CHAPTER 7. RECURSION – ADVANTAGES AND DISADVANTAGES

You have already seen two examples of recursion. One example is mathematical induction, which is sometimes called proof by recurrence. A proposition about a big number n is proved assuming that the same proposition holds true for some numbers smaller than n: just $n - 1$ if you're doing standard induction, some other set of numbers smaller than n otherwise.

The other example is the recursive definition of the Fibonacci numbers:

$$f(0) = 0, f(1) = 1; f(n) = f(n\text{-}1) + f(n\text{-}2) \text{ if } n \geq 2.$$

In this case, the value of $f(n)$ for a big n – that is, $n \geq 2$ – is defined in terms of the value of $f(m)$ for some numbers m smaller than n.

7.1 Recusively defined functions

A recursive definition of a function is useful for constructing a table of values of that function. There are non-recursive definitions of the Fibonacci numbers, but they are much less useful than the above recursive definition even for calculating a single Fibonacci number, let alone a table of values. Even if a simple non-recursive definition is known, it is usually easier to use a recursive one to construct a table of values, and people usually do so without even realizing it. A case in point is the factorial function. The non-recursive definition of $n!$ is

$$0! = 1,\ n! = 1 \times 2 \times 3 \times \cdots \times n \text{ if } n \geq 1.$$

Now start constructing a table of values of $n!$. You begin as follows:

n	0	1	2	3	4
$n!$	1	1	2	6	

Now calculate 4! Did you use the non-recursive definition? If so, you would have multiplied 1 by 2, then by 3 and then by 4, but I'll bet that isn't what you did. Instead, you probably realized that you had already written down the product $1 \times 2 \times 3 = 6$ as 3! and you multiplied that number by 4. By so doing, you unwittingly used the recursive definition of $n!$, which is

$$0! = 1,\ n! = n \times (n-1)! \text{ if } n \geq 1.$$

Another recursively defined function that will be used later is

$$h(0) = 0, h(n) = 2 \times h(n-1) + 1.$$

Can you find a non-recursive definition for this function? The first step is to construct a table of values.

n	0	1	2	3	4	5	6	7	8
$h(n)$	0	1	3	7	15	31	63	127	255

Do you recognize the sequence of numbers in this table of values? If not, try adding 1 to each of the numbers. Now you should recognize these numbers: they are the powers of 2; so $h(n)$ must

be $2^n - 1$. You can prove this assertion by standard induction because $h(n)$ depends only upon $h(n\text{-}1)$, but, given the form of the recursive definition, you will want to go from $n - 1$ to n instead of from n to $n + 1$.

Basic step: $n = 0$. According to the definition, $h(0) = 0$ and $2^0 - 1 = 0$.

Induction step. Suppose that $n \geq 1$ and that $h(n - 1) = 2^{n-1} - 1$. Required to prove: $h(n) = 2^n - 1$. From the definition, $h(n) = 2 \times (2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1$, QED.

Sets too can be defined recursively – by adding elements to a set if certain other elements are already in the set – but since I won't be using recursively defined sets in this monograph, I wont discuss them here. If you're interested, you can find a discussion of these objects in many discrete mathematics texts including [Kenneth H. Rosen, Discrete Mathematics and its Applications, Seventh Edition, McGraw-Hill Education, 2011, 1072 pages].

7.2 Recursive algorithms

A recursive algorithm is one that calls itself with a smaller value of a parameter unless that parameter is already small enough. To give a non-mathematical example, a non-recursive algorithm for walking to a wall would be "walk towards the wall until you reach it", whereas a recursive algorithm would be "If you're just one step from the wall, take that step; otherwise, take one step towards the wall, and now that you're one step closer to the wall, execute the same algorithm to walk to the wall."

This may seem a highly unnatural way to describe the way to deal with real life situations, but it is useful for programming recursively defined functions because a recursive definition of a function is easy to translate into a recursive algorithm. Besides, you don't need a loop invariant to prove that a recursive algorithm is correct. Induction, either simple or generalized, will work, and that's a lot easier than using a loop invariant, because you know what proposition you have to assume – that the algorithm works for a smaller parameter – whereas finding a loop invariant is notoriously difficult. On the other hand, a recursive algorithm is usually slower than a non-recursive one for solving the same problem. The trick is to decide which is greater, the difference in execution speed or the difference in ease of programming and proving correctness, and to do this, you have to be able to measure them both.

As a first example, consider the recursive definition of n!. It is easily translated into the following algorithm:

```
natural function fact(n: natural)
   if n=0 then return 1;
   else return n*fact(n-1);
   end if;
end fact.
```

How would a computer execute this program? Let's trace the execution of the instruction f = fact(4). The computer sets $n = 4$ and starts executing the function from the beginning. Since $n > 0$, it will execute the **else** alternative: it will first execute fact(n-1) – that is, fact(3) – and then multiply the result by n, which is 4. To execute fact(3), it must change n from 4 to 3, but since it

has to multiply by the original value of *n*, it has to store 4 somewhere, and since this may have to be done several times, it stores the 4 on a stack. Then it changes *n* to 3 and executes the function again with this new value of *n*. When it finishes executing fact(3) it returns 6, sets *n* to the number on the top of the stack, which is 4, deletes it from the stack and then resumes executing fact(4). It now multiplies the value returned (6) by *n*, which is 4, and since the stack is now, empty, the computer returns the product (24) to the main program, which assigns this value to *f*. Here is a trace of that execution, with the stack drawn horizontally to save space.

```
n  stack   value returned
4
3  4
2  4 3
1  4 3 2
0  4 3 2 1       1
1  4 3 2         1
2  4 3           2
3  4             6
4               24
```

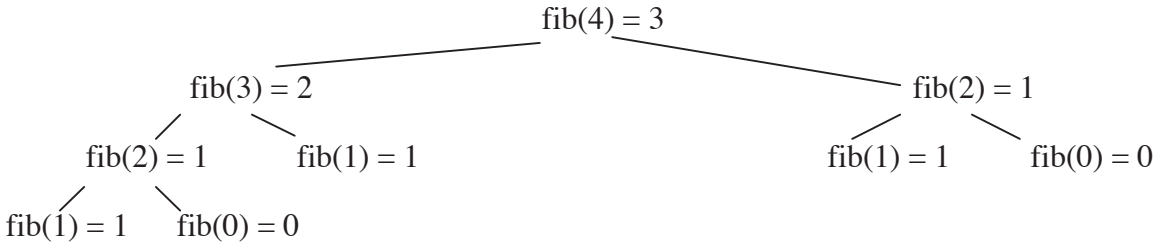
As you can see, the computer does all the same multipications as it would do when executing the non-recursive program, which sets *f* to 1 and then multiplies it by $i = 1, 2, \ldots, n$, and in addition it has to manage the stack. The time complexity is $O(n)$, the same as for the non-recursive algorithm, but the big *O* is now bigger. In addition, since all the numbers from 1 to *n* have to be on the stack at the same time, the space complexity is $O(n)$, whereas with the non-recursive algorithm it is $O(1)$ because all you have to store is *f*, *n* and the loop index *i*. Of course, it is a bit easier to translate the recursive definition into the recursive algorithm than into the non-recursive one and you don't need a loop invariant to prove that this algorithm works, but the loop invariant you need to prove that the non-recursive algorithm works is easy to find: at the beginning of the *i*th iteration of the loop, $f = (i - 1)!$. I think you would agree with me that in this case the disadvantages of recursion outweight the advantages, although the difference isn't all that great.

A more extreme example is the calculation of the *n*th Fibonacci number. The recursive definition can be easily translated into the following recursive algorithm:

```
natural function fib(n: natural)
   if n ≤ 1 then return n;
   else return fib(n-1) + fib(n-2);
   end if;
end fib.
```

Now the computer has to execute fib(n-1) and then execute fib(n-2) and **then** add them together. To know where to resume the execution of fib(n), the computer has to store on another stack the address of the machine-language instruction following the last instruction of the recursive call it intends to execute – either fib(n-1) or fib(n-2). Since stack management isn't the greatest obstacle to efficiency, I leave you to include the contents of these two stacks in the following trace of the execution of the instruction f = fib(4). The trace is represented as a binary tree. You traverse the tree, starting at the top. For each node, if there's nothing below it, you write the value of either fib(1) or fib(0). Otherwise you follow the left branch if you see the node

for the first time and the right branch if you see it for the second time (you've got to it from the left). When you see it for the third time (you've got to it from the right), you add the two numbers just below it, write the sum beside the function call and go to the node above unless you're at the top, in which case the trace is over.



As you can see, fib(2) and fib(0) are calculated twice and fib(1) is calculated three times. This seems redundant even for fib(4). What is the total number of additions that are done when executing fib(n)? Let $t(n)$ be this number. To execute fib(1) or fib(0), no additions have to be done. To execute fib(n) for $n > 1$, the computer has to execute fib(n-1), which takes $t(n$-1) additions, and then execute fib(n-2), which takes $t(n$-2) additions, and then do one more addition to compute fib(n-1) + fib(n-2); so $t(n)$ is defined recursiveley by

$$t(0) = t(1) = 0,\ t(n) = t(n\text{-}1) + t(n\text{-}2) + 1 \text{ if } n > 1.$$

This equation looks a bit like the recursive definition of fib(n); so one might suspect that $t(n)$ bears some resemblance to fib(n). Here is a table of value of both of these functions.

n	0	1	2	3	4	5	6	7	8	9	10	11	12
$t(n)$	0	0	1	2	4	7	12	20	33	54	88	143	232
fib(n)	0	1	1	2	3	5	8	13	21	34	55	89	144

From these values one could guess that $t(n) = \text{fib}(n+1) - 1$. As an exercise, you can prove this assertion by generalized induction, using the recursive definitions of both functions. What I wanted to establish here is how inefficient the recursive algorithm is. Remember that fib(n) grows exponentially with n: it is at least as big as $(1{,}5)^{n-2}$ for all $n \geq 2$; so $t(n)$ also grows exponentially with n. There is an obvious non-recursive algorithm for calculating fib(n) – you just construct the whole table of values of fib(i) for i from 0 to n using the recursive definition of fib(n). This algorithm takes $O(n)$ time and $O(n)$ space. But to calculate fib(n) you don't need the whole table. All you need is fib(n-1) and fib(n-2). The following algorithm uses this fact to calculate fib(n) using $O(n)$ time and only $O(1)$ space. The loop invariant and the comments needed to prove the induction step of the correctness proof are given; I leave it to you to complete the correctness proof.

```
natural function fib(n: natural)
local variables x, y, z: natural;
if n ≤ 1 then return n; end if;
   x ← 0;
   y ← 1;
   for i ← 2 to n do                    {loop invariant: x = fib(i-2) and y = fib(i-1)}
      z ← x + y;                                                  {now z = fib(i)}
      x ← y;                                                      {now x = fib(i-1)}
      y ← z;                                                      {now y = fib(i)}
   end for;                  {i changes to i + 1; so x is now fib(i-2) and y is now fib(i-1)}
   return z;                          {When i was n, z was fib(n) and it doesn't change.}
end fib.
```

Now this algorithm is **a little** harder to program and prove correct than the recursive one but it is **a lot** more efficient. Later in this chapter I will show you an algorithm for calculating fib(n) in $O(\log(n))$ arithmetic operations, but I'm sure you'll agree with me that even the above algorithm is so much better than the recursive one that the recursive one is to be avoided like the plague. At this point you are probably asking yourself what good recursive algorithms could possibly be! In the following two sections I present two problems for which the recursive algorithm is only **a little** less efficient than the non-recursive one but a **lot easier** to program and prove correct.

7.3 Computing the *n*th power in $O(\log n)$ arithmetic operations

The naïve algorithm for computing a^n, which consists of multiplying 1 by a n times, does n multiplications. The following recursive definition of a^n leads to a much more efficient algorithm, at least in terms of the number of arithmetic operations:

$$a^0 = \begin{cases} 1 \text{ if } n = 0 \\ a \times a^{n-1} \text{ if } n \text{ is odd} \\ \left(a^{n/2}\right)^2 \text{ if } n \text{ is even and } n > 0 \end{cases}$$

This recursive definition can be easily translated into the following recursive algorithm:

```
real expod(a: real, n: natural)          {precondition: a and n are not both 0}
   local variable z: natural;
   if n = 0 then return 1; end if;
   if n is odd then return a*expod(a,n-1);
   else
      z ← expod(a,n/2); return z*z;
   end if;
end expod.                                {postcondition: expod(a, n) returns a^n.}
```

This algorithm and various variations on it have been known for 2000 years. A good acccount of this subject can be found in [Donald E. Knuth. Seminumerical Algorithms, volume 2 of The Art of Computer Programming. Addison-Wesley, Reading, Massachusetts, second

edition, 1981]. It's easy to prove by induction that this algorithm works. Of course, you have to use generalized induction because $n - 1$ isn't the only parameter used in the recursive call.

Basic step: $n = 0$. The algorithm returns 1 and $a^0 = 1$ if $a \neq 0$.

Induction step. Suppose that $n > 0$ and that expod(a,m) returns a^m for every integer m such that $0 \leq m < n$. Required to prove: expod(a,n) returns a^n. Since $n > 0$, $0 \leq n - 1 < n$ and $0 \leq n/2 < n$. If n is odd, then expod(a, n-1) returns a^{n-1}; so expod(a, n) returns $a^* a^{n-1} = a^n$ even if $a = 0$. If n is even, then expod(a, n/2) returns $a^{n/2}$; so $z = a^{n/2}$ and expod(a, n) returns $\left(a^{n/2}\right)^2 = a^n$.

How many arithmetic operations does a call to expod(a, n) do? If n is odd, n is decreased by 1 and becomes even and then one multiplication is done. If n is even, n is divided by 2 and then one multiplication is done. For each division of n by 2, either one or two multiplications are done. The number of times that n has to be divided by 2, rounding down if necessary by first subtracting 1, before it drops to 0 is $\lfloor \log_2 n \rfloor$; so the total number of arithmetic operations is in $O(\log n)$.

Here is a non-recursive version of that algorithm.

```
real function power(a: real , n: natural)          {precondition: a and n are not both 0}
   t ← n;  p ← 1;  z ← a;
   loop
      if t is odd then p ← p*z; end if;
      if t ≤ 1 then exit the loop end if;              {This avoids squaring z uselessly.}
      t ← floor(t/2);
      z ← z*z;
   end loop;
   return p;                                          {postcondition: power(a, n) returns a^n}
end power.
```

The number of iterations of the loop is $\lfloor \log_2 n \rfloor$; so the number of multiplications is between $\lfloor \log_2 n \rfloor$ (if t is always even because n is a power of 2) and $2\lfloor \log_2 n \rfloor$ (if t is always odd because n is one less than a power of 2).

To prove that this algorithm is correct, we have to find a loop invariant, and this isn't easy. We begin by tracing the algorithm for some value of n, say 21 (see the trace below). Use your powers of observation to see if you can find a pattern. The answer is below the trace; so cover up the answer, observe the trace carefully and try to find the pattern before you uncover the answer.

t	21	10	5	2	1	
p	a^0	a^1	a^1	a^5	a^5	a^{21}
z	a^1	a^2	a^4	a^8	a^{16}	

DON'T PEEK UNTIL YOU'VE TRIED TO FIND IT YOURSELF!

If you multiply the exponent in z by t and add the exponent in p, you always get 21, which is n. Is this the pattern you found? I told you that loop invariants are notoriously difficult to find! Anyway, this pattern suggests a candidate for a loop invariant: $z^t \times p = a^n$. And here's a proof, using that loop invariant, that the algorithm works.

If $n = 0$, the algorithm returns 1, which is equal to a^0 unless $a = 0$. Now suppose that $n > 0$.

Basic step: At the beginning of the first iteration of the loop, $t = n$, $p = 1$ and $z = a$; so $z^t \times p = a^n \times 1 = a^n$.

Induction step. Now suppose that at the beginning of some iteration of the loop, $t > 1$ and $z^t \times p = a^n$. If t is odd, then p changes to $p \times z$, t changes to $(t-1)/2$ and z changes to $z \times z$; so $z^t \times p$ changes to $\left(z^2\right)^{(t-1)/2} \times (p \times z) = z^{(t-1)+1} \times p = z^t \times p = a^n$. And if t is even, then t changes to $t/2$ and z changes to $z \times z$; so $z^t \times p$ changes to $\left(z^2\right)^{t/2} \times p = z^t \times p = a^n$.

Terminal step. After a finite number of iterations of the loop, t drops to 1. Then since t is now odd, p changes to $p \times z$, but since $t = 1$, $p \times z = p^t \times z = a^n$. Then the algorithm exits the loop and returns the final value of p, which is a^n.

This algorithm has other applications besides finding powers of a real number. Since it never divides, a doesn't have to belong to a set in which every element has a multiplicative inverse. And since every quantity it ever computes is a power of a, it isn't necessary that multiplication be commutative either. It is enough that the product of two elements of the set is an element of the set, that multiplication is associative and that there is a multiplicative identity, that is, an element 1 such that $a*1 = 1*a$ for every element a in the set. The set of square matrices, all of the same size, satisfies these conditions with the multiplicative identity the matrix with 1 everywhere on the main diagonal and 0 everywhere else (prove that statement or look it up in any linear algebra text); so a, t, p and z can be square matrices, all of the same size and the algorithm will still work.

In particular, it works for matrices with 2 rows and 2 columns, including $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. This matrix is of particular interest because the Fibonacci numbers satisfy the matrix equation $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} f(n-1) \\ f(n-2) \end{pmatrix} = \begin{pmatrix} f(n) \\ f(n-1) \end{pmatrix}$ (prove it!). Since $f(0) = 0$ and $f(1) = 1$, it follows by standard induction on n that $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} f(n) \\ f(n-1) \end{pmatrix}$ (prove it!). So to calculate $f(n)$, all you have to do is to execute Power(a, n-1) with $a = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ and 1 replaced by $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ and return the upper left element of the resulting matrix and you will have calculated $f(n)$ in at most $2\lfloor \log_2 n \rfloor$ multiplications of two 2 by 2 matrices, and since each such matrix multiplication involves 8 multiplications of of two numbers, the total number of multiplications of two numbers is at most

$16\lfloor \log_2 n \rfloor$.

But the factor 16 can be reduced considerably by taking advantage of certain properties of the powers of a. We show that the matrices p and z are always symmetrical and that the upper left element is always the sum of the lower right element and the lower left (or upper right) element.

After the initialisation,

$$z = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \text{ and } p = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

These matrices have these two properties.

Suppose now that this is true at the beginning of a given iteration of the loop:

$$z = \begin{pmatrix} x+y & y \\ y & x \end{pmatrix} \text{ and } p = \begin{pmatrix} i+j & j \\ j & i \end{pmatrix}.$$

At the end of this iteration, if t is odd,

$$p = p * z = \begin{pmatrix} i+j & j \\ j & i \end{pmatrix}\begin{pmatrix} x+y & y \\ y & x \end{pmatrix} = \begin{pmatrix} ix+iy+jx+jy+jy & iy+jy+jx \\ jx+jy+iy & jy+ix \end{pmatrix},$$

which has these two properties. If $t \geq 1$,

$$z = z * z = \begin{pmatrix} x+y & y \\ y & x \end{pmatrix}\begin{pmatrix} x+y & y \\ y & x \end{pmatrix} = \begin{pmatrix} x^2+2xy+2y^2 & 2xy+y^2 \\ 2xy+y^2 & x^2+y^2 \end{pmatrix},$$

which has these two properties.

Initialise y and i to 1 and x and j to 0. Replace $p \leftarrow p * z$ by a series of instructions that changes j to $jx + jy + iy$ and i to $jy + ix$. Replace $z \leftarrow z * z$ by a series of instructions that changes y to $2xy + y^2$ and x to $x^2 + y^2$. And at the end, return $i + j$.

This does 4 multiplications of numbers for $p * z$ (evaluate jy and use it twice) and 4 mutliplications for $z * z$ (evaluate y^2 and use it twice); so the factor 16 can be reduced to 8. This is approximately what can be found in the literature. But even this can be improved upon. The new value of $i - j$ is $i(x-y) - jx$ and the new value of $x - y$ is $x(x-2y)$. These observations lead to the following algorithm:

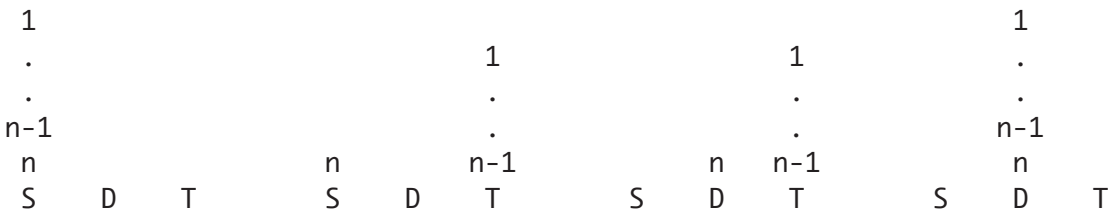
```
natural function Fibo(n:natural)
local variables t, i, j, x, y, u natural;
   t ← n – 1;
   y ← 1;
   i ← 1;
   x ← 0;
   j ← 0;
   loop
      if t is odd then
         u ← j*x;
         j ← u + (j+i)*y;
         i ← j – u - i*(y-x);                          {3 multiplications instead of 4}
      end if
      if t ≤ 1 then exit end if;
      u ← y;
      y ← y*(x+x+y);
      x ←y - x*(u+u-x);          {2 multiplications instead of 4; so the factor 8 is reduced to 5}
      t ← t div 2;
   end loop;
   return i+j;
end Fibo.
```

This idea can be generalized to the solution of any recursively defined function as long as the coefficients are constant; the amount of optimization that can be done depends upon the properties of the corresponding square matrix.

7.4 The Tower of Hanoi

Once upon a time a group of monks decided to put an end to the world. They were given three pegs and 64 rings, all of different sizes, that were stacked on one of the pegs in order of size, with the biggest ring on the bottom and the smallest ring on top. And they were told that if they stacked all the rings on one of the other pegs without ever moving more than one ring at a time or putting a big ring on top of a smaller ring, the world would come to an end. But if they ever put a big ring on top of a smaller ring, the big ring would crush the smaller one and they would never be able to destroy the world.

This puzzle, and probably this myth as well, were invented by the French mathematician Edouard Lucas in 1883 as an illustration of recursion. Here is his recursive solution. Let's call the peg on which the rings are originally stacked *S* for source, the peg on which the rings are to be stacked *D* for destination and the third peg *T* for the French word tampon, which means buffer. Since I'm too lazy to draw rings and pegs, I'll just represent the pegs by these letters and the rings by numbers, with the smallest ring labeled 1 and the largest one *n*. Here are the initial position of the rings, the final position and two intermediate positions that are necessary, because to be able to move ring *n* from *S* to *D* without violating either of the rules, the monks would have had to stack all the other pegs onto *T*.



Lucas' recursive algorithm makes a recursive call to move rings 1 through $n - 1$ from the initial position to the first intermediate position shown above, then moves ring n, and finally makes a second recursive call to stack rings 1 through $n - 1$ on top of ring n.

```
procedure EL(n: natural, S, D, T: pegs)     {precondition: n > 0 and the rings are stacked on S}
   if (n > 1) then  EL(n-1, S, T, D) end if;
   move ring n from peg S to peg D;
   if (n > 1) then  EL(n-1, T, D, S) end if;
end EL.          {postcondition: the rings are stacked on D and the rules have never been violated}
```

It is easy to prove by standard induction on n that this algorithm is correct.

Basic step: $n = 1$. The algorithm simply moves ring 1 from S to D. It can't move more than one ring at a time or put a big ring on a smaller ring because there is only one ring.

Induction step. Suppose that the algorithm works properly for $n - 1$ rings. Required to prove: that it works properly for n rings. The first recursive call takes rings 1 through $n - 1$, which are now on S, and stacks them on T without ever moving more than one ring at a time or putting a big ring on top of a smaller ring among these rings. Since all these rings are smaller than ring n, it never puts a big ring on top of a smaller ring among all the rings. The next instruction moves ring n from S to D, and since there are no rings on top of ring n and peg D is unoccupied, it moves only one ring and doesn't put a big ring on top of a smaller ring. The second recursive call takes rings 1 through $n - 1$, which are now on T, and stacks them on D, and thus on top of ring n, without ever moving more than one ring at a time or putting a big ring on top of a smaller one among these rings. Since all these rings are smaller than ring n, it never puts a big ring on top of a smaller ring among all the rings. So the whole algorithm stacks all the rings on D in order of size without ever violating either of the rules, QED.

How many moves does it take to execute this algorithm? Let $h(n)$ be the number of moves made for n rings. If $n = 1$ it takes 1 move; so $h(n) = 1$. If $n > 1$, the first recursive call makes $h(n\text{-}1)$ moves, the next instruction makes 1 move and the second recursive call also makes $h(n\text{-}1)$ moves; so $h(n) = 2h(n\text{-}1) + 1$. If we set $h(0)$ to 0, which is reasonable because if there are no rings there is nothing to be done, then from the recurrence we have $h(1) = 1$; so we can change the equation $h(1) = 1$ to $h(0) = 0$. Do you remember seeing this recursively defined function before? Look through the section on recursively defined functions and see if you can find this one. Yes, it's there, and the non-recursive definition of that function is $2^n - 1$.

With 64 rings, the number of moves is $2^{64} - 1$, which is about $1.844674407 \times 10^{19}$. Suppose the monks made one move per second, day and night, without either stopping or making a wrong move. Since there are 31557600 seconds in an average year (counting one leap year

every four years), it would take them about $5.84542046 \times 10^{11}$ years to put an end to the world. But the age of the universe is only about 1.37×10^{10} years; so it seems that the world is safe for a while anyway.

But suppose that there were only 32 rings instead of 64. Then the number of moves would be only $2^{32} - 1 = 4294967295$, and at one move per second, it would take the monks only about 136.1 years to destroy the world. If they started in 1883 when the recursive solution to the puzzle was first published, they would be able to bring the world to an end some time in 2019 – provided that they never made any mistakes. It is highly unlikely that they would be able to do so, however, because, if you recall, executing a recursive algorithm requires a stack, in this case of size 32, and keeping that many things in your head at one time is beyond human capability. They would have had to write the stack down and update it with each move, which would make the whole process considerably slower.

At around the same time, a non-recursive solution to this puzzle was proposed. It is clear that the first move must be to move ring 1, since it is the only ring that has nothing on top of it. It is also clear that it doesn't pay to move the same ring twice in succession; so moves of ring 1 must alternate with moves of some other ring. And if you're moving a ring bigger than ring 1, there is only one place to move it – to the peg that doesn't contain ring 1. The only choice you have is: when you're moving ring 1, which of two possible pegs should you move it to? Here is a trace of the algorithm EL for $n = 1, 2$ and 3.

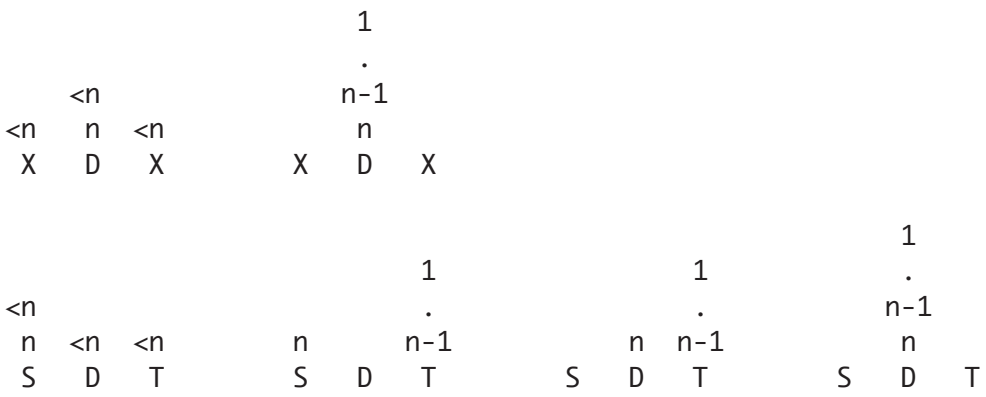
```
1         1
S D T   S D T


1                       1
2       2   1     2 1     2
S D T   S D T   S D T   S D T


1                                                       1
2       2                   1       1             2       2
3       3 1     3 1 2   3   2     3 2   1 3 2   1 3       3
S D T   S D T   S D T   S D T   S D T   S D T   S D T   S D T
```

When n is odd (1 or 3), ring 1 moves from peg to peg in the cyclic order S, D, T, S and when n is even (2), ring 1 moves from peg to peg in the cyclic order S, T, D, S. Generalizing this observation to all n yields a non-recursive algorithm for solving the puzzle, for which you should be able to write a pseudocode. It is much more difficult to program than the recursive algorithm because you have to find the second-smallest topmost ring, and this entails keeping track of where all the rings are, whereas all the recursive algorithm has to do is print out instructions for moving the rings. In addition, the non-recursive algorithm is much harder to prove correct. It was suggested around 1883, but its correctness proof appeared in [P. Buneman and L. Levy, The towers of Hanoi problem, Information Processing Letters 10 (1980), p. 243-244] – nearly a century later!

But even this algorithm would have been insufficient to help the monks destroy the world any time soon. You see, human beings do make errors, and this algorithm shows no way to

recover from an error. In addition, if the monk who was currently working on the puzzle suddenly died without writing down his last move and none of the other monks was watching him, his successor would have no way of knowing whether or not to resume by moving the first ring or the second-smallest topmost ring.

An algorithm that does correct errors appeared in [F. Scarioni and M.G. Speranza, A probabilistic analysis of an error-correcting algorithmfor the towers of Hanoi puzzle, Information Processing Letters 18 (1984), p. 99-103]. Starting from the rings in any legal position (no big ring on top of a smaller ring), it stacks the rings on a given peg *D*. There are two cases to consider, depending upon whether ring *n* starts on peg *D*. The sequences of positions below illustrates these two cases. Here *<n* means some stack of rings smaller than ring *n*.



And here is the algorithm:

procedure SS(n: **natural**, D: peg) {precondition: n > 0 and the rings are in a legal position}
local variables S, T: peg;
if ring n is on D **then** SS(n-1, D);
  else
    let S be the peg that contains ring n and T the peg other that S and D;
    SS(n-1,T);
    nove ring n from peg S to peg D;
    SS(n-1,D);
  end if;
end SS. {postcondition: the rings are stacked on D and the rules have never been violated}

I leave it to you to prove by standard induction that this algorithm is correct. The proof is similar to the one for the algorithm EL.

Of course, this algorithm too would not have helped the monks destroy the world because it is recursive. Two non-recursive algorithms for solving the same problem appeared in 1983. They are both based on the concept that for each ring there is a target peg – the peg to which that ring must be moved once there are no rings on top of it and the peg is unoccupied. The target peg for ring *n* is *D*. From the above diagram it is clear that for any ring *i*, if ring *i* is on its target peg, then the target peg for ring *i* –1 is the same peg: it must be put on top of ring *i*. And if ring *i* is not on its target peg, then the target peg for ring *i* – 1 is the peg other than the peg containing ring *i* and the target peg for ring *i*: ring *i* –1 must be put on that peg so that ring *i* can be moved to its target peg.

The algorithm in [M.C. Er, An iterative solution to the generalized towers of Hanoi problem, BIT 23 (1983), 295-302] stores an array containing the target peg for each ring and updates it with every move. This algorithm would not have helped the monks to destroy the world because they would have had to update the array of target pegs by hand, which would take them as long as updating the stack necessary for solving the recursive version.

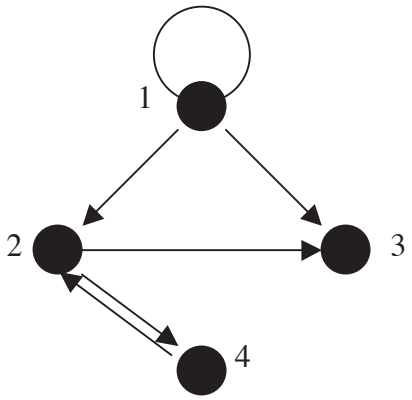
My algorithm in [T.R. Walsh, A case for iteration, Proceedings of the 14th Southeastern Conference on Computing, Graph Theory and Combinatorics, 1983, Congressus Numerantium 40 (1983), p. 38-43] doesn't store an array of target pegs for each ring. Instead, it computes the target peg for ring 1 and uses it to decide whether the first move should be of ring 1 or some other ring. Then it uses the following observation: if ring i >1 has just been moved, then the target peg for ring 1 is the peg containing ring i if and only if i is even. If ring i >1 has just been moved, suppose that it was moved from peg A to peg C and call the remaining peg B. All the rings from i – 1 down to 1 are now stacked on peg B. Since ring i is now on its target peg, the target peg for ring i –1 must be peg C. Since ring i –1 is on peg B, the target peg for ring i –2 must be peg A. Since ring i –2 is on peg B, the target peg for ring i – 3 must be peg C, and so on. The target pegs for rings i –1, i –2, i –3, …, 1 alternate between peg C and peg A, beginning with peg C. So the target peg for ring 1 is peg C (the peg containing ring i) if i is even and peg A (the peg containing neither ring i nor ring 1) if i is odd. And here is the algorithm:

```
procedure TW(n: natural, D: peg)      {precondition: n >0 and the rings are in a legal position}
local variables: T: peg; i: natural;
   T ← D;                                                    {calculating the target peg for ring 1}
   for i ← n-1 downto 1
      if ring i is not on peg T then T ← the peg other than T and the peg containing ring i end if;
   end for;                                                  {peg T is the target peg for ring 1}
   if ring 1 is not on peg t then move ring 1 to peg t end if;
   while the the rings are not yet stacked on peg D
      move the second-smallest topmost ring i to the peg not containing either ring i or ring 1;
      if i is even then
         move ring 1 on top of ring i;
      else
         move ring 1 to the peg not containing either ring i or ring 1;
      end if;
   end while
end TW.       {postcondition: the rings are stacked on D and the rules have never been violated}
```

A monk who took over the puzzle from his predecessor could have calculated which ring to move first and could have corrected any errors made by his predecessor by executing the algorithm from the beginning. Since it stores no arrays except for the rings themselves, which the monks could see, they could have destroyed the world by the year 2019 if only this algorithm had been published as early as 1883. Fortunately it didn't appear until 1983; so when the world blows up in 2119, the aliens who visit it will know whom to blame.

CHAPTER 8. ALGORITHMS ON GRAPHS

A *directed graph* $G=(V,A)$ is a set V of *vertices* and a set A of *arcs*. An *arc* is an ordered pair (u,v) of vertices in V. If $u = v$, then the arc (u,v) is called a *loop*. In the diagram below, $V = \{1,2,3,4\}$ and $A = \{(1,1),(1,2),(1,3),(2,3),(2,4),(4,2)\}$.

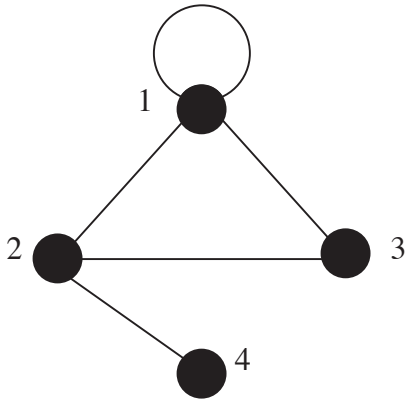


If there is an arc $a = (u,v)$, then a is said to be *incident from* u and *incident to* v, u is *adjacent to* v, v is *adjacent from* u and v is a *neighbour* of u. A loop (u,u) is incident from u and incident to u, making u adjacent to and from itself and a neighbour of itself. The *in-degree* of a vertex v is the number of arcs incident to v and the *out-degree* of a vertex u is the number of arcs incident from u. In the above diagram, vertex 1 has in-degree 1 and out-degree 3, vertex 2 has in-degree 2 and out-degree 2, vertex 3 has in-degree 2 and out-degree 0, and vertex 4 has in-degree 1 and out-degree 1. Note that the sum of the in-degrees of all the vertices is 6, the sum of the out-degrees of all the vertices is 6 and the number of arcs is 6.

In general, the sum of the in-degrees of all the vertices, the sum of the out-degrees of all the vertices and the number of arcs of any directed graph are all the same number. Every arc (u,v) contributes 1 to the in-degree of v and therefore 1 to the sum of the in-degrees of all the vertices, and it also contributes 1 to the out-degree of u and therefore 1 to the sum of the out-degrees of all the vertices, whether or not $u = v$.

A directed graph with no loops is called a *simple* directed graph. For a directed graph, simple or not, since A is a set, all the arcs are distinct. If we are going to allow multiple arcs but not loops, then we call G a directed *multigraph*. If we are going to allow both multiple arcs and loops, then we call G a directed *pseudograph*. Since I'm not going to be using multigraphs or pseudographs, I'm not going to take the trouble to draw one. I confess at this point that the notation I use is not standard. In the literature there is no special name given to a directed graph with loops but with no multiple arcs. Since I will be using such graphs often, I just refer to them as directed graphs. But then, there isn't really any standard notation. There are many different notations, and if you read a lot of literature on graph theory, you will encounter quite a few of them. Get used to it.

In an *undirected graph* (V,E), instead of ordered pairs (u,v) there are unordered pairs $\{u,v\}$ which are called *edges*. An edge that is not a loop is sometimes referred to as a *link*. In the diagram below, $V = \{1,2,3,4\}$ and $E = \{\{1,1\},\{1,2\},\{1,3\},\{2,3\},\{2,4\}\}$.



If there is an edge $e = \{u,v\}$, then e is incident to both u and v, u and v are adjacent to each other and u and v are each other's neighbours. The *degree* of a vertex v is the number of edges incident to v, with a loop counting twice. In the above diagram, vertex 1 has degree 4, vertex 2 has degree 3, vertex 3 has degree 2 and vertex 4 has degree 1. The sum of the degrees of all the vertices is 10, which is twice the number of edges.

In general, the sum of the degrees of the vertices of any undirected graph is twice the number of edges. A link $\{u,v\}$ contributes 1 to the degree of u and 1 to the degree of v and therefore 2 to the sum of the degrees of all the vertices. A loop $\{v,v\}$ contributes 2 to the degree of v and therefore 2 to the sum of the degrees of all the vertices. The adjective *simple* means no loops allowed. The prefix *multi* means multiple edges allowed but not loops. The prefix *pseudo* means anything goes. And these last two prefixes mean that I'm not going to draw diagrams.

If you want to represent a graph in a computer, you can't very well draw it on the computer's circuits. You could represent it as a list of the elements of V and a list of the elements of A or E, but there are more efficient ways. One such way is by creating the number n of vertices and, for each vertex u, a list of all the neighbours of u. For both of the above graphs, $n = 4$. The lists for the directed graph are shown below on the left and those for the undirected graph are shown on the right.

```
1: 1,2,3
2: 3,4
3:
4: 2
```

```
1: 1,2,3
2: 1,3,4
3: 1,2
4: 2
```

This representation is called *adjacency lists*. For a graph with n vertices and m arcs or edges, the space occupied by this representation is n words for the headings of the lists and m words for the lists themselves if the graph is directed or $2m$ words otherwise for a total space-complexity in $O(n+m)$. Note that it is necessary to include both n and m in the estimate because even if $m = 0$ you would still need n words for the list headings. In a directed graph, $m \leq n^2$, and if the graph is simple, $m \leq n^2 - n$. In an undirected graph, $m \leq n(n+1)/2$, and if the graph is simple, $m \leq n(n-1)/2$. In any of these cases, the space-complexity of the adjacency list representation is in $O(n^2)$, but we still keep the estimate $O(n+m)$ in case m is much less than n^2.

Another commonly used representation is an *adjacency matrix*. It is a square array with n rows and n columns each of whose n^2 elements is either 0 or 1. If we call a matrix M, then the element in row i, column j, is denoted by $M[i,j]$. If M is the adjacency matrix of a graph G, then $M[i,j] = 1$ if the vertex j is a neighbour of the vertex i and 0 otherwise. The matrix on the left below is the adjacency matrix of the directed graph above and the matrix on the right is the adjacency matrix of the undirected graph.

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

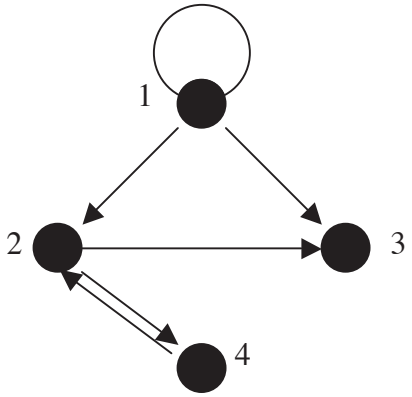
The adjacency matrix of a graph with n vertices and m arcs or edges occupies n^2 bits no matter how small m is. Since a bit occupies less space than a word, an adjacency matrix can be made more space-efficient than adjacency lists if m is almost as big as n^2; otherwise, adjacency lists are more space-efficient. For example, it can be shown (but I'm not going to do it here) that any simple undirected graph with $n \geq 3$ vertices that can be drawn on the plane so that the edges don't cross each other has at most $3n - 6$ edges. If such a graph has 1000 vertices, then it has at most 2994 edges, so that its adjacency lists would occupy only 6988 words and its adjacency matrix would occupy 1000000 bits. This type of graph is used to model many real-life things, like ground or sea transport networks; so it is useful to know adjacency lists. In general, if the number of edges in the graphs you're dealing with is bounded by a function of n that grows slower than n^2, then those graphs are called *sparse*, and for big enough n adjacency lists are more space-efficient than adjacency matrices. If not, then the graphs are called *dense* and adjacency matrices can be made more efficient than adjacency lists by packing as many bits of the matrices as possible into a word.

Suppose you want to find the number of edges or arcs in a graph. Using an adjacency matrix, you count all the 1s if the graph is directed and you count the 1s above and on the main (descending) diagonal otherwise (try it). The time-complexity is in $O(n^2)$. Using adjacency lists, you count the total number of elements in all the lists if the graph is directed; otherwise you count the total number of elements that are greater than or equal to the heading of the list (I won't keep repeating it, but try it anyway). The time-complexity is in $O(n+m)$.

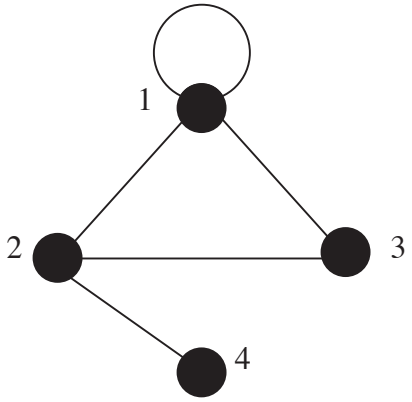
Suppose you want to find the degree of a given vertex i. Using an adjacency matrix, you count the 1s in row i for the out-degree or in column i for the in-degree, and for the degree in an undirected graph you count the 1s in row (or column) i except that a 1 in row i, column i counts twice. The time-complexity is in $O(n)$. Using adjacency lists, the procedure is a little more complex. To find the degree of i in an undirected graph, you count the total number of elements in the list with heading i, except that if i appears in that list, it counts twice. To find the out-degree of i in a directed graph, you count the total number of elements in the list with heading i. Both of these procedures take $O(n)$ time. But to find the in-degree of i, you have to count the total number of occurrences of i in all the lists, and that takes $O(n+m)$ time! For that particular problem, an adjacency matrix is more time-efficient even if m is considerably less than n^2. Both of these representations are good to know.

8.1 Paths in graphs and an algorithm to find the shortest paths

A *path* in a graph is a list of n+1 vertices $v_0, v_1, ..., v_n$ such that for each i=1,2,…,n, there is an arc (v_{i-1}, v_i) for a directed graph (an edge $\{v_{i-1}, v_i\}$ for an undirected graph). In other words, in an undirected graph you can drive on the roads in either direction, whereas in a directed graph you have to obey the one-way signs or else you may lose your driver's licence. The *length* of the path is n, the number of arcs (or edges). Note that a list consisting of a single vertex v is a path of length 0. A path is called *elementary* if all its vertices are distinct. A *cycle* is a path of length at least 1 whose first and last vertices are the same. A cycle is called elementary if all its vertices are distinct except that the first and last vertices are the same (without this exception there couldn't be any such thing as an elementary cycle).



In the directed graph above, 1,2,4,2,3 is a non-elementary path, 2 and 4,2,3 are elementary paths, 1,1 and 2,4,2 are elementary cycles, 2,4,2,4,2 is a non-elementary cycle and 2 is not a cycle.



In the undirected graph above, 2,3,1,1 is a non-elementary path, 1,3,2,4 is an elementary path, 2,3,1,1,2 is a non-elementary cycle and 1,2,3,1 and 1,1 are elementary cycles.

It is clear that if there is a path from a vertex u to a vertex v, then there is an elementary path from u to v: you follow the path, starting from u, and every time you get to some vertex x you've already been to, you delete every vertex from the first occurrence of x (exclusive) to the second one (inclusive). You will eventually get to the last vertex in the path, which is v, and you

will have eliminated all multiple occurrences of vertices. Any cycle from *v* to *v* can be made into an elementary cycle in the same way, except that when you get to the last vertex *v*, you don't delete the whole cycle just because you've been to *v* before as the first vertex. For example, in the undirected graph above, the non-elementary path 2,3,1,1,2,3,1,2,4 gets changed first to 2,3,1,2,3,1,2,4, then to 2,3,1,2,4 and finally to 2,4 and the non-elementary cycle 1,2,3,1,1,1,2,3,1 gets changed first to 1,1,1,2,3,1 and then to 1,1,2,3,1 and then to 1,2,3,1 but not to 1.

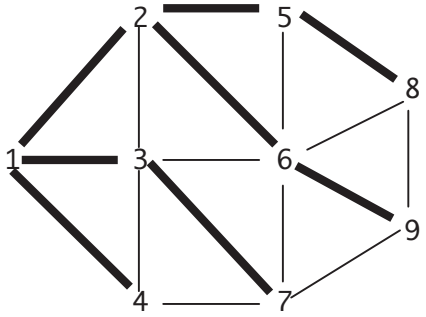
The *distance* from the vertex *u* to the vertex *v*, denoted by dis(*u*,*v*), is the length of any of the shortest paths from *u* to *v*. For example, in the undirected graph above, dis(1,4) = 2 (there is one shortest path 1,2,4, of length 2) and dis(1,1) = 0 (the shortest path is 1, of length 0).

The following algorithm, called *breadth-first search*, finds one of the shortest paths from a given vertex *s*, called the *source*, to every other vertex *v* of a simple graph, whether directed or undirected, or else indicates that there is no path from *s* to *v*. In any path-finding graph algorithm, whenever you are visiting a vertex *u*, having found a path from *s* to *u*, and see one of its neighbours *v*, you mark *v* with *u*, the *predecessor* of *v* in a path from *s* to *v*, so that later you can follow the marks you made from any vertex that has a mark all the way back to *s*. This is sort of like Hansel and Gretel scattering bread crumbs to find their way back home after their idiotic mother sent them into the woods to gather berries as punishment for having broken a milk jug. Unfortunately the bread crumbs were eaten up by birds and the poor kids got lost and then got captured by a witch. Fortunately there are no birds in the computer to eat the marks you make, only bugs in your program and those you can swat.

In any path-finding algorithm, while you're visiting some vertex *u*, you mark, and later visit, all the unmarked neighbours of *u*, and by doing this with every vertex you visit, starting with *s*, you will eventually mark, and then visit, every vertex *v* such that there is some path from *s* to *v*. What distinguishes breadth-first search from other path-finding algorithms is that you don't go blindly off to visit the first unmarked neighbour of *u* that you see. Instead, you keep a list of vertices to visit, you add to the end of the list all the neighbours of *u* that you haven't already marked, you mark them all at once, and you always visit the first vertex on your list and delete it from the list. A data structure that does this is called a *queue*, in which elements are added to the back of the queue and deleted from the front – first come, first served, like any real-life queue in which you can't bribe your way to the front. In this way you make sure that you visit the vertices in increasing order of distance from *s* so that that path you find from *s* to any vertex *v* will always be one of the shortest ones. Here is the algorithm.

```
procedure BFS(G: n-vertex graph; s: vertex; P: array[1..n] of vertices);
   local variables: u,v: vertices; Q: queue of vertices;
   for every vertex v in G do P[v]←0 end for;
   P[s]←s;        {Mark s so that it will not be seen as an unmarked neighbour of another vertex.}
   Q←the empty queue; add s to the end of Q;
   while Q is not empty do
      copy the first element of Q into u and delete it from Q;
      for every neighbour v of u do
         if P[v] = 0 then                                  {This is the first time v has been seen.}
            add v to the end of Q;
            P[v]←u;
         end if;
      end for;
   end while;
end BFS.
```

We trace this algorithm on the undirected graph shown below with s=1. The thick lines are the edges of the paths found by this algorithm.



		i=1	2	3	4	5	6	7	8	9	10	Q
	u	P[i]	1	0	0	0	0	0	0	0	0	1
	1		1	1	1	1	0	0	0	0	0	2 3 4
	2		1	1	1	1	2	2	0	0	0	3 4 5 6
10	3		1	1	1	1	2	2	3	0	0	4 5 6 7
	4		1	1	1	1	2	2	3	0	0	5 6 7
	5		1	1	1	1	2	2	3	5	0	6 7 8
	6		1	1	1	1	2	2	3	5	6	7 8 9
	7		1	1	1	1	2	2	3	5	6	8 9
	8		1	1	1	1	2	2	3	5	6	9
	9		1	1	1	1	2	2	3	5	6	

After this algorithm has been executed, $P[10]$ is still 0 and there is no path from 1 to 10. For every other vertex v there is a path from 1 to v and $P[v] \neq 0$. To get the path from s to v you follow the predecessors from v to s and then trace backwards the list of vertices you encountered. For example, $P[9] = 6$, $P[6] = 2$, $P[2] = 1$ and the path from 1 to 9 is 1,2,6,9.

You may be surprised to learn that I am going to prove that this algorithm works without using a loop invariant. A loop invariant is a proposition that you prove by induction on the number of iterations of the loop. For this algorithm the most convenient number on which to use induction is not the number of iterations of the loop but the distance of a vertex from the source. The proposition to be proved is that the vertices will be added to, and therefore deleted from, Q in increasing order of distance from the source. More precisely, **during the time that the vertices of distance *d* from *s* are being visited, all the vertices of distance *d*+1 and only those vertices will be marked.**

Basic step: $d = 0$. The only vertex u such that $\text{dis}(s,u) = 0$ is $u = s$. When $u = s$, all the neighbours v of u are unmarked – that is, $P[v] = 0$. They will get added to Q and $P[v]$ will get set to u. These are just the vertices v such that $\text{dis}(s,v) = 1$, so that the statement is true for $d = 0$.

Induction step. Suppose that $d > 0$ and that the statement holds for every number $< d$. Then after the vertices of distance 1 are marked (in the basic step), they will be visited, during which time the vertices of distance 2 will be marked (their predecessors will be of distance 1). Then, while they are being visited, the vertices of distance 3 will be marked (their predecessors will be of distance 2) and so on until, while the vertices of distance d - 1 are being visited the vertices of distance d will be given predecessors of distance d - 1, and Q will contain just these vertices.

Now the vertices of distance d will be visited. Let u be such a vertex. All its neighbours are of distance at most $d + 1$ from the source. Any neighbour v of u of distance less than $d + 1$ will have already been marked while the vertices of distance less than d were being visited; so it will be ignored. This shows that the only vertices that **can** be marked while the vertices of distance d are being visited must be of distance $d + 1$. Now we show that all these vertices **will** be marked. Let v be a vertex of distance $d + 1$ from s. Then the second-last vertex on one of the shortest paths from s to v is a vertex u of distance d from s, and v is a neighbour of u. While u is being visited, v will be seen as a neighbour of u. If v has already been marked, then its predecessor must be of distance d by what has already been proved. If not, then v will be given the predecessor u, which is of distance d. This completes the induction.

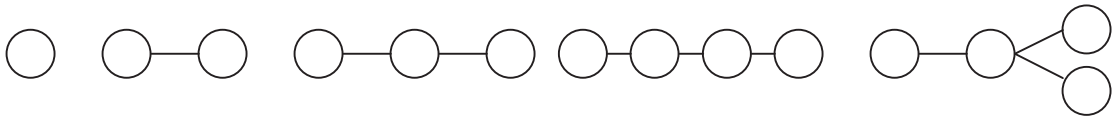
Terminal step (yes, we need one even if we're not using a loop invariant). Since any vertex added to Q will be given a non-zero predecessor and only vertices with predecessor zero get added to Q, every vertex will get added to Q at most once. Therefore, every vertex gets deleted from Q at most once; so there are at most n iterations of the outer loop. For each of these, there are at most n iterations of the inner loop because a vertex cannot have more than n neighbours. Each iteration of the outer loop gets completed in finite time, and after a finite number of these, Q must be empty and the algorithm terminates with $P[v] = 0$ for just those vertices v such that no path exists from s to v. For every vertex v of distance d from s, $P[v]$ is of distance d - 1, $P[P[v]]$ is of distance d - 2 and so on, so that the number of edges (or arcs) traversed backwards in following the predessors from v to s must be d. This shows that the path from s to v consisting of the same sequence of vertices traced backwards is of length d, the length of a shortest path from s to v; so it is one of the shortest paths from s to v. The algorithm works, QED.

During the terminal step we showed that there are at most n^2 iterations of the inner loop, so that the time-complexity of this algorithm is in $O(n^2)$. If an adjacency matrix is used to represent the graph, then the inner loop will be iterated n^2 times if all the vertices are accessible from s because you have to traverse the whole uth row of the matrix to find all of u's neighbours. On the other hand, if you use adjacency lists, the neighbours of u can be found by traversing u's adjacency list, so that the total number of iterations of the inner loop is the sum of the lengths of the adjacency lists, which is the number of arcs in a directed graph or twice the number of edges in an undirected graph, so that the time-complexity is $O(n+m)$. Adjacency lists make this algorithm more efficient than an adjacency matrix for sparse graphs.

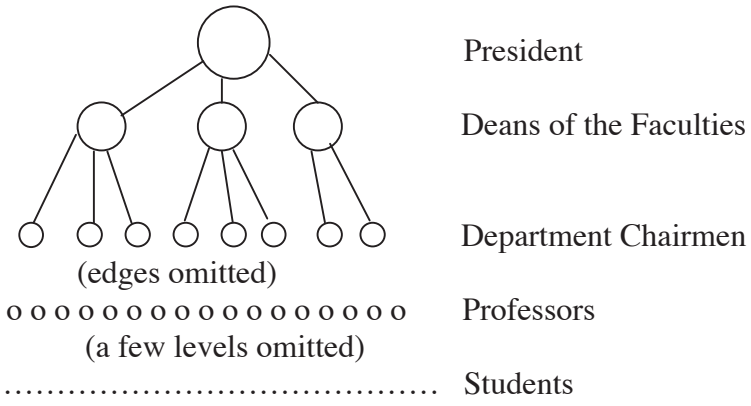
This algorithm has a lot of applications.

An undirected graph is called *connected* if for any two vertices u and v there is a path from u to v. For example, the graph drawn above is not connected because there is no path from vertex 1 to vertex 10, but if you delete vertex 10, then the graph becomes connected. Given a graph G, pick any vertex and call it s and run BFS(G,s,P). If $P[v] = 0$ for some vertex v, then G is not connected because there is no path from s to v. Suppose that $P[v] \neq 0$ for every vertex v in G. Then there is a path from s to v for every vertex v in G. Does this mean that there is a path from any vertex u to any vertex v? There is a path from s to u, and since the graph is undirected, you can follow that path backwards to get from u to s. Then you continue along some path from s to v and you find a path from u to v; so G is connected. This algorithm decides in time $O(n+m)$ whether or not an undirected graph with n vertices and m edges is connected.

A connected graph with no cycles is called a *tree*. The smallest trees are drawn below.

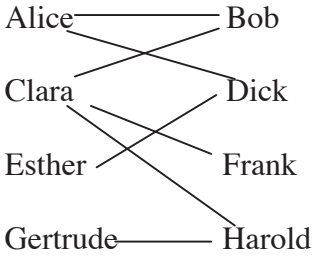


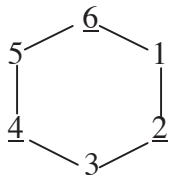
If you distinguish a vertex of a tree and call it the *root*, then you get a *rooted tree*, which computer types just call a tree. You have seen a rooted tree in the previous section – a binary tree – for representing a binary heap. It models a hierarchy, in that case a hierarchy of prisoners, each of whom have at most two slaves and each of whom have one boss except the best fighter, the root of the tree. If we remove the restriction on the number of slaves each person can have, a (rooted) tree models a more general hierarchy. The tree below models a hierarchy between the members (aside from the staff) of a very small university.

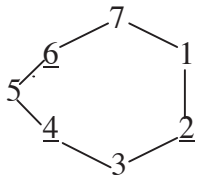


A graph H is called a *subgraph* of the undirected graph G if all the vertices of H are vertices of G and all the edges of H are edges of G that join two vertices of H. In the graph above (the one on which I traced BFS), the vertices from 1 to 9 and the thick edges form a subgraph that is a tree. If we delete the vertex 10 from the graph, so that it is connected, then the vertex set of the tree is equal to the vertex set of the graph. In this case, the tree is a *spanning tree* of the graph. The algorithm BFS(G,s,P) constructs a spanning tree, rooted at s, of any connected simple graph G in $O(n+m)$ time. The edges of this tree are the edges $\{v,P(v)\}$ for every vertex v except s. There are n - 1 such vertices; so the spanning tree will have n - 1 edges. If G is a tree, then the spanning tree constructed by this algorithm will be G itself. This shows that any tree with n vertices must have n - 1 edges. There are easier ways to prove that assertion, but since we already have the algorithm, we may as well use it.

A simple undirected graph G is called *bipartite* if its set V of vertices can be divided into two disjoint subsets V_1 and V_2 so that all the edges of G are incident with a vertex of V_1 and a vertex of V_2. The graph on the left below represents a set of young people at a party, where two people are connected by an edge if they – well, it would be indiscrete of me to spell out the details, except to assure the homophobic reader that the graph is indeed bipartite.







A cycle of even length is bipartite: if its vertices are labeled 1,2,…,n so that the edges are {1,2}, {2,3}, … , {n-1,n}, {n,1}, then all the vertices with an odd label can be put into V_1 and all the vertices with an even label can be put into V_2 and all the edges will connect a vertex in V_1 with a vertex in V_2 (see the graph in the middle above, where the even numbers are underlined). But a cycle of odd length is not bipartite. With the same labeling scheme, suppose that we put vertex 1 into V_1. Then for that graph to be bipartite, vertex 2 has to go into V_2 because it is adjacent to vertex 1, vertex 3 into V_1 because it is adjacent to vertex 2, and so on, putting all the odd vertices into V_1 **including vertex *n*** and all the even vertices into V_2. But now vertex 1 is adjacent to vertex n and they are both in V_1. And it wouldn't do us any good to put vertex 1 into V_2 instead because then vertex n, which is adjacent to vertex 1, would also be put into V_2. A graph that has an odd cycle as a subgraph cannot be bipartite, because if you can't divide the set of vertices of the odd cycle into two subsets so that no two vertices in the same subset are adjacent, then adding more vertices and edges isn't going to help you.

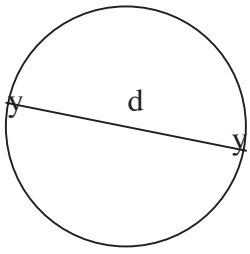
The algorithm BFS can be used to decide whether a simple undirected graph is bipartite. Pick any vertex s of the graph and execute BFS(G,s,P). If the graph isn't connected, some of the vertices won't be given non-zero predecessors. Pick one of them as s and apply BFS again, and so on until all the vertices have non-zero predecessors. Although there could be several applications of BFS, the total time complexity is still in $O(n+m)$. The number of iterations of the outer loop is equal to the number of vertices accessible by a path from the source; by applying BFS with several sources we do an iteration for each vertex accessible from one of the sources, and there are n of them.

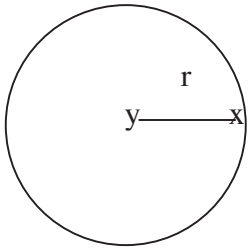
Now from each vertex v, follow the predecessors until we get to a vertex s such that $P[s] = s$ (the source for that vertex v), count the number of times we had to go from one vertex to another, which is the length of the path from s to v, and set $L[v]$ equal to this number. This takes $O(n)$ time for each vertex, because a shortest path is of necessity elementary and an elementary path in an n-vertex graph can't contain more than n - 1 edges; so it takes $O(n^2)$ time altogether. Now check all the edges of G to see whether there is one that joins two vertices u and v such that either $L[u]$ and $L[v]$ are both even or else $L[u]$ and $L[v]$ are both odd - this takes $O(n+m)$ time. If not, then G is bipartite, because the vertices v with odd $L[v]$ can be put into V_1 and the vertices v with even $L[v]$ can be put into V_2 and no edge will join two vertices in the same subset. But

suppose that there is an edge $\{u,v\}$ such that $L[u]$ and $L[v]$ are either both even or both odd. Since u and v are adjacent, the source s from which you found a path to u is the same one from which you found a path to v, and either both of these paths are of odd length or else both of these paths are of even length. Start from u. Follow the path from s to u backwards to get to s. Then follow the path from s to v (so far you have traversed an even number of edges). Finally, follow the edge $\{u,v\}$ to get back to u. You have found an odd cycle in G. This cycle may not be elementary, but the process of turning a cycle into an elementary cycle removes elementary cycles, and if the total length of the elementary cycles removed and the remaining elementary cycle is odd, then one of these elementary cycles must be of odd length; so G is not bipartite. And so, with the help of BFS, we have proved that **a simple undirected graph is bipartite if and only if it contains no cycles of odd length**.

The algorithm BSF decides in time $O(n^2)$ whether a graph with n vertices is bipartite, and it can be modified to work in $O(n+m)$ without having to fool with the paths afterwards. Instead of keeping an array of predecessors, keep an array P of subset labels, initialized to 0 everywhere. For each source s, set $P[s] = 1$. For each neighbour v of a given vertex u the algorithm is visiting, if $P[v] = 0$, then set $P[v] = 1$ if $P[u] = 2$ or set $P[v] = 2$ if $P[u] = 1$ and add v to the end of Q. If $P[v] \neq 0$, then check whether $P[v] = P[u]$ and, if so, declare G non-bipartite and stop execution. If the algorithm terminates without declaring G non-bipartite, then declare G bipartite. You write a pseudocode for this algorithm and test it on a few graphs, not all of them connected.

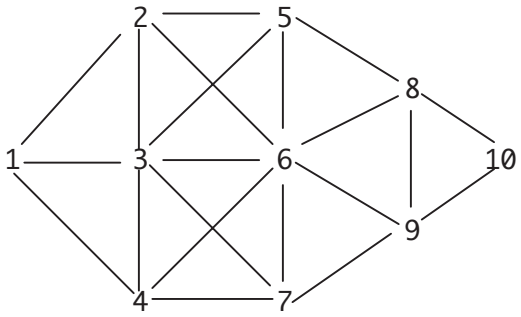
Suppose that two boys who are afraid of each other are confined to the interior or the circumference of a circle. To get as far apart as possible, they will both stand on the circumference of the circle at antipodal points and the distance between them will be the diameter of the circle (see the diagram on the left below, where each boy is represented by a letter y). Confine them instead to the vertices of a connected undirected graph and they will go to a pair of vertices that maximize the distance between them. By analogy with a circle, the maximum distance between any two vertices of a connected undirected graph is called the *diameter* of the graph. Now suppose that one of the boys is replaced by a girl whom the other boy loves but who despises him because he is a coward. He will soon discover that wherever he goes, she will go as far away from him as possible; so he will go to the spot that minimizes the maximum distance that she can put between them. In the circle, he will go to the centre of the circle, she will go to some point of the circumference and the distance between them will be the radius of the circle (see the diagram on the right below, where the girl is represented by the letter x). On the graph, he will go to one of the vertices that minimize the maximum distance between it and any other vertex of the graph and she will then go to one of the vertices that is as far from him as possible. By analogy with a circle, the *radius* of a connected undirected graph G is defined as $\min_{x\in G} \max_{y\in G} dis(x,y)$ and the *centres* of G are the vertices x that realize this minimax.





Once the algorithm BFS(G,s,P) has been executed on the connected undirected graph G for some vertex s, the distance from s to each vertex v in G can be calculated by following the predecessors from v to s, but BFS can be modified to avoid having to fool with the paths. Instead of calculating a table P of predecessors we calculate a table P of distances from s to each of the vertices. We initialize $P[s]$ to 0 because we know that dis(s,s) = 0, and for every other vertex v we initailize $P[v]$ to infinity, indicating that we haven't yet found a path from s to v. The symbol for infinity is ∞, but you can't type this symbol by twisting the key for 8 by 90 degrees. Normally you represent infinity by the biggest integer that can be represented on the computer, but it suffices to represent it by n, the number of vertices of G, because a shortest path must be elementary and no elementary path can have as many as n edges. Now, for each neighbour v of u, if $P[v] = n$, you add v to Q and you set $P[v] = P[u]+1$. When the algorithm terminates, $P[v]$ will be the equal to dis(s,v) for each vertex v in G unless $P[v] = n$ for some v, in which case G is not connected. For each vertex s in G, run this modified version BFS(G,s,P) and copy P into the sth row of an $n \times n$ matrix D. Then for each ordered pair (i,j) of vertices of G, $D[i,j]$ will be equal to dis(i,j). This process takes $O(n^3)$ operations on a dense graph and can be made to run in $O(n(n+m))$ on a sparse graph by representing it using adjacency lists.

Once the matrix D has been calculated, the maximum element in each row can be calculated in $O(n)$ operations, for a total of $O(n^2)$, and then, in another $O(n)$ operations, the maximum of these maxima, the minimum of these maxima and the vertices that realize this minimum can be calculated. The time-complexity estimate of computing the diameter, radius and centres of a graph is dominated by the cost of executing BFS n times. The space-complexity is $O(n^2)$, the size of D, but it can be reduced to $O(n+m)$, the size of the adjacency-list representation of G, by calculating the maximum of the elements of P without copying them into a matrix (write a pseudocode for this version of the algorithm). In the diagram below, the graph is shown on the left, the matrix of distances in the centre and the column vector of row maxima on the right. From this column vector it can be seen that the diameter of the graph is 4, the radius is 2 and the centres are the vertices 5, 6 and 7. The two boys who fear each other will go to vertices 1 and 10 and be 4 units apart. When the girl replaces one of the boys, the other one will go to one of the vertices 5, 6 or 7 to minimize the maximum distance she can put between them and she will go to one of the vertices of maximum distance 2 from him.




```
0 1 1 1 2 2 2 3 3 4       4
1 0 1 2 1 1 2 2 2 3       3
1 1 0 1 1 1 1 2 2 3       3
1 2 1 0 2 1 1 2 2 3       3
2 1 1 2 0 1 2 1 2 2       2
2 1 1 1 1 0 1 1 1 2       2
2 2 1 1 2 1 0 2 1 2       2
3 2 2 2 1 1 2 0 1 1       3
3 2 2 2 2 1 1 1 0 1       3
4 3 3 3 2 2 2 1 1 0       4
```

8.2 Flows in networks

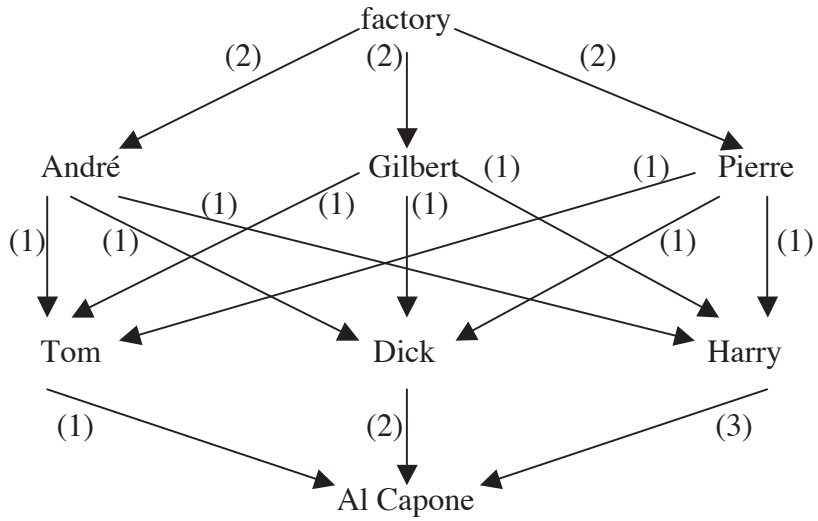
A *network* is a simple directed graph with two distinguished vertices s, the *source*, with in-degree 0, and t, the *sink*, with out-degree 0, together with a function c from the set of arcs into the set of non-negative real numbers. The number $c(u,v)$ is called the *capacity* of the arc(u,v). A *flow* in a network N is a function f from the set of arcs into the set of non-negative real numbers such that:

1. For every arc (u,v), $f(u,v) \leq c(u,v)$ (the flow in an arc cannot exceed the capacity of the arc), and
2. For every vertex v except s and t, the sum of the flows in all the arcs entering v is equal to the sum of the flows in all the arcs exiting from v (no vertex except the source and the sink can produce or consume).

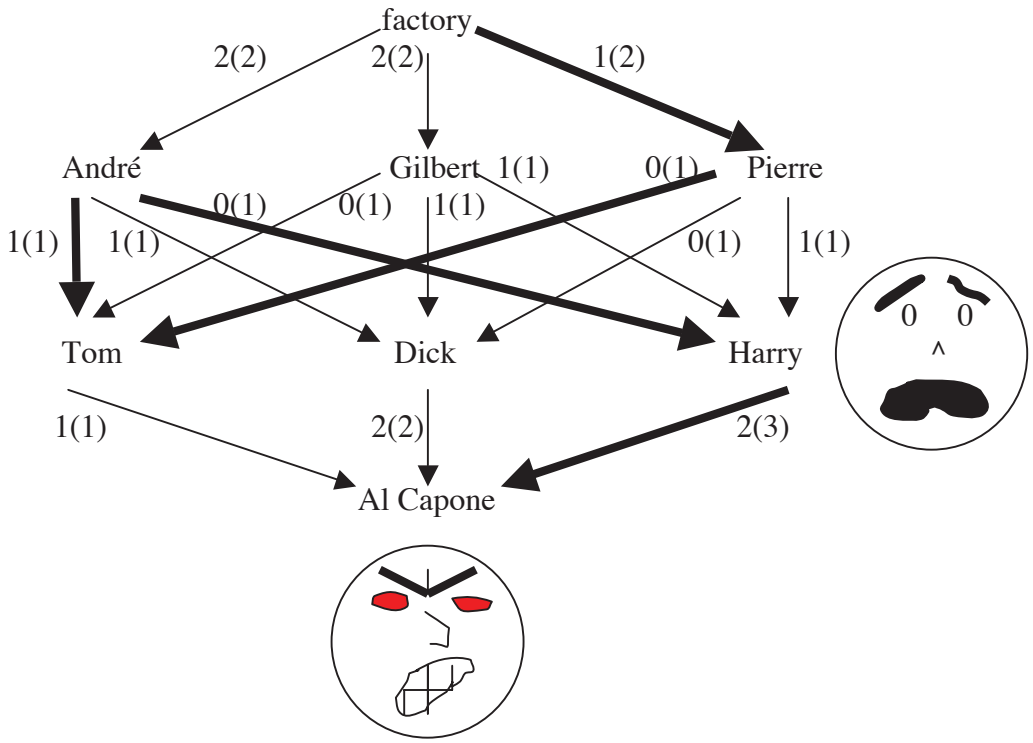
Since the set of vertices of N neither produces nor consumes, the sink consumes exactly what the source produces, so that the sum of flows in all the arcs exiting from the source is equal to the sum of the flows in all the arcs entering the sink; this number is called the *value* of the flow. The problem is to find a flow of maximum value.

The following model illustrates this problem. In the prehistoric days of the 1930s, before even I was born, let alone you, the United States experimented with the prohibition of alcohol. The experiment turned out to be a disaster, at least in part because of the 4000-mile-long undefended border between the United States and Canada, which did not institute such a futile measure. As you can imagine, it became extremely profitable to transport alcohol across this border, and a great many networks of bootleggers were established for this purpose.

A typical such network is the one shown below. The source is a factory in Quebec that makes booze of some sort. Among the men who work in this factory are three thieves called André, Gilbert and Pierre, who used to be mathematics professors until the lean, mean times forced them to find work beneath their qualifications, like many mathematicians even today. Each of them can steal 2 barrels of booze per day, as indicated by the capacity of the arc from the source to each of the three thieves (the numbers in parentheses). The sink is the chief of the network, the American gangster Al Capone. He has 3 fences working for him - Tom, Dick and Harry - who can buy (respectively) 1, 2, and 3 barrels of stolen booze a day and resell it to their boss. Between each thief and each fence there is a human mule who can transport 1 barrel a day across the border.

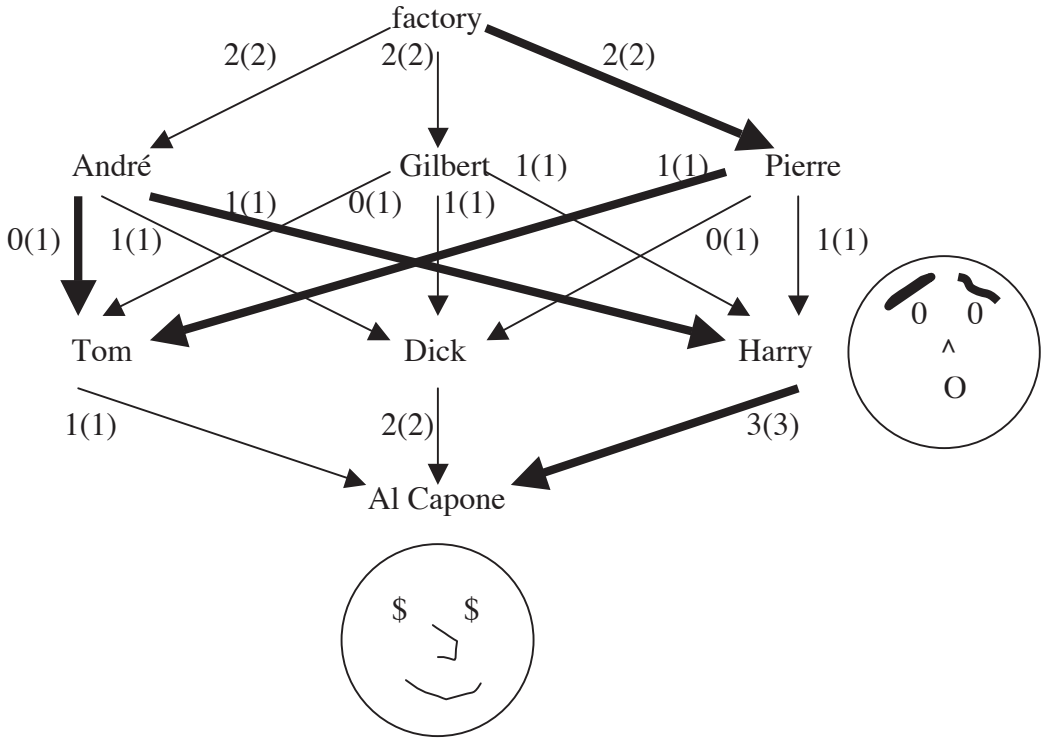


On the first day of operations, Al Capone orders his fences to establish a flow of maximum value. They, in turn, send messages to the thieves, asking them to steal as much as they can and sell it to them. André steals his 2 barrels, sells the first one to Tom via their mutual mule and the second one to Dick. Then Gilbert steals his 2 barrels, sells the first one to Dick and the second one to Harry. Finally, Pierre steals 1 barrel and sells it to Harry. The network, with the flow indicated by the numbers outside the parentheses appears in the diagram below (the thickened arcs will be explained below).



Harry has good reason to be frightened. He has delivered only 2 barrels instead of the 3 he can afford to buy, and his boss is mad enough to kill. But there appears to be no way for Pierre to get his second barrel to Harry. The mule from Pierre to Harry is *saturated* (the flow in this arc is equal to the capacity of the arc). The arc from Pierre to Tom and the arc from Pierre to Dick are not saturated, but the arc from Tom to Al Capone and the arc from Dick to Al Capone are saturated. There is no way to get the sixth barrel of booze from the factory to Al Capone by traversing the arcs along the direction indicated by the arrows.

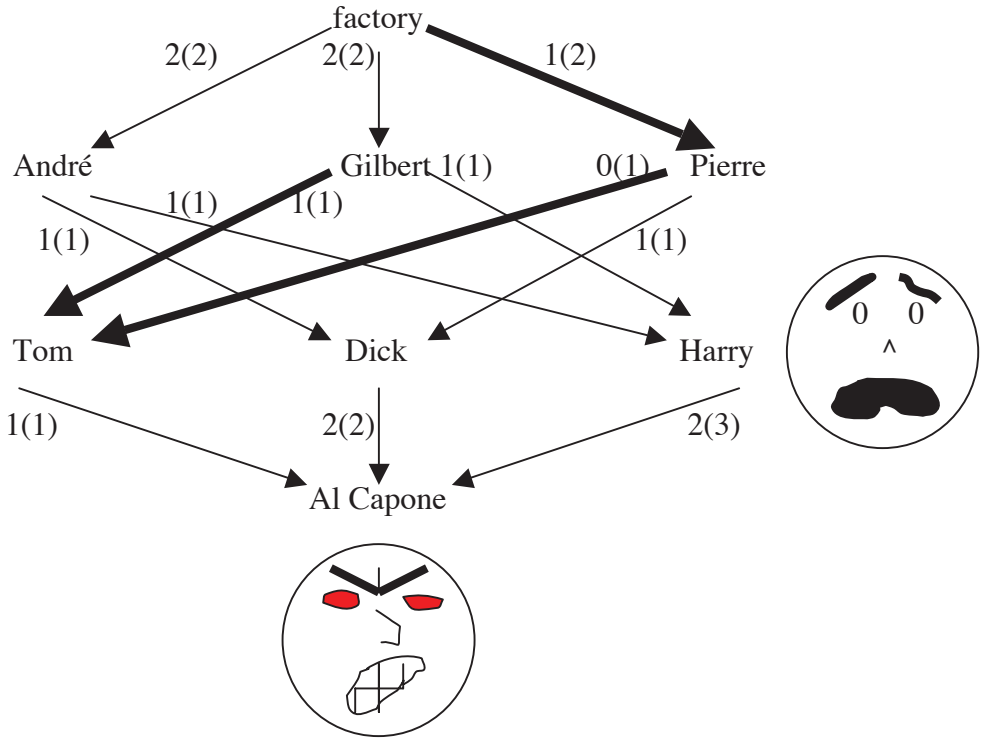
But Pierre, the greatest of the three mathematicians, figures out a way to augment the flow in the network. He steals his second barrel and sends it to Tom via their mutual mule, who is not saturated. Tom cannot buy that barrel (the arc from Tom to Al Capone is saturated) but, on Pierre's instructions he sends the barrel to André via **their** mutual mule. That arc has flow 1; transporting the barrel **in the opposite direction from the arrow** reduces the flow in the arc from 1 to 0. André sends the barrel to Harry via their mutual mule, who is not saturated, and a much relieved Harry buys his third barrel and sells it to his boss, who is no longer angry because the network now has a flow of maximum value (see the diagram below), which will be repeated every day thenceforth without having to find it by trial and error.



The thickened arcs in the two diagrams above are the arcs in the path by which the last barrel could be sent from the source to the sink. Such a path is called an *augmenting path*. It's a path from the source to the sink that consists of arcs of two sorts: *direct arcs* that are not saturated so that the flow in them can be increased by sending some booze down the arc in the right direction according to the arrow, and *inverse arcs* with positive flow that can be decreased by sending some booze up the arc in the wrong direction (like the arc from André to Tom). The *residual capacity* of a direct arc (u,v) is equal to $c(u,v) - f(u,v)$, the amount by which the flow in

that arc can be increased, and the residual capacity of an inverse arc (u,v) is equal to $f(u,v)$, the amount by which the flow in that arc can be decreased. The residual capacity of an augmenting path from s to t is the minimum of the residual capacities of all the arcs, both direct and inverse, in the path, and this is the maximum amount by which the value of the flow can be augmented by sending booze down this path. In this example, the residual capacity in the augmenting path Factory, Pierre, Tom, André, Harry, Al Capone is 1, and sending 1 barrel down this path augments the flow from 5 to 6.

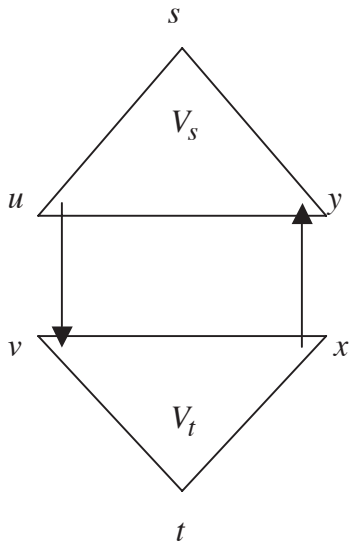
After some time disaster strikes. The cops make a raid and arrest all the mules. Each of them has stolen one bottle from the barrel he was transporting. Six of them give their bottles to the cops and get released, but the other three have already drunk the contents of their bottles; so they get flung into jail. Al Capone thinks that a flow of value 6 can be re-established because the thieves, the remaining mules and the fences all have a total capacity of 6, and he once again orders his fences to do so. André steals his 2 barrels, selling the first one to Dick and the second one to Harry. Gilbert steals his 2 barrels, selling the first one to Tom and the second one to Harry. Pierre steals 1 barrel and sells it to Dick. Once again, Al Capone demands his sixth barrel and once again Harry, having underfulfilled his quota, is feeling threatened (see the diagram below).



But this time Pierre cannot help Harry. He can't send a second barrel to Dick because their mutual mule is saturated. He could send it to Tom, but Tom couldn't buy that barrel because the mule from him to Al Capone is saturated. He could send it back to Pierre but that wouldn't do much good because Pierre could do nothing else with it except give it back to the factory from which he stole it. Tom could send the barrel to Gilbert, but Gilbert couldn't send it anywhere else except back to Tom or to the factory because the mule from Gilbert to Harry is already saturated. The set of vertices of the network accessible to this barrel via direct and

inverse arcs is {Factory, Pierre, Tom, Gilbert}. The set of vertices inaccessible to this barrel is {Dick, André, Harry, Al Capone}. A partition of the vertex set V of a network into 2 disjoint subsets, V_s containing the source and V_t containing the sink, is called a *cut*. The *capacity* of a cut is the sum of the capacities of all the arcs from a vertex in V_s to a vertex in V_t. The capacity of this cut is 5 (2 from the factory to André, 1 from Gilbert to Harry, 1 from Pierre to Dick and 1 from Tom to Al Capone); so no more than 5 barrels can be sent from the first set (containing the factory) to the second set (containing Al Capone). Since the flow already has value 5, it is at its maximum. Pierre explains the situation to Harry and advises him to explain it to Al Capone. But Al Capone is no mathematician, and to him a cut is something to be given to under-performing fences.

In general, **the value of any flow cannot exceed the capacity of any cut**. This is clear from the diagram below. The value of the flow is the amount produced by the source and consumed by the sink, which is the amount that gets transported from V_s to V_t, which, in turn, is equal to the sum of the flows in the arcs from V_s to V_t minus the sum of the flows in the arcs from V_t to V_s. The sum of the flows in the arcs from V_s to V_t cannot exceed the sum of the capacities of these arcs – the capacity of the cut – minus zero, and this bound achieved if all the arcs from V_s to V_t are saturated and all the arcs from V_t to V_s have flow 0.



Suppose you start with a flow of 0 in every arc and augment the flow until there is no augmenting path. Let V_s be the set of vertices accessible from s by a path consisting of unsaturated direct arcs and inverse arcs with positive flow and let V_t be the set of inaccessible vertices. Clearly $s \in V_s$ (you can reach s from s by a path of length 0) and $t \in V_t$ (otherwise there would be an augmenting path); so V_s and V_t constitute a cut. We show that every arc from V_s to V_t is saturated. If not, let (u,v) be an unsaturated arc from V_s to V_t (see the diagram above). Then, since $u \in V_s$, u is accessible, and since (u,v) is unsaturated, v must be accessible too, which contradicts the fact that $v \in V_t$. We show that every arc from V_t to V_s has flow 0. If not, let (x,y) be an arc from V_t to V_s with positive flow. Then, since $y \in V_s$, y is accessible, and since (x,y) has positive flow, it is an inverse arc and x is accessible too, which contradicts the fact that $x \in V_t$.

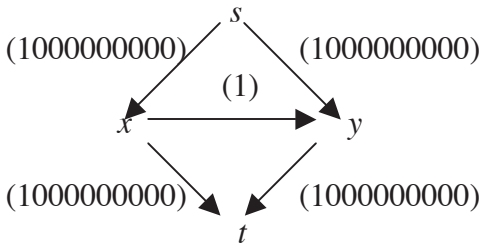
Since every arc from V_s to V_t is saturated and every arc from V_t to V_s has flow 0, the value of the flow is equal to the capacity of the cut, and since the value of any flow cannot exceed

the capacity of any cut, the flow has the maximum value of any flow in this network, the cut has the minimum capacity of any cut in this network, and **the maximum value of a flow in a network is equal to the minimum capacity of a cut in the network**.

This theorem, called the Ford-Fulkerson theorem after its authors [Ford, L. R.; Fulkerson, D. R. (1956). "Maximal flow through a network". Canadian Journal of Mathematics 8: 399–404], assumes that there is an algorithm that augments the flow until there is no augmenting path. Such an algorithm, not surprisingly, is called the Ford-Fulkerson algorithm.

```
procedure FF(N: network, f: array of real){Creates a maximum flow and a minimum cut in N.}
   local variables x, y: vertices;
   for every arc (x,y) in N do f(x,y)←0 end for;
   loop
      Search for an augmenting path from the source to the sink;
      if no such path is found then exit from the loop end if;          {Otherwise you found a path.}
      Augment the flow by pushing the residual capacity of the path you found down that path;
   end loop;                                                           {The flow now has maximum value.}
   Let VS be the set of vertices reached in the failed search for an augmenting path;
   Let VT be the set of vertices not reached;  {VS and VT constitute a cut of minimum capacity.}
end FF.
```

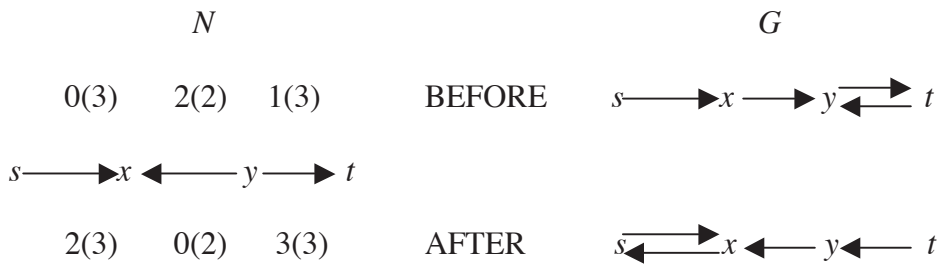
The correctness of this algorithm assumes that it is guaranteed to terminate for any network. If the capacities are arbitrary non-negative real numbers, then not only is this algorithm not guaranteed to terminate, but it could produce a succession of flows whose values approach a limit that is less than that of some valid flow. If the capacities are all integers, then the algorithm must eventually terminate because each augmentation increases the value of the flow by at least 1 and the cut separating the source from all the other vertices has finite capacity; so the theorem is established in this case, but this bound can be attained if the paths are chosen with systematic stupidity. Suppose that in the network shown below, the computer happens to choose alternately the paths *s*,*x*,*y*,*t* and *s*,*y*,*x*,*t*, of residual capacity 1. Then it will take two billion steps to reach the maximum flow, which could have been reached in two steps by choosing the shortest paths *s*,*x*,*t* and *s*,*y*,*t*.



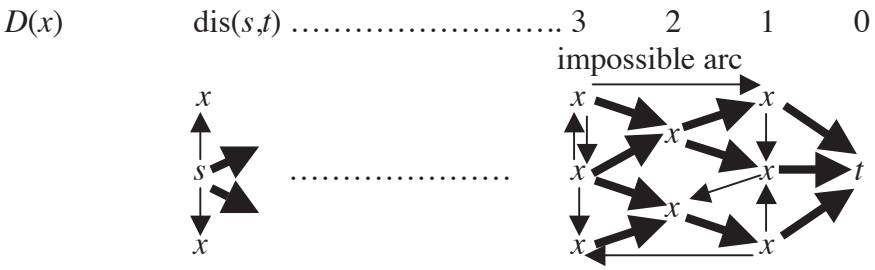
It was shown by J. Edmonds and R.M. Karp [Edmonds, Jack; Karp, Richard M. (1972). Theoretical improvements in algorithmic efficiency for network flow problems, Journal of the ACM (Association for Computing Machinery) 19 (2), 248–264], and independently by E. A. Dinic [E.A. Dinic, Algorithm for solution of a problem of maximum flow in a network with power estimation, *Soviet Math. Dokl.* 11 (5),1 277-1280,(1970). (English translation by R.F. Rinehart)], that **if a shortest augmenting path is systematically chosen, then the algorithm, executed on a network with *n* edges and *m* arcs, will terminate after at most *mn***

augmentations.

Given a network N with a given flow, define the associated directed graph G whose vertices are those of N and whose arcs are the direct arcs and the inverse arcs of N. Searching for an augmenting path in N is equivalent to searching for a path from s to t in G. When you push the residual capacity of the augmenting path down that path in N, then in the corresponding path in G, at least one arc will be destroyed and none will be created, although you may create some arcs in G that go in the opposite direction (see the diagram below).



Initially, when the flow is 0 in every arc of N, G is identical to N except that no capacities are assigned to the arcs of G. For every vertex v of G, let $D(v) = \text{dis}(v,t)$, the length of a shortest parth from v to the sink. We call an arc (x,y) *approaching* if $D(y) < D(x)$, in which case $D(y) = D(x) - 1$. A shortest path from s to t must consist entirely of approaching arcs. In the diagram below we show a graph with the approaching arcs thickened.



Suppose the Ford-Fulkerson algorithm is modified so that a shortest augmenting path is systematically chosen. Then the corresponding path in G will consist entirely of approaching arcs. When the flow is augmented by pushing the residual capacity of the augmenting path in N down this path, at least one approaching arc in G will be destroyed and none will be created (some non-approaching arcs may be created). It may be that $\text{dis}(x,t)$ has changed for some vertices x, but for the moment we don't redefine $D(x)$. Since we can't create any super-approaching arcs (arcs (x,y) such that $D(y) < D(x) - 1$), for each x, $\text{dis}(x,t)$ cannot decrease. In particular, $\text{dis}(s,t)$ cannot decrease. If $\text{dis}(s,t)$ remains the same, there will still be a path in G consisting entirely of approching arcs, and the algorithm will find one of these paths and destroy at least one of its arcs without creating any approaching arcs.

Since there are at most m approaching arcs, after at most m augmentations there will no longer be any path from s to t consisting entirely of approaching arcs; so the $\text{dis}(s,t)$ has become greater than $D(s)$. At this point, for each vertex x we redefine $D(x)$ to be $\text{dis}(x,t)$. Since $\text{dis}(s,t)$ has increased by at least 1, so has $D(s)$. Since the length of a shortest path in a graph with n vertices cannot exceed $n - 1$, after at most n reevaluations of the function D there will no longer

be any path from s to t in G. This implies that there is no longer any augmenting path in N; so the algorithm terminates, having created a flow of maximum value, after at most mn augmentations even if the capacities are arbitrary non-negative real numbers. Ford and Fulkerson, with some help from Edmonds, Karp and Dinic, got it right, QED.

The Edmonds-Karp algorithm is just the Ford-Fulkerson algorithm modified to search for the shortest augmenting path in N. The BFS algorithm finds a shortest path in $O(m+n)$ operations and pushing some flow down a path of length at most $n - 1$ takes $O(n)$ operations, for a total of $O(m+n)$ operations, which can be simplified to $O(m)$ because in a connected graph $m \geq n - 1$. Since the number of augmentations is in $O(mn)$, the total time-complexity of the Edmonds-Karp algorithm is in $O(m^2n)$, which is in $O(n^5)$ because m is in $O(n^2)$.

Dinic's algorithm creates explicitly the subgraph of G that consists of the vertices of G and the approaching arcs. Suppose you modify BFS to create an array D of distances from the source s to each vertex v, as we did for computing the diameter and radius of an undirected graph. In addition, you create a subgraph B, which has the same vertices as G but initially has no arcs. For each neighbour v of u, whether or not $D(v)$ was initially infinity (i.e. n), if $D(v)$ is now greater than $D(u)$, you add the arc (u,v) to B. When the algorithm terminates, B will consist of those arcs that recede from s. This is called a *directory* of arcs that can be part of some shortest path from s to some vertex. But you want to find the arcs that can be part of some shortest path from some vertex to t; so instead you apply this algorithm to the graph obtained from G by reversing the direction of all the arcs and you use t as the source rather than s. These modifications of BFS don't change its time-complexity, which is $O(m)$.

If the directory B contains s, then any path from s to t in B will of necessity be a shortest one. To find a path, instead of using breadth-first search, you use depth-first search [Robert Tarjan, Depth first search and linear graph algorithms, Siam J. Comput. Vol. 1, No. 2, 1972, 146-160]. If you are visiting u, you immediately visit the first unmarked neighbour v of u you see and set to u. To identify the first unmarked neighbour of u, you create an array F, where $F[x]$ is the current neighbour of x. Initially $F[x]$ is the first vertex in the ordered list of neighbours of x created during the construction of B. When you visit a vertex u, you start by setting v to $F[u]$. If v turns out to be marked, you reset $F[u]$ and v to the next neighbour of u, and so on until either you find an unmarked neighbour of u or $F[u]$ advances past the last neighbour of u and is set to 0, indicating that u is now dead. If you reach t, you have completed the search; following the predecessors back to s, you retrace the path twice, once to find the residual capacity and a second time to change the flows in the arcs of the path. When an arc (u,v) in B gets destroyed, and at least one of them will, you indicate this fact by advancing $F[u]$ to the next neighbour of u in the list. If you hit a vertex v with no unmarked neighbours ($F[v] = 0$), you retreat to $u = P[v]$ and advance $F[u]$ to the next neighbour of u after v. If instead of reaching t you kill s, then there are no paths in B from s to t and you create a new directory B. The algorithm terminates when the new directory B does not contain s because there is no path in G from s to t. At this point the flow is of maximum value.

How long does this algorithm take? Each new directory takes $O(m)$ operations to create and $O(m)$ operations to destroy (it has at most m arcs). For each directory there are at most m augmentations, since each one destroys at least one arc in the directory. Each operation in the path-finding algorithm either advances towards t or destroys an arc; so aside from destroying arcs there are $O(n)$ operations per augmentation, both for finding the path from s to t in the directory

and for changing the flow. For each directory, the total number of operations is dominated by the cost of the augmentations, since this number is in $O(mn)$ and the time-complexity of creating and destroying the directory is only in $O(m)$. A new directory will be created at most n times; so the total cost of the algorithm is in $O(mn^2)$, which is in $O(n^4)$.

This algorithm is simple to program (about 50 lines of code in Fortran), and it runs faster than the Edmonds-Karp algorithm. There are asymptotically faster ones, and they run faster on networks designed to force both the Edmonds-Karp algorithm and Dinic's algorithm to their worst case, but for randomly generated dense networks, Dinic's algorithm is at least as fast as any of the others.

For those interested in graph algorithms, there are a great many to choose from. I mention only a few of them here. The flow algorithm called "the Three Indians' algorithm" after the nationality of its authors [Malhotra, V and Kumar, MP and Maheshwari, SN (1978) An O(|V|3) algorithm for finding maximum flows in networks. Information Processing Letters, 7 (6). pp. 277-278] runs in $O(n^3)$ time on n-vertex graphs independently of their density. The Tarjan-Sleator algorithm [Sleator, D. D., and Tarjan, R. E. A data structure for dynamic trees. J. Comput. Syst. Sci. 26 (1983), 362-391] is more efficient than the Three Indians' Algorithm for sparse graphs. Dijkstra's algorithm for finding a shortest path from a source to each vertex in a graph with non-negatively weighed arcs [Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". Numerische Mathematik 1: 269–271] runs in $O(n^2)$ time on dense graphs and can be made to run faster on sparse graphs by using a very complicated data structure. Prim's algorithm for finding the shortest spanning tree in a connected undirected graph with weighted edges [R. C. Prim: Shortest connection networks and some generalizations. In: Bell System Technical Journal, 36 (1957), pp. 1389–1401] is similar to Dijkstra's algorithm and has the same time complexity, and Kruskal's algorithm for solving the same problem [Joseph. B. Kruskal: On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. In: Proceedings of the American Mathematical Society, Vol 7, No. 1 (Feb, 1956), pp. 48–50] is almost as efficient. The above-mentioned article by Edmonds and Karp finds the maximum flow with minimal cost in a network whose arcs have costs as well as capacities, but it only works when the capacities and the costs are integers and its time-complexity depends on the size of these integers. An algorithm that solves this problem for arbitrary non-negative real costs and capacities in a time polynomially bounded by n and m appears was later found by Eva Tardos [Tardos, E. (1985): A strongly polynomial minimum cost circulation algorithm. Combinatorica 5, 247-255].

Another application of BFS will be given in the next chapter.