Algorithms course book

Version 1.3 (20 June 2009)

Free University of Bolzano Bozen – A.Y. 2008-09 – Prof. Paolo Coletti

Introduction

This book contains the Introduction to Information Science course's lessons held at the Faculty of Science and Technology of the Free University of Bolzano Bozen. It contains only the last parts of the course for Engineering students, namely the lessons on:

- programming techniques,
- computational complexity,
- typical problems and algorithms,
- cryptography.

It does not contain the parts on Microsoft Windows, Word, Excel, Access, Visual Basic for Applications, on basic computer, computer networks, computer dangers and security and SQL language which are very well covered by the course's suggested books.

This book is in continuous development, please take a look at the version date.

Disclaimers

This book is designed for novice programmers. It often contains oversimplifications of theory and many technical details are purposely omitted. Experts will find this book useless and, for certain aspects, questionable.

Table of contents

1. Programming techniques

This chapter presents a brief description of some widely used programming techniques.

When the problem has a finite number of possible situations, the easiest technique is a brute force attack to the problem, which consists in exploring all the possible solution's candidates until one which satisfies every problem's request is found. This technique can be implemented straightforwardly and represents the basic algorithm, giving at the same time proof that the solution can be found. Many times brute force can be improved skipping automatically obvious wrong candidates, but usually it is not improved further this step, and sometimes other programming techniques can reach the solution much faster.

1.1. Recursion

In mathematics recursion is a method of defining functions in which the function being defined is applied within its own definition. In practical applications, recursion must obviously have a termination condition and the application must be simpler than the original, where simpler means that it must be closer to the termination condition. For example, the factorial of a number, n!, may be calculated using an iterative rule as

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

or may be calculated with a recursive rule as

$$n!=n\cdot(n-1)!\text{ and }1!=1.$$

In general, every iterative rule can be derived expanding the corresponding recursive rule until the termination condition is met. However, in many cases the iterative formula is not as simple as the one of this example while the recursive formula is usually very compact.

In computer science, recursion can be implemented through a recursive function. This function checks first whether the termination condition is met and, if not, it recalls itself passing different parameters.

```
Function factorial(n As Long) As Long
        If (n = 1) Then
                factorial = 1
        Else
                factorial = n * factorial(n-1)
        End If
end Function
```

This function is thus able to suspend its execution (storing all the temporary values, such as the value of n, in a special memory area) and call itself again with another parameter, continuing this self-calling until n is equal to 1. Obviously this implies the automatic building of a stack of values in the computer memory to keep track of all the functions which were called but left unfinished. When n is finally 1, the last called function returns its value (in the factorial case it is 1) to the second last, which performs the suspended operations (in the factorial case, it multiplies 2 by the received value) and returns the calculated value to the third-last function (in the factorial case, the returned value is $1\cdot 2$, which gets multiplied by 3 by the third last function) and so on until the value is returned to the original first called function.

1.1.1. Exercises

Build a recursive and an iterative/direct program to calculate these functions.

- f(n) = f(n-1) + 10; f(1)=0; n is any positive integer number;
- f(n) = f(n-1) + n; f(1)=0; n is any positive integer number;

- f(n) = 2·f(n-1); f(1)=1; n is any positive integer number;
- f(x) = f(x/2) + 10; f(x)=0 for each x<1; x is any positive real number;
- f(x) = f(x/2) + x; f(x)=0 for each x<1; x is any positive real number;
- f(x) = 2·f(x/2) + 10; f(x)=0 for each x<1; x is any positive real number;
- f(x) = 3·f(x/2). ; f(x)=0 for each x<1; x is any positive real number;

1.2. Divide and conquer

The divide and conquer technique works by recursively breaking a problem into two or more sub problems of the same type until these become simple enough to be solved directly. The solutions of the sub problems are then combined giving the solution of the original problem.

For example, if we want to find the largest element in a list of numbers, we can use the iterative algorithm scanning all the elements remembering the largest one

```
Function largest (list As Single()) As Single
        largest = list(LBound(list))
        For i = LBound(list)+1 To
UBound(list)
                If (list(i) > large) Then
                        largest = list(i)
                End If
        Next i
End Function
```

or we can build a divide and conquer algorithm which first checks that the array has more than one element, then divides it in two parts and finds the largest element of both parts. In order to find the maximum of each part the function obviously recalls itself, thus setting on a recursive process with the termination condition that the length of the array be 1.

```
Function largest (list() As Single, a as Integer, b as Integer) As Single
        If (a=b) Then
                largest = list (a)
        Else
                Dim middle As Integer
                middle= (a+b) \ 2 'note that division is not / but \ which is integer division
                largest=WorkSheetFunction.Max(largest(list, a, middle), largest(list, middle+1, b) )
        End If
End Function
```

1.3. Reduction

The reduction technique transforms the problem into a better known problem for which we already have working algorithms. It can be used also to simply reducing the amount of code to write relying on functions already written and tested. This reduction may be a smart use of a mathematical formula or a programming technique.

For example, to calculate n! / m!, with m smaller than n, instead of applying twice the factorial function, which requires a lot of multiplications, we can mathematically simplify the problem and calculate only $n \cdot (n – 1) \cdot (m + 1)$. Another mathematical transformation can be applied when we want to calculate $\log_b x$ but the only available logarithm function calculates only natural logarithms:

we can apply the base transformation formula for logarithms and calculate ln x / ln b, using an already present and optimized function.

An example which does not use an a priori mathematical transformation is the calculation of the median[1] for an unsorted list of numbers. The brute force approach is to find the largest and the smallest numbers and exclude them from the list, then find again the smallest and largest and exclude them from the list and so on until one or two values remain. However, finding the largest and smallest so many times is very time consuming and an easier way to solve this problem can be ordering the set using one of the many sorting algorithms (see section 3.3 on page 14) and then taking the value in the middle position.

1.3.1. Exercises

Build algorithms which solve these three problems with and without the reduction technique.

[1] The median, not to be confused with the mean, is a statistical centrality measure which, for a finite set of numbers, is defined as the value which splits the ordered set into two equal parts. If the number of values is odd the median is well defined, if the number is even we may take anyone of the two central values.

2. Computational complexity

Whenever we run a program we are using computer's resources, which are a limited amount even through in constant grow every year. When the amount of input data is very large, for example a large set of numbers to sort or a large number of possibilities to choose from, the algorithm must do an extremely large amount of operations, typically multiplications, divisions and comparisons, and it must store the original data and all the temporary data into its memory.

Therefore it is necessary to study in advance the amount of time, expressed usually in the form of binary operations, and the amount of space, expressed usually in the form of used bytes, that the algorithm requires with respect to the original input. Sometimes these quantities grow proportionally, and in these cases the algorithm can handle also very large amounts of input data, other times these quantities increase more than proportionally and in these cases the algorithm takes too much time or uses too much memory when the amount of input data is large.

2.1. Binary numbers

Computers think through a series of logical electronic circuits whose only possible states are on and off and therefore their reasoning unit is the bit and all their arithmetic and logics is binary.

2.1.1. Integers' representation

The procedure to convert a number in base 10 to its corresponding in base 2 is the following:

1. divide the number by 2 using an integer division[2], write down the remainder as the last binary digit;
2. use the quotient as the new number and return to the previous operation;
3. stop when the number is 0.

Since remainders may only be 0s or 1s, the result of this operation is a binary number. For example, converting 13 is:

- divide 13 by 2 and obtain 6 with the rest of 1. Write down 1 and use the 6 in the next step;
- divide 6 by 2 and obtain 3 with the rest of 0. Write down 0, which now makes 0 1, and use 3 in the next step;
- divide 3 by 2 and obtain 1 with the rest of 1. Write down 1, which now makes 1 0 1, and use 1 in the next step;
- divide 1 by 2 and obtain 0 with the rest of 1. Write down 1, which now makes 1 1 0 1, and use 0 in the next step;
- the procedure is over since the number is now 0. The result is 1101.

To convert a binary number into a decimal number, the procedure is:

1. multiply each number by 2^{k-1} where k is its place in the binary format;
2. sum all the results.

For example, starting from 1101:

- we multiply 1 by 2^0, which is 1, obtaining 1;
- we multiply 0 by 2^1, which is 2, obtaining 0;
- we multiply 1 by 2^2, which is 4, obtaining 4;
- we multiply 1 by 2^3, which is 8, obtaining 8;
- we sum the results 1+0+4+8 obtaining 13.

It is also easy to determine how many bits does a decimal number n use. The formula simply calculates how many steps the conversion algorithm will take: $\text{int}(\log_2 n) + 1$, or, using natural

[2] An integer division is a division of an integer by an integer which produces an integer result. Many times its result, the quotient, is not exact and produces also a remainder which is always smaller than the divisor. For example, dividing 13 by 5 produces a quotient of 2 and a remainder of 3 (which is smaller than the divisor 5).

logarithms, int(ln n/ln 2) + 1. The reverse formula, 2^k-1 is the maximum number which may be represented using k bits.

In order to represent negative numbers a bit is used for the sign, and the maximum range goes from $-(2^{k-1}-1)$ to $+(2^{k-1}-1)$.

In modern programming languages usually the integer types which may be used are:

- boolean, which uses 1 bit to represent 0 and 1;
- integer, which uses 16 bits (2 bytes) and can therefore represent integer unsigned numbers from 0 to 65535 or signed numbers from –32767 to +32767;
- long, which uses 32 bits (4 bytes) and can represent signed numbers from –2,147,483,647 to +2,147,483,647.

2.1.2. Exercises

Build a function which takes an integer number and returns its binary representation (for example as an array of integers or as an array of booleans). Build another function which, using the previous one, build a string containing the binary representation.

Build a function which takes an array of integers "0" and "1" and returns the corresponding integer number.

2.1.3. Real numbers' representation

In order to represent a real number computers adopt a floating point notation, which is a representation that does not cover all the infinite possible real numbers but is able to represent every number with a constant relative precision. This means that very small numbers are represented with a fixed amount of decimal digits preceded by many zeros and very large numbers are represented with the same amount of digits followed by many zeros.

For example, using 7 digits of precision, we can represent the real numbers:

- π as 3.141592,
- 1/350 as 0.002857142,
- e^5 as 146.1660,
- e^{35} as 1425362000000000.

It is clear that these representations are not exact, but only approximated since they leave out the less significant digits, rounding the number in such a way to use only 7 significant digits.

Moreover, numbers are represented, using appropriate powers of 10, with a single digit before the decimal points and the rest of the significant digits after, for example:

- π as $3.141592 \cdot 10^0$,
- 1/350 as $2.857142 \cdot 10^{-3}$,
- e^5 as $1.461660 \cdot 10^2$,
- e^{35} as $1.425362 \cdot 10^{15}$.

Therefore this notation uses for every number a constant number of decimal digits: the precision plus the maximum allowed number of digits for the power of 10. These decimal digits are in fact two integer numbers[3], called significand and exponent, which may be represented using binary numbers exactly in the same way as two integer numbers. In fact, the most common real number types in modern programming languages are:

- single, which uses 32 bits (4 bytes): 1 bit for sign, 23 for the significand, 1 bit for exponent's sign and 7 for the exponent value, thus going from $-8.388607 \cdot 10^{127}$ to $-0.000001 \cdot 10^{-127}$, including 0 and going from to $0.000001 \cdot 10^{-127}$ to $8.388607 \cdot 10^{127}$;

[3] The first number is not officially integer, but the computer may use it as integer and simply remember to put a decimal dot after the first digit.

- double, which uses 64 bits (8 bytes): 1 bit for sign, 52 for the significand, 1 bit for exponent's sign and 10 for the exponent value, thus having $4.503599627370495 \cdot 10^{1023}$ as the largest positive number and $0.000000000000001 \cdot 10^{-1023}$as the smallest positive number.

2.1.4. Exercises

Build a function which takes an array of 32 integers "0" and "1" and returns the corresponding real number.

Build a function which takes a real number and returns its binary representation using 32 bits (for example as an array of integers or as an array of booleans). Build another function which, using the previous one, build a string containing the binary representation. Note: this exercise is quite hard.

2.1.5. Strings representation

A string is a sequence of symbols which may be digits, capital or small letters, punctuation symbols, and sometimes particular symbols (such as accented letters or mathematical symbols) or control characters (such as "end of paragraph" or "end of line"). Each of these symbols, generally called characters, is converted into an integer number using coding tables, the most famous being the ASCII-7 and ASCII-8 (American Standard Code for Information Interchange) which have 127 or 255 symbols[4] and the UTF-8 (Unicode Transformation Format) which uses for each character a variable amount of 1 to 4 bytes, each using 8 bits and UTF-16, which uses a variable amount of two bytes. These integer numbers may then be easily represented by the computer in binary format.

Clearly ASCII coding is very limited, representing only the most common Latin letters and computer symbols, while the UTF coding can represent other alphabets and advanced mathematical or musical symbols.

Character	ASCII-7 Code	Character	ASCII-7 Code	Character	ASCII-7 Code	Character	ASCII-7 Code	Character	ASCII-7 Code
Control	0	Control	26	4	52	N	78	g	103
Control	1	Escape	27	5	53	O	79	h	104
Control	2	Control	28	6	54	P	80	i	105
End of text	3	Control	29	7	55	Q	81	j	106
Control	4	Control	30	8	56	R	82	k	107
Control	5	Control	31	9	57	S	83	l	108
Control	6	Space	32	:	58	T	84	m	109
Control	7	!	33	;	59	U	85	n	110
Backspace	8	"	34	<	60	V	86	o	111
Horizontal tab	9	#	35	=	61	W	87	p	112
Line feed	10	$	36	>	62	X	88	q	113
Vertical tab	11	%	37	?	63	Y	89	r	114
Form feed	12	&	38	@	64	Z	90	s	115
Carriage return	13	'	39	A	65	[	91	t	116
Control	14	(	40	B	66	\	92	u	117
Control	15	)	41	C	67	]	93	v	118
Control	16	*	42	D	68	^	94	w	119
Control	17	+	43	E	69	_	95	x	120
Control	18	,	44	F	70	`	96	y	121
Control	19	-	45	G	71	a	97	z	122
Control	20	.	46	H	72	b	98	{	123
Control	21	/	47	I	73	c	99	|	124
Control	22	0	48	J	74	d	100	}	125
Control	23	1	49	K	75	e	101	~	126
Cancel	24	2	50	L	76	f	102	Delete	127
Control	25	3	51	M	77				

[4] plus the symbol used as "string termination" control symbol.

2.2. Binary operations

Since the computer performs only operations with binary numbers, we need to investigate how many single binary operations does a decimal number operation require. We will suppose that comparing two bits, adding two bits and multiplying two bits all require the same amount of time, here called "bit operation", since in fact these basic operations are performed by very similar electronic circuits.

The comparison, in order to check whether two unsigned numbers of k bits are equals, requires a maximum of k binary comparisons, since the operation must simply check that the bits have all the same value. Also the comparison to check which is the largest requires a maximum of k comparisons: the procedure needs simply to start from the left and stop when a number has a 0 and the other has a 1, and is thus the largest, but in the unlucky case that the difference lies in the last bit it must all the k bits.

The addition of two unsigned binary numbers is done in columns exactly as the elementary addition of decimal numbers. It therefore requires k bit operations plus the k additions of 1s or 0s carried on from the previous column. For example, summing 6, which is 110, with 13, which is 1101:

"carried on"	1				
first addendum	0	1	1	0	+
second addendum	1	1	0	1	=
	---	---	---	---	---
result	1	0	0	1	1

The multiplication is also performed using the well known elementary procedure, which requires k^2 binary multiplications and $k \cdot 2k$ additions for a total of $3k^2$ bit operations. For example multiplying 3, which is 11, by 5, which is 101:

first factor	0	1	1	×
second factor	1	0	1	=
	---	---	---	---
	0	1	1	+
	0	0	0	- +
	0	1	1	- - =
	---	---	---	---
Result	0	1	1	1 1

Subtraction is very similar to addition since bit subtraction takes the same time as bit addition. Signed numbers multiplication takes only one bit operation more to get the correct sign. Division has a very similar number of bit operations to multiplication.

Operations on floating point numbers use a similar number of bit operations which depends[5] on k for comparisons and additions and on k^2 for multiplications and divisions. The main difference lies in the fact that floating point numbers usually have more bits and therefore their operations are always more expensive.

There can be other time consuming operations, such as function calling or even simple instructions and memory access. Usually the time required to perform these operations is negligible compared

[5] Multiplication of floating point numbers is in fact cheaper than $3k^2$ since exponent and significand are multiplied separately, but the important aspect, as can be seen in section 2.2.2, is that the number of bit operations goes as k^2.

with comparisons, additions and especially multiplications and therefore we will consider it only for algorithms which do not contain any other heavier operation.

2.2.1. Worst and average case

As we have already experienced with binary addition, the number of requested operations is not always fixed but depends on our luck, other than on the number of bits. For the addition, for example, the number of operations ranges from k to 2k according to the number of carried on 1s.

We will always consider the largest possible number of operations and thus analyze the worst case as the most interesting one. Sometimes, when there is a significant difference between the worst theoretical case and what happens in practice, we will analyze also the average case, which for example in the case of additions requires 3k/2 operations and for the equality comparison requires k/2 operations.

Another possible analysis that we will do when it is significantly different from the worst and average cases is the solution verification. For many problems verifying that the solution is correct is much easier and faster than finding it. An example is the square root: finding its value is a hard task which requires advanced mathematical techniques and many calculations, while verifying the solution simply requires a multiplication.

2.2.2. Big Oh notation

Computational complexity studies the time and space requirements of algorithms when the amount of input data becomes very large. Therefore we are interested only in the behaviour of the functions when n, which represents the amount of input data, goes to infinity. We define these relations between f(n) and g(n), two functions of n which go to infinity when n goes to infinity:

- we say that $f = O(g)$ when there is a positive constant c such that $f(n) \leq c \cdot g(n)$ for every sufficiently large n; this is equivalent to saying that $\lim f(n)/g(n)$ when n goes to infinity is a finite number. In this case f(n) goes to infinity slower or with the same speed as g(n);
- we say that $f = \Omega(g)$ when $g = O(f)$. In this case f(n) goes to infinity faster or with the same speed as g(n);
- we say that $f = \theta(g)$ when $f = O(g)$ and $g = O(f)$. In this case f(n) goes to infinity with the same speed as g(n);
- we say that $f = o(g)$ when, for every positive ε, $f(n) \leq \varepsilon \cdot g(n)$ for every sufficiently large n; this is equivalent to saying that $\lim f(n)/g(n)$ when n goes to infinity is zero. In this case f(n) goes to infinity slower than g(n);
- we say that $f = \omega(g)$ when $g = o(f)$. In this case f(n) goes to infinity faster than g(n).

Example: for the function $f(n) = 100 \cdot n \cdot \log n$ and $g(n) = n^2$ we may calculate $\lim f(n)/g(n) = \lim 100 \cdot \log n / n$ which, using L'Hôpital rule[6], becomes $\lim 100 \cdot 1 / n = 0$. Therefore $f = o(g)$ and we can easily deduce that log n goes to infinity slower than n regardless of the multiplicative coefficient in front. This is a general rule: if $f = o(g)$ every coefficient in front of f does not change the relation.

Example: $f(n) = 3 \cdot n^2 + 5 \cdot n + 2 \cdot \log n$ and $g(n) = n^2$. Calculating $\lim f(n)/g(n) = \lim 3 + 5 / n + 2 \cdot \log n / n^2 = 3$ and therefore $f = O(g)$. Calculating $\lim g(n)/f(n)$ using L'Hôpital rule twice we get $\lim (2 \cdot n) / (6 \cdot n + 5 + 2 / n) = \lim 2 / (6 - 2 / n^2) = 1 / 3$ and therefore $f = \theta(g)$. This is also a general rule: any slower function added to the fastest function does not modify its big Oh relations.

Example: $f(n) = \min \{ n , 1000 \}$ and $g(n) = \log n$. Calculating $\lim f(n)/g(n) = \lim \min \{ n , 1000 \} / \log n$. Since we are considering the limit when n goes to infinity, after 1001 f(n) is always 1000 and therefore the limit is $1000 / \log n = 0$. Therefore $f = o(g)$. Another general rule is that the behaviour

[6] L'Hôpital rule can be used when the limit has the undetermined form 0/0 or ∞/∞, i.e. the nominator and denominator both go towards 0 or infinity. In this case derive the nominator and denominator independently and recalculate the limit: if it is still an undetermined form 0/0 or ∞/∞ the rule may be applied again.

of the function for the early n does not count and only the behaviour for the very large n must be taken into account.

Example: $f(n) = 2^n$ and $g(n) = n^5$. Calculating $\lim f(n)/g(n)$, using L'Hôpital rule five times[7], we get $\lim 2^n / n^5 = \lim 2^n \cdot \ln 2 / 5 \cdot n^4 = \lim 2^n \cdot (\ln 2)^2 / 20 \cdot n^3 = \lim 2^n \cdot (\ln 2)^3 / 60 \cdot n^2 = \lim 2^n \cdot (\ln 2)^4 / 120 \cdot n = \lim 2^n \cdot (\ln 2)^5 / 120 = \infty$. Calculating $\lim g(n)/f(n)$ in the same way we get 0. Therefore $f = \omega(g)$.

It is easy to see that there are some classes of functions which can be put in sequence of increasing speed. The elements of a class are $\theta(\ )$ among themselves and each one is $o(\ )$ of the elements of the next class:

1. the classes of logarithmic functions $\theta(\log_b n)$ with $b > 1$, from the largest bases b to the smallest ones;
2. the classes of polynomial functions $\theta(n^c)$ with $c > 0$, from the smallest exponents c to the largest ones;
3. the classes of exponential functions $\theta(c^n)$ with $c > 1$, from the smallest c to the largest ones;
4. the class of factorials $\theta(n!)$;
5. the classes of exponential of exponential functions $\theta(n^{cn})$ with $c > 0$, from the smallest exponents c to the largest ones.

2.2.3. Exercises

Find the relations among these functions:

n^2	$2n^2 + 100\sqrt{n}$	n^{100}	$2^{n/100}$	$2^{n^{1/100}}$
$\sqrt{n}$	$2^{\sqrt{\log n}}$	$2^{(\log n)^2}$	$n \log n$	$1000n$

2.3. Classes

Each algorithm uses a maximum (and an average) amount of time and space. These quantities are called complexity and space-complexity. They are never expressed as the exact number of maximum binary operations or bits used by the algorithm, which is often a very complicated formula, but more simply with an easy θ equivalent function of the amount n of input data. For example, a multiplication of two integer numbers has a complexity, expressed in terms of bits' operations, of $\theta(k^2)$ for two numbers of no more than k bits or $\theta((\log_2 n)^2)$ for two numbers smaller than n.

Since we are considering the maximum number of operations and since we will always make estimations by excess, we will use the $O(\ )$ notation instead of the $\theta(\ )$ notation since the real implementation of the algorithm may be cheaper than our estimation.

When dealing with algorithms, many times it will be more convenient to use a unit of measure larger than the bit operation, such as the most expensive operation involved. For example, we will measure the complexity of a real numbers' sorting algorithm, which needs to perform a lot of comparisons, with the number of singles', or doubles', comparisons and its space-complexity with the number singles, or doubles, that the algorithm must store.

The complexity of a problem is the smallest complexity among all the algorithms able to solve that problem. It is clear that, while there is a clear upper bound for the complexity of a problem given by the current known algorithms, there is usually no lower limit since a new cheaper algorithm can be invented in the future.

[7] The derivative of an exponential c^n is $c^n \cdot \ln c$, where ln is the logarithm in base e.

2.3.1. P

P is the class of problems, or algorithms, which can be solved maximum in a polynomial time or faster. P-SPACE is the class of problems, or algorithms, which require maximum a polynomial amount of space or less.

A typical representative of this class is the problem of finding the maximum in an unsorted list of length n. The easiest algorithm is:

- take the first element as solution candidate;
- go through the elements from the second to the last one comparing them with the candidate: the largest is chosen as candidate;
- the remaining candidate after the last comparison is the largest number in the list.

Using the comparison of two numbers as unit of measure, the complexity of the algorithm is $n - 1$ which means $O(n)$. This problem is very simple and it is clear that the complexity of the problem can not be further reduced, since being the list unsorted all the numbers must be compared at least once to find the maximum.

Space-complexity of this algorithm is $n + 1$ which means $O(n)$. Space-complexity for this problem can be reduced, using the space of the first element of the list to store the candidate, to n which is still $O(n)$. It may not be further reduced since the number of input elements is n and they must be stored. This is also a general rule: the space-complexity can never be less than the space necessary to store the input[8].

2.3.2. NP

NP is the class of problems, or algorithms, which can be solved, in the worst case, in a larger than polynomial time or faster, but whose solution verification requires no more than a polynomial time.

A typical example is the problem of colouring a geographic map containing n countries using only a fixed set of colours taking care that bordering countries have always different colours. Using the colour comparison as unit of measure, once the solution is given the verification takes maximum $O(n^2)$ comparisons[9]. However, every algorithm invented until now is not able to solve this problem in less than an exponential number of comparisons.

For example, the easiest algorithm is to try all the possible colour combinations, and for each one to check all the borders, until one combination which works is found. This brute force algorithm has a worst case complexity of $O(n^2 \cdot c^n)$ where c is the number of colours, while its average complexity depends strongly on the number of available colours.

[8] Except for interactive algorithms, which will not be analyzed in this book, which receive the input sequentially during the elaboration and therefore do not need to store all of it at the beginning of execution.

[9] Each country can border with all the other $n - 1$ countries and therefore maximum $n - 1$ comparisons must be done for every country. Since the countries are n, maximum $n \cdot (n - 1)$ comparisons are necessary.

Central-southern Europe coloured with 13 colours plus the sea.

A more efficient algorithm called backtracking (see section 4.4 on page 26) for this problem is trying to colour one country at the time avoiding the colour of the bordering countries, going back changing the previous decisions if no colour can be found for a country. For example, we assign red to Portugal, yellow to Spain, blue to France, green to Italy, yellow to Germany, red to Austria but we notice that Switzerland has no color available (its four bordering countries use up all the four colors). Therefore we go back and change the color of Germany to green and now Switzerland may be colored yellow. However the stepping back sometimes can cause the change of several countries in a row if there are countries with many borders such Hungary.

It is clear that class P is contained in class NP both for algorithms and problems and that there are algorithms contained in class NP which are not contained in class P. However, it is still an open question whether for problems these two classes coincide. This means that there are problems whose algorithms invented until now solve them in a slower than polynomial time but it still must be proven that no polynomial solving algorithm exists. This is currently the greatest challenge of theoretical computer science.

NP-SPACE is the class of problems, or algorithms, which require maximum a larger than polynomial amount of space or less. A theorem shows that NP-SPACE coincides with P-SPACE.

2.3.3. L and NL

L is the class of problems, or algorithms, which can be solved maximum in a logarithmic time or faster.

NL is the class of problems, or algorithms, which can be solved, in the worst case, in a larger than logarithmic time or faster, but whose solution verification requires no more than a logarithmic time.

3. Typical problems and algorithms

We present here a collection of typical computer science problems with their complexities and their easiest algorithms.

3.1. Majority

The majority problem receives as input a list of n booleans and must detect whether the majority of them is true. It is the problem of counting a ballot's votes discovering whether the majority is "yes" or "no".

Clearly the easiest algorithm, which is also the most efficient invented until now, is counting the "true" and the "false" until one of these values reaches int(n/2 + 1). If this never happens, there is a tie. Using the integer comparison (or the addition of integers) as unit of measure, its worst case complexity is n and its average case complexity, supposing that on average we reach majority after ¾ of the votes, is 3n/4, therefore this algorithm is in P. Its space-complexity is n booleans and 3 integers.

```
Function majority(n As Integer, votes() As Boolean) As Integer
        Dim threshold As Integer, i As Integer
        Dim countTrue As Integer, countFalse As Integer
        countTrue = 0
        countFalse = 0
        threshold = int( n / 2 + 1 )
        For i = 1 To n
                If votes(i) Then
                        countTrue = countTrue + 1
                        If ( countTrue >= threshold) Then
                                majority = 1
                                GoTo end_of_program
                        End If
                Else
                        countFalse = countFalse + 1
                        If ( countFalse >= threshold) Then
                                majority = 2
                                GoTo end_of_program
                        End If
                End If
        Next i
        majority = 3 ' we use value 3 for the tie
end_of_program:
End Function
```

The validation of the result has the same complexity and same space-complexity classes, since it use the same algorithm with the only difference that it counts and checks only the majority votes. Its complexity is n / 2 + 1.

Variations of this algorithm are possible, for example counting only the true and decide the result based only on this number, but every algorithm invented until now is not faster than O(n) and therefore probably this problem is in P and P-SPACE and is not in L nor in NL.

3.2. Search

The search problem is finding a specific number in a sorted list of n numbers returning the position of the number. Clearly the problem can be reduced to the search in an unsorted list, scanning all the numbers one by one, with a complexity of n and an average case complexity of n / 2 when we use the comparison as unit of measure. But in this case, we do not take any advantage of the sorting of the list. A better algorithm is the binary search which is a divide and conquer algorithm which splits the list into two parts and, according to the central number, searches only in the appropriate part.

```
Function binarySearch(goal As Single, list() As Single, a As Integer, b As Integer) As Integer
        If ( a = b ) Then
                If ( goal = list(a) ) Then
                        binarySearch = a
                Else
                        binarySearch = –1 ‘ we use –1 when the number does not exist
                End If
        Else
                Dim pivot As Integer
                pivot = int ( (a + b) / 2 )
                If ( goal <= list(pivot) ) Then
                        binarySearch = binarySearch(goal, list, a, pivot)
                Else
                        binarySearch = binarySearch(goal, list, pivot + 1, b)
                End If
        End If
End Function
```

This algorithm has a complexity equal to the maximum number of times the list is split, which is $1 + \log_2 n$, therefore the algorithm is in L. With minor improvements the average complexity can be slightly reduced but never below $O(\log_2 n)$. Its space-complexity, considering space used by singles, is n and can obviously never be reduced. Validation of the result requires clearly only one operation if the number has been found and the repetition of the whole algorithm when the number has not been found.

3.3. Sorting

The problem of sorting a list of numbers is very important in computer science since it is widely used for databases and, in general, to speed up searching processed. This problem can be solved using a recursive algorithm scanning the whole list looking for the largest number, putting it in the first place and re-applying the algorithm[10] to the list without the already sorted elements.

[10] In order to apply the algorithm over and over to the same list we need a subroutine and not a function and we need to pass the list by reference and not by value.

```
Sub directSort ( ByRef list() As Single, step As Integer ) 'start with step = 0
        Dim i As Integer, largestIndex As Integer
        Dim largest As Single
        largestIndex = LBound(list) + step
        largest = list(largestIndex)
        For i = LBound(list) + step + 1 To UBound(list)
                If ( list(i) > largest ) Then
                        largestIndex = i
                        largest = list(i)
                End If
        Next i
        list(largestIndex) = list( LBound(list) + step )
        list(LBound(list) + step) = largest
        If ( LBound(list) + step + 1 < UBound(list) ) Then
                Call directSort(list, step + 1) 'stop when only one element
        End If
End Sub
```

This algorithm must perform n-1 comparisons in the first step, n-2 in the second, and so on until 1 comparison in the last step. The sum of the numbers from 1 to n-1 is $n \cdot (n-1) / 2$ and therefore, using comparison of single as unit of measure, the algorithm has a complexity of $O(n^2)$ and the problem is for sure in P. If we want to consider also number switching as an operation, it performs only n switches. Its average complexity is the same, its space-complexity is O(n) and the solution verification requires n-1 comparisons.

3.3.1. BubbleSort

There are however many other sorting algorithms. A similar one is BubbleSort[11] which is basically the iterative version of the previous algorithm. It scans the list comparing the neighbouring numbers and switching the numbers every time the second is larger than the first. After the first step, the largest number is automatically in the last place and therefore it will never be scanned again. Therefore, the second step is performed from the first element to place n-1 and leaves the second-largest element in the second-last place. The third step is performed up to place n-2 and so on.

```
Sub bubbleSort ( ByRef list() As Single )
        Dim i As Integer, step As Integer
        For step = 0 To UBound(list) – LBound(list) – 1
                For i = LBound(list) + 1 To UBound(list) – step
                        If ( list(i-1) > list(i) ) Then
                                Dim temp As Single
                                temp = list(i)
                                list(i) = list(i-1)
                                list(i-1) = temp
                        End If
                Next i
        Next step
End Sub
```

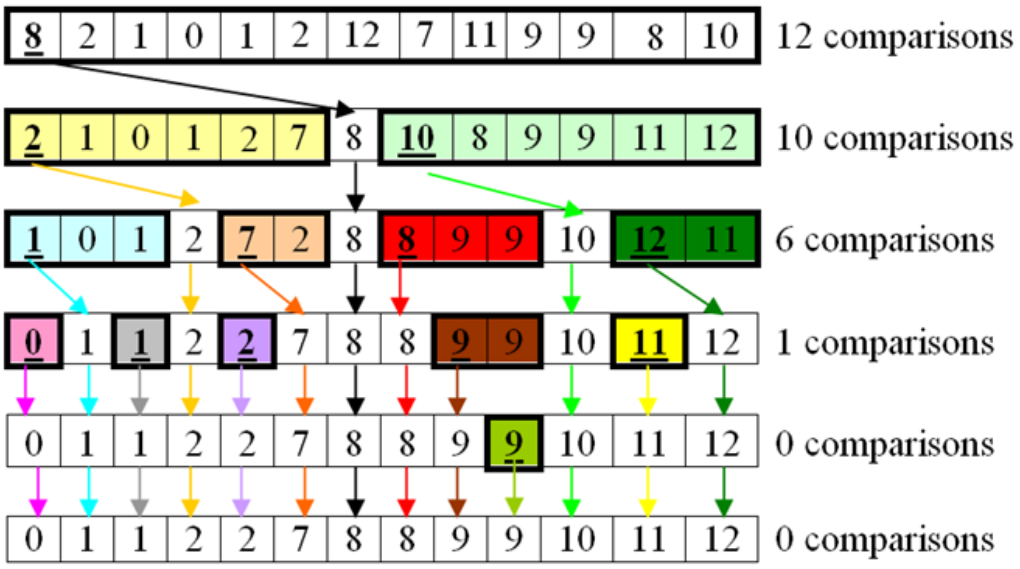
[11] The name derives from the idea of air bubbles in water, where the largest bubble comes out faster.

This algorithm performs n – step – 1 comparisons each step, where step goes from 0 to n – 2, therefore n-1 + n-2 + … + 2 + 1 which makes n · (n-1) / 2 and therefore BubbleSort has the same complexity as the previous algorithm. Its only advantage is that it does not have the overhead of the subroutine calling operations which can slightly slow down the process, with the disadvantage that it performs on average n · (n-1) / 4 numbers' switches and in the worst case n · (n-1) / 2.

3.3.2. QuickSort

A more efficient algorithm is QuickSort which uses a divide and conquer strategy. It fixes a pivot number, which may be taken at the beginning of the list or at any other place, and rebuilds the list putting all the smaller numbers on the left of the pivot and all the larger numbers on the right. From now on the position of the pivot is fixed and the algorithm recalls itself twice passing as a list the numbers on the left and then the numbers on the right.

```
Sub quickSort ( ByRef list() As Single, lower As Integer, upper As Integer)
        If ( upper – lower > 1 ) Then
                Dim i As Integer, lower2 As Integer, upper2 As Integer
                Dim list2(lower To upper) As Single
                lower2 = lower
                upper2 = upper
                For i = lower + 1 to upper
                        If ( list(i) < list(lower) ) Then 'list(lower) is taken as pivot
                                list2(lower2) = list(i)
                                lower2 = lower2 + 1
                        Else
                                list2(upper2) = list(i)
                                upper2 = upper2 – 1 'values put on the far right of list
                        End If
                Next i
                list2(lower2) = list(lower) 'pivot is put at the center
                For i = lower to upper
                        list(i) = list2(i)
                Next i
                Call quickSort(list, lower, lower2–1)
                Call quickSort(list, lower2+1, upper)
        End If
End Sub
```



Each partition of the list requires k-1 comparisons, where k is the number of elements in that list. In the worst case, for example when we take always the lowest or highest number[12], thus requiring n-1 steps to perform the algorithm. In this case the number of comparisons is $(n-2) \cdot (n-1) / 2$ and the algorithm is therefore $O(n^2)$. But considering the average case of a randomly sorted list, the number of steps is only $\log_2 n$ and therefore the complexity can be improved to $O(n \cdot \log_2 n)$; in fact in real applications, unless the list is deliberately built against QuickSort, it runs faster than the other two algorithms. Space-complexity for this algorithm is, as usual, $\theta(n)$.

3.3.3. MergeSort

Among the easy sorting algorithms, the most efficient is MergeSort invented by John Von Neumann in 1945. It starts from the consideration that merging two already sorted lists into a unique sorted list is a cheap operation, since with a basic algorithm it requires less than 2k-1 comparisons, where k is the size of each list.

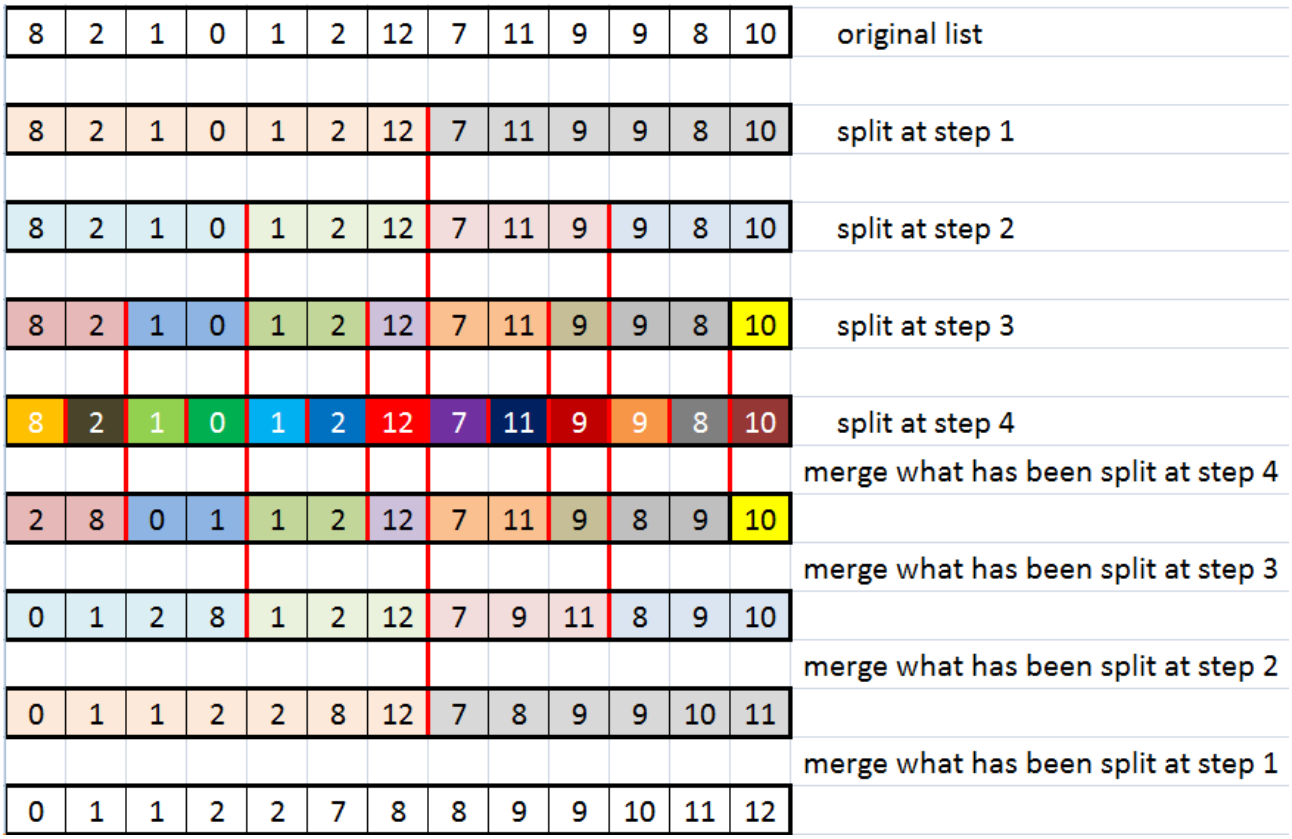
In detail, the merging subroutine takes the smallest value between the two lists until one part runs out of values and then it takes all the values of the other. Using comparisons as unit of measure, complexity of the merging operation is 2k-1 in the most unlucky case when the two lists run out of elements at the same time, and k-1 in the lucky case when a list has all the elements smaller that the other.

[12] Very unlucky cases are when the whole list is already sorted or reversely sorted and we take always the first or the last element of each list.

```
Sub merge (list1( ) As Single, list2( ) As Single, ByRef list3( ) As Single)
        Dim i1 As Integer, i2 As Integer, i3 As Integer, j As Integer
        i1 = LBound(list1)
        i2 = LBound(list2)
        i3 = 1
        ReDim list3(1 To UBound(list1) – Lbound(list1) + UBound(list2) – Lbound(list2) +2)
        While ( i1 <= UBound(list1) And i2 <= UBound(list2) )
                If ( list1(i1) > list2(i2) ) Then
                        list3(i3) = list1(i1)
                        i1 = i1 + 1
                Else
                        list3(i3) = list2(i1)
                        i2 = i2 + 1
                End If
                i3 = i3 + 1
        WEnd
        ‘ at this point either list1 or list2 is over
        If ( i1 <= UBound(list1) ) Then
                For j = i1 To UBound(list1) ‘ list1 is not over
                        list3( i3 + j – i1 ) = list1( j )
                Next j
        Else
                For j = i2 To UBound(list2) ‘ list2 is not over
                        list3( i3 + j – i2 ) = list2( j )
                Next j
        End If
End Sub
```

Once we have the merging subroutine, the algorithm recursively splits the list into two parts of the same size without doing any sorting until each part is a 1-element list. Every 1-element list is automatically sorted, and then the algorithm applies sequentially the merging subroutine.

```
Sub mergeSort( ByRef list( ) As Single, lower As Integer, upper As Integer)
        If ( UBound(list) > LBound(list) ) Then ‘ list has at least 2 elements
                Dim central As Integer, i as Integer
                central = ( lower + upper ) \ 2
                Call mergeSort ( list, lower, central )
                Call mergeSort ( list, central+1, upper )
                ‘ build list1 and put inside elements from lower to central
                ‘ build list2 and put inside elements from central+1 to upper
                merge ( list1, list2, list3 )
                ‘ put elements from list3 into list at places from lower to upper
        End If
End Sub
```



The main difference between QuickSort and MergeSort is that the latter is sure to perform no more than $\log_2 n$ splits, regardless of the numbers involved, since it is us who decide how to split, and is sure that the length of every list is no more than n/2. Therefore its worst case complexity is $O(n \cdot \log_2 n)$, exactly as its average case complexity. Its space-complexity is always O(n).

3.4. Prime numbers

Prime numbers are very important in computer science because, since they represent an example of a numerical sequence on which we still do not have enough mathematical knowledge, they are the basis for cryptography.

3.4.1. Primality test

Algorithms to detect whether number n is prime or not have been studied in the last thousands of years. The brute force method is to divide the candidate n by 2 and by every odd number from 3 to the square root[13] of n and stop when a division returns a remainder of 0; if the algorithm arrives at the end without stopping, the number is prime. This method requires, on the worst case if the number is prime, a number of divisions $O(\sqrt{n})$, but, if the number is not prime, the average number of divisions is much smaller, since many non prime numbers have small numbers as divisors[14].

[13] or divide up to the half of the candidate, if no procedure to calculate the square root is available.

[14] In fact, with a 50% probability the candidate is even and the algorithm stops after the first division. With a further 16.6% of probability the candidate is a multiple of 3 and the algorithm stops after the second division. With a further 6.6% of probability the candidate is a multiple of 5 and the algorithm stops after only three divisions. Therefore, for more than 73% of candidates the algorithm requires no more than three divisions, regardless of the size of the candidate.

```
Function isPrime(n As Long) As Boolean
        Dim i As Long, quotient As Double
        isPrime = True
        'first we try to divide by 2, since 50% of numbers stop at this test
        If ( n / 2 = n \ 2 ) Then 'this is a trick to see if the remainder is 0
                isPrime = False
                GoTo end_of_function
        End If
        For i = 3 To int(sqrt(n)) Step 2 'now we divide by every odd number up to sqrt(n)
                If ( n / i  = n \ i ) Then
                        isPrime = False
                        GoTo end_of_function
                End If
        Next i
end_of_program:
End Function
```

This algorithm can be speed up a lot if we have in advance a list of prime numbers. In this case, instead of trying all the odd numbers, we can try the prime numbers of our list and then, if we still have not reached the square root of n (for example because we have a short list), go on with the odd numbers.

There are also much more complicated algorithms which are much faster: the best algorithm implemented until now is $O((\log_2 n)^{7.5})$.

3.4.2. Primes extraction

Finding all the prime numbers from 1 to a fixed number n is a useful task both to speed up primality tests but also to build good cryptographic keys (see section 5.3 at page 29).

The most famous, and probably most antique, algorithm is the Sieve of Eratosthenes, invented by Eratosthenes of Cyrene in the III century BC. It starts with 2, which is notably prime, and eliminates all its multiples up to n, then it goes on to the next non-eliminated number and performs the algorithm again. The algorithm is increasingly faster since the non eliminated numbers become fewer and fewer. Its complexity, using the multiplication of integer numbers as unit of measure, can be calculated using complicated bound on the amount of prime numbers as $O((n \cdot \log_2 n) \cdot (\log_2 \log_2 n))$ putting this algorithm in the P class.

There are better algorithms which extracts primes in the less than linear time $O(n / \log_2 \log_2 n)$.

3.4.3. Integer factorization

The top problem in cryptography is finding the prime numbers which factorize[15] a prime number n and almost every cryptographic tool relies in the fact that no algorithm exists that performs integer factorization in a short time.

Clearly the solution verification for this algorithm is very fast and it requires never more than $\log_2 n$ multiplications[16] which means, remembering that each multiplication requires k^2 bit operations and number n uses 2^k bits, k^3 bit operations.

[15] To factorize means that the multiplication of those prime numbers among themselves gives the number. For example, number 132 is factorized by primes 2, 3 and 11 since $2 \cdot 2 \cdot 3 \cdot 11 = 132$.

[16] $\log_2 n$ is the maximum amount of prime factors that number n may have. It is easy to prove, since the numbers which have the largest amount of factors are powers of 2, such as 128 (7 factors all equal to 2) or 256 (8 factors, all equal to 2), etc.

This problem can be solved by a brute force approach dividing n by every prime up to the square root of n and extracting as factors those primes whose division gives a remainder of zero. If no factor is found before the square root of n, then the number itself is a prime. A good improvement of this brute force technique is to do, whenever a factor is found:

- go on with the quotient of the division rather than with the original number, since the quotient is much smaller and divisions are faster;
- redo the division of the quotient by that factor, instead of going to the next one, to try to reduce it even further;
- if the quotient is a prime, it automatically is the last factor.

For example, to find the factorization of 1989:

Division	Quotient	Remainder
1989 / 2	994	1
1989 / 3	663	0 → 3 is a factor
663 / 3	221	0 → 3 is a multiple factor
221 / 3	73	2
221 / 5	44	1
221 / 7	31	4
221 / 11	20	1
221 / 13	17	0 → 13 is a factor
17 is prime → 17 is the last factor		

Using the division of integer as unit of measure, the complexity of this algorithm is O(n) when we do not have a list of primes in advance and can be proven to be, using complex bounds on the amount of primes, $O(\sqrt{n} / \log_2 n)$ when we have already a list of primes. Considering instead the number of bit operations as unit of measure, and remembering that $n=2^k$ and that each division is $3k^2$ operations, the complexity becomes $O(2^{k/2} k)$ which is an exponential time. Therefore this algorithm, when considering bit operations, is in class NP.

Many algorithms exists to perform integer factorization and the best ever invented until now is $O(\exp(k^{1/3} (\log_2 k)^{2/3}))$, which means that in terms of bit operations the problem for the moment is in class NP and not in class P. This means that when we increase the number of bits with which n is built, the number of required bit operations becomes extremely large making factorization impossible within a reasonable amount of time. This fact is the keystone on which cryptography is based.

3.5. Timetable

The problem of building a timetable is an everyday problem for which people spend a lot of their time. Its basic formulation is simply to assign p professors to the h hours of c classes, with the following bounds:

- maximum one professor in each hour in each class,
- maximum one class for each professor in each hour,
- each professor has a fixed amount of hours per class that must be assigned.

Moreover, often many other real life bounds come into play. For example:

- the same professor can not have more than 3 consecutive hours,

- the same professor can not have always the same type of hours (always the first hour, always the afternoon hours, etc.),
- each professor must have a complete free day in the timetable,
- in case of empty hours (no-lesson hours), they may be only at the beginning or at the end of the day.

The brute force approach is to try all the combinations putting all the p+1 professors in c·h places (the extra professor is to indicate no-lesson) until we find one which satisfies all the bounds. Using the bounds' check as unit of measure, this algorithm's complexity is $(p+1)^{ch}$, which is polynomial with respect to professors and exponential with respect to classes or hours.

The algorithm can be improved avoiding automatically the obviously wrong configurations, for example always putting a professor in an empty classroom for which he still has hours to do. However, the worst case remains always exponential with respect to classes and hours.

```
Global c as Integer
Global h as Integer
Global p as Integer

Sub timetable( )
   c = InputBox( "Tell me number of classes" )
   h = InputBox( "Tell me number of hours" )
   p = InputBox( "Tell me number of professors" )
   Dim table( ) As Integer
   ReDim table( 1 To c , 1 To h )
   Dim jc As Integer
   Dim jh As Integer
   Dim jp As Integer
   ' build an empty timetable to start
   For jh = 1 To h
      For jc = 1 To c
         table( jc , jh ) = 0
      Next jc
   Next jh
‘ increment professor in cell jc , jh. If prof is already jp,
‘ it will be put to 0 and increment the next one by 1
   Do While ( Not isLastOne(table) )
      Call increment(table, 1, 1)
      If ( checkBounds(table)  ) Then
         Call printSolution(table)
         Exit Do
      End If
   Loop
End Sub
```

```
Function isLastOne( table( ) As Integer ) As Boolean
   ‘ we use this function to avoid incrementing the last one
   Dim jc As Integer
   Dim jh As Integer
   Dim jp As Integer
   isLastOne = True
   For jc = 1 To c
      For jh = 1 To h
         If ( table( jc , jh )  <> p) Then
            isLastOne = False
            Exit For
         End If
      Next jh
   Next jc
End Function

Sub increment ( table( ) As Integer, jc As Integer, jh As
Integer )
   If ( table( jc, jh ) < p ) Then
      table( jc , jh ) = table( jc , jh ) + 1
   Else
      table( jc , jh ) = 0
      If ( jh < h ) Then Call increment( table , jc , jh + 1)
      Else Call increment( table,  jc + 1 , 1)
   End If
End Sub

Function checkBounds( table( ) As Integer ) As Boolean
   ‘ check the bounds according to school's rules
   …
End Function
```

3.6. Knapsack problem

The knapsack problem consists of filling in a knapsack, which has a maximum weight, with items trying to maximise the utility. Each item has its weight and its utility and may be used only once. It is a maximisation problem because we have a function, the sum of the utilities of the items in the knapsack, that we wish to maximise with a bound, the maximum weight.

Also for this problem we can use the brute force approach, exploring all the possible configurations, including and excluding each item, excluding those configurations which do not satisfy the bound and selecting the one with the largest utility. If we call n the number of items, there are 2^n combinations and for each one we need to sum from zero to n (in the worst case) weights and utilities. Therefore, using the integers' addition as unit of measure, the complexity of this algorithm is $O(2 \cdot n \cdot 2^n) = O(n \cdot 2^n)$. The average complexity is $O(n/2 \cdot 2^n) = O(n \cdot 2^n)$.

The brute force algorithm can be obviously improved, for example avoiding to add other items when the weight has already reached the maximum, or trying to insert the best items (those with a high utility over weight ratio), or using a branch and bound technique (see page 25), but the improvement is only in the average case. No better worst case algorithm is known. The validation of the solution requires maximum 2n sums for the bound checking but requires also the exploration of all the possibilities to be sure that the utility is the maximum and therefore, for the present knowledge, this problem is not even in NP.

3.7. Travelling salesman problem

In the travelling salesman problem we have a table with all the distances between each pair of towns and we must find the shortest route which touches once all the cities and then goes back to the original one. This is a minimization problem[17], where the function to be minimized is the distance and the bound is that the patch must touch all the cities once with the original one as the last one.

Also for this problem the brute force can be applied, but we must use a smart strategy to avoid an infinite amount of combinations. Starting from the first city c_1, we can go to each c_2 of the other n-1, then from c_2 we can go to each c_3 of the other n-2 cities and so on. The possible combinations are $(n-1) \cdot (n-2) \cdots 2 \cdot 1 = (n-1)!$ and for each one we need to do exactly n integers' additions. Therefore the complexity of the algorithm is O(n!) and this is also the average complexity since we need to explore all the possibilities to find the minimum.

The brute force algorithm can be obviously improved, for example using a branch and bound technique (see page 25), but the improvement is only in the average case. No better worst case algorithm is known. The validation of the solution requires the exploration of all the possibilities to be sure that the distance is the minimum and therefore, for the present knowledge, this problem is not even in NP.

4. Alternative algorithms

This chapter presents a list of alternative algorithms which can be applied to the previous problems. All these algorithms do not reduce the complexity of solving the problem, like algorithms in chapter 1, but try to find alternative ways for an approximated solution or to reduce average case complexity.

4.1. Approximation

Approximation algorithms do not look for the solution of the problem, but try to approximate it searching for a similar solution or for an almost optimal one.

[17] It is exactly equivalent to a maximisation problem. We simply have to take as maximisation function "minus the distance".

In the discrete case, where the solution exists and theoretically can be found exploring all the cases, these algorithms find a sub optimal solution, which probably is not the optimal one but at least has guarantees to be among the best ones.

In the continuous case this approach finds a solution which is very close to the real one, which is impossible to find for a computer since it is often an irrational number[18], often with a required precision.

4.1.1. Newton's method

Newton's method is an approximation algorithm to find the root of the equation $f(x) = 0$ with a certain precision ε, where f can be also a complicated function which can not be inverted analytically. The algorithm is very simple:

1. start with a value x_0 for which $f'(x_0) \neq 0$;
2. if $|f(x_i)| < \varepsilon$ the solution is x_i;
3. calculate $x_{i+1} = x_i - f(x_i) / f'(x_i)$ and go back to the previous step.

For example, for $f(x) = 3 \cdot \ln x - x$ finding the x for which $f(x) = 0$ is analytically impossible. However, we can use Newton's method which, choosing $x_0 = 1$ and remembering that $f'(x) = 3 / x - 1$, we get 1, 1.5, 1.7836, 1.8535, 1.8571, 1.8571 and so on.

The stop condition can be on the value of f, asking it to be very small, or directly on the x value $|x_{i+1} - x_i| < \varepsilon$, requiring the x value to be very close to the previous x.

The only drawback of this simple and very efficient algorithm is that, if there is a local maximum or minimum of f, the derivative is very close to zero and the next x will jump away, or, even worse, the derivative is zero and the program returns a division by zero error. Its complexity can not be calculated a priori since it depends strongly on the function and on the starting value.

4.2. Randomization

Randomization is used to get a faster average running time when dealing with some malicious problems or a deliberate bad input. The algorithm's strategy is not deterministic but includes some random decisions which are able to avoid particular situations when traditional algorithms move towards the solution very slowly.

Consider for example the problem of finding any surname starting with letter P (9.3% of the Italian surnames starts with P) in a long list of n surnames on which you do not have any information. The obvious approach is to look sequentially at each element of the list, but this would take very long if the list were ordered alphabetically; there is a similar drawback with checking in the reverse order in case it is ordered in reverse alphabetical order. In fact, using any deterministic algorithm with a fixed strategy to check the elements, we cannot guarantee that the algorithm completes quickly for all possible inputs. On the other hand, if we check the list elements at random, then we find on average a right surname in less than 11 trials.

Another use of randomization is to solve problems with a small probability to commit a mistake: the solution is exact and not approximated, but we are not completely sure of this.

4.2.1. Fermat's primality test

This test can check whether a number is prime with an exact result when it says that the number is not prime and with a probability to commit a mistake when it says that the number is prime.

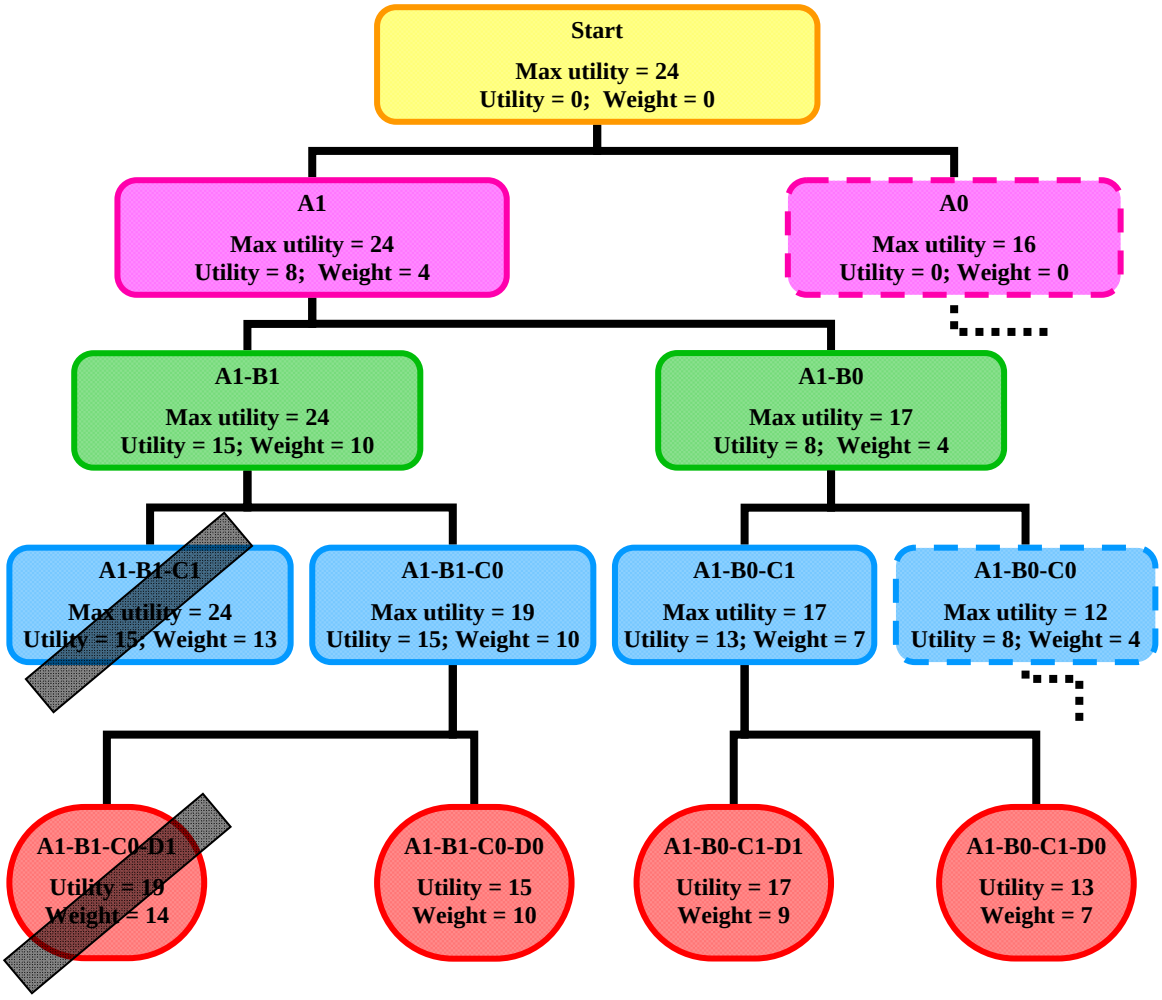
The algorithm chooses random numbers m smaller than n and, if the remainder of m^{n-1} / n is not 1 then n is not a prime, otherwise if the remainder is equal to 1 for every tried m, then n is very probably prime. Performing this check has an integers' multiplication complexity $O((\log_2 n)^2 + n$

[18] An irrational number is a number with an infinite amount of non periodic decimal digits, for example the square root of 2. Using floating point representation these numbers are impossible to be represented exactly.

$(\log_2 n)^2$) but which can be reduced to approximately O($(\log_2 n)^2$) using a smart raise to the power operation. This must be obviously multiplied by the number of times we do this check.

4.3. Branch and bound

Branch and bound is a heuristic technique for maximisation problems which does not improve the worst case complexity but only the average case. Its strategy is to branch the problem into sub problems, each with a bound on the maximisation function, and exploring the sub problem which is more promising. This sub problem is then branched again into sub sub problems, each with a bound, and so on until we may not go on anymore and a solution is found. If the solution's utility is largest than the bound on every open sub problem, then it is also the maximum; otherwise the appropriate branches are examined until a better solution is found or until no branch with a bound higher than the current best solution exists.



Branch and bound tree for the knapsack problem with a maximum weight of 10, four objects of weights 4, 6, 3, 3 and utilities 8, 7, 5, 4. Cancelled out branches are not allowed situations, dashed branches are unexplored and circles are final solutions.

For example, consider a knapsack problem with a maximum weight of 10, four objects of weights 4, 6, 3, 3 and utilities 8, 7, 5, 4. We can immediately put a bound on the utility function which clearly may never be larger than 24, the sum of the utilities of all the objects, while the effective utility for the moment is 0 and the weight is 0 (see yellow box). Then, starting from the object with the largest utility, called object A with utility 8 and weight 4, we branch into the two sub problem A1 "object A is included" and A0 "object A is excluded". The bound on these two sub problems is now 24 for the first and 16 for the second, while the effective values are 8 and 0 for the utilities and 4 and 0 for the weights (see the two pink boxes). We go on with the first and most promising branch

and consider now object B with utility 7 and weight 6. We branch into A1-D1 with bound on utility of 24, an effective utility of 15 and weight of 10 and A1-D0 with a bound of 17, an effective utility of 8 and a weight of 5. If we go on with the more promising branch adding object B with a utility of 4 and a weight of 4 (see the two green boxes). Choosing again the most promising branch, A1-B1, we discover that there is no possibility A1-B1-C1 since the effective weight becomes 13 which exceeds the maximum of 10. Therefore this branch is cancelled and from here the only possibility is A1-B1-C0 which has a bound on utility equal to 19 and an effective utility of 15 and weight of 10 (see the two left blue boxes). Choosing between branch A1-B1-C0, A1-B0 and A0 we take the first and branch it into the impossible solution A1-B1-C0-D1 which exceeds the allowed weight and the solution A1-B1-C0-D0 with a weight of 10 and a utility of 15 (see the two right red circles). This is a possible solution, however we are not sure that it is the best one because there are still unexplored branches with a bound larger than 15. We therefore explore the most promising one, which is A1-B0 (the left green box). We branch it into A1-B0-C1 with a bound of 17 and A1-B0-C0 with a bound of 12 (the two right blue boxes). We go on with the most promising one which is A1-B0-C1 and branch it into the two possible solutions A1-B0-C1-D1 with a utility of 17 and a weight of 9 and A1-B0-C1-D0 with a utility of 13 and a weight of 7 (the two right red circles). The best solution found until now is A1-B0-C1-D1 whose utility of 17 is larger than every unexplored branch, which are A0 with bound 16 and A1-B0-C0 with bound 12, and therefore this is the best solution.

As can be seen, particular nasty problems can force the branch and bound technique to explore a lot of branches and, in some cases, even all the branches. Therefore this algorithm's complexity remains $O(2^{2n})$, while the average case complexity is reduced because this algorithm skips all the branches leading to only impossible solutions and all the branches which for sure lead to non optimal solutions without exploring them. In the best case the number of integers' sums is 3n, while, supposing a quite pessimist estimate that we need to reopen on average one branch every two, the average complexity is $O(2^n)$, still exponential but of lower grade.

The branch and bound algorithm can be applied also to the traveling salesman using directly the distance already covered as a minimum bound. A very similar strategy is used in modern GPS navigation route planners.

4.4. Backtracking

Backtracking in another heuristic technique to improve the average case complexity in problems where the solution must be found without any maximisation but only with some bounds. It tries to build a solution choosing a random route, or a route which seems the most interesting according to an a priori heurist criterion, and goes on until the solution is found or some bounds are broken. In the latter case, it steps back and explores all the other possibilities. If all these possibilities lead to broken bounds, it steps back again exploring all the other possibilities of the previous level. If the choice criterion is smart enough, usually backtracking goes almost directly to the solution avoiding to become stuck in a lot of steps back.

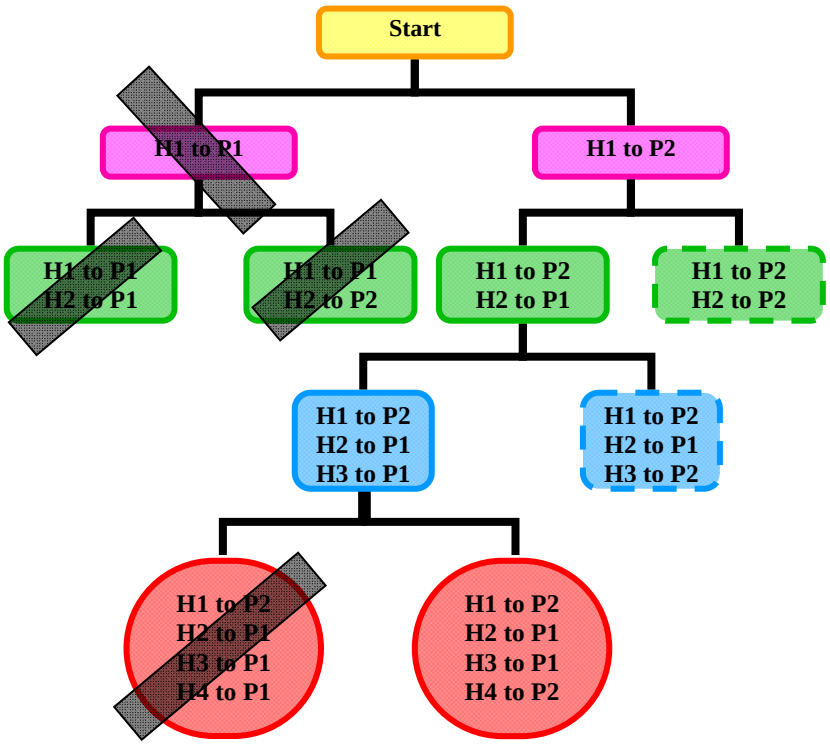
A typical problem which can be solved using backtracking, and which is solved also by human beings using backtracking, is Sudoku[19].

Often a good strategy is to try the most constrained branches first. For example, when colouring a map (see problem in section 2.3.2 on page 11) it is a good idea to start with the country with the most borders and its neighbours to immediately avoid a lot of possible backtracks. When dealing with a timetable, it is a better to fix the most problematic professor first.

The difference between backtracking and branch and bound is that backtracking is not for maximisation problems, therefore it chooses the closest branch and not the most interesting one. Moreover, once backtracking arrives to a solution the algorithm is over.

[19] For details, see http://en.wikipedia.org/wiki/Sudoku and http://en.wikipedia.org/wiki/Algorithmics_of_sudoku.

Considering for example the timetable problem, we take two classes C1 and C2 with two hours each H1 to H2 for C1 and H3 to H4 for C2, and two professors P1 and P2 which must be assigned to the classes with the bound of one lesson per class. Moreover, professor P2 does not want to make lesson in class C1 in the second hour. We start the algorithm facing with the decision to whom assigning H1. We assign it to P1 and this does not break any bound. Then we must decide to whom assigning H2 and we assign it to P1. But we notice that this breaks the bound and therefore we step back and cancel this whole branch (which implies cancelling automatically one quarter of the 2^4 possible cases) and we assign hour H2 to P2. But also this breaks a bound and we must step back again. Now all the current level branches are cancelled, and therefore we must step back another time returning to the choice for H1. Since the current branch has lead to cancelled sub branches, we cancel it completely (which implies cancelling automatically half of the 2^4 possible cases) and we choose the only remaining one which is assigning H1 to P2. Then we must decide to whom assign H2. We assign it to P1 and it does not break any bound; we do not care whether the other sub-branch breaks the bound, since we will not use it unless all the current sub-sub-branches are deleted. Going to classroom C2, we assign H3 to P1 and this does not break any bound. We then assign H4 to P1, but this breaks a bound and therefore we cancel this branch and step back, assigning H4 to P2 which does not break any bound. All the hours have been assigned without breaking any bound and therefore the solution has been found. Note that we do not care whether there are still open branches, since backtracking simply finds a solution.



Backtracking tree for the timetable problem with classes C1 and C2, professors P1 and P2, two possible hours per class and the bound that professor P2 may not teach in class C1 in hour H2. Cancelled out branches are not allowed situations and the circles are the solutions. Dashed branches are branches which are left unexplored, since in this problem the aim is simply finding one solution.

Complexity is reduced more or less in the same way as branch and bound: worst case complexity remains the same, but the number of average explored possibilities is strongly reduced.

5. Cryptography

Cryptography studies the techniques to send a message between two people avoiding that the content be read by others. The message itself may be looked at by other people, but these should not be able to understand it. Cryptography is one of the top military research topics for millennia, to Julius Caesar is attributed one of the first cryptographic techniques. However, until the nineteenth century cryptographic techniques were trivial and, having the appropriate statistical tools, easy to break. From the beginning of the twentieth century, mostly due to the two world wars first, the cold war and Internet transactions later, cryptography has made giant steps and now it is currently used by everybody on the WWW.

We will call encrypt the act, done by the sender, of rewriting the message in such a way that it becomes unreadable and decrypt the act, done by the receiver, of rewriting the message in such a way that it becomes readable again.

5.1. Letters scrambling

The naïve cryptographic technique consists into converting the letters to another alphabet, made for example of numbers (ASCII coding can be used) or of other invented symbols. Sender and receiver must know the conversion table perfectly in order. This system works for a single message, but if the flow of messages is continuous, such as when transmitting information for military use or over the Internet, the coding table can easily be guessed using statistics. In fact, every human language has strong differences in the frequency of appearance of every letter[20] and, if the number of messages is high enough, it is very easy to detect to which letters the most frequent symbols correspond. Once the most frequent letters are detected, everybody looking at the text can guess the other letters.

Exactly the same technique, with exactly the same drawback, is moving the letters a fixed amount of places ahead, for example converting “a” into “d”, “b” into “e”, “c” into “f” and so on up to “x” into “a”, “y” into “b” and “z” into “c”, or even converting the letters into other letters decided in advance without any mathematical relation.

A more interesting letter scrambling was used at the beginning of the twentieth century and lasted for over 50 years. It consists into changing the letters following a dynamic rule, for example converting the first letter into its ASCII code, then converting the second letter into its ASCII code plus 3, the third one into its ASCII code plus 7, the fourth one into its ASCII code minus 2 and so on according to a predetermined number sequence. This technique is very difficult to attack using statistics, even through with powerful calculations it may be broken.

However, it has a problem which is common to all the basic scrambling techniques: the sender and the receiver must communicate the coding at the beginning of the communication. If somebody is listening to this early communication, he can easily decrypt all the subsequent messages, and, moreover, encrypt fake messages.

The Enigma machine, used by the German army to encrypt and decrypt messages during WWII, uses a dynamic letter scrambling technique.

[20] For example, in English letter “e” appears more than 12% of the times, while letter “j” less than 0.2% of the times. In Italian the four vowels “a”, “e”, “i” and “o” make up about 10% each. In German letter “e” appears more than 17% of the times, while letter “j” less than 0.3%. Source: http://en.wikipedia.org/wiki/Letter_frequencies.

5.2. Diffie and Hellman's key sharing

Bailey Whitfield Diffie and Martin Hellman invented in the 1976 an algorithm[21] which lets two people exchange a number which remains secret even if somebody is listening to the communication. In this way, a whole sequence of numbers can be exchanged using this algorithm and this sequence can be used for dynamic encryption of the next message.

The algorithm works as follow:

- the sender decides a prime number P and a smaller number G (for example P=23 and G=5) and openly sends them to the receiver;
- the sender chooses a number A (for example A=6) and then openly sends to the receiver number $\hat{A}$ calculated as the remainder of G^A / P (in our example 5^6 = 15625, 15625 / 23 makes 679 with the remainder of 8);
- the receiver chooses a number C (for example C=15) and then openly sends to the receiver number $\hat{C}$ calculated as the remainder of G^C / P (in our example, 5^{15} / 23 makes 1326851222 with the remainder of 19);
- the sender calculates the secret number S computing the remainder of $\hat{C}^A$ / P (in our example, 19^6 / 23 makes 2045473 with the remainder of S=2);
- the sender calculates the same secret number S computing the remainder of $\hat{A}^C$ / P (in our example, 8^{15} / 23 makes 1529755308210 with the remainder of S=2).

Even if somebody intercepts the communication, using only P, G, $\hat{A}$ and $\hat{C}$ and knowing perfectly the algorithm, it is very hard to calculate the secret number S without knowing either A or C. In fact this calculation is, for the moment, a NP problem.

5.3. RSA

RSA encryption algorithm takes its name from its inventors in 1977: Ron Rivest, Adi Shamir, and Leonard Adleman. The algorithm is asymmetric because the receiver knows all the keys to encrypt and decrypt while the sender has only the key to encrypt.

The algorithm works as follow:

- the receiver chooses two primes a and b and computes $n = a \cdot b$. Then he calculates $(a - 1) \cdot (b - 1)$ and finds a number P which is coprime (it has no factors in common) to this quantity. For example, choosing a = 3, b = 5, n is 15, $(a - 1) \cdot (b - 1)$ is 8 and he can take P = 3;
- the receiver finds a number S such that $S \cdot P - 1$ is a multiple of $(a - 1) \cdot (b - 1)$. In our example, he must choose 3S – 1 as a multiple of 8, for example 32, and S = 11;
- the receiver openly sends P and n to the sender;
- the sender converts the message in numbers using any naïve letter scrambling and gets a long sequence of digits. This sequence is then split into pieces of the same length. We call every single piece M, paying attention that the largest M be smaller than n;
- the sender encrypts every number M simply calculating C as the remainder of M^P / n and openly sending it to the receiver. In our example, number M = 12 becomes encrypted into 12^3= 1728 divided by 15 which makes 115 with the rest of 3. Therefore 3 is sent;
- the receiver can decrypt C to calculate M as the remainder of C^S / n;
- the receiver rejoins the parts together and unscrambles the message.

A possible interceptor gets to know only P and n and is thus able to send encrypted messages but is not able to decrypt messages. In order to do so, he should be able to find S, which requires the knowledge of a and b. These numbers can be found as factors of n, but as we have seen in section 3.4.3 this is a NP problems using bits operations as unit of measure. Therefore, if the receiver

[21] B.W. Diffie and M. Hellman, New Directions in Cryptography, IEEE Transactions on Information Theory, vol. IT-22, Nov. 1976, pp: 644-654.

chooses a and b with many bits, factorizing n becomes very difficult. RSA algorithms used on the Internet typically take a and b with at least 128 bits which are numbers with 7 digits.

Clearly if the receiver wants instead to send messages, they simply need to build other two keys for the communication in the other direction.

RSA algorithm can be used to electronically sign documents, using the two keys exactly in the reverse way: S to encrypt and P to decrypt. The signer publishes key P on his website and writes his document, ending it with his name, encrypting it using key S. Everybody can use key P to decrypt the document and reading its content: if the document is correctly decrypted and the signer's name appears at the end the message, it means that the document was originally encrypted with S and therefore it is original. Otherwise, if only nonsense letters appear, the document is not encrypted with S and it is counterfeited.

5.3.1. Exercise

Supposing you are an interceptor and get n=3233 and P=17, build a program to find S.

The solution is a = 61, b = 53, S=2753.

Index