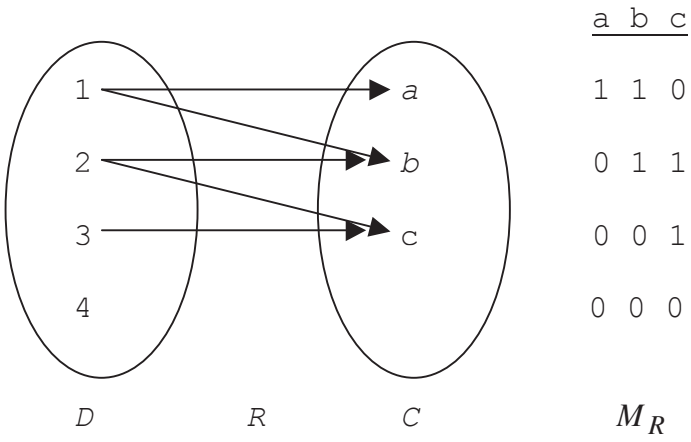
CHAPTER 9. RELATIONS

Recall that a function f from the domain D to the codomain C associates to each element $x \in D$ exactly one element $y = f(x) \in C$. If instead we allow each element $x \in D$ to be associated to any number of elements in C, then we have a generalization of a function called a *relation*, as illustrated by the diagram below.



a	b	c
1	1	0
0	1	1
0	0	1
0	0	0

M_R

Each arrow in the above diagram represents an ordered pair (x,y), where $x \in D$ and $y \in C$. The set of all the ordered pairs (x,y) such that $x \in D$ and $y \in C$ is the cartesian product $D \times C$. In a relation, each of the ordered pairs (x,y) in $D \times C$ can be present or absent; so **a *relation* R from the domain D to the codomain C is a subset of $D \times C$.** Recall that in Chapter 3 we defined what a function **does**. We can now define what a function **is**: a *function* f from the domain D to the codomain C is a relation from D to C such that for each x in D there is exactly one ordered pair (x,y) in f.

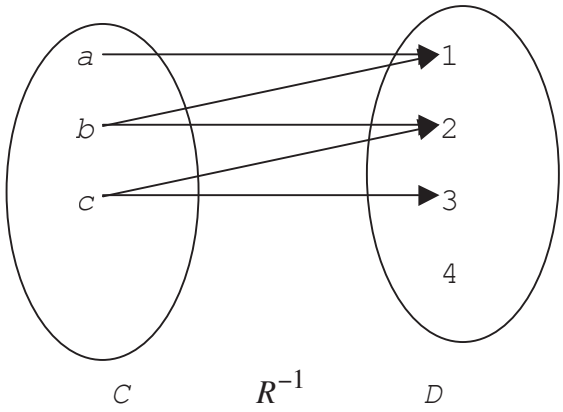
If D and C are finite, then a relation R from D to C can be represented by a graph or by a matrix M_R, which has one row for every element of D and one column for every element of C. The element $M_R[x,y]$ is equal to 1 if $(x,y) \in R$ or 0 otherwise, as illustrated above.

9.1 Operations on relations

Since a relation is a set, the usual set operations can be done on them. If R and S have the same domain D and codomain C, then so do the relations $R \cup S$, $R \cap S$, $R \oplus S$, $R - S$ and $\overline{R}$. The matrix of each of these relations can be found from M_R and M_S in the obvious way. For instance, for every $x \in D$ and every $y \in C$, $M_{R \cap S}[x,y]=1$ if $M_R[x,y]=1$ and $M_S[x,y]=1$; otherwise $M_{R \cap S}[x,y]=0$. I leave it to you to find a formula for each of the matrices of the other relations obtained by applying set operations to R and S and to verify your formulae (and the one given here for set intersection) on some examples.

Recall that the inverse of a bijective function $f : D \to C$ is a function $f^{-1} : C \to D$ defined by saying that for every y in C, $f^{-1}(y)$ is equal to the unique x in D such that $f(x) = y$. That is, if f

sends x into y, then f^{-1} sends y into x. The inverse of a relation can be defined similarly except that no restriction has to be made on the sort of relation that can have an inverse. The *inverse* of a relation $R : D \to C$ is the relation $R^{-1} : C \to D$ such that for every $y \in C$ and every $x \in D$, $(y,x) \in R^{-1}$ if and only if $(x,y) \in R$. It will sometimes be convenient to write xRy instead of $(x,y) \in R$. In this notation, $y R^{-1} x$ if and only if xRy. The diagram below shows the inverse of the relation shown in the diagram above and the matrix of this inverse relation.



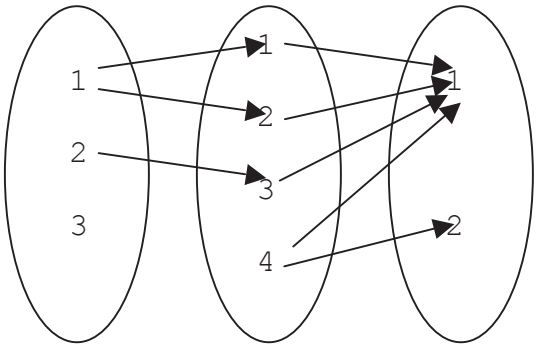
a	1	0	0	0
b	1	1	0	0
c	0	1	1	0

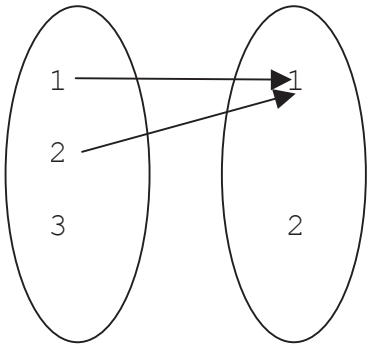
$$M_{R^{-1}} = (M_R)^T$$

The matrix of R^{-1} is not the inverse of M_R. It is the *transpose* of M_R, obtained from M_R by exchanging the rows and columns, because a row of M_R represents an element $x \in D$, a column of M_R represents an element $y \in C$, and R^{-1} is obtained from R by exchanging x and y.

Recall that the composition of the function $g:A \to B$ by the function $f : B \to C$ is the function $f \circ g : A \to C$ defined by the proposition that for every element x in A, if $g(x) = y$ and $f(y) = z$, then $(f \circ g)(x) = z$. That is, if g sends x into y and f sends y into z, then $f \circ g$ sends x into z. The composition of a relation S by a relation R can be defined similarly except that in general there is no guarantee that S is going to send x into anything; so we instead insist that there be at least one y in B such that S sends x into y and R sends y into z. More formally, if S is a relation from A to B and R is a relation from B to C, then $R \circ S$ is the relation from A to C such that for every $x \in A$ and every $z \in C$, $xR \circ Sz$ if and only if there exists some y in B such that xSy and yRx.

The composition of one relation by another can be illustrated by the following model. Each of the sets A, B and C is represented by a set of rocks sticking up above the surface of a river. A frog is sitting on one of the rocks x in A and a fly is sitting on one of the rocks z in C. The frog, of course, wants to eat the fly, but if it jumps into the river it will make noise, scaring the fly away; so it has to hop from its own rock x to the fly's rock z. It can't hop from one rock to another in the same set or from any rock in A to any rock in C, but it can hop from rock x in A to rock y in B if xSy and it can hop from rock y in B to rock z in C if yRz. Then $xR \circ Sz$ if the frog can hop from x to z in two hops, the first one from x to some rock y in B and the second one from y to z. Below we show two relations, the composition of one by the other and the matrices of all three relations.





A	S	B	R	C
1 1 0 0			1 0	
0 0 1 0			1 0	
0 0 0 0			1 0	
			1 1	

A	$R \circ S$	C
1 0		
1 0		
0 0		

If the frog is on 1 and the fly is on 1, then there are 2 ways the frog can catch the fly: via the rock 1 in B and via the rock 2 in B. If the fly is on 2, then the frog can't catch the fly because it can't hop to 2 in C from either of the rocks 1 and 2 in B it can hop to. If the frog is on 2 and the fly is on 1, then there is 1 way the frog can catch the fly: via the rock 3 in B. If the fly is on 2, then the frog can't catch the fly because it can't hop to 2 in C from the one rock 3 in B it can hop to. If the frog is on 3, then it can't catch the fly no matter which rock the fly is on, because the frog can't hop anywhere from where it is.

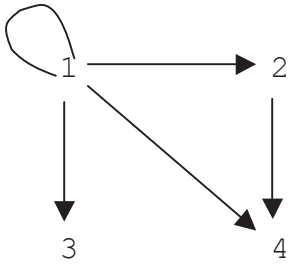
How could we evaluate $M_{R \circ S}$ from M_R and M_S without drawing pictures? Suppose the frog is on rock x in A and the fly is on rock z in C. The frog can catch the fly if there is a rock y in B such xSy and yRz. Now xSy if $M_S[x,y] = 1$ and yRz if $M_R[y,z] = 1$; so rock y is useful for the frog if $M_S[x,y] = 1 \wedge M_R[y,z] = 1$. The frog can catch the fly if $M_S[x,y] = 1 \wedge M_R[y,z] = 1$ for at least one y in B – that is, if the disjunction over all y in B of the conjunctions $M_S[x,y] = 1 \wedge M_R[y,z] = 1$ is equal to 1. To calculate this disjunction, the algorithm given in some elementary texts runs all the way through row x of M_S and column z of M_R, evaluating all the conjunctions and updating their disjunction. This is what you do when you multiply two matrices, except that here you replace addition by disjunction and multiplication by conjunction. The essential difference is that here $1 + 1 = 1$ instead of 2. If you multiplied these two matrices the usual way, you would calculate the number of ways in which the frog could catch the fly, whereas with this sort of multiplication you are determining whether there is at least one way for the frog to catch the fly. The product you get is called the *Boolean product*, denoted by $M_R \otimes M_R$.

But the average frog is smarter than such an algorithm. If in the above example the frog is on rock 1 in A and the fly is on rock 1 in C, then after looking at rock 1 in B, the frog knows that it can catch the fly. It isn't going to take the time to look at any of the other rocks in B to determine whether there is more than one way to catch the fly, because the fly may not stick around long enough. As soon as the frog finds a way to get to the fly, it will hop to it. A smart algorithm, or at least a smart algorithm designer, should be able to simulate the decision-making

capability of a smart frog and should set $M_{R\circ S}[x,z]$ equal to 1 as soon as a y is found such that $M_S[x,y] = 1$ and $M_R[y,z] = 1$. Can you?

9.2 Relations on a set and their properties

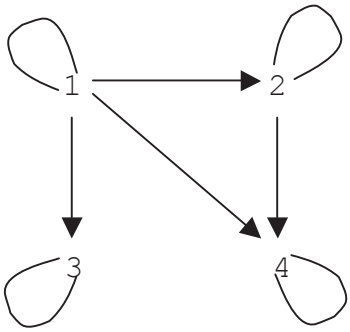
A relation R *on* a set S is a relation from S to S. Since the domain and the codomain of R are the same, it isn't necessary to draw two copies of it. The graph of R will have S as its set of vertices and there will be an arc (x,y) if and only if xRy. Such a graph will be a directed graph in which loops are allowed but not multiple arcs; so it will be called a directed graph with no other adjectives in our notation. The matrix of this graph is just M_R (see the diagram below). This method of representing a relation is more economical than drawing two copies of S, but to find $R\circ R$ it may be easier to draw 3 copies of S rather than look for paths of length 2 in the graph drawn with only one copy of S (try it both ways and decide for yourself).



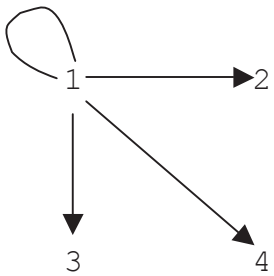

```
1 1 1 1
0 0 0 1
0 0 0 0
0 0 0 0
```

A relation R on a set S is called *reflexive* if xRx for every x in S and *irreflexive* if xRx for no x in S. For example, on the set of real numbers, the relation $\geq$ is reflexive because $x \geq x$ for every real number x, the relation $>$ is irreflexive because no number is greater than itself, and the relation $y \geq x^2$ is neither reflexive nor irreflexive because $1/2 \geq (1/2)^2$ but $2 < 2^2$.

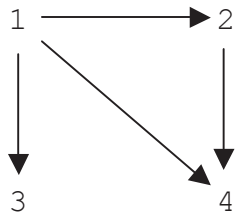
In the diagrams below, the relation on the left is reflexive, the one on the right is irreflexive and the one in the middle is neither reflexive nor irreflexive.




```
1 1 1 1
0 1 0 1
0 0 1 0
0 0 0 1
```




```
1 1 1 1
0 0 0 1
0 0 0 0
0 0 0 0
```




```
0 1 1 1
0 0 0 1
0 0 0 0
0 0 0 0
```

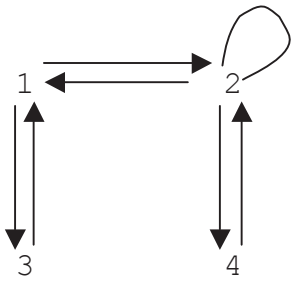
From the graphs and the matrices above it is clear that a relation is reflexive if and only if the graph has a loop at every vertex and the matrix has a 1 in every place in the *principal diagonal* (the one from the top left corner to the bottom right corner) and a relation is irreflexive if and only if the graph has no loops and the matrix has a 0 in every place in the principal diagonal. For the graph of a reflexive relation, the forbidden configuration is a a vertex without a loop (here a dashed line or curve indicates the absence of an arc) o

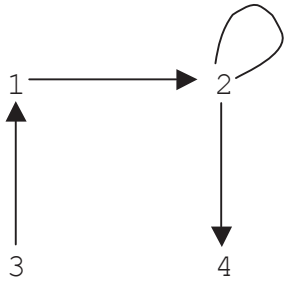
and for the graph of an irreflexive relation, the forbidden configuration is a loop. o

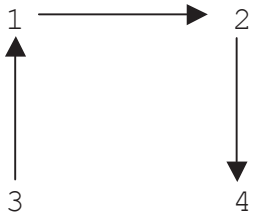
It takes $O(n)$ operations to determine whether a relation R on a set S with n elements is reflexive or irreflexive or neither – you test the n elements on the principal diagonal of M_R.

A relation R on a set S is called *symmetric* if for every x and y in S, $xRy \leftrightarrow yRx$. For example, the relation that a line is parallel to another line is symmetric, and so is the relation that they are perpendicular. A relation R on a set S is called *asymmetric* if xRy and yRx cannot both be true for any x and y in S. For example, the relation $>$ is asymmetric because there is no pair x,y such that $x > y$ and $y > x$. The relation $\geq$ is not asymmetric because $x \geq y$ and $y \geq x$ can both be true if $x = y$. A relation R on a set S is called *antisymmetric* if xRy and yRx cannot both be true for any **distinct** x and y in S. Another way of stating this condition is that if xRy and yRx, then $x = y$. The relation $\geq$ is antisymmetric and so is the relation that x divides y on the set of positive integers because if x and y are positive integers and x divides y, then $x \leq y$, but this relation is neither symmetric nor antisymmetric on the set of non-zero integers because 1 divides -1 and -1 divides 1 but 1 and -1 are distinct. The relation = is both symmetric and antisymmetric: if $x = y$, then $y = x$, and if $x = y$ and $y = x$, then $x = y$. Conversely, any relation that is both symmetric and antisymmetric is contained in the relation of equality: if xRy, then yRx by symmetry, so that $x = y$ by antisymmetry. The only relation on any set that is symmetric, antisymmetric and reflexive is the relation of equality: if xRy, then $x = y$ by symmetry and antisymmetry, and if $x = y$, then xRy by reflexivity, so that xRy if and only if $x = y$.

In the diagrams below, the relation on the left is symmetric, the relation on the right is asymmetric and the relation in the middle is antisymmetric but not asymmetric.





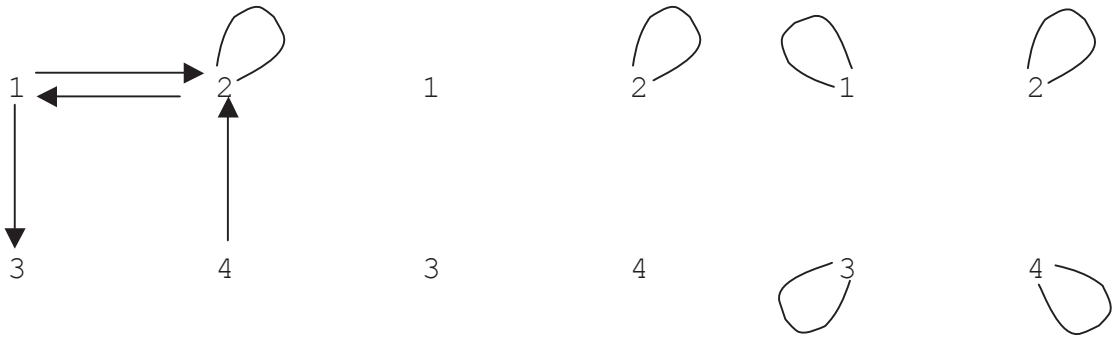



```
0 1 1 0
1 1 0 1
1 0 0 0
0 1 0 0
```

```
0 1 0 0
0 1 0 1
1 0 0 0
0 0 0 0
```

```
0 1 0 0
0 0 0 1
1 0 0 0
0 0 0 0
```

In the diagrams below, the relation on the left is neither symmetric nor anti-symmetric, the one on the right is symmetric and antisymmetric and reflexive and the one in the middle is symmetric and antisymmetric but not reflexive.

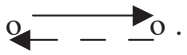


0	1	1	0
1	1	0	0
0	0	0	0
0	1	0	0

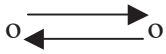
0	0	0	0
0	1	0	0
0	0	0	0
0	0	0	0

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

From the graphs and matrices above we can deduce the following tests for these properties. A relation is symmetric if and only if for every non-loop arc (x,y) in the graph there is a return arc (y,x) and the matrix is symmetric: every element not on the principal diagonal sees an element equal to itself when it uses the principal diagonal as a mirror. The forbidden configuration for a symmetric relation is an arc without a return arc:



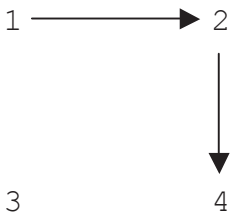
The forbidden configuration for an antisymmetric relation is an arc with a return arc:

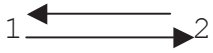


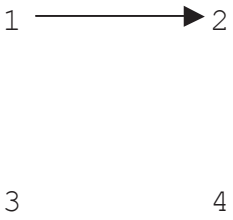
A relation is asymmetric if and only if it is both antisymmetric and irreflexive; so it has two forbidden configurations: an arc with a return arc and a loop. To test a relation on a set S with n elements for symmetry or antisymmetry you check the off-diagonal elements to see whether a 1 sees a 1 in the mirror (if so, no antisymmetry) or a 1 sees a 0 (if so, no symmetry) and to test for asymmetry you check the diagonal elements too; so these tests take $O(n^2)$ operations.

A relation R on a set S is called *transitive* if for all x, y and z in S, if xRy and yRz, then xRz. An example of a transitive relation is >: if a cow is bigger than a dog and the dog is bigger than a mouse, then the cow is (much) bigger than the mouse. It's funnier in French: si une vache est plus grande qu'un chien et le chien est plus grand qu'une souris, alors la vache est vachement plus grande que la souris. Other transitive relations: one integer divides another, one set is a subset of another, one function is an estimate of another (you prove that one). The relation that a line on a plane is perpendicular to another line on the same plane is not transitive: if line x is perpendicular to line y and line y is perpendicular to line z, then line x is parallel to line z.

The relation drawn in the first diagram in this section is transitive and so are the three relations drawn just below it – the left one is the relation that one of the integers 1,2,3,4 divides another. Of the three relations shown below, the left one and the middle one are not transitive but the right one is.

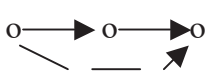






$$\begin{matrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{matrix} \qquad \begin{matrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{matrix} \qquad \begin{matrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{matrix}$$

The first one isn't transitive because 1R2 and 2R4 but 1R4 is false. The second one isn't transitive because 1R2 and 2R1 but 1R1 is false (also, 2R1 and 1R2 but 2R2 is false). In the proposition that if xRy and yRz then xRz, set $z = x$; then it becomes the proposition that if xRy and yRx, then xRx. For transitivity there are two forbidden configurations: two consecutive arcs without an arc from the initial vertex to the terminal one and an arc with a return arc but without a loop at both incident vertices.



Note that setting $y = x$ does not add any more configurations because if xRx and xRz then xRz for any relation, and the same holds true for setting $y = z$; so these are the only two forbidden configurations. The relation on the right has neither of these two configurations; so it is transitive. For all x, y, z, the proposition that $xRy \wedge yRz$ is false; so the implication that $(xRy \wedge yRz) \rightarrow xRz$ is true whether or not xRz is true.

How would you test a relation R for transitivity using the matrix M_R? You could, of course, test, for every triple (x,y,z), whether $M_R[x,y] = 1$ and $M_R[y,z] = 1$ but $M_R[x,z] = 0$, but, like the smart frog, you want to stop once you know the answer. Here is an algorithm that does so.

```
boolean function Transitive(n: natural; M: array[1..n][1..n]  of boolean)
                                                        {boolean means true or false}
      {Transitive[n,M] = true if the relation whose matrix is M is transitive and false otherwise.}
   local variables x, y, z: natural;
   for x←1 to n do
      for z←1 to n do
         if M[x,z] = 0 then
            for y←1 to n do
               if M[x,y]=1 and M[y,z]=1 then return false end if;
            end for y;
         end if;
      end for z;
   end for x;
   return true;
end transitive.
```

In the worst case this algorithm does $O(n^3)$ operations because there are 3 nested loops, each doing n iterations. Is this a good estimate? Suppose that $M[x,z] = 0$ for all x and z. Then the inner loop will get iterated n^3 times; so in this case the algorithm is neither better nor worse than testing every triple, but in any other case, some time will be saved.

If $M[x,z] = 0$, the inner loop calculates $(M \otimes M)[x,z]$ using the smart-frog approach and declares the relation not to be transitive if $(M \otimes M)[x,z] = 1$; if this never happens, the algorithm declares the relation to be transitive. Its correctness is based on the following theorem (you prove it).

A relation R on a finite set S is transitive if and only if for each x and z in S, $(M_R \otimes M_R)[x,z] = 1 \rightarrow M_R[x,z] = 1$.

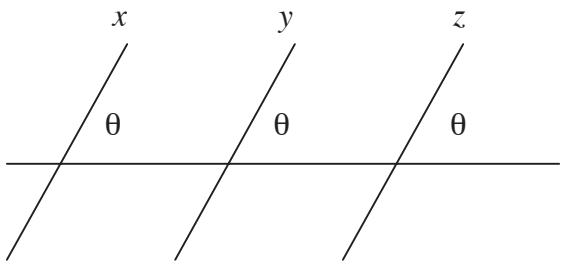
9.3 Equivalence relations

A relation R on a set S is called an *equivalence relation* if it is reflexive, symmetric and transitive. For each element $x \in S$, the set $\{y \in S: xRy\}$ of elements y of S such that x is related to y is called the *class of x under R* and is denoted by $[x]_R$, or just $[x]$ if only one relation is being discussed.

Example 1. The most obvious equivalence relation, and the one responsible for the name equivalence, is the relation of equality on any set. For this relation, $[x]=\{x\}$ for any x.

Example 2. The relation R on the set of real numbers defined by xRy iff either $x = y$ or $x = -y$ is an equivalence relation. Since $x = x$ for any x, R is reflexive. If $x = y$, then $y = x$, and if $x = -y$, then $y = -x$; so R is symmeteric. Suppose that xRy and yRz. Then there are four possibilities. If $x = y$ and $y = z$, then $x = z$. If $x = y$ and $y = -z$, then $x = -z$. If $x = -y$ and $y = z$, then $x = -z$. Finally, if $x = -y$ and $y = -z$, then $x = z$. In each of those cases xRz ; so R is transitive. Under this relation, $[0] = \{0\}$ and $[x] = [-x] = \{x,-x\}$ for every real number $x \neq 0$.

Example 3. Let S be the set of lines on a plane and R be the relation on S defined by xRy if x is parallel to y – that is, x and y do not have a single point of intersection. According to this definition, any line is parallel to itself; so R is reflexive. The expression "x and y" is the same as "y and x"; so R is symmetric. To prove that R is transitive, we use a theorem of Euclidean geometry that says that two lines are parallel if and only if they make the same angle with a transversal (another line parallel to neither of them). From the diagram below the reader should be able to derive a proof of the transitivity of R.



Under this relation, $[x]$ is the set of lines that are parallel to x; by the above theorem, $[x]$ is the set of lines making the same angle as x with a transversal.

Example 4. Let R be the relation on any set S defined by xRy for all x and y in S. This is (trivially) an equivalence relation and $[x] = S$ for all x.

Example 5. Let S be the set of integers and m an integer greater than 1, and let R be the relation on S defined by xRy if m divides $x - y$. Since m divides $x - x = 0$, R is reflexive. If m divides $x - y$, then m divides $y - x = -(x-y)$; so R is symmetric. If m divides both $x - y$ and $y - z$, then m divides $x - z = (x-y) + (y-z)$; so R is transitive. Under this relation, $[x]$ is the set of integers y such that m divides $x - y$ so that $y = x + km$ for some integer k. For $m = 5$ we have the following equivalence classes:

$[0] = \{\ldots, -15, -10, -5, 0, 5, 10, 15, \ldots\}$
$[1] = \{\ldots, -14, -9, -4, 1, 6, 11, 16, \ldots\}$
$[2] = \{\ldots, -13, -8, -3, 2, 7, 12, 17, \ldots\}$
$[3] = \{\ldots, -12, -7, -2, 3, 8, 13, 18, \ldots\}$
$[4] = \{\ldots, -11, -6, -1, 4, 9, 14, 19, \ldots\}$.

Note that $[5] = [0]$, $[6] = [1]$ and so on, so that the 5 classes $[0],\ldots,[4]$ cover S.

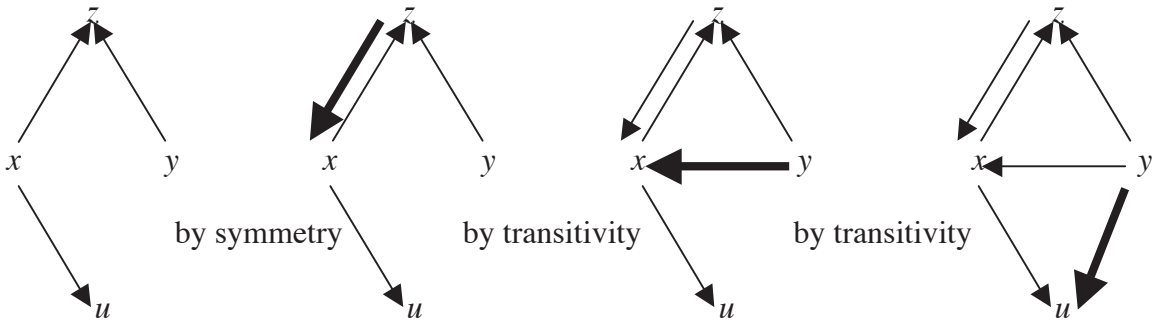
In each of those five examples, the equivalence classes are non-empty, they cover S (their union is equal to S) and any two distinct classes are disjoint. Is this a coincidence or is this a property common to all equivalence relations?

Theorem. Let R be an equivalence relation on any set S. Then the equivalence classes under R have the following properties:

1) every equivalence class is non-empty;
2) every element of S is in some equivalence class;
3) any two distinct equivalence classes are disjoint, so that no element of S is in more than one equivalence class.

Proof. To prove properties 1 and 2, we use the fact that R is reflexive – that is, xRx for any x in S – so $x \in [x]$. It follows that $[x]$ is not empty because it contains at least the element x (property 1) and that x belongs to at least the equivalence class $[x]$ (property 2).

To prove property 3 we use the fact that R is symmetric and transitive. We prove the contrapositive of property 3: if $[x]$ and $[y]$ are not disjoint, then they are equal (as sets). Suppose that $[x]$ and $[y]$ are not disjoint, so that there is an element z of S in both $[x]$ and $[y]$. Then xRz and yRz. To show that $[x] \subseteq [y]$, we need to show that if u is any element of $[x]$, then $u \in [y]$ – that is, if xRu, then yRu. In the leftmost diagram below, we show the facts we already know: xRz, yRz and xRu. Since xRz and R is symmetric, zRx (see the second diagram below). Since yRz and zRx and R is transitive, yRx (see the third diagram). Finally, since yRx and xRu and R is transitive, yRu (see the last diagram).



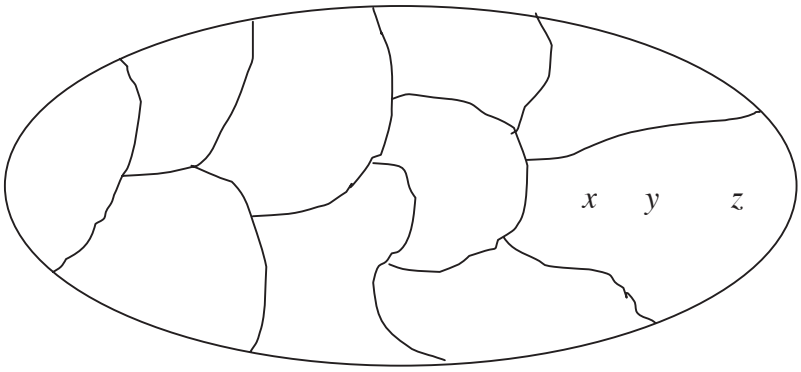
This proves that $[x] \subseteq [y]$. To prove that $[x] = [y]$ we need to prove that $[y] \subseteq [x]$. To do this, we repeat the proof that $[x] \subseteq [y]$, replacing each occurrence of x by y and each occurrence of y by x. Once you do this, you will be entitled to write QED.

Suppose we choose one representative of each family of identical equivalence classes. Then we have a **set** P of non-empty subsets of S that are pairwise disjoint and whose union covers S, so that each element of S belongs to exactly one member of P. The set P is called a *partition* of S and the members of P are called the *parts* of S. The above theorem says that **the distinct representatives of the equivalences classes of an equivalence relation R on a set S constitute a partition of S**.

Conversely, suppose we have a partition P of a set S. Since each element of S belongs to exactly one member of P, for each x in S we can define uniquely the part of S to which x belongs. Now let R be the relation on S defined by xRy if x belongs to the same part of S as y. Since x belongs to the same part as x, R is reflexive. If x belongs to the same part as y, then y belongs to the same part as x; so R is symmetric. If x belongs to the same part as y and y belongs to the same part as z, then x belongs to the same part as z; so R is transitive. Thus R is an equivalence relation, and for each x in S, $[x]$ is the part of S to which x belongs. We have just proved the converse of the previous theorem: **a partition P of a set S defines an equivalence relation whose equivalence classes are the parts of S under the partition P.**

In the diagram below, S is an Easter egg that has been coloured by a child. Each region of the egg has been coloured a different colour. The regions are not empty, every bit of the surface

of the egg has been coloured, and no part has been coloured with more than one colour; so the regions constitute a partition of the surface of the egg. By examining this egg and the location of x, y and z on its surface, you should be able to understand the above proof.



The following theorem is related to the previous one and could be proved from it as a corollary.

Theorem. Let R be a relation on a set S. Suppose there is a function f from S to some set such that $f(x) = f(y)$ if and only if xRy. Then R is an equivalence relation.

Proof. Since $f(x) = f(x)$ for any x, R is reflexive. If $f(x) = f(y)$, then $f(y) = f(x)$; so R is symmetric. If $f(x) = f(y)$ and $f(y) = f(z)$, then $f(x) = f(z)$; so R is transitive.

In example 1, $f(x) = x$ for any x. In example 2, $f(x)$ could be $|x|$ or x^2. In example 3, $f(x)$ is the angle that the line x makes with a given line y ($f(x)$ is defined to be 0 if x is parallel to or identical to y). In example 4, $f(x)$ has the same value for any x. Finally, in example 5, $f(x) = x \bmod m$. I leave you to prove that $f(x) = f(y)$ if and only if xRy for the first four examples. Here is a proof for example 5. By the formula for the quotient and remainder of a division, there are integers q and r such that $x = qm + r$ and $0 \le r < m$, and there are integers k and l such that $y = km + l$ and $0 \le l < m$. Now $r = x \bmod m$ and $l = y \bmod m$. If $x \bmod m = y \bmod m$, then $l = r$ so that $x - y = qm - km = (q-k)m$, a multiple of m; thus xRy. Conversely, supppose that m divides $x - y = (q - k)m + (r - l)$. Then $r - l$ must also be a multiple of m. Now $0 \le r < m$ and $0 \le l < m$; so $-m < r - l < m$. To see this, imagine that you start at 0 and go r units to the right and then l units to the left, so that your net displacement is $r - l$. You can't go as many as m units to the right or less than 0 units to the left; so your net displacement can't be as much as m units to the right. You can't go less than 0 units to the right or as many as m units to the left; so your net displacement can't be as much as m units to the left – that is, $-m$ units. Now the only multiple of m that lies strictly between $-m$ and m is 0; so $r - l = 0$ and $x \bmod m = y \bmod m$.

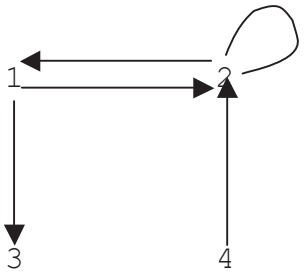
There are examples for which this theorem makes it easier to prove that a relation is an equivalence relation. It is easier to prove that $x = y$ or $x = -y$ if and only if $|x|=|y|$ than to prove that the relation of example 2 is an equivalence relation directly. Here is a more extreme example. Let S be the set of ordered pairs of non-zero real numbers and let R be the relation on S defined by $(a,b)R(c,d)$ if and only if $a/c = b/d$. Here is the direct proof that R is an equivalence

relation. To show that $(a,b)R(a,b)$, substitute a for c and b for d in the definition of R: $(a,b)R(a,b)$ if and only if $a/a = b/b$, which is true because both fractions are equal to 1. Suppose that $(a,b)R(c,d)$ so that $a/c = b/d$. To prove that $(c,d)R(a,b)$ you have to prove that $c/a = b/d$, which you do by inverting both of the fractions a/c and b/d. Finally, suppose that $(a,b)R(c,d)$ and $(c,d)R(e,f)$ so that $a/c = b/d$ and $c/e = d/f$. To prove that $(a,b)R(e,f)$ you have to prove that $a/e = b/f$, which you do by multiplying the equation $a/c = b/d$ by the equation $c/e = d/f$. Now here is the short proof: $a/c = b/d$ if and only if $a/b = c/d$; so the function $f((a,b)) = a/b$ satisfies the sufficient condition for R to be an equivalence relation.

9.4 Closures of relations

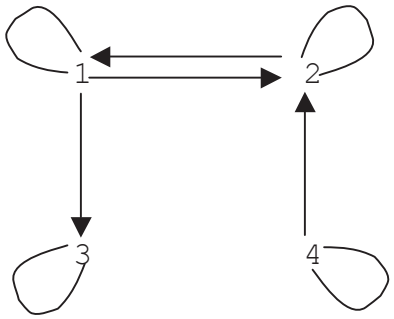
Let R be a relation on a set S. The *reflexive closure* of R is the relation obtained from R by adding just the ordered pairs necessary to make it reflexive. For example, suppose S is the set of natural numbers and R is the relation $<$. Then R consists of all the pairs (x,y) such that $x < y$. To make R reflexive you have to add the pairs (x,y) such that $x = y$, and now you have the relation $\leq$. In general, if R is defined by a condition under which xRy, then to get the reflexive closure of R, you add "or $x = y$" to that condition. In this example, the condition "$x < y$" becomes "$x < y$ or $x = y$", which is equivalent to "$x \leq y$".

Now suppose that S is of finite size n and R is represented by a graph or a matrix. How do you get the reflexive closure of R? Suppose, for example, that R is represented by the graph and the matrix below.




```
0 1 1 0
1 1 0 0
0 0 0 0
0 1 0 0
```

To make R reflexive you have to add the pairs (1,1), (3,3) and (4,4). Now the graph and the matrix have been changed to the ones shown below.

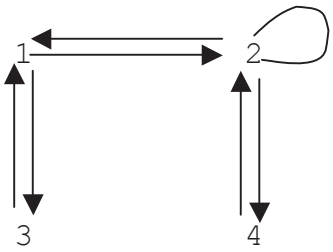



```
1 1 1 0
1 1 0 0
0 0 1 0
0 1 0 1
```

To get from R to its reflexive closure, you draw loops on all the vertices (withoout loops) of the graph of the relation and you change all the 0s to 1s in the principal diagonal of the matrix, which takes $O(n)$ operations.

The *symmetric closure* of R is the relation obtained from R by adding just the ordered pairs necessary to make it symmetric. In the infinite example above, S is the set of natural numbers and R is the relation $<$, which consists of the pairs (x,y) such that $x < y$. To make R symmetric, you have to add the pairs (x,y) such that $y < x$, and you now have the relation $\neq$. In general, if R is defined by a condition under which xRy, then to get the symmetric closure of R, you make the disjunction of that condition and a condition equivalent to yRx. In this example, the condition "$x < y$" becomes "$x < y$ or $y < x$", which is equivalent to "$x \neq y$".

Now suppose that S is of finite size n and R is represented by a graph or a matrix. In the finite example above, since (1,3) and (4,2) are arcs that don't have return arcs, you have to add the return arcs (3,1) and (2,4), so that the graph becomes the one that is shown below together with its matrix, in which the elements in positions (3,1) and (2,4) have been changed from 0 to 1.

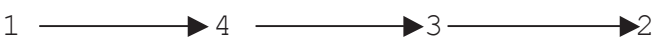



```
0 1 1 0
1 1 0 1
1 0 0 0
0 1 0 0
```

In the graph of a relation, for every arc without a return arc you draw the return arc. In the matrix, for every off-diagonal 1 that sees a 0 in the principal-diagonal mirror, you change that 0 to 1: that is, for every x and y, if $M_R[x,y] = 1$ and $M_R[y,x] = 0$, then $M_R[y,x] \leftarrow 1$ (it isn't really necessary to test whether $M_R[y,x] = 0$). This takes $O(n^2)$ operations.

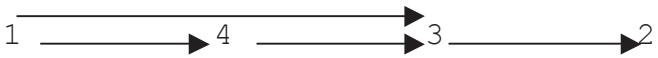
The *reflexive and symmetric closure* of R is the relation obtained from R by adding just the ordered pairs necessary to make it reflexive and symmetric. You could first make the reflexive closure and then the symmetric closure of the reflexive closure or do it the other way around and the resulting relation will be the same. This is obvious for graphs and matrices because adding loops and adding return arcs are independent operations as are changing diagonal elements and changing off-diagonal elements (you try it with the finite example above). In the infinite example, the symmetric closure of $<$ is $\neq$, and to get the reflexive closure of $\neq$ you make the disjunction with $=$, so that x is related to y by the reflexive and symmetric closure if $x \neq y$ or $x = y$, that is, for all x and all y.

The *transitive closure* of R is the relation obtained from R by adding just the ordered pairs necessary to make it transitive. Finding the transitive closure of a relation is trickier than finding the reflexive or symmetric closure. You may think that all you have to do is to look at all the ordered triples (x,y,z) and add the pair (x,z) if xRy and yRz, but if you look at the triples in the wrong order the resulting relation may not be transitive, as the following example will show. Suppose R is the relation whose graph is shown below.



Suppose you examine the ordered triples in the order given in the algorithm above that determines whether a relation is transitive: x goes from 1 to 4 in the outer loop, z does the same thing in the intermediate loop and so does y in the inner loop. When $x = 1$ and $z = 2$, there is no y

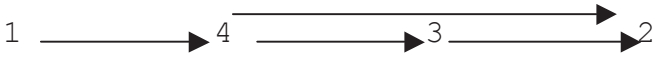
such that xRy and yRz; so no new pairs are added. When $x = 1$ and $z = 3$, there is such a y ($y = 4$); so the pair (1,3) is added and the graph now looks like the one below.



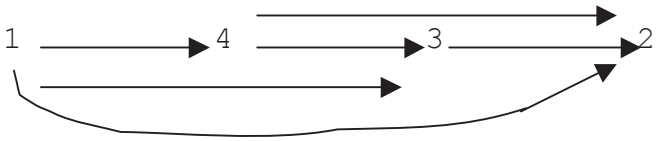
Now since 1 is related to 3 and 3 is related to 2, for the relation to be transitive, 1 has to be related to 2. But you have already exhausted all the triples with $x = 1$ and $z = 2$; so the pair (1,2) will never be added.

The same thing happens if y changes in the intermediate loop. You don't add the pair (1,3) until y gets to 4, and by then it's too late to add the pair (1,2), which should have been done when y was equal to 3.

But now suppose that y changes in the outer loop. When $y = 3$, $x = 4$ and $z = 2$, since $4R3$ and $3R2$, the pair (4,2) will be added and the graph looks like the one below.



Later on, y will advance to 4. When x=1 and z=2, since 1 is related to 4 and 4 is now related to 2, the pair (1,2) will get added, and then, when z advances to 3, the pair (1,3) will get added. The graph now looks like the one below, and you can verify that this relation is transitive.



Will this approach always work? If you examine the original graph and the final one, you will notice that for every pair (x,z), if there is a path of length > 0 from x to z in the original graph, then there will be an arc (x,z) in the final one. In general, **for every pair of vertices (x,z) in the graph of a relation R, if there is a path of length > 0 from x to z, then there will be an arc (x,z) in the graph of the transitive closure of R.** We prove this proposition by induction on the length n of the path from x to z.

Basic step: $n = 1$. A path of length 1 is an arc; so this path is the required arc.

Induction step. Suppose that $n \geq 1$ and that the proposition is true for paths of length n. Suppose that the graph of R has a path of length $n + 1$ from x to z. Let y be the second-last vertex on this path. Then there is a path of length n from x to y and an arc from y to z. By the induction hypothesis, the graph of the transitive closure of R has an arc from x to y, and it also has an arc from y to z. For the relation to be transitive, there must also be an arc from x to z, QED.

Suppose you take the graph of a relation R and draw an arc (x,z) for every pair of vertices (x,z) such that there is a path of length > 0 from x to z. This is a necessary condition for the resulting relation T to be transitive, but is it a sufficient condition? Well, if xTy, then in the graph of R there is a path of positive length from x to y, and if yTz, then there is also a path of positive length from y to z. By following the first path from x to y and then the second path from y to z you get a path of positive length from x to z, so that the graph of T must contain the arc (x,z) and thus xTz. **The graph of the transitive closure of a relation R is obtained from the graph of R**

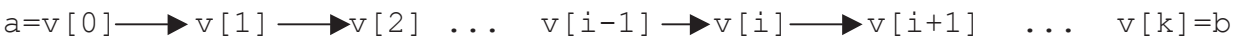
by drawing an arc from *x* to *z* for every pair of vertices (*x*,*z*) such that there is a path of positive length from *x* to *z* in the graph of *R*.

Let *S* be the set of integers and *R* the relation on *S* defined by *xRy* if $y = x + 1$. Use the above theorem to prove that the transitive closure of *R* is the relation <.

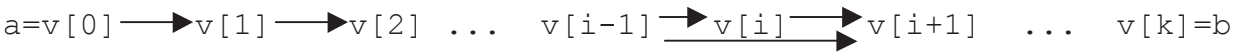
Using BFS we can determine, for each pair (*x*,*z*), whether there is a positive-length path from *x* to *z*, but there is an easier way. The algorithm below looks at all the ordered triples (*x*,*y*,*z*), with *y* changing in the outer loop, and draws the arc (*x*,*z*) if there is an arc (*x*,*y*) and an arc (*y*,*z*).

```
procedure Transitive_closure(G: graph of a relation R)
   for every vertex y in G do
      for every non-loop arc (x,y) entering y do
         for every non-loop arc (y,z) exiting y do
            if there isn't an arc (x,z) then draw one end if;
         end for (y,z);
      end for (x,y);
   end for y;
end Transitive_closure.
```

This algorithm has already been traced on the graph drawn above and it worked on this graph. Will it always work? Let *a* and *b* be two vertices of *G* such that there is a positive-length path from *a* to *b* (see the diagram below).



During the execution of this algorithm, *y* will be set to each of the vertices of *G* including all the intermediate vertices *v*[1], *v*[2],…,*v*[*k*-1] of the path from *a* to *b*. Let *v*[*i*] be the (chronologically) first intermediate vertex of the path such that *y* is set equal to *v*[*i*]. While *y* is equal to *v*[*i*], *x* will be set equal to *v*[*i* -1] and *z* will be set equal to *v*[*i* +1]. Since there is an arc from *x* to *y* and an arc from *y* to *z*, if there isn't already an arc from *x* to *z*, the algorithm will draw one; so that either way there will now be an arc from *x* to *z*. There is now a path from *a* to *b* that avoids *v*[*i*] and doesn't contain any of the vertices that weren't originally there (see the diagram below).



The same thing will happen with the second intermediate vertex of the path to which *y* is assigned, and the third one, and so on, until a path has been made from *a* to *b* that contains none of the intermediate vertices of the original path and no other vertices either. This path is an arc from *a* to *b*. If there is a positive-length path from *a* to *b*, the algorithm will draw an arc from *a* to *b*. If there isn't a positive-length path from *a* to *b*, then the algorithm will not draw an arc from *a* to *b*, because it only draws an arc from *x* to *z* if there is a path of length 2 from *x* to *z*. The algorithm correctly draws the graph of the transitive closure of the relation whose graph is *G*.

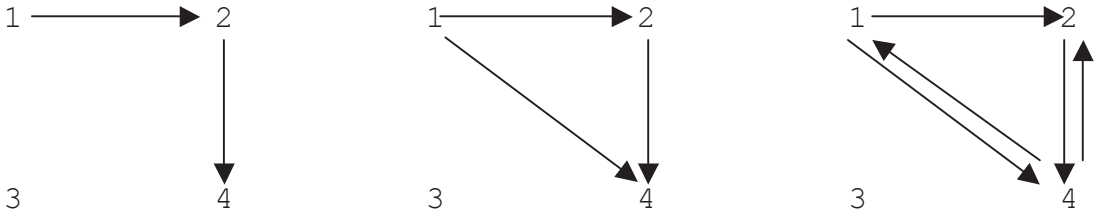
If *G* is represented by a matrix *M*, then the algorithm becomes the one shown below, which is Warshall's algorithm.

```
procedure Warshall(n: natural; M: matrix[1..n][1..n] of a relation R)
local variables x,y,z: natural;
   for y ← 1 to n do
      for x ← 1 to n do
         for z ← 1 to n do
            if M[x,y] = 1 and M[y,z] = 1 then M[x,z] ← 1 end if;
         end for z;
      end for x;
   end for y;
end Warshall.
```

This algorithm takes $O(n^3)$ operations because there are 3 nested loops, each of which is iterated n times. The correctness proof of this algorithm in some elementary texts uses the last intermediate vertex in the path and an implied loop invariant. I think my proof is easier to understand.

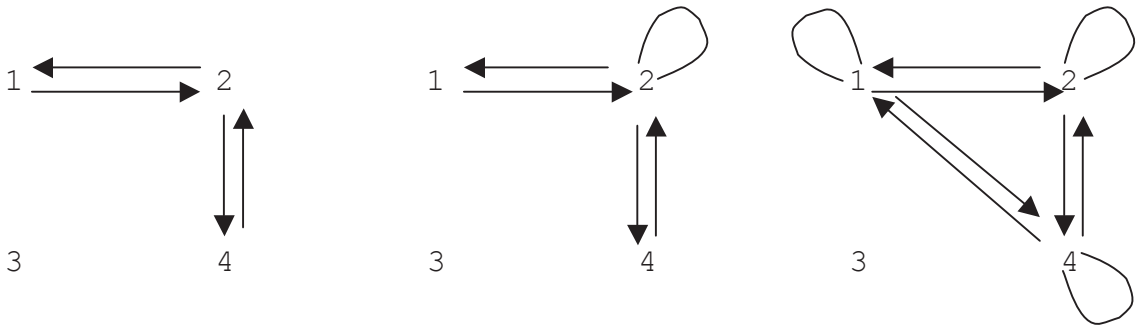
The *reflexive and transitive closure* of a relation R is the relation obtained from R by adding just the pairs necessary to make it reflexive and transitive. Adding all the pairs (x,x) beforehand to get the reflexive closure doesn't change the pairs (x,y) of distinct vertices that need to be added to get the transitive closure because if xRx and xRy then xRy; so you can take these two closures in either order.

The *symmetric and transitive closure* of a relation R is the relation obtained from R by adding just the pairs necessary to make it symmetric and transitive. Let R be the relation whose graph is shown on the left below. Its transitive closure T is shown in the middle and the symmetric closure of T is shown on the right.



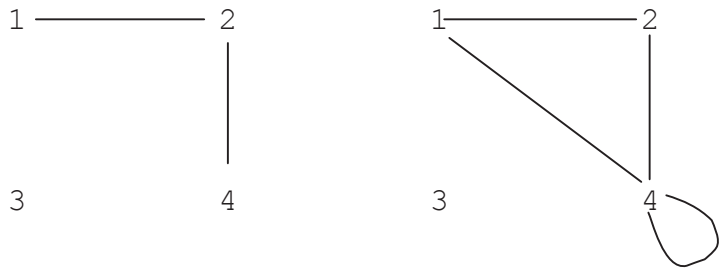
Is that relation transitive? It contains one of the forbidden configurations – an arc with a return arc but no loop at either vertex – so the relation is not transitive.

Now we start with the original relation and make the symmetric closure S (the diagram on the left below).



To make the transitive closure of this relation S, we apply the graph version of Warshall's algorithm. When $y = 1$, there is one entering arc (2,1) and one exiting arc (1,2); so you have to draw the loop (2,2) (the middle diagram above). When $y = 2$, there are two entering arcs (1,2) and (4,2) and two exiting arcs (2,1) and (2,4). The pair of arcs (1,2), (2,1) makes you draw the loop (1,1). The pair of arcs (1,2), (2,4) makes you draw the arc (1,4). The pair of arcs (4,2), (2,1) makes you draw the arc (4,1). The pair of arcs (4,2), (2,4) makes you draw the loop (4,4). Now the graph looks like the one on the right above. When $y = 3$ there are no entering arcs and no exiting arcs; so no extra arcs have to be drawn. When $y = 4$ there are two entering arcs (1,4) and (2,4) and two exiting arcs (4,1) and (4,2). The pair of arcs (1,4), (4,1) would make you draw the loop (1,1), but that loop is already there. The other three pairs of arcs too add no new arcs; so the graph on the right above is the transitive closure of the graph on the left – and it is symmetric.

The symmetric and transitive closure of a relation is made by first taking the symmetric closure and then the transitive closure and not the other way around. But Warshall's algorithm is not the most efficient way of constructing the transitive closure of a symmetric relation. In the graph of a symmetric relation, you can replace each pair of arcs (x,y), (y,x) by the edge $\{x,y\}$ and get an undirected graph. For the directed graph on the left above, the undirected version is shown on the left below, and for the directed graph on the right above, the undirected version is shown on the right below.



Now how would you get from the graph on the left above to the graph on the right? Notice that the graph on the left is not connected. It has two "pieces", one consisting of the vertices 1,2,4 and one consisting of the vertex 3. In the graph on the right, every pair of vertices in the same piece is connected by an edge and there is a loop at every vertex except vertex 3, which is *isolated* (degree 0). Does this process always construct the transitive closure of a symmetric relation?

First, let's define "piece" more rigourously. Given an undirected graph G with vertex set V, define the relation R on V as xRy if there is a path in G from x to y. This relation is reflexive because there is a path of length 0 from x to x for every vertex x. The relation is symmetric because the graph is undirected; so if there is a path from x to y, following the path backwards makes a path from y to x. And the relation is transitive because if there is a path from x to y and a path from y to z, then by following the first path from x to y and then the second path from y to z you get a path from x to z. Thus R is an equivalence relation. The equivalence classes are called the *connected components* of G, and for each vertex x, $[x]$ is the set of vertices y such that there is a path from x to y.

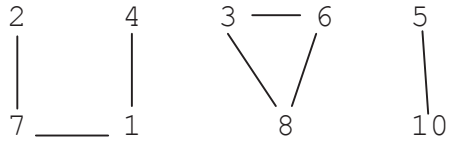
Suppose that x and y are two distinct vertices in the same connected component of G. Then there is a positive-length path from x to y in the directed version of G, so that there must be an arc from x to y in the transitive closure of (the relation whose graph is) G. But since the

relation is symmetric, there must be a path from y to x in G; so there is also an arc from y to x in the transitive closure of G, and in the undirected version of G there is an edge $\{x,y\}$. If x and y are in different connected components of G, then there is no path from x to y, so there is no edge $\{x,y\}$ in the transitive closure of G. Now suppose that x is a non-isolated vertex. Then x has at least one neighbour y. In the undirected version of G there is an edge $\{x,y\}$; so in the directed version there are the arcs (x,y) and (y,x). But for the relation to be transitive, there must be a loop at the vertex x. On the other hand, if x is an isolated vertex, there is no positive-length path from x to any vertex; so in the transitive closure there is no arc from x to any vertex including x, hence no loop at x. **To get the transitive closure of a symmetric relation, divide the undirected graph of the relation into connected components, draw an edge between every pair of non-adjacent vertices in the same component and draw a loop at every non-isolated vertex that doesn't already have a loop.**

This process is clearly easier to execute by hand than Warshall's algorithm. Is it any more efficient by computer? Remember that at the end of Chapter 8 I promised you that there would be another application of BFS in Chapter 9? A slight modification of BFS divides a graph into connected components. You don't initialize the array P to zero, and instead of setting $P[v]$ to u you set $P[v]$ to s. With this modification of BFS, the algorithm below divides an undirected graph into connected components.

```
procedure Components(n: natural; G: n-vertex graph)
   local variable s: natural;
   for s←1 to n do P[s]←0 end for;
   for s←1 to n do
      if P[s] = 0 then BFS(G,s,P) end if;
   end for;
end Components.
```

We trace this algorithm on the graph drawn below.




```
9   i= 1  2  3  4  5  6  7  8  9 10
  P[i] 0  0  0  0  0  0  0  0  0  0
  s=1: 1  1  0  1  0  0  1  0  0  0
  s=2: no change because P[2]≠0
  s=3: 1  1  3  1  0  3  1  3  0  0
  s=4: no change
  s=5: 1  1  3  1  5  3  1  3  0  5
  s=6,7,8: no change
  s=9: 1  1  3  1  5  3  1  3  9  5
  s=10: no change.  P is final.
```

After this algorithm has been executed, for each pair of vertices (x,y), x and y are in the same component if and only if $P[x] = P[y]$. For each pair of vertices x and y such that $x < y$, if $P[x] = P[y]$ and there isn't an edge between x and y, then you draw one (in matrix form, you set $M[x,y]$ and $M[y,x]$ to 1 without testing whether those elements were initially 0); this does $O(n^2)$ operations. Then, for each vertex x of degree > 0 you draw a loop at x if there isn't already a loop there (you set $M[x,x] = 1$) – this does $O(n)$ operations. If you want the transitive and reflexive

closure of a symmetric relation, or the reflexive, symmetric and transitive closure of an arbitrary relation, then you draw a loop at every vertex whether or not it is of degree 0.

How many operations does the algorithm Components do? Each execution of BFS(G,s,P) looks at all the neighbours of each vertex u in the connected component $[s]$ of G containing s; so the number of operations done by this call to BFS is proportional to n times the number of vertices in $[s]$. The total number of operations done by all the calls to BFS is thus proportional to n times the total number of vertices, which is in $O(n^2)$. The number of operations done to manage both of the loops is $O(n)$; so the algorithm Components runs in $O(n^2)$ time (or $O(m+n)$ for sparse graphs with m edges represented by adjacency lists). Since the rest of the procedure for finding the transitive closure of a symmetric relation also runs in $O(n^2)$ time, the whole procedure runs in $O(n^2)$ time, which is more efficient than Warshall's algorithm, which runs in $O(n^3)$ time. There are asymptotically faster transitive closure algorithms than Warshall's, but none of them run in $O(n^2)$ time; so this way of finding the transitive closure of a symmetric relation is an improvement over using an algorithm for finding the transitive closure of an arbitrary relation. This is another result that I obtained independently and I haven't found it in the literature yet, but I'd be very surprised if it isn't there somewhere!

CHAPTER 10. GENERATING COMBINATORIAL OBJECTS

Suppose you want to construct the truth table of some long and complicated Boolean expression with many distinct variables. Finding the truth value of such an expression for each assignment of truth values to the variables is slow and error-prone by hand but it is easy to write a computer program to do it. But then you would need to write a program to generate all the strings of 0 and 1 that represent the assignment of truth values to the variables. More generally, you may want to write a computer program to generate a list of *words* (strings of symbols) that represents a set of combinatorial objects such as subsets of a set, combinations, permutations, compositions of an integer, partitions of a set, partitions of an integer, balanced parenthesis systems, trees, graphs or maps (graphs drawn on surfaces) so that you can get your computer to study the properties of these objects faster and more accurately than you could by hand. There are many books and articles on this subject. One of the most comprehensive is [Albert Nijenhuis and Herbert S. Wilf, Combinatorial Algorithms for Computers and Calculators, Academic Press, 1978]. But I'll restrict myself to three of these objects – the strings of 0 and 1 that represent subsets of a set (and truth values), the strings of positive integers that represent permutations and the strings of 0 and 1 that represent balanced parenthesis systems.

10.1 Lexicographical order

The easiest way to generate a set of words is to generate them in *lexicographical order*, the order in which words are listed in a dictionary. Assuming that the set of *letters* (the symbols of which the words are composed) is ordered, words can be ordered by comparing the first (that is, leftmost) letter in which the words differ; if none of the letters differ, then the shorter word comes before the longer one. The word the word **string** comes before the word **supper** because they have the same first letter but the second letter of **string** comes before the second letter of **supper** and the word **ball** comes before the word **ballistic**.

Here is a list of the 16 *binary strings* (strings of 0 and 1, which are called *bits*) of length 4 in lexicographical order (ignore the number to the right of each string for the moment):

0000 1
0001 2
0010 1
0011 3
0100 1
0101 2
0110 1
0111 4
1000 1
1001 2
1010 1
1011 3
1100 1
1101 2
1110 1
1111 5
total 31

You could generate the list of binary strings of length n recursively by generating the strings of length $n - 1$ twice, putting a 0 in front of all the strings in the first list and a 1 in front of all the strings in the second list, but if you want to know what string follows the current one you need to find a non-recursive description of this list. Examining the list above, you can observe that the first string is 00…00 and that to get from one string to the next, you start from the right end of the string, changing each 1 you meet to a 0 until you come to a 0; then you change that 0 to a 1 and then quit. If you applied that algorithm to the last string 11…11, you would change it to 00…00 and never quit; so you create a bit before the first one – a sentinel – and set it to 0. Then after the real string gets changed to 00…00, the sentinel gets changed to 1, which tells you that 11…11 was the last string and the new one isn't to be processed. This is the way an odometer in a car works; for instance, the number 345999 gets changed to 346000 and the number 999999 gets changed to 000000 – the sentinel then tells you that it's time to buy a new car. With an odometer the biggest digit is 9 and with binary strings it's 1, but in either case it gets changed to the smallest digit, which is 0, and the digit to the left of the string of biggest digits gets increased by 1.

What is the total number of changes that are made when you generate the whole list? The number to the right of each binary string is the number of bits, including the sentinel, that get changed in passing from that string to the next one. The total number of bits that get changed is 31, and you may guess from this number that for binary strings of length n the total number of changes is $2^{n+1} - 1$. You could prove this assertion by standard induction on n, but if you don't know the answer in advance, you'd want to be able to derive it from scratch. There's a way to do this that takes horse power and a way that takes brain power.

The horse power method involves counting the number of strings that require i changes, multiplying by i and then summing over all possible i. If the last bit is 0, only 1 bit gets changed. This is the case for half of the 2^n binary strings of length n; that is, for 2^{n-1} strings. If the last bit is 1 but the second-last bit is 0, then 2 bits get changed. This is the case for a quarter of the strings – for 2^{n-2} strings. Similarly, 3 bits get changed in 2^{n-3} strings, and so on until all n bits get changed in 1 string (0111…1) and $n + 1$ bits including the sentinel get changed in 1 string (11…11). So the total number of changes is

$$2^{n-1} + 2 \times 2^{n-2} + 3 \times 2^{n-3} + \cdots + (n-2) \times 2^2 + (n-1) \times 2^1 + n \times 2^0 + (n+1).$$

How would you find a formula for this monstrous sum? Before attempting this feat, I'll tell you another story. One New Year's Eve a father, hoping to teach his son to work for his money, offered to pay him to do the dishes every day instead of giving him an allowance. His son agreed on condition that he get paid 1 cent on January 1, 2 cents on January 2, 4 cents on January 3, 8 cents on January 4 and so on until the end of the month. Not having studied mathematics beyond grade 10, the father agreed. After a week the father realized that the price he would have to pay would soon grow beyond his means; so he cancelled the agreement and excused his son from any more dishwashing. How much would he have had to pay his son had he continued to double the salary each day until the end of the month?

For a month with $n + 1$ days, the total price, denoted by S, is given by the formula

$$S = 2^0 + 2^1 + 2^2 + \cdots + 2^{n+1} + 2^n.$$

To be able to cancel out most of the terms in the above equation, we multiply each of them by 2:

$$2S = 2^1 + 2^2 + 2^3 \cdots + 2^n + 2^{n+1}.$$

Now we subtract the first equation from the second one. Most of the terms cancel out and we get $S = 2^{n+1} - 1$. With $n = 30$ (since January has 31 days), this means that the father would have had to pay his son \$21474836.47.

This kind of series of numbers is called a *geometric series*. The most general form of such a series is $a + ar + ar^2 + \cdots + ar^n$. I invite you to find a formula for the sum of this series in two cases: when $r \neq 1$ and when $r = 1$.

Now the series we have to sum is more complicated than a geometric series, but the same trick used to sum a geometric series will reduce the series we have to sum to a geometric series. Ignoring the term $n + 1$ for the moment, writing the rest of the series backwards and equating it to S, we get

$$S = n \times 2^0 + (n-1) \times 2^1 + (n-2) \times 2^2 + \cdots + 3 \times 2^{n-3} + 2 \times 2^{n-2} + 2^{n-1}.$$

Again we multiply by 2:

$$2S = n \times 2^1 + (n-1) \times 2^2 + (n-2) \times 2^3 + \cdots + 3 \times 2^{n-2} + 2 \times 2^{n-1} + 2^n.$$

And again we subtract the first equation from the second one:

$$S = 2^1 + 2^2 + 2^3 + \cdots + 2^{n-2} + 2^{n-1} + 2^n - n.$$

Now except for the the $-n$ we have a geometric series which is the same as the one we have already summed except that the term $2^0 = 1$ is missing; so its sum is $2^{n+1} - 2$. We have to subtract n from this sum and we also have to add $n + 1$; so the total number of changes is $2^{n+1} - 1$.

The brain power method involves counting the number of times that each bit gets changed. The rightmost bit gets changed in half the strings; that is, 2^{n-1} times. The second rightmost bit gets changed in a quarter of the strings - 2^{n-2} times - the third rightmost bit 2^{n-3} times and so on until the leftmost bit gets changed twice and the sentinel once; so the total number of changes is $2^0 + 2^1 + 2^2 + \cdots + 2^{n+1} + 2^n = 2^{n+1} - 1$. The average number of changes over all the 2^n binary strings of length n is $2 - 1/2^n$, which approaches 2 as n approaches infinity. The algorithm for generating the binary strings in lexicographical order runs in constant average time, or CAT for short. Well, if the truth be told, CAT stands for constant **amortized** time, but whatever the A stands for, I can assure you that this acronym has nothing to do with either felines or DNA sequences.

If you want to generate a random binary string, all you have to do is toss a fair coin for each of the n bits, making the bit 1 if the coin lands heads and 0 if the coin lands tails. On a computer, that would mean making each bit either a 0 or a 1 with equal probability using a random number generator.

In what follows we need to define two parts of a word $x_1, \cdots, x_n$: a *prefix* $x_1, \cdots, x_i$ of length i consists of the i leftmost symbols of the word and a *suffix* $x_{n-i+1}, \cdots, x_n$ of length i consists of the i rightmost symbols of the word. Here i can be any number from 0 to n inclusive. If $i = 0$, the prefix or the suffix is empty and if $i = n$, it's the whole word.

A *permutation* of length n is an arrangement of the numbers 1, 2, … , n. There are $n!$ permutations of length n. There are n ways to choose which of the n numbers to put first, and for each of these ways there are $n - 1$ ways to choose which of the remaining $n - 1$ numbers to put second, for a total of $n(n\text{-}1)$ ways to fill the first two positions, and for each of these ways there are $n - 2$ ways to choose which of the remaining $n - 2$ numbers to put third, for a total of $n(n\text{-}1)(n\text{-}2)$ ways to fill the first three positions, and so on until the last position is filled with the last remaining number; so the total number of ways to arrange the n numbers is $n!$

Here is a list of the 24 permutations of length 4 in lexicographical order:

1234
1243
1324
1342
1423
1432
2134
2143
2314
2341
2413
2431
3124
3142
3214
3241
3412
3421
4123
4132
4213
4231
4312
4321

In the first permutation the numbers are increasing order and in the last string they are in decreasing order, but these strings of numbers are too short to illustrate the method of changing each permutation to its lexicographical successor; so we choose a longer example: 62358741.

The only way to change a permutation is to exchange two of its numbers. If you want to make the pemutation lexicographically bigger, you have to exchange a number with a bigger number on its right. Now the last four numbers of 62358741 are in decreasing order from left to right; so none of them have a bigger number on its right and you have to choose some number to the left of these four numbers. The further left you go, the bigger the lexicographical increase you'll make; so you have to choose the rightmost number that has a bigger number on its right, that is, the 5. And you have to exchange it with the smallest number on its right that is bigger than 5, that is, the 7. Once you do this, the permutation becomes 62378541. Note that the last four numbers are still in decreasing order from left to right, but not the last five numbers. Of all the permutations that start with 6237, this one is the biggest and we have to make it the smallest; so we reverse the suffix 8541 and we get 62371458. This is the analogue of changing each 1 on the right of a binary string to 0 and the rightmost 0 to 1: we increase by the minimal amount the rightmost number that can be increased and then we make everything to its right as small as possible.

Here is the algorithm for finding the successor of a given permutation $a[1]a[2]\ldots a[n]$:

```
Boolean function nextperm(n: natural, a: array[1..n] of natural)
                        {precondition: The array a consists of distinct members of the set {1,2,…,n}}.
   local variables i, j: natural;
   i ← n – 1;                    {First we look for the rightmost element a[i] such that a[i] < a[i+1].}
   while (i > 0) and a[i] > a[i+1] do
      i ← i – 1;
   end while;
   if i = 0 then return true end if;                    {true means that the permutation was n…21.}
                    {We haven't returned; so i > 0 and a[i+1] > a[i+2] > … > a[n] but a[i] < a[i+1].}
   j ← n;                {Now we look for the smallest element among a[i+1],…,a[n] that is > a[i].}
   while a[j] < a[i] do
      j ← j – 1;
   end while;                                                          {a[j] > a[i] > a[j+1]}
   exchange (a[i], a[j]);          {a[i+1] > a[i+2] > … > a[n] is still true. We reverse this suffix.}
   for j from 1 to floor((n-i)/2)
      exchange(a[i+j], a[n+1-j]);
   end for;
   return false;
end nextperm.
```

What is the average number of exchanges that this algorithm will do for big n? The first step is to find the total number of exchanges, or at least an approximation to this number that is accurate enough that the error tends to 0 as n tends to infinity. The horse power method is to count the number of permutations that require i exchanges, multiply by i and sum over i. This can be done, but it takes several pages and is error-prone; so I'll spare you the details and show you the brain power method: you count the number of permutations for which each element gets exchanged with an element on its left. For the moment we ignore the fact that the last permutation is not changed.

The first exchange instruction – exchange(a[i], a[j]) – is done for every permutation; so it happens $n!$ times. When the longest suffix such that $a[i+1] > a[i+2] > \ldots > a[n]$ gets reversed,

$a[n]$ gets exchanged with a number on its left if this suffix is of length at least 2. This happens if $a[n] < a[n\text{-}1]$. For every permutation that satisfies this condition, there are 2! permutations altogether - you get them by making the 2! permutations of $a[n]$ and $a[n\text{-}1]$; so that the number of permutations that satisfy this condition is $n!/2!$. Also, $a[n\text{-}1]$ gets exchanged with a number on its left if this suffix is of length at least 4. This happens if $a[n] < a[n\text{-}1] < a[n\text{-}2] < a[n\text{-}3]$, and since there are 4! ways to arrange that suffix, the number of permutations that satisfy that condition is $n!/4!$. Continuing this process, we find that the total number of exchanges is

$$n!(1 + 1/2! + 1/4! + 1/6! + \ldots).$$

Of course, this series doesn't really go on forever, because the length of the suffix can't exceed $n - 1$. The expression for the last term is complicated – it depends upon whether n is odd or even – but we can ignore the fact that the series stops because we want to divide by n and then take the limit as n approaches infinity. And we can also continue to ignore the fact that the last permutation doesn't get changed because the number of exchanges wouldn't exceed n, and when you divide n by $n!$ and let n tend to infinity, the quotient tends to 0. In the limit, then, the average number of exchanges is $1 + 1/2! + 1/4! + 1/6! + \ldots$ to infinity.

Now you may recall a series that looks something like this sum. It's in Section 3.2:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

Substituting $x = 1$ into this series, we get

$$e = 1 + 1 + 1/2! + 1/3! + 1/4! + 1/5! + 1/6! + \ldots$$

Substituting $x = -1$ into the same series, we get

$$1/e = 1 - 1 + 1/2! - 1/3! + 1/4! - 1/5! + 1/6! - \ldots$$

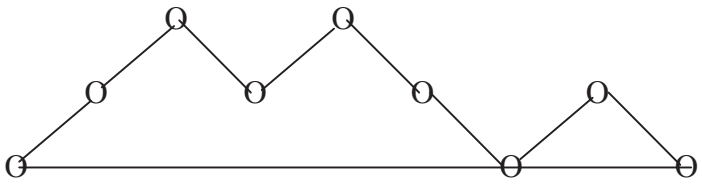
Adding these two equations, we get $e + 1/e = 2 + 2/2! + 2/4! + 2/6! + \ldots$; so as n tends to infinity, the average number of exchanges tends to $(e + 1/e)/2 = 1.543080635\ldots$

Nextperm too is a CAT algorithm.

To generate a random permutation of length n, you would do a sort of randomized selection sort. For each i from 1 to $n - 1$, instead of exchanging $a[i]$ with the smallest element among $a[i]\ldots a[n]$, you exchange it with $a[j]$, where j is an integer chosen at random from the integers $i,\ldots,n$.

A *balanced parenthesis system* of length $2n$, also called a *Dyck word* after the German mathematician Walther Franz Anton von Dyck, consists of n left parentheses and n right parentheses such that among the first i parentheses there are at least as many left parentheses as right parentheses. If you change each left parenthesis to a 1 and each right parenthesis to a 0, the Dyck word becomes a binary string $b_1 b_2 b_3 \cdots b_{2n}$ such that any prefix $b_1 b_2 b_3 \cdots b_i$ has at least as many 1s as 0s. For example, the binary string 11010010 represents the Dyck word (( ) ( )) ( ). A *Dyck path* is a series of steps in the plane, starting at the origin, each one going east one unit

and either north (for a 1) or south (for a 0) one unit, that ends on the x axis and never goes below it. The Dyck path corresponding to the binary string 11010010 is shown below.

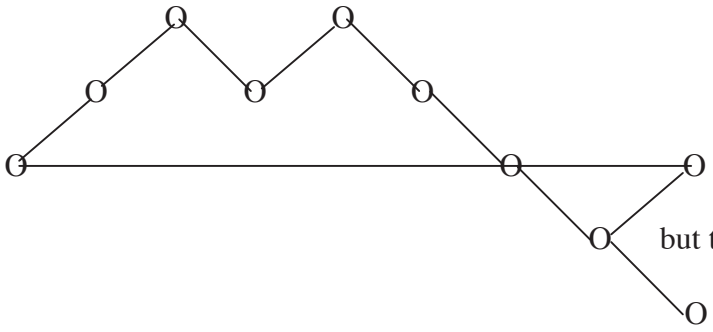


How many Dyck words are there of length $2n$? This problem was solved by several people in 1887; the most direct solution appears in [D. André, Solution directe du problème résolu par M. Bertrand, Comptes Rendus de l'Académie des Sciences, Paris 105 (1887) 436–437]. It was stated in terms of ballot counting: if two candidates got the same number of votes, in how many ways can the scrutineer count the ballots so that his favourite candidate never trails his opponent? If you have read this far, you will know by now that I wouldn't be satisfied with such a serious model. Instead, suppose that a drunkard gets booted out of a bar by the bouncer, who threatens to twist his neck if he takes one step inside the bar before he sobers up. Starting at the entrance to the bar, the drunkard takes $2n$ random steps, either away from the bar or towards it, without ever entering the bar, until he collapses in a heap at the entrance to the bar. In how many ways can he do this?

If the bouncer had not made this threat, the drunkard could have taken any sequence of n steps north (away from the bar) and n steps south. Among the $2n$ steps, n of them are chosen to be north. In how many ways can you choose n objects out of $2n$? More generally, in how many ways can you choose r objects out of n? Recall that there are $n!$ ways to permute n objects. Suppose you want to choose r of them and permute them – that is, among the n objects, you want to choose one to be first, one to be second and so on up to r of them. There are n ways to choose the first object, $n-1$ ways to choose the second object and so on – the object to be put in the rth place can be chosen in $n - r + 1$ ways, so that the total number of ways to choose all the r objects and permute them is $n(n\text{-}1)(n\text{-}2)\ldots(n\text{-}r\text{+}1) = n!/(n\text{-}r)!$. But you only want to choose them, not to permute them too. For each choice of those r objects, there are $r!$ ways to permute them; so the number of ways to choose them without permuting them is $\frac{n!}{(n-r)!r!}$. This is sort of like counting the cows in a field by counting the legs and dividing by 4; it sounds more complicated than counting the cows directly, but in this case it's actually easier. Anyway, since the drunkard is choosing n of his $2n$ steps to be north, the number of ways he can to it is $\frac{(2n)!}{n!\times n!}$.

Now some of these choices will lead him into the bar. As soon as he takes one step inside the bar, the bartender will twist his neck by 180 degrees, which will reverse all of the drunkard's subsequent steps. So instead of going from one step inside the bar to the entrance, he will go from one step inside the bar to two steps inside the bar before he collapses, meaning that he will have taken n - 1 steps north and n + 1 steps south (see the diagram below). The transformation done by the bouncer is uniquely reversible; if the drunkard had taken any combination of n - 1 steps north and n + 1 steps south without getting his neck twisted, he would have collapsed two steps inside the bar, but instead, when he gets one step inside the bar, the bouncer twists his neck and he collapses at the entrance. This establishes a bijection between the set of walks that go inside the bar and end at the entrance and the set of all the walks that end two steps inside the bar;

so the number of walks that go inside the bar and end up at the entrance is equal to the number of ways of choosing $n - 1$ steps out of $2n$ to go north, which is $\frac{(2n)!}{(n-1)!(n+1)!}$.



Among the $\frac{(2n)!}{n! \times n!}$ walks that end at the entrance, $\frac{(2n)!}{(n-1)!(n+1)!}$ enter the bar; so the number of walks that don't enter the bar is equal to $\frac{(2n)!}{n! \times n!} - \frac{(2n)!}{(n-1)!(n+1)!} = \frac{(2n)!}{n!(n+1)!}$. The numbers $C(n) = \frac{(2n)!}{n!(n+1)!}$ are called the *Catalan numbers*, named after the Belgian mathematician Eugène Charles Catalan, and they count a large number of combinatorial objects, including the triangulations of a polygon with $n + 2$ sides, the ways of parenthesizing $n + 1$ factors and various sorts of trees including binary trees. A table of Catalan numbers can be constructed by using the recursive definition $C(0) = 1$, $C(n) = (4n\text{-}2)C(n\text{-}1)/(n+1)$ if $n \geq 1$ (you prove it). The first few Catalan numbers are 1, 1, 2, 5, 14, 42, 132 (you calculate these numbers and the next few). Below are the 14 Dyck words of length 8 in lexicographical order:

10101010
10101100
10110010
10110100
10111000
11001010
11001100
11010010
11010100
11011000
11100010
11100100
11101000
11110000

The first Dyck word of any (even) length is 101010…10 and the last one is 11…1100…00. How do you transform a given Dyck word into its successor? You have to change to a 1 the rightmost 0 that can be changed. If a 0 is followed only by 0s, it can't be changed to a 1 because there would be no way to restore the equality between the number of 0s and the number of 1s by changing any of the numbers to the right of that 0. If the drunkard was going to go straight to the entrance to the bar before he collapsed and then changed his mind and

took one step away from the bar, he'd collapse before he made it to the entrance. So you have to change to a 1 the rightmost 0 that has a 1 to its right. You find it by starting at the rightmost bit and passing over all the 0s until you get to a 1 and then over all the 1s until you get to a 0. If you never get to a 0 after encountering 1s, the Dyck word is 11…1100…00, the last one in lexicographical order, and you quit generating them. Otherwise you change to a 1 the first 0 you encounter after encountering some 1s. Then you want to make the suffix following that new 1 as small as possible lexicographically; so you pad with 0s until there are as many 0s as 1s in the prefix (after you make the drunkard change a southward step to a northward one, you send him straight to the entrance to the bar) and then you pad with 10…10 until the word has $2n$ numbers (you make him alternate northward steps with southward ones until he collapses). How do you know how many 0s to add before you start adding 10? You count the 0s as you pass over them, then you subtract 1 from that number every time you pass a 1. In this way, you're tracing backwards the drunkard's last few steps and keeping track of the number of steps he is away from the bar. When (or if) you next encounter a 0, the number you have is the number of steps the drunkard is away from the entrance to the bar **after** he made the southward step represented by that 0. When you change the 0 to a 1, you've sent him 2 steps farther from the bar; so you add 2 to the number, and that's the number of 0s you have to add before you start adding 01.

Here is the algorithm:

```
Boolean function NextDyck (n; natural; b: array[1..2*n] of {0,1})
   local variables d, k: natural;
   d ← 0;
   k ← 2*n;
   do                                          {Search for the rightmost occurrence of 1.}
      d ← d + 1;
      k ← k – 1;
   while b[k] = 0;
   do                                          {Search for the rightmost occurrence of 01.}
      d ← d – 1;
      k ← k – 1;
   while (k > 0) and (b[k] = 0);  { d is now the number of 1s minus the number of 0s in b[1..k].}
   if k = 0 then return true end if;                                   {b is 11…1100…00}
   b[k] ← 1;                                   {Change the 0 in the rightmost 01 to 1.}
   d ← d + 2;
   do                          {Append 0s to the right of this new 1 until the prefix is balanced.}
      k ← k + 1;
      b[k] ← 0;
      d = d ← 1;
   while d > 0;
   while k < 2*n                               {Pad with 10 until the word is of length 2n.}
      k ← k + 2;
      b[k-1] ← 1;
      b[k] ← 0;
   end while;
   return false;
end NextDyck.
```

How many bits get written, on the average, in transforming each Dyck word to its successor? This number tends to 16/3 as the length of the word approaches infinity. The proof is too long and complicated to be included here, but I'll tell you where to find it and how I discovered it.

I found four articles, each of which use a different method for generating binary trees, and I wanted to know the extent to which the theoretical estimate of the average time complexity coincides with the experimental estimate found by programming and executing each of those methods. Being too lazy to do the programming myself, I hired a second-year undergraduate student, Pierre Auger (who is now Dr. Auger) to do the programming for me. If that's all he did, I would merely have acknowledged his contribution at the end of the article I submitted on the subject. But instead he found errors in the description of some of the algorithms and corrected them; so I included him as a joint author. One of those articles [T.R. Walsh, Generating nonisomorphic maps without storing them, SIAM Journal of Algebraic and Discrete Methods, Vol. 4 (1983), 161-178] contains the above algorithm without a time-complexity analysis; so I proved, using the horse power method, that the total number of bits that get written in transforming all the Dyck words of length n is $C(1) + C(2) + \ldots + C(n+1)$ and that when you divide that number by $C(n)$ the quotient tends to 16/3 as n tends to infinity. The joint paper [P. Auger and T.R. Walsh, Theoretical and Experimental Comparison of Four Binary-tree Generation Algorithms, Congressus Numerantium 93 (1993), 99-109] omits most of the tedious details of the derivation of the formula $C(1) + C(2) + \ldots + C(n+1)$, which occupy several pages. I later used the brain power method to derive this formula and published it in [P. Auger and T.R. Walsh, Addendum to Theoretical and Experimental Comparison of Four Binary-tree Generation Algorithms, Congressus Numerantium 112 (1995), 3-5]; so the entire proof is contained in the union of these two articles.

It turns out that the above algorithm is neither the first nor the most efficient one for generating Dyck words in lexicographical order. The one in [I. Semba, Generation of all the balanced parenthesis strings in lexicographical order, Information Processing Letters 21 (1981), 188-192] examines an average of 3 bits instead of 16/3. I include my algorithm here because it's mine and because it gives me the opportunity to pursue the model of the drunkard's walk.

The methods known for generating a random Dyck word are also too complicated to be included here. If you're interested, you can find one of these methods and a reference to another one in [Dominique Gouyou-Beauchamps, Combinatorics and Random Generation, Algorithms Seminar 2001-2002, F. Chyzak (ed.), INRIA, (2003), 177-182].

10.2 Gray codes

In another computer game I played, you travel from one time period to another in a time machine. In each time period there are things to do and you have to do them quickly because the time machine is programmed to return to its original time period after a certain period of time. In one of these time periods there is a locked sarcophagus you have to open. The sarcophagus has 8 levers, each in the down position, and you can toggle them between the down position and the up position, but only one lever at a time. One combination of positions of all the levers will unlock the sarcophagus, but the game gives you no hint as to what combination that is; so you have to keep trying them until you get the right one. The positions of the levers can be expressed as a binary string of length 8, with down represented by 0 and up by 1. I started generating all these

combinations of positions in lexicographical order, but I was too slow: before I could get the right combination, the time machine vanished, stranding me in that time period, and I lost the game. Apparently, doing an average of two lever pulls to get from one combination of positions to the next just won't cut it.

Fortunately there is a way of ordering the length-n binary strings so that each string differs from its predecessor in a single bit. It was invented by Frank Gray to prevent spurious output from electromechanical switches and published in [Gray, F.: Pulse Code Communication. U.S. Patent 2 632 058 (March 17, 1953)]. Here are the lists of length-n in Gray's order for the first few values of n.

$n = 0$	$n = 1$	$n = 2$	$n = 3$	$n = 4$
	0	00	000	0000
	1	01	001	0001
		11	011	0011
		10	010	0010
			110	0110
			111	0111
			101	0101
			100	0100
				1100
				1101
				1111
				1110
				1010
				1011
				1001
				1000

By examining these lists you can discern the pattern: to construct the list of words of length n, you write down the list of words of length $n - 1$, putting a 0 to the left of each word, and then you write down the list of words of length $n - 1$ backwards, putting a 1 to the left of each word. In each list, only one bit changes in passing from one binary string to the next one. But a recursive description is difficult to execute by hand when you can't see all the words in front of you. A non-recursive description was in order.

Look again at the list of length-4 binary strings in lexicographical order. All the words with the same prefix are grouped together, and the bit immediately to the right of this prefix is first 0 and then 1, so that it follows the sequence (0, 1). The leftmost bit that changes is the 0 immediately to the left of a suffix of 1s **because 1 is the last member of the sequence of values attained by each bit.** Now in Gray's list, putting a 1 in front of a list reverses that list; so putting an odd number of 1s in front of a list will reverse the sequence followed by each bit: if the prefix has an odd number of 1s, then the next bit will follow the sequence (1, 0) whereas if the prefix has an even number of 1s, then the next bit will follow the sequence (0, 1). Check out the lists above to verify this assertion.

Now suppose the whole word has an even number of 1s. If the rightmost bit is 0, it has an even number of 1s to its left; so its sequence is (0, 1) and it is not at its last value. If the

rightmost bit is 1, it has an odd number of 1s to its left; so its sequence is (1, 0) and it is not at its last value. Either way, it's the rightmost bit that changes.

Now suppose the whole word has an odd number of 1s. If the last bit is a 0, it has an odd number of 1s to its left; so its sequence is (1, 0). It is at its last value; so it can't change. The same holds true for any 0 in a suffix of 0s. If there is an odd number of 1s, there must be at least one of them. The rightmost 1 has an even number of 1s to its left; so it's sequence is (0, 1). This bit too is at its last value; so it too can't change. But the bit immediatetly to left of the rightmost 1 is the rightmost bit of a prefix with an even number of 1s; so its situation is the same as the rightmost bit of a whole binary string with an even number of 1s: whether it is a 0 or a 1 it is not at its last value; so it changes. And if there aren't any bits to the left of the rightmost 1; then the string is 10…0, the last string on the list.

Each time a bit is changed, the parity of the number of bits that are 1 changes too. Starting with the string 00…00 and a Boolean variable Odd set to false, you execute the following procedure:

if (Odd) **then**
  search the string from right to left until you get to a 1;
  if this 1 is the leftmost bit **then** quit generating;
  else change the bit immediately to the left of this 1; **end if**;
else
  change the rightmost bit;
end if;
change Odd to not(Odd).

Following this procedure, I managed to find the right combination of positions of the levers to open the sarcophagus in time to get to the time machine before it vanished, and I eventually won the game.

In honour of Frank Gray, any list of words that satisfies some closeness criterion for adjacent words in a list is called a Gray code. Gray codes have been invented for many combinatorial objects, including combinations, permutations, compositions of an integer, partitions of a set, partitions of an integer and Dyck words. A survey of Gray codes can be found in [C. D. Savage, "A survey of combinatorial Gray codes", SIAM Review , 39, No. 4, 1997 605-629].

Of course, Gray codes are useful for more than just playing computer games. If two successive words representing a combinatorial object differ only slightly, then some of the properties of that object can be updated quickly. For example, binary strings of length n represent subsets of a set of n elements, and when one bit changes, the cardinality of the subset changes by 1; so as you generate all the subsets, you can update the cardinality quickly.

Another example of a property that can be updated quickly with the help of a Gray code is the number of inversions of a permutation. An *inversion* of a permutation is a pair of elements such that the bigger one is to the left of the smaller one. For example, the permutation 53241 has 8 inversions: 53, 52, 54, 51, 32, 31, 21 and 41. When you exchange two adjacent elements $p(i)$ and $p(i+1)$ of a permutation, the number of inversions increases by 1 if $p(i) < p(i+1)$ or decreases

by 1 if $p(i) > p(i+1)$; so if the set of permutations of $\{1, 2, \ldots, n\}$ is generated in an order such that each permutation differs from its predecessor by a transposition of adjacent elements, the number of inversions can be updated quickly.

Among the Gray codes for permutations there is one that transposes adjacent elements. It was discovered independently by Johson [S.M. Johnson, Generation of permutations by adjacent transpositions, Mathematics of Computation 17 (1963), p. 282-285] and Trotter [H.F. Trotter, Algorithm 115: Perm, Communications of the ACM 5 (1962), 434-435]; so it's called the Johnson-Trotter Gray code. Here is a list of the 24 permutations of length 4 as they are generated by this Gray code, with the number of inversions to the right of each permutation.

Perm.	#inversions
1234	0
1243	1
1423	2
4123	3
4132	4
1432	3
1342	2
1324	1
3124	2
3142	3
3412	4
4312	5
4321	6
3421	5
3241	4
3214	3
2314	2
2341	3
2431	4
4231	5
4213	4
2413	3
2143	2
2134	1

Examining this list of words, you can see that the largest number (4) moves first from right to left until it gets to its final position – at the left end of the word. Then the second largest number (3) moves one spot from right to left, and then the largest number moves from left to right until it gets to its final position - at the right end of the word - and then the second largest number moves one spot from right to left again, and so on until the second largest number gets to its final position, which is not at the left end of the word but just to the right of the largest number – a number isn't allowed to swap with a bigger number. Now both the largest and the second largest number are in their final positions; so the third largest number (2) moves one spot from right to left and the two largest numbers change direction. This continues until it is the turn of the smallest number 1 to move, which would turn the permutation into the first one again; so the generation stops. I leave it to you to write a pseudocode for generating this Gray code.

There are several Gray codes for Dyck words. The descriptions of these Gray codes are too complicated to be given here; I'll just describe the properties of three of them and give references. In all of these Gray codes, one Dyck word is transformed into its successor by changing a single 1 to a 0 and a single 0 to a 1. In the first of these to be published [Ruskey, F., Proskurowski, A.: Generating Binary Trees by Transpositions. J. Algorithms 11 (1990) 68-84], there could be an arbitrary number of bits including some 1s between the 1 and the 0 that change. In the next one [Bultena, B., Ruskey, F.: An Eades-McKay Algorithm for Well-Formed Parentheses Strings. Inform. Process. Lett. 68 (1998), no. 5, 255-259] there could be an arbitrary number of bits between the 1 and the 0 that change, but each of these bits must be 0. And in the third one [V. Vajnovszki and T.R. Walsh, *A loopless two-close Gray-code algorithm for listing k-ary Dyck words*, Journal of Discrete Algorithms, Vol. 4, No. 4 (2006) 633-648] there can be at most one bit between the 1 and the 0 that change and that bit must be 0.

Notice the word "loopless" in the title of that last reference. That means that each word is transformed into its successor in $O(1)$ time **even in the worst case**. This term and the method for designing such an algorithm was invented by Gideon Ehrlich [G. Ehrlich, Loopless algorithms for generating permutations, combinations, and other combinatorial configurations, J. ACM 20 (1973), p. 500-513]. Other researchers then jumped on the bandwagon to design loopless algorithms for other sets of combinatorial algorithms, including some colleagues of Ehrlich [James R. Bitner, Gideon Ehrlich, Edward M. Reingold: Efficient Generation of the Binary Reflected Gray Code and Its Applications. Commun. ACM 19(9): 517-521 (1976)], Stanley Gill Williamson [S.G. Williamson, Combinatorics for computer science, Computer Science Press, Rockville, 1985] and, of course, me. I did it for the Ruskey-Proskurowski Gray code [T.R. Walsh, *Generation of well-formed parenthesis strings in constant worst-case time*, The Journal of Algorithms 29, 1998, 165-173] and the one discovered by Vincent Vajnovski (see the reference above with his name and mine). Professor Vajnovski invited me to come to Besançon to work with him, which I was only too glad to do. After our work was done, I still had some time before having to use my return ticket from Paris to Montreal; so I was able to spend a few days and nights in Paris, sightseeing during the day and taking in Paris' night life each evening – by attending a concert of classical music. Since my trip was subsidized by both my research grant and Prof. Vajnovski's university, this was another use I managed to make of Gray codes.

I also discovered a couple of Gray codes of my own. In one case, Ehrich's original method worked [T.R. Walsh, *Gray codes for involutions*, The Journal of Combinatorial Mathematics and Combinatorial Computing 36, 2001, 95-118]; in the other I had to modify it to make it work [T.R. Walsh, *Loop-free sequencing of bounded integer compositions*, The Journal of Combinatorial Mathematics and Combinatorial Computing 33, 2000, 323-345]. This latter article illustrates yet another use I made of Gray codes. While working on my M. Sc. I got interested in finding $f(n,r)$, the number of length-n permutations with r inversions. There was already a recursive definition of $f(n,r)$, but I wanted to find a non-recursive formula. The one I finally found was so complicated that it was less efficient than the recurrence; so the article I submitted was condemned, derided and dismissed by both referees. But I was determined to claim authorship of this result, and 35 years later I finally got the chance. I applied a special case of my Gray code for bounded integer compositions to design a Gray code for permutations with a given number of inversions. Having thus introduced the subject, I slipped my monstrous formula into my article, and fortunately the referees did not insist that I remove it. And so it was that I used Gray codes to claim authorship of an unpublishable discovery.

CHAPTER 11. NP-COMPLETE PROBLEMS

Unlike the previous chapters, this last chapter is not intended to be self-contained. That would make it too long for its purpose, which is to amuse you with an article I published in The Mathematical Intelligencer. Any notion not contained in this chapter can be found in the book [Alfred V. Aho, J.E. Hopcroft, Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley Series in Computer Science and Information Processing (1974)], referred to henceforth as AHU. Here I intend to give you enough of the flavour of the subject that you will be able to appreciate the article.

The sort of problems we will be dealing with here are called *decision problems* – problems to which the solution is either the answer "yes" or the answer "no". An example of such a problem is the *knapsack problem*. Given an array of k integers $(m_1, m_2, ..., m_k)$ and another integer S, is there a subarray of these integers whose sum is S? Suppose that the array is (1,3,6,10). If $S = 10$, then the answer to that instance of the problem is "yes"; if $S = 8$, then the answer to this instance of the problem is "no".

The P in NP-COMPLETE (the P in NP, not the P in COMPLETE) stands for polynomial. Here we will not be concerned with whether an algorithm runs in $O(n^2)$ or $O(n^3)$ time. All we want to know is whether the time in which it runs is bounded by some polynomial $P(n)$. A problem is said to be *in the class P* if there is a polynomial P and an algorithm that, for any instance of the problem, correctly decides whether the answer to this instance of the problem is yes or no in a time bounded by $P(n)$, where n is the size of an efficient representation of this instance of the problem.

Why was it necessary to add the condition that the representation has to be efficient? There is an obvious algorithm for solving the knapsack problem that runs in $O((n+k)^2)$, where n is the sum of the absolute values of all the k integers. Take an array $A[-n..n]$ of Boolean and initialize it to 0 everywhere except that $A[0] = 1$. Then execute the following algorithm:

```
for i ← 1 to k do
   for j ← max(-n,-n- m_i) to min(n,n- m_i) do
      if A[i] = 1 then A[i+ m_i] ← 1 end if;
   end for j;
end for i.
```

This algorithm will put a 1 in each $A[i]$ such that there is some sub-array of integers whose sum is i. We trace this algorithm for the array (1,3,6,10). Since all the integers in this array are positive, we can make the array A start at 0 instead of –20.

```
 j= 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
A[j]1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
i=1:1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
i=2:1  1  0  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
i=3:1  1  0  1  1  0  1  1  0  1  1  0  0  0  0  0  0  0  0  0  0
i=4:1  1  0  1  1  0  1  1  0  1  1  1  0  1  1  0  1  1  0  1  1
```

The number of operations is bounded by k times the array length, which is $2n+1$, and $k(2n+1) \le (n+k)^2$ if $k \ge 1$. Then for any S, if $-n \le S \le n$ and $A[S] = 1$, then the answer to that instance of the problem is "yes"; otherwise the answer is "no".

Now you can always represent an instance of the problem so inefficiently that $(n+k)^2$ will be bounded by a polynomial in the size of the representation. You represent the integer m_i by a string of $|m_i|$ 1s, preceded by a minus sign if $m_i < 0$, and followed by a comma. The total length will then be at least $n + k$, so that the time taken to solve the problem is bounded by the square of the input size.

But if an integer $m_i \neq 0$ is represented in binary or decimal, then the number of digits needed to represent m_i is not m_i but rather proportional to $\log(|m_i|)$, where the constant of proportionality depends upon whether you're using binary or decimal or some other base. There is no known algorithm for solving the knapsack problem in a time bounded by any polynomial in the size of this more efficient representation of the problem; so blowing up the input size to make the problem solvable in a time bounded by a polynomial in that over-inflated input size is cheating, and it is to prevent this sort of cheating that the condition "efficient" is introduced.

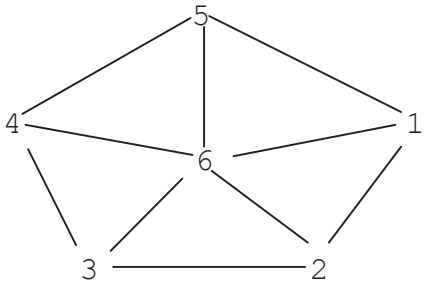
The N in NP-COMPLETE stands for non-deterministic. A non-deterministic algorithm is allowed to make guesses. In the knapsack problem, suppose that the array M of integers is (1,3,6,10) and $S = 10$. A non-deterministic algorithm could run through the array M and, for each integer m_i in M, guess whether to include or exclude m_i, add up all the integers it includes and say "yes" to the input if the sum of all the included integers is equal to S. If it decides to include 1, 3 and 6 and exclude 10, or to exclude 1, 3 and 6 and include 10, then the sum of the included integers will be equal to S and the algorithm will say "yes". Such an algorithm runs in polynomial time in the size of the input because all the algorithm has to do is to run through the array M, include or exclude each of the elements m_i of M and add up the integers it includes. There other sequence of guesses for which the sum of the included integers will not be equal to S and the algorithm will not say "yes", but the algorithm is said to *accept the input* **if there exists** a sequence of lucky guesses that allows the algorithm to say "yes" to the input. If S is 8 instead of 10, then the algorithm cannot say "yes" no matter what sequence of guesses it makes. A problem is said to be *in the class NP* if there is a non-deterministic algorithm that accepts any "yes" instance of the problem in a time bounded by a polynomial in an efficient input size and doesn't accept any "no" instance. Such an algorithm is not required to say "no" to a "no" instance in polynomial time. To say "no" to a "no" instance of the knapsack problem, the algorithm would have to try all the 2^k possible sequences of guesses and this cannot be done in polynomial time. But because there is a non-determininistic algorithm that accepts a "yes" instance of the knapsack problem in polynomial time, albeit through a sequence of lucky guesses, and accepts no "no" instances, the knapsack problem is in the class NP.

Another example of a problem in the class NP is the *partition problem*. Given an array of integers, can this array be partitioned into two sub-arrays such that the sum of the integers in one sub-array is equal to the sum of the integers in the other sub-array? A non-deterministic algorithm would run through the array and, for each element, guess whether to put it into the first sub-array or the second one, add the elements in each sub-array and say yes to the input if the sum of the elements in the first sub-array is equal to the sum of the elements in the second one.

For example, suppose the array was (1,2,3,4). A good sequence of guesses would be to put 1 and 4 in one sub-array and 2 and 3 in the other one.

Another example is the *exact cover problem*. Given a set S and a family F of subsets of S, is there a sub-family of F such that the members of the sub-family are pairwise disjoint and their union is equal to S? A non-deterministic algorithm would run through F and, for each member of F, guess whether to include it in the sub-family or exclude it. Then the algorithm would check whether the members of the sub-family are pairwise disjoint and whether their union is equal to S and, if so, say yes to the input. For example, suppose that S={1,2,3,4} and F consists of the subsets {1}, {1,2}, {2,3}, {2,4} and {4}. A good sequence of guesses would be to include {1}, {2,3} and {4} in the sub-family and exclude {1,2} and {2,4}.

Another example of wide interest is the *graph colouring problem*. Given a simple undirected graph G and a positive integer k, is it possible to colour the vertices of G using at most k colours so that no two adjacent vertices get the same colour? A non-deterministic algorithm would guess one of the k colours for each vertex and then check whether any two adjacent vertices got the same colour and say yes to the input if not. For example, if the graph were the one drawn below and $k = 4$, then a good sequence of guesses would be to colour vertices 1 and 3 red, vertices 2 and 4 blue, vertex 5 yellow and vertex 6 green.



Another example is the *satisfiability problem*. A *Boolean expression* consists of variable names, conjunction symbols, disjunction symbols, negation symbols, and enough parentheses to make it clear the order in which the operations are to be applied. An example of a Boolean expression is $(p \wedge \neg q \wedge r) \vee (p \wedge q \wedge r)$. A Boolean expression is said to be *satisfiable* if there is an attribution of truth values (true or false) to each of the variables that gives the expression the value true. Given a Boolean expression, is it satisfiable? A non-deterministic algorithm would guess a truth value for each of the (distinct) variables, evaluate the Boolean expression and say "yes" to the input if the Boolean expression turns out to be true. For the above Boolean expression, a good sequence of guesses would be to make p true, q false and r true.

A special case of the satisfiablility problem is the *CNF-satisfiability problem*. A Boolean expression is said to be in *conjunctive normal form* if it consists of the conjunction of Boolean expressions called *clauses*, each of which is the disjunction of variables, each of which may or may not be preceded by a negation sign. The above Boolean expression is not in conjunctive normal form, but $(p \vee \neg q \vee r \vee s) \wedge (p \vee q \vee r \vee \neg s)$ is. Given a Boolean expression in conjunctive normal form, is it satisfiable? A non-deterministic algorithm would proceed as it would for an arbitrary Boolean expression.

A special case of the CNF-satisfiability is *3-SAT*. Given a Boolean expression in conjunctive normal form in which each of the clauses has exactly three variables, is it satisfiable? The expression $(p \vee \neg q \vee r \vee s) \wedge (p \vee q \vee r \vee \neg s)$ is not an instance of 3-SAT, but the expression $(p \vee \neg\ q \vee r) \wedge (p \vee q \vee r) \wedge (q \vee r \vee s) \wedge (\neg\ q \vee r \vee \neg s)$ is. A non-deterministic algorithm would proceed as it would for an arbitrary Boolean expression.

It is not surprising that all these problems are in the class NP. What is perhaps more surprising is that none of these problems has been proved to be, or not to be, in the class P. Even more surprising is that either all of them are in the class P or none of them are!

Suppose you're interested in a problem D and you want to find a polynomial-time algorithm to decide whether or not a given instance D, coded efficiently by a string x of n symbols, is a "yes" instance or a "no" instance. If you know of a problem E that is in class P, you may be able use problem E to find the desired algorithm. The trick is to transform each input x of D into an input $f(x)$ of E such that $f(x)$ represents a "yes" instance of E if and only if x represents a "yes" instance of D. Suppose you can do the transformation in a time bounded by a polynomial $p(n)$. Then $f(x)$ can't be any longer than $p(n)$ symbols because it takes a transformation step just to write a symbol of $f(x)$. Since E is in class P, there is an algorithm A that decides whether $f(x)$ represents a "yes" instance or a "no" instance of E in a time bounded by some polynomial $q(p(n))$. You transform x into $f(x)$ in time $p(n)$, and then you apply the algorithm A that decides in a time bounded by $q(p(n))$ whether $f(x)$ represents a "yes" instance or a "no" instance of E. If A says "yes" to $f(x)$, then you say "yes" to x; if A says "no" to $f(x)$, then you say "no" to x. You now have an algorithm that decides whether x represents a "yes" instance or a "no" instance of D in a time bounded by $p(n) + q(p(n))$, which is a polynomial in n; so D is in class P.

A *polynomial transformation* of a problem D into a problem E transforms each "yes" instance of D into a "yes" instance of E and each "no" instance of D into a "no" instance of E in a time bounded by some polynomial of the length of the input that represents the instance of D.

As an example, here is a polynomial transformation of the knapsack problem into the partition problem. Recall that the knapsack problem asks whether there is a sub-array of the array $M = (m_1, m_2, ..., m_k)$ whose sum is equal to a given integer S. To transform this problem into the partition problem, add another element to the array: $m_{k+1} = 2S - (m_1 + m_2 + ... + m_k)$. This transformation can be done in polynomial time because two integers can be added in a number of operations bounded by the number of digits of the bigger one. The sum of all the integers in the extended array $M^* = (m_1, m_2, ..., m_k, m_{k+1})$ is equal to $2S$. Suppose there is a sub-array of M the sum of whose members is S. This sub-array is also a sub-array of M^*, and the sum of the other members of M^* is also equal to S; so M^* has been partitioned into two sub-arrays of equal sum. Conversely, suppose that there is a partition of $(m_1, m_2, ..., m_k, m_{k+1})$ into two sub-arrays of equal sum, which must be S because the sum of all the elements of M^* is $2S$. Then m_{k+1} has to belong to exactly one of the two sub-arrays. The other sub-array is a sub-array of M and the sum of its elements is equal to S. To give a more concrete example, the instance (1,3,6,10) with $S = 16$ of the knapsack problem gets transformed into the instance (1,3,6,10,12) of the partition problem. The sub-array (6,10), whose elements add to 16, induces the partition of (1,3,6,10,12) into the sub-arrays (6,10) and (1,3,12) of equal sum 16.

As a harder example, here is a polynomial transformation of the exact cover problem into the knapsack problem. Let E be the finite set $\{e_1,e_2,...,e_n\}$ and let F be some family of k subsets of E. The exact cover problem asks whether there is a subfamily of F whose members are pairwise disjoint and whose union is equal to E. The trick here is to express each subset of E as a non-negative integer in base $k+1$. Transform e_1 into 1, e_2 into $(k+1)$, … , e_i into $(k+1)^{i-1}$, … and e_n into $(k+1)^{n-1}$. Then transform each subset S of E into the sum of the images of the elements of S. For example, the empty set gets transformed into 0, the set $\{e_1,e_3,e_4\}$ gets transformed into $1+(k+1)^2+(k+1)^3$ and E itself gets transformed into $1+(k+1)+(k+1)^2+...+(k+1)^{n-1}$. Each member of F, which is a subset of E, gets transformed into a non-negative integer, so that F, when its elements are ordered, gets transformed into an array of k integers $M=(m_1,m_2,...m_k)$. Each sub-family f of F gets transformed into a sub-array of M. For example, if $f=\{\{e_1,e_2\},\{e_1,e_2\},\{e_1,e_3\}\}$, then f is transformed into the array $(1+(k+1),1+(k+1),1+(k+1)^2)$, and the sum of the elements of this array is $3 + 2(k+1) + (k+1)^2$. For each sub-family f of F, let $s(f)$ be the sum of the images of the members of f. The coefficient of $(k+1)^{i-1}$ in $s(f)$ is the number of occurrences of e_i among all the members of f. Now the members of a sub-family f are pairwise disjoint if and only if no element of E occurs more than once among the members of f so that no coefficient of a power of $k+1$ in $s(f)$ is greater than 1. Also, the union of the members of f is equal to E if and only if every element of E occurs at least once among the members of f so that no coefficient of a power of $k+1$ in $s(f)$ is less than 1. It follows that f is an exact cover of E if and only if every coefficient of a power of $k + 1$ in $s(f)$ is exactly 1. Since F has only k members, every sub-family f of F must have at most k members, so that the coefficient of each power of $k + 1$ in $s(f)$ must be at most k. It follows that the coefficients of the powers of $k + 1$ are uniquely determined by the representation of $s(f)$ in base $k + 1$. For example, if $k = 3$, then $3 + 2(k+1) + (k+1)^2 = 27$ and 27 can be expressed in only one way in base 4. It follows that the coefficient of every power of $k + 1$ in $s(f)$ is exactly 1 if and only if $s(f)$ is equal to the integer $1+(k+1)+(k+1)^2+...+(k+1)^{n-1}$. This means that there is a sub-family f of F whose members are pairwise disjoint and whose union is equal to E if and only if there is a sub-array of $(m_1,m_2,...m_k)$, where m_j is the image of the the jth element of F under this transformation, whose elements add up to the integer $1+(k+1)+(k+1)^2+...+(k+1)^{n-1}$. This transformation can be done in polynomial time; so it is a polynomial transformation of the exact cover problem into the knapsack problem.

Since there is a polynomial transformation of the exact cover problem into the knapsack problem and a polynomial transformation of the knapsack problem into the partition problem, there is a polynomial transformation of the exact cover problem into the partition problem. You transform the exact cover problem into the knapsack problem, and the length of the input to the knapsack problem will be bounded by a polynomial in the length of the input to the exact cover problem. Then you transform the knapsack problem into the partition problem. The time taken will be bounded by a polynomial in the length of the input to the knapsack problem, which, in turn, is bounded by a polynomial in the length of the input to the exact cover problem; so the time taken by the pair of transformations is bounded by a polynomial in the length of the input into the exact cover problem. In general, **if there is a polynomial transformation from problem *A* into problem *B* and a polynomial transformation of problem *B* into problem *C*, then there is a polynomial transformation from problem *A* into problem *C***.

In 1971 Stephen Cook published a proof that for any problem *A* in the class NP there is a polynomial transformation from problem *A* into the CNF-satisfiability problem. His proof uses a Turing machine, as does any satisfactory proof I've ever seen. Reproducing this proof here would over-inflate the length of this monograph; if you're interested, you can find it in AHU. A problem *B* is said to be *NP-hard* if for every problem *A* in the class NP there is a polynomial transformation from problem *A* into problem *B*. If, in addition, problem *B* is in the class NP, then it is said to be *NP-complete*. This is what the word COMPLETE means in the title of this chapter. What Cook proved is that CNF-satisfiability is NP-complete.

A polynomial transformation from CNF-satisfiability to 3-SAT appears in AHU. Since there is a polynomial transformation from any problem in the class NP into CNF-satisfiability and there is a polynomial transformation from CNF-satisfiability into 3-SAT, there is a polynomial transformation from any problem in the class NP into 3-SAT; so 3-SAT is NP-hard, and since it is in the class NP, it is NP-complete. In general, the way to prove that a problem *D* is NP-hard is to find a polynomial transformation from a problem *H*, which has already been proved to be NP-hard, into the problem *D*. In AHU there are polynomial transformations from 3-SAT into the graph-colouring problem and from the graph-colouring problem into the exact cover problem. In this monograph there are polynomial transformations from the exact cover problem into the knapsack problem and from the knapsack problem into the partition problem (I take credit for neither of these transformations – the latter one was found by a student taking a graduate course from me). It follows that all these problems are NP-hard and, since they're all in the class NP, they are all NP-complete.

By now there are hundreds of problems that have been shown to be NP-complete. If any of these problems, say problem *E*, were also in the class P, then any problem in the class NP, say problem *H*, would also be in the class P, because there is a polynomial transformation from any problem, including *H*, in the class NP into an NP-complete problem, including *E*, and by combining this transformation with a polynomial-time algorithm that solves problem *E* you'd get a polynomial-time algorithm that solves problem *H*. It follows that there are only two possibilities: **either all the NP-complete problems can be solved in polynomial time or none of them can be.** Now, among these hundreds of NP-complete problems are many on which some very bright people have been working for a very long time, and none of them has found a polynomial-time algorithm to solve their pet problem. If all of these problems had polynomial-time algorithms, it would be rather unlikely that all of these brilliant researchers would have missed them; so most people who work in the field think that none of the NP-complete problems are in the class P. But an argument based on the intelligence of the researchers who have failed to find polynomial-time algorithms for their favrourite problems is not a proof that no such algorithm exists. This is the fundamental unsolved problem of theoretical computer science: is P = NP or not? If you solve it, you will no doubt earn a Fields Medal. On the other hand, if you attempt to solve it, you'd better do other research as well; otherwise you may end up having to work beneath your qualifications like the mathematicians André, Gilbert and Pierre who ended up as menial labourers in a booze factory.

While teaching a course in algorithm analysis using AHU as a text, I thought of a story that I could use to illustrate the polynomial transformations from CNF-satisfiability right through to the partition problem. I submitted it to The Mathematical Intelligencer, and the editor-in-chief Prof. Chandler Davis, an old friend of mine, accepted it (he suggested the title). Throughout this monograph I have been trying to combat the contempt that some computer types have for

mathematics. This story, on the other hand, makes fun of mathematicians who have contempt for computer science.

Completely Non-Plussed

by Timothy Walsh

This story appeared in The Mathematical Intelligencer 9, #4 (1997), p77.

Once upon a time (in 1971), an Associate Professor of Mathematics went to his Chairman (the only Full Professor in the Department) and demanded a promotion. The Chairman handed him a book the size of a Toronto telephone directory, entitled "Conditions for Promotion to Full Professor". Being an algebraist specializing in Boolean Algebras, the Associate Professor not only wrote the Conditions as a Boolean expression but also put it into Conjunctive Normal Form. He hoped to be able to determine which of the thousands of variables must be assigned the value TRUE in order to satisfy the Conditions, or else to prove that no such assignment is possible. In the latter case he could threaten to show up the Chairman as a fraud and then trade his silence for the desired promotion. But he soon calculated that even a supercomputer would have to work longer than the expected lifetime of the the universe to decide whether that expression is satisfiable by trying all possible assignments of truth values to the variables; so he set out to find a more efficient algorithm.

Now in those days mathematicians tended to scorn anything as dirty as Computer Science; their idea of an efficient algorithm was one that ran faster on the computer (if indeed they knew how to program) than the other algorithm they knew. In his case, the efficient algorithm first transformed the Boolean expression into one in which each clause contains exactly three variables. But even the efficient algorithm proved unequal to the task; so our Associate Professor thought: "Well, perhaps if I transformed the problem into a simpler one ...". Day and night he worked, neglecting both teaching and research, until he found a graph that has a proper colouring in a certain number of colours if (and only if) his Boolean expression can be satisfied.

Our Associate Professor did not realize it, but he had just proved that 3-SAT and the graph-colouring problem are NP-complete. But instead of using this result to further his case for promotion, he rushed out of his office and down the hall to the office of an Assistant Professor, a graph theorist.

"I have an interesting problem for you," panted the Associate Professor, unrolling the wallpaper on which he had drawn his graph. "Can this graph be coloured in 3124 colours? I know you're coming up for tenure soon, and I'm on the Tenure Committee. Solve this problem for me, and I'll see to it that you get in."

The Assistant Professor tried the two graph-colouring algorithms he knew and found them both wanting; so he too decided to transform his problem into a simpler one. Several student complaints later, he came up with a set and a family of subsets such that some pairwise-disjoint subfamily can cover the original set if and only if the graph has the desired colouring. But instead of backing up his tenure application with his NP-completeness proof of the exact cover problem, he promised his graduate student, a set theorist, an M.Sc. for covering his set.

The graduate student didn't know how to program a computer, but he had learned from masters how to tackle problems he couldn't solve himself: transform them and give them to someone lower on the hierarchy than you, a procedure made possible by his position as an Instructor. While stacks of assignments lay unmarked on his desk, he laboured mightily until he found a sequence of integers that has a subsequence adding up to a certain number if and only if his set has an exact cover. But instead of writing up his proof of the NP-completeness of the knapsack problem as an M. Sc. thesis, he presented his problem to his class, promising a mark of 100% in the course to the first student to solve it.

The brightest student in his class took up the challenge. Instead of studying for his exams, he transformed the sequence into another one that can be partitioned into two equal-sum subsequences if the original sequence has the required subsequence, and gave his new sequence to his girlfriend, promising her just about anything if she would write a program to partition it. After all, he reasoned, his girlfriend is a Computer Science student, and what are Computer Science students good for besides writing programs?

"Where did you get these numbers?" she asked. His male ego swelling, he showed her the numbers he had been given and the transformation he had discovered. "And who gave you those numbers?" she asked. On being told the name of his Instructor, she said, "Well, I'll get to work on it. See you around!"

Following the leads, she went from Instructor to Assistant Professor to Associate Professor, coaxing a transformation out of each of them and tracing the partition problem all the way back to CNF-satisfiability. Since she attended Computer Science seminars, she knew that the CNF-satisfiability problem had just been proved to be NP-complete, so that the transformations she had been given constituted NP-completeness proofs of the 3-SAT, graph-colouring, exact cover, knapsack and partition problems. As soon as her final exams were over, she wrote up these proofs in a paper, thanking all four mathematicians "for their invaluable assistance". She was confident that they wouldn't dare challenge the originality of her results even if they did happen to find out about them from someone who reads Computer Science literature lest they admit having missed the significance of the transformations they had unwittingly given her. And upon entering Graduate School the following year she presented her results as an M. Sc. thesis, thus achieving the distinction of being the first student in the history of that Computer Science Department to obtain an M. Sc. before running out of funds.