAG, AACTAG. All of these are proper substrings of s, except for AACTAG.

Note that the terms *substring, prefix, suffix*, and *subsequence* are relative to another string. Something can either be a string or not, depending on whether it fits the definition of *string*. However, it cannot be a substring as such. A string can only be a substring *of another string*.

[3] Most definitions of strings also allow the *empty string*, usually denoted ε, which has length 0. We do not need the empty string and therefore exclude it in our definition.

[4] "$s(i) \in \Sigma$" is pronounced like this: "$s(i)$ is an element of Σ", or "$s(i)$ is in Σ.

If t is a substring of s, then it is also a subsequence of s, but not vice versa. I.e., there can be subsequences of s that are not substrings, see Example 3.3.

Terminology. The term *string* originates from mathematics/computer science, while biologists usually refer to the same objects by the term *sequence*. The latter has a different meaning in mathematics. However, we will follow the bioinformatics tradition and use these two terms interchangeably. Thus, from now on, string = sequence. We do, however make a distinction between *sub*strings and *sub*sequences. The term *word* is also often used: in mathematical tradition, it means the same as *string*, while in biological literature, it is usually used in the loose meaning "short string". Other terms used are *k–mers* and *k–tuples*; they both mean "strings of length k".

3.2 The Exact Matching Problem

> The Exact Matching Problem: Given two strings s and p, where $|s| = n, |p| = m$, and $m \leq n$, determine whether p is a substring of s. If the answer is yes, output the position(s) where p occurs in s. We will refer to p and the *pattern* as s as the *text*.

Example 3.4. $s =$ `xabxyabxyabxz`, $p =$ `abxyabxz`. Then p is a substring of s, occurring in positions $i = 6$ to $j = 13$.

3.3 A naive algorithm for exact matching

Consider the following algorithm: First, try to match the pattern with the first m positions of s. Then shift the pattern by one and try to match it with positions 2 to $m + 1$ of s, and so on, see Figure 1 for an example. We will call this algorithm naive, because it is the first one that comes to mind.

In this example, the naive algorithm makes $1 + 8 + 1 + 1 + 1 + 8 = 18$ comparisons. In the worst case, however, it could try and match up to the end of the pattern in each iteration. E.g. for $s =$ `aaaaaaaaaaaaa` and $p =$ `aaaaaaab`. Thus, in terms of n and m, the naive algorithm could make up to $m(n - m + 1)$ comparisons.

Here is a pseudo–code for the naive exact matching algorithm[5]:

[5]A note on pseudo–code conventions: I am using a loosely Pascal–based syntax. := stands for an assignment, = for a comparison, and // at the beginning of comments.

Figure 1: The naive algorithm. In each iteration, it tries to align pattern p at a certain position of s. The appropriate positions are compared until a mismatch is found, then the pattern is shifted by one. A vertical line (|) denotes a match, a star ($*$) a mismatch.

1	2	3	4	5	6	7	8	9	10	11	12	13
x	a	b	x	y	a	b	x	y	a	b	x	z
*												
a	b	x	y	a	b	x	z					
	|	|	|	|	|	|	|	*				
	a	b	x	y	a	b	x	z				
		*										
		a	b	x	y	a	b	x	z			
			*									
			a	b	x	y	a	b	x	z		
				*								
				a	b	x	y	a	b	x	z	
					|	|	|	|	|	|	|	|
					a	b	x	y	a	b	x	z

```
algorithm Naive Exact Match
input: two strings p and s

m := |p|; n := |s|;                          // length of input strings
if n < m exit                                // reject stupid inputs
end if;
for j = 1 to n − m + 1 do                    // go through s
  i := 1;                                    // current position in p
  while ( i ≤ m and p(i) = s(j + i − 1)) do  // while positions match,
    i := i + 1                               //    compare next posi-
  end while;                                 //    tions of p and s
  if i = m + 1 then                          // did we match all of p?
    print "Found match at position" j
  end if;
end for;
```

Figure 2: **Improvement 1:** Shift pattern by more than one position.

x	a	b	x	y	a	b	x	y	a	b	x	z
1	2	3	4	5	6	7	8	9	10	11	12	13
*												
a	b	x	y	a	b	x	z					
	|	|	|	|	|	|	|	*				
	a	b	x	y	a	b	x	z				
					|	|	|	|	|	|	|	|
					a	b	x	y	a	b	x	z

3.4 The Knuth–Morris–Pratt algorithm

Let's look once more at Figure 1. Consider the situation after the second iteration. The naive algorithm goes on to try to match p starting at position 3 of s, even though by looking at the pattern, one can see that the next chance to match p is at position 6, since no `a` occurs in s before that. A more intelligent algorithm could use this information and skip the three comparisons $p(1) - s(3), p(1) - s(4)$, and $p(1) - s(5)$ in between: See Figure 3.4.

Furthermore, the three comparisons $p(1) - s(6), p(2) - s(7)$, and $p(3) - s(8)$ at the beginning of the last iteration are not necessary, since in the second iteration, it has already been established that those positions of s match positions $5, 6$, and 7 of p. But $p(1, 3)$ is identical to $p(5, 7)$, and therefore, $s(6, 8)$ must also match $p(1, 3)$.

In order to be able to use this kind of information, we have to take a closer look at the pattern *before* we start the exact matching algorithm with given s. A step like this, that goes before the main algorithm, working only on one of the several inputs, is called *preprocessing*. In this case, we want to preprocess the pattern. We need to gather information on p beforehand that will allow us to introduce the two improvements while running an exact matching algorithm on p and s.

Let us therefore forget s for a while, and just look at p. We define the prefix–function π that takes as arguments positions of p and returns the value by which we will have to shift the pattern if our last match was at this position. Formally:

Figure 3: **Improvement 2:** Do not repeat comparisons where a match has already been established in the previous iteration.

x	a	b	x	y	a	b	x	y	a	b	x	z
1	2	3	4	5	6	7	8	9	10	11	12	13
*												
a	b	x	y	a	b	x	z					
	|	|	|	|	|	|	|	*				
	a	b	x	y	a	b	x	z				
								|	|	|	|	|
					a	b	x	y	a	b	x	z

$$\pi(i) \quad = \quad \text{length of the longest prefix of } p \text{ that is identical to a proper suffix of } p(1,i)$$

If there is no such prefix, we set $\pi(i) = 0$. In other words, given position i, we want to know how long the longest prefix of p is that matches a proper substring of s ending in position i. For example, the last match we found in the second iteration was in position 7 of p, and we want to shift it by 3 positions. This is because the substring of p `abx` is the longest prefix of p that also occurs ending in position 7 of p. Let us compute π for all positions of p:

Figure 4: The prefix–function: π–values for pattern p

i	1	2	3	4	5	6	7	8
$p(i)$	a	b	x	y	a	b	x	z
$\pi(i)$	0	0	0	0	1	2	3	0

We will look more closely at the preprocessing step that computes π in the next section. Given π, we can now formalize the Knuth–Morris–Pratt (KMP) algorithm that employs both Improvement 1 and Improvement 2 of the naive algorithm. The algorithm first tries to match the pattern p at the first position of s. Then, if it finds a mismatch, it looks at the position in p of the last match, i.e., one position back. Let's call this position i. The

Figure 5: The shift in the Knuth–Morris–Pratt algorithm: The first mismatch is printed in bold; i is the position in p of the last match. The next comparison will be made between the mismatched position j in s and position $\pi(i) + 1$.

								j				
x	a	b	x	y	a	b	x	**y**	a	b	x	z
1	2	3	4	5	6	7	8	**9**	10	11	12	13
...												
	|	|	|	|	|	|	|	*				
	a	b	x	y	a	b	x	**z**				
	1	2	3	4	5	6	7	**8**				
			$\pi(i)$				i	$i+1$				
								|	|	|	|	|
					a	b	x	y	a	b	x	z
								$\pi(i)+1$				

algorithm then looks up the π–value of that position, $\pi(i)$, and shifts the pattern by $i - \pi(i)$ or, if $i - \pi(i) = 0$, then by 1; rather than always shifting it by 1 (Improvement 1). The next comparison it makes will be the last mismatched position of s with the position in p just after the end of the matching longest prefix, i.e., position $\pi(i) + 1$ (Improvement 2). After each iteration, it continues in the same way. If it has not found a mismatch in an iteration, i.e., if the whole pattern matches there, then instead of the current position of s it will compare the next one in s after the appropriate shift of p: See Figure 3.4.

We can now present a pseudo–code for the complete algorithm:

```
algorithm Knuth-Morris-Pratt
input: two strings p and s
preprocessing: compute prefix–function π for p

m := |p|; n := |s|;                          // length of input strings
if n < m then exit                           // reject stupid inputs
end if;
i := 0; j := 1;                              // pos. variables: i for p, j for s
while j < n − m + i + 1 do
  while (i < m and p(i + 1) = s(j)) do
    i := i + 1; j := j + 1;
  end while;                                 // Now i and j − 1 are pos.s
                                             //    of last match found.
  if i = m then                              // did we match all of p?
    print "Found match at position" j − i
  end if;
  if i = 0 then j := j + 1
  else i := π(i)                             // shift p for next
  end if;                                    //    iteration
end while;
```

Note: The pattern is never really "shifted", i.e., no copying within the computer takes place. The shifts are just for the visualization of the algorithm.

How many comparisons does the Knuth–Morris–Pratt algorithm make? We could go about the analysis the same way we did for the naive algorithm: KMP goes through at most $n - m + 1$ iterations, and in each iteration it makes at most m comparisons, yielding altogether at most $m(n - m + 1)$ comparisons. However, this analysis vastly overestimates the actual number of comparisons of KMP.

Instead, let us split the comparisons into matches and mismatches. What happens when a position of s is compared? If a match is found, then the algorithm moves onto the next position of s. If a mismatch is found, the algorithm shifts the pattern and compares the same position of s to a new position of p. Note that the algorithm always advances with respect to the current position of s it is working on: Once it has passed a position j of s, every position k of s that is compared later on will be larger than j. So one position of s may be compared several times, but once it is finished with, it

will never be looked at again. The total number of comparisons is

$$\text{no. of comparisons} \;=\; \text{no. of matches} \;+\; \text{no. of mismatches.}$$

The first number is at most n, since s has only n characters that can be compared. The second number is equal to the number of iterations of the algorithm, i.e., the number of times the pattern is shifted. This can happen at most $n - m + 1$ times, thus we have

$$\text{no. of comparisons} \;\leq n + (n - m + 1) \leq 2n.$$

Storage space is only required for storing the prefix–function π, which has m entries. Thus, KMP requires m units of storage space, additionally to the space needed for storing the input.

3.5 Preprocessing of the Knuth–Morris–Pratt algorithm

How do we compute the prefix–function π? We can do it in the following way: First, generate a list of all proper prefixes of p. Next, for each $i = 1, \ldots, m$, compute every proper suffix of $p(1, i)$ and check whether it is in the prefix list. Choose the longest found in the prefix list and set $\pi(i)$ equal to its length. If none is found, set $\pi(i) = 0$.

This is a very inefficient procedure. Depending on the details of the computation, both running time and storage space used will vary. However, we will at least have to look up each substring of s ending in i in our prefix list. Even if we count this lookup as one step (one comparison), it will result in

$$\sum_{i=1}^{m} \text{no. of proper substrings of } s \text{ ending in pos. } i \;=\; \sum_{i=1}^{m} (i-1)$$

$$= \sum_{i=1}^{m-1} i \quad \overset{\text{see Section 5.2}}{=} \quad \frac{m(m-1)}{2} \;=\; \frac{m^2}{2} - \frac{m}{2}$$

steps. The storage space used will be at least the space for listing all proper prefixes of p, and since there is one prefix of length 1, one of length 2, and so on until length $m - 1$, this will yield a storage space of at least $\sum_{i=1}^{m-1} i = \frac{m(m-1)}{2} = \frac{m^2}{2} - \frac{m}{2}$. Thus the running time and the storage space are both proportional to $m^2 - m$, while there are ways of computing the prefix–function π in both time and space proportional to m (see e.g. [1] or [4]).

We will take a closer look at measuring running time and storage space of algorithms in Section 4.

3.6 Preprocessing

Many algorithms use some kind of preprocessing before they tackle the "big job." It is often useful to spend some time on preprocessing beforehand, because typically, the preprocessing will only have to be done once, while the information gathered there can be reused every time the main algorithm is run. For this reason, running time and storage space of the preprocessing step is measured separately from running time and storage space of the main algorithm. Often, one is quite willing to put up with a fairly slow algorithm for the preprocessing, if it allows for a very efficient main algorithm in return.

It may even be the case that an algorithm that has a preprocessing step is more efficient than one that doesn't, even if one adds up the running time and storage space used both in the preprocessing step and the main algorithm. This is the case for the Knuth–Morris–Pratt algorithm: If an "intelligent" preprocessing is used, then the preprocessing and the main algorithm taken together are still faster than the naive algorithm, and use only little more storage space.

In the exact matching problem, one can either preprocess the pattern or the string, depending on the use the algorithm will be put to. For example, if one wants to compare a particular pattern many times, then it can make sense to preprocess the pattern. On the other hand, database search algorithms, where the goal is to find an exact match in a large database of strings, will usually preprocess the database, i.e., the text in our terminology.

3.7 Problems

1. Answer at least two of the following. First try out small examples and then try to generalize your findings to a formula in n.

   (a) What is the maximum number of non–empty substrings a string s of length n can have?

   (b) (*) What is the maximum number of non–empty subsequences a string s of length n can have?

   (c) What is the number of prefixes of a string s of length n? And the number of proper prefixes?

   (d) What is the number of suffixes of a string s of length n? And the number of proper prefixes?

   (e) Given an alphabet of size c (i.e., there are c different characters), how many different strings of length n are there?

2. List all comparisons the naive algorithm makes for $p = \texttt{0001}$ and $s = \texttt{000010001010001}$ (source: [1]).

3. (*) What is the worst–case time of the naive algorithm for finding the first occurrence of p (as opposed to finding all occurrences)?

4. Compute π for $p = \texttt{ababbabbababbababbabb}$ (source: [1]).

5. (a) Work out how many comparisons the KMP–algorithm makes for Example 3.4.

   (b) How many comparisons could it make on inputs of the same size, i.e. $|p| = 9$ and $|s| = 13$?

   (c) Try to find a bad pair of p and s that would force the algorithm to make many comparisons.

4 Algorithm Analysis

The efficiency of an algorithm is measured in two different values: its running time, and its storage space requirements (short: time and space). In addition, if the algorithm has a preprocessing step, like the KMP exact matching algorithm, then one can split the analysis into the time and space required for the preprocessing, and the time and space required for the main algorithm. Storage space is usually measured *in addition* to the storage space required for storing the input. Both time and space are measured relative to the input size. The reason is that one expects an algorithm to take longer and use up more space when it is working on a large input than on a small one. E.g., it is not surprising if the same algorithm takes more time and uses more space when it is matching a pattern p of length 1000 against a text s of length 1000000, than when p has length 5 and s length 20. The crucial question is *how much* more.

There are two principal ways to analyze time and space requirements of an algorithm: *worst–case analysis* and *average–case analysis*. In worst–case analysis, one wants to find an upper bound on the requirements of the algorithm: If the algorithm has worst–case running time, say, $m(n-m+1)$, then we know that it will never make more steps than $m(n-m+1)$, whatever the input. Thus, this yields a *guarantee* that we will never have to wait longer than $m(n-m+1)$ steps. In average–case analysis, one is interested in how long the algorithm takes *on average*. Algorithms may have bad worst–case time, but still run pretty fast most of the time. However, one may be unlucky and have an input that the algorithm will spend much more time on that

its average–case running time. Average–case analysis requires assumptions about the probability distribution of the inputs (do all inputs occur with the same probability or do these differ?), and is thus much more complicated than worst–case analysis.

We will discuss worst–case analysis only.

4.1 Big–O–notation

Consider three algorithms $\mathcal{A}$, with running time n, $\mathcal{B}$, with running time n^2, and $\mathcal{C}$, with running time 2^n. If we have an input of size 10, then $\mathcal{A}$ will do 10 steps, $\mathcal{B}$ will need 100 steps, and $\mathcal{C}$ $2^{10} = 1024$ steps. Now we want to run it on an input twice the size of the first one, i.e., of size 20. $\mathcal{A}$ will need 20 steps, $\mathcal{B}$ will need 400, and $\mathcal{C}$ will run in time $2^{20} = 1048576$. The running time of $\mathcal{A}$ has doubled, that of $\mathcal{B}$ has quadrupled, and that of $\mathcal{C}$ has squared (i.e., $2^{20} = (2^{10})^2$)!

Now let's look at algorithms $\mathcal{A}'$, running in $3n$ steps, $\mathcal{B}'$, that needs $2n^2$ steps, and $\mathcal{C}'$, with running time $\frac{1}{4}2^n$. The running times on an input of size 10 will be 30, and 200, and 256, respectively. On an input of size 20, the algorithms will run in 60, 800, and 262144 steps, respectively. I.e., doubling the size of the input has doubled the running time of algorithm $\mathcal{A}'$, quadrupled that of $\mathcal{B}'$, and more than squared that of $\mathcal{C}'$. In fact, the running time of $\mathcal{C}'$ on input size 20 is 4 times the square of that on input size 10.

The so–called O–notation has been introduced to be able to make statements about the order of growth of functions, or their asymptotic behaviour (i.e., what happens when n becomes very large). The idea is first, to get rid of constant factors such as the 3 in $3n^2$, and second, to be able to concentrate on those terms that dominate the growth behaviour of a function. For example, given an algorithm with running time $5n^2 + 15n + 4$, then, when n grows large, both terms $15n$ and 4 become negligible in comparison with the term $15n^2$ (try $n = 100000$).

Definition 4.1 (O-classes)**.** Given a function $f : \mathbb{N} \to \mathbb{R}$, then $O(f(n))$ is the class (set) of all functions $g(n)$ with the following property:

There exist $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0 :\ g(n) \leq c \cdot f(n)$.

Example 4.1. $10n + 5 \in O(n^2)$.
Proof: Choose $c = 1$ and $n_0 = 11$. We have to show that for all $n \geq n_0$, $10n + 5 \leq 1 \cdot n^2$.

$$10n + 5 \overset{\text{since } n \geq 5}{\leq} 11n \overset{\text{since } n \geq 11}{\leq} n^2.$$

Actually, one can be very generous with both the constant c and the bound n_0. In the last example, we could also have used $c = 1$ and $n_0 = 15$:

$$10n + 5 \overset{\text{since } n \geq 1}{\leq} 10n + 5n = 15n \overset{\text{since } n \geq 15}{\leq} n^2.$$

Lemma 4.2. [6] *For any function f and any positive number C: If $g(n) = C \cdot f(n)$, then $g(n) \in O(f(n))$.*

Proof. Choose $c = C$ and $n_0 = 1$. Then, $g(n) = C \cdot f(n) \leq c \cdot f(n)$, as required. □

Lemma 4.3. *If for all n, $g(n) \leq f(n)$, then $g(n) \in O(f(n))$.*

Proof. Choose $c = 1$ and $n_0 = 1$. □

Note that the converse is not true: e.g., $2n^2 \in O(n^2)$ (choose $c = 2$ and $n_0 = 1$, or use Lemma 4.2). However, $2n^2$ is larger than n^2 for all values of n.

Example 4.2. $5n^2 \in O(2^n)$.
Proof: We have to find c and n_0 such that for all $n \geq n_0$:

$$5n^2 \leq c \cdot 2^n.$$

If we choose $c = 5$, then we still need to show $n^2 \leq 2^n$. But this is true for all $n \geq 4$, so choosing $c = 5$ and $n_0 = 4$ will do.

It is a little more tricky to show that a function is *not* in a particular O–class:

Example 4.3. $n^2 \notin O(n)$.
Proof: Assume otherwise. Then there exist $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$: $n^2 \leq c \cdot n$. Now choose $n_1 := \max(n_0 + 1, \lceil c \rceil + 1)$. (For a positive real number x, $\lceil x \rceil$ is defined as the smallest integer greater than or equal to x.) Then $n_1 > c$ by definition. However, since $n_1 \geq n_0$, it must hold that $n_1^2 \leq c \cdot n_1$. This implies $c \geq n_1$, yielding a contradiction to $c < n_1$.

Example 4.4. $5n^2 + 3n - 5 \in O(n^2)$. *Proof:* Choose $c = 8$ and $n_0 = 1$. Then

$$5n^2 + 3n - 5 \leq 5n^2 + 3n \leq 5n^2 + 3n^2 = 8n^2.$$

In fact, the following lemma holds. I will skip the proof; it is simple but technical.

[6] A lemma is a technical theorem that is typically used to prove "big" theorems.

Lemma 4.4. *If $g(n) \in O(f(n))$, then $f(n) + g(n) \in O(f(n))$.*

Note that a function such as $f(n) = 5$ is also a function of n. It is a constant function, because it does not change with n. For example, if an algorithm uses 5 units of storage space, independent of the input size, then we say that it uses constant space, or $O(1)$ space.

4.2 Important O–classes

Here is a list of the most important functions, ordered by increasing O–classes. Each function f_i is in the O–class of the next function f_{i+1}, but $f_{i+1}(n) \notin O(f_i(n))$.

<table>
<tr><td>1</td><td>$\log\log n$</td><td>$\log n$</td><td>$\sqrt{n}$</td><td>n</td><td>$n \log n$</td><td>n^2</td><td>n^3</td><td>...</td><td>...</td><td>2^n</td><td>$n!$</td><td>n^n</td></tr>
<tr><td>constant</td><td></td><td>logarithmic</td><td></td><td>linear</td><td></td><td>quadratic</td><td>cubic</td><td></td><td></td><td>exponential</td><td></td><td></td></tr>
<tr><td colspan="3"></td><td colspan="6">polynomial (of the form n^c for some constant c)
(all except $n \log n$ are polynomials)</td><td colspan="4"></td></tr>
<tr><td colspan="9">F E A S I B L E</td><td colspan="4">not feasible</td></tr>
</table>

In actual fact, an algorithm that has running time, say, n^{1000}, would not be considered useful, even though it counts as polynomial–time from a theoretical point of view. In real life, one doesn't really want algorithms whose running time is not, say, in $O(n^4)$. Another point to note is that even though any function of the form $c \cdot f(n)$ is in $O(f(n))$, if c is very large, then an algorithm with running time $c \cdot f(n)$ may not be practical. For example, an algorithm with running time $100000n^2$ is quadratic in theory, but will in practice often not be considered quadratic. The reason is that for values of n smaller than 100000, this algorithm has, in fact, at least cubic running time: if $n \leq 100000$, then $100000n^2 \leq n^3$. Just be careful when you are "dropping the constants" that they are not too large!

4.3 Analysis of the exact string matching algorithms

In Section 3, we looked in detail at two algorithms for the Exact String Matching Problem. Our inputs were two strings of length n and m. We found that the naive algorithm takes at most $m(n - m + 1)$ steps and uses no storage space (additional to the that needed for storing the two strings). The Knuth–Morris–Pratt algorithm needed at most $2n$ steps, and used m units of storage space, for storing the prefix–function π.

Note that we now have two input sizes that we want to be able to relate to the time and space requirements of our algorithm: n, the size of the text,

and m, the size of the pattern. We can resolve this in one of two ways: Either, we include both in our O–classes, or we note that since $m \leq n$, we can substitute n wherever we have m, since this will at most increase the function.

We are now able to state the efficiency of our algorithms in terms of O–classes: The naive algorithm has running time $O(n \cdot m)$, and uses constant storage space. The KMP–algorithm has running time $O(n)$ and uses $O(m)$ additional storage space. As mentioned in that section, the preprocessing step of KMP can be done very efficiently, namely in time $O(m)$ and space $O(m)$. Thus, preprocessing and main algorithm of KMP put together have $O(n+m)$ running time and $O(m)$ storage space, which is much faster than naive algorithm and uses only little more space.

In terms only of n, this works out to: The naive algorithm has running time $O(n^2)$ and constant storage space. The KMP–algorithm has running time and storage space $O(n)$. Thus, the one is a quadratic–time algorithm, and the other a linear–time algorithm.

4.4 Problems

1. Which of the following statements is correct? (multiple checks possible)
   ☐ $n^2 \in O(n^3)$ ☐ $n^3 \in O(n^2)$ ☐ $2^n \in O(n^k)$ ☐ $n^2 + n \in O(n^2)$

2. Let $k = 2^x$, what is x then? (multiple checks possible)
   ☐ $x = \log_2(k)$ ☐ $x = 2^k$ ☐ $x = \ln(k)/\ln(2)$

3. (*) The function $f(n)$ is defined recursively as $f(1) = 1$ and $f(n) = n + f(n/2)$ for $n = 2^k, k \in \mathbb{N}$. Which of the following statements is correct? (multiple checks possible)
   ☐ $f(8) = 16$ ☐ $f(n) = \sum_{i=0}^{k} 2^i$ ☐ $f(n) = 2n - 1$ ☐ $f(n) = 2^n - 1$

4. Let $f(n) = \sum_{i=1}^{n} i$. Then $f(n)$ grows
   ☐ linearly ☐ quadratically ☐ exponentially in n.

5. What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine? (source: [1])

6. Order the following functions in "O–increasing" order, i.e., produce a list $f_1, f_2, \ldots, f_n$ such that, for all $i = 1, \ldots, n-1$: $f_i \in O(f_{i+1})$.

   (a) $2^{\log n}$, n^2, $n!$, 2^n, n^3+n+15, $\sqrt{n}$, $\log n$, n^{42}, 5, $\log\log n$, $n \log n$

(b) (*) Now insert the following functions into your list of part (a) (multiple answers possible): $5n,\ n^n,\ 4^n,\ \ln n,\ n^3,\ 37n,\ 37,\ 2^{n+1},\ 2^{2n}$

5 Basic Combinatorics

In this section, we look at important numbers that come up frequently in algorithm analysis.

5.1 Exponentiation and logarithms

What is the number of different strings of length n? Obviously, this depends on the number of letters in the alphabet. Let $|\Sigma| = c$. Then, there are c^n different strings of length n over Σ: You have c choices for the first letter, c choices for the second, c choices for the third, and so on. Thus, when working with a binary alphabet (an alphabet of size 2), e.g. $\Sigma = \{0, 1\}$, then there are 2^n strings of length n.

$$\text{If } k = 2^n, \text{ then } n = \log_2 k.$$

In computer science, one normally just writes $\log x$, meaning $\log_2 x$. The natural logarithm, the logarithm to the base e, is denoted $\ln x$. Note that two logarithms of the same number x to different bases differ only by *a multiplicative constant*, namely

$$\log_a x = \frac{\ln b}{\ln a} \cdot \log_b x.$$

Proof. Let's set $y := \frac{\ln b}{\ln a} \cdot \log_b x$. We want to show that $\log_a x = y$. This is the case if and only if $a^y = x$. Let us write $a = e^{\ln a}$.

$$a^y = (e^{\ln a})^y = e^{\ln a \cdot y} = e^{\ln a \frac{\ln b}{\ln a} \log_b x} = e^{\ln b \cdot \log_b x} = (e^{\ln b})^{\log_b x} = b^{\log_b x} = x.$$

□

The number 2^n occurs frequently in combinatorics, since it is the total number of subsets of a given set S with n elements. Thus, the set $\{a, b, c\}$ has $2^3 = 8$ different subsets[7]: $\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}$, and $\{a, b, c\}$. Likewise, the number of different subsequences of a string s with length $|s| = n$ is $2^n - 1$: A subsequence t of s with length $|t| = m$ is defined by the positions $i_1 < i_2 < \ldots < i_m$. This is equivalent to saying that for

[7] $\emptyset$ denotes the empty set

each position i of s, we either choose that position to be a character of t or don't, yielding 2 choices for each position. Since we are excluding the empty string, we have to deduct 1 from this number. Note the growth behaviour of function $f(n) = 2^n$: Increasing n by 1 will double the value of $f(n)$, since $2^{n+1} = 2 \cdot 2^n$.

5.2 Sum of the first n consecutive integers

Let s be a string of length n. How many different non–empty substrings can s have? Let's classify the substrings according to their lengths: s has one substring of length n, namely itself. It has at most two substrings of length $n-1$, namely $s(1, n-1)$ and $s(2, n)$. "At most" because these two could also be identical, as in $s =$ `aaa`. Likewise, it has at most 3 different substrings of length $n-2$, etc., down to at most n different substrings of length 1. Thus, this adds up to $1 + 2 + \ldots + n = \sum_{k=1}^{n} k$. We have a nice formula for this sum:

$$\sum_{k=1}^{n} k \quad = \quad \frac{n(n+1)}{2}$$

The proof is due to Johann Carl Friedrich Gauss (German mathematician, 1777–1855) who came up with this formula as a child. For the proof write down all numbers from 1 to n twice, once in increasing and once in decreasing order:

1	2	3	$\ldots$	$\ldots$	$\ldots$	$n-1$	n
n	$n-1$	$n-2$	$\ldots$	$\ldots$	$\ldots$	2	1
$n+1$	$n+1$	$n+1$	$\ldots$	$\ldots$	$\ldots$	$n+1$	$n+1$

Adding up each column will result in the sum $n+1$. There are n pairs, so we get a total of $n(n+1)$. This is twice the sum we are looking for, since we now added each number twice, so the original sum evaluates to $\frac{n(n+1)}{2}$.

5.3 n factorial

In how many different ways can one arrange the letters of the word `SCRAMBLE`? Let's say we first decide which letter to put in the first position: We have 8 choices. For each of these choices we have 7 letters to choose from for the second position, etc., until we still have 2 choices, for the second to last position, and only one letter left for the last. Altogether, this gives us

$8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 8! = 40320$ possibilities. In general, $n!$ (pronounce: n factorial) is defined as

$$n! \quad := \quad n(n-1)(n-2)\cdots 3 \cdot 2 \cdot 1$$

By definition, we set $0! := 1$. Note that $n!$ grows very fast with increasing n: $2! = 2, 3! = 6, 4! = 24, 5! = 120, 6! = 720, 7! = 5040, 8! = 40320$. Indeed, it is easy to see that $n!$ grows at least exponentially in n:

Lemma 5.1. *For all $n \geq 4$, $\quad 2^n \leq n! \leq n^n$.*

Proof. We prove the two inequalities separately. First, look at the second and third terms. Since we have the same number of (positive) factors on both sides, and each one on the left is smaller than or equal to its counterpart on the right, we can deduce that the product on the left is smaller than or equal to the product on the right:[8]

$$n! = \underbrace{n(n-1)(n-2)\cdots 2 \cdot 1}_{n \text{ factors}} \quad \leq \quad \underbrace{n \cdot n \cdots n}_{n \text{ factors}} = n^n$$

Now to the second inequality in Lemma 5.1: Again, we have n factors on each side, and we want to show that:

$$2 \cdot 2 \cdots 2 \quad \leq \quad n(n-1)(n-2)\cdots 2 \cdot 1.$$

Since $n \geq 4$, each of the factors on the right is greater than or equal to its counterpart on the left, except for the last one, which is 2 on the left and 1 on the right. However, $\frac{n}{2} \geq 2$, since $n \geq 4$, and thus

$$\begin{array}{lrcll} & \underbrace{2 \cdot 2 \cdots 2}_{(n-1) \text{ times}} & \leq & \frac{n}{2}(n-1)(n-2)\cdots 2 & \text{multiply both sides by 2} \\ \Rightarrow & \underbrace{2 \cdot 2 \cdots 2}_{n \text{ times}} & \leq & n(n-1)(n-2)\cdots 2 \cdot 1 & \\ \Rightarrow & 2^n & \leq & n! & \end{array}$$

and we are done. □

[8] In general terms: For positive numbers $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n$: If for all $i = 1, \ldots, n$, $a_i \leq b_i$, then $a_1 \cdot a_2 \cdots a_n \leq b_1 \cdot b_2 \cdots b_n$.

5.4 Binomial coefficients

Given a group of n people, if each two are going to shake hands, how many handshakes will there be altogether? For 2 people, it is 2 handshakes, for 4, there will be 6, and for 10 people, 45 handshakes. The question is this: How many different pairs of people are there in a group of n? Say, we choose the first one: we have n choices. Now we have still to choose the second person he or she is going to shake hands with: $n-1$ choices. This would give us $n(n-1)$ pairs. However, now we have counted each handshake twice: if person A and B are shaking hands, then we have counted (A,B) and (B,A) separately. Thus, we have to divide the number $n(n-1)$ by 2: There will be $\binom{n}{2} := \frac{n(n-1)}{2}$ handshakes. In general,

$$\binom{n}{k} \quad := \quad \frac{n!}{k!(n-k)!}$$

is the number of k–sets in an n–sets, i.e., given a set S with n elements, $\binom{n}{k}$ is the number of different subsets of S that have exactly k elements.[9] For example, $\binom{49}{6}$ is the number of different combinations you can put on a lotto slip, so your chances of winning are $\frac{1}{\binom{49}{6}}$, which is about one in 2 million!

We can now give a second proof for the fact that a string s of length n can have at most $\frac{n(n+1)}{2}$ substrings. First note that we can find all substrings of s by looking at all pairs of positions (i,j) where $1 \le i \le j \le n$: the beginning and end of the substring. E.g., if $s =$ `babbbc`, then the pair $(1,2)$ will yield `ba`, the pair $(2,6)$, `abbbc`, etc. Again, two different pairs of positions will not necessarily yield different substring, as can be seen by $(3,4)$ and $(4,5)$ both yielding `bb`, so we are merely getting an upper bound on the number of substrings. How many of these pairs (i,j) are there? Distinguish two cases: $i = j$, these are the substrings of length 1, and $i < j$, those with length greater than 1. Clearly, there are n different pairs of the first type, namely $(1,1),(2,2),\ldots,(n,n)$. For the second type, we have to choose two different positions between 1 and n, i.e., we have to choose two elements from a set of n elements. Thus, there are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of the second type. Altogether, we get

$$\text{no. of substrings of } s \quad = \quad n + \frac{n(n-1)}{2} = \frac{2n + n(n-1)}{2} = \frac{(n+1)n}{2}.$$

[9] $\binom{n}{k}$ is pronounced "n choose k."

5.5 Further useful formulas

Here follow a few formulas involving the numbers introduced. The proofs can be found in any introductory book on combinatorics.

1. Pascal–triangle equality: $\binom{n}{k} + \binom{n}{k-1} = \binom{n+1}{k}$.
2. Symmetry of binomial coefficients: $\binom{n}{k} = \binom{n}{n-k}$.
   (*Hint:* Look at the definition.)
3. Binomial theorem: For $a, b \in \mathbb{R}$: $(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$.
4. Sum of binomial coefficients: $\sum_{k=0}^{n} \binom{n}{k} = 2^n$.
   (*Hint:* Use the binomial theorem with $a = b = 1$.)
5. Sum of consecutive powers of 2: $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$.

5.6 Problems

1. Say, the dean specifies that each student should take exactly 4 of the 7 courses offered. The lecturers report the following numbers of students who subscribed to their course: $51, 30, 30, 20, 25, 12$, and 18. What does this tell us? (source: [2]).
2. (*) There are 151 seats in a country's parliament, and 3 parties A, B, and C. How many different distributions of seats are possible that exclude an absolute majority[10] of any party. (source: [2]) (*Note:* We count the seat distribution $(40, 60, 51)$, meaning that party A has 40 seats, party B 60, and party C 51, as different from the case $(60, 40, 51)$.)
3. How many different words can be built by using all the letters of the word `ALGORITHM` exactly once? Compute the actual value.
4. (*) How many different words can be built by using all the letters of the word `ABRACADABRA` exactly once? (source: [2]) You do not need to compute the actual value. How does this compare to the number of words that can be built using all letters exactly once of the word `COPYRIGHTED`?

[10] *absolute majority*: more than half of the seats, in this case 75.

6 Individual Projects

The aim of having each student present an algorithm is twofold: Firstly, the presenter learns to analyze an algorithm independently. Secondly, all students learn about important concepts in a short time. It is therefore important to insist that the students prepare handouts. Each presentation should include an example. The presentations should take at most 30 minutes each. *Note:* All these sorting algorithms and the binary search algorithm are described in any algorithms book or any introductory computer science book. In addition, there are lots of descriptions on the web, including many nice animations that can be very helpful in understanding how the algorithms work.

1. Explain and analyze the Bubble Sort algorithm. When analyzing the running time, first count the number of comparisons and swaps separately, and then add them up.

2. Explain and analyze the Selection Sort algorithm. When analyzing the running time, first count the number of comparisons and swaps separately and then add them up.

3. Explain and analyze the Insertion Sort algorithm. When analyzing the running time, first count the number of comparisons and swaps separately and then add them up.

4. Explain and analyze Binary Search. When analyzing running time, count the number of comparisons.

5. Explain in detail and analyze the simple (inefficient) preprocessing of the Knuth–Morris–Pratt algorithm.

6. (*) Explain and analyze the intelligent preprocessing of the KMP–algorithm. *Note:* Use [1] for preparing this presentation. The preprocessing is also explained in [4], but a slightly different prefix–function is used. Do not browse the web to find other preprocessing algorithms, because there are many different ones out there, all called Knuth–Morris–Pratt, and it will only confuse the other students.

7 Conclusion

An algorithm is a recipe for solving a certain type of problem. Given an algorithm, after having checked its correctness (does it do what it should?) and whether it terminates (does it always stop?), one needs to analyze its efficiency (how efficiently does it do the job?). Efficiency is measured in running time and storage space requirements, measured relative to the input size. If the algorithm has a preprocessing phase, then time and space of the preprocessing and time and space of the main algorithm are measured separately. Worst–case analysis delivers an upper bound on these values: The algorithm is guaranteed to take at most this number of steps and this much storage space, on any input of a given size.

This upper bound is given in form of a function of n, if n is the input size. One is mainly interested in the growth behaviour of these functions, thus we express them using O–classes. The most important classes are, in increasing order: constant ($O(1)$), logarithmic ($O(\log n)$), linear ($O(n)$), $n \log n$ ($O(n \log n)$), quadratic ($O(n^2)$), cubic ($O(n^3)$), and exponential ($O(2^n)$). Classes of the form $O(n^c)$ are called polynomial classes, and algorithms with running time $O(n^c)$ are called polynomial–time algorithms. In real life, only those polynomial–time algorithms are considered efficient whose running time and storage space is $O(n^c)$ with c small, say, not greater than 4.

References

[1] Thomas H. Cormen, Charles E. Leiserston, Ronald L. Rivest, *Introduction to Algorithms.* MIT Press and McGraw–Hill, 1990.

[2] Martin Aigner, *Diskrete Mathematik.* Vieweg, 1996.

[3] João Setubal and João Meidanis, *Introduction to Computational Molecular Biology.* PWS Boston, 1997.

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