Stable matching

- This problem gets used, *e.g.*, when assigning medicins to hospitals. Usually described in a more naturalistic way:
- Instance: N ladies, N gentlemen. Each person has a list of acceptability of persons of the other gender (this list is a permutation).
- Question: Find a stable matching with respect to the input. The matching is stable, if all persons are matched and there exists no pair ij such that the matching is aj, ib and on the list of j, person i preceeds a and on the list of i, person j preceeds b (*i.e.*, if we re-match this pair of pairs, both, i and j will be more satisfied.
- At the first sight, even unclear whether always such a matching can be found.
- Easy algorithm which always finishes, proof of partial-correctness is a bit harder.

Stable matching

- "Gentlemen, please, ask ladies for a dance!"
- Getlemen start asking ladies in the ordering given by their (individual) permutations.
- Each lady chooses among a current partner and all the newly coming gentlemen the best one (with respect to her permutation).
- The refused gentlemen continue asking the ladies in the ordering given by their (individual) permutations.
- Why the algorithm is finite?
- Because ladies are still getting better and better partners.
- Why does it find a stable matching?
- In fact we show even more...

Can we show something better?

Fact: *The algorithm finds a stable matching optimal for all gentlemen.*
It means: For any gentlemen there is no stable matching permitting him a better partner than the matching found by this algorithm.

Definition

A *sin* is a situation, when *a lady refuses a getleman,* who would be acceptable for her in *any* stable matching.

We show that our algorithm permits no "sin".

Lemma

Our algorithm permits no "sin".

Proof.

By contradiction: The algorithm does not permit the first sin. For contradiction let there is (a first) sin.

- Let *Eve* refused *Adam* acceptable in some stable matching. Now, she is matched to *Žibřid.*
- Let us take a look at the orderings.
- Is *Žibřid* better or worse than *Adam* for *Eve*!? [worse, thus our algorithm does not permit this behavior!]

□

Searching in Graphs

- Motivation: Mínótaurus in a Labyrinth is consuming citizens of Athens.
  Among them, prince Théseus appeared and killed Mínótaurus.
- Algoritmus:
  1. Find Mínótaurus,
  2. Kill Mínótaurus.
- We keep the latter part to him, we are interested in the former part.

Searching – depth first search

- Théseus obtained a thread from Ariadne,
- either he used a randomized algorithm, or he obtained a bucket with a color, as well.
- Algorithm (Interesting only for crossings, through the corridors we just pass):
  While we did not find Mínótaurus yet:
  - If there is a corridor not yet colored (containing no thread), color this corridor (its beginning and end) and pass through it.
  - Otherwise reel the thread in (return to the previous crossing).

Why the algorithm is correct? Finiteness? [trick with numbers on edges]
Partial correctness? Invariants? [a thread always denotes a sequence to Ariadne]
Each corridor gets passed through at most twice.
There is no reachable crossing (or even a corridor) which we do not reach before returning.

Different algorithm:

While we are not by Mínótaurus:

- If you see two corridors containing a thread, reel the thread in (return).
- Else, if there is an uncolored corridor, pass through it and color it.
- Else, reel the thread in [if it is possible].

Lemma

Even this algorithm is correct.

Although we return while we still can go further, because we know that we return at some moment (and then we go further). Moreover, this algorithm keeps a "return path to Ariadne", not just a "return sequence to Ariadne".

Remarks

- The former algorithm is called depth-first search. It passes as long as possible. When it is impossible, it starts returning (but only when necessary). The latter is search with return.
- While exploring graphs, we may use also a breadth-first search (also called the wave-algorithm):
- The wave-algorithm: Labyrinth gets explored with unbounded number of warriors who are spreading through the labyrinth like a flood. This algorithm finds the shortest path to Minotaurus (will be better described later).

How to notate an algorithm?

While notating an algorithm, we do:

- operate the variables (of different types) and constants,
- modify the values of variables,
- call subroutines,
- compare the content of individual variables,
- decide based on these comparisons,
- perform cycles,
- read the input, write the output.

Program in Pascal

Example:

```
program useless;
const   x=10;
        text='ten';
var a,b:integer;
    c:string;
begin
```

Program begins with the keyword `program`!
We divide individual statements (commands) by a semicolon!

A section of constants follows introduced by keyword `const.` We assign the constants: `constant = value` Next is the section defining variables (keyword `var`). We are defining integer variables `a` and `b`.

About the importance of indentation:

```
program useless; const x=10; text='ten'; var
a,b:integer; c:string;
begin write('Enter a number:  '); readln(a);
write('Enter yet another number:  '); readln(b);
writeln('Their sum is ',a+b); writeln(x,' is ',text);
end.
```

Variables and their types

Each variable has an underlying (data-)type. Possible (Pascal) types are

- **byte:** 0 .. 255 (integers),
- **integer:** $-32\,768$ .. $32\,768$,
- **longint:** -2^{31} .. 2^{31},
- **real:** -10^{38} .. 10^{38} (non-integers),
- **word:** 0 .. 65 535 (integers),
- **char:** character (one 8-bit ASCII-character),
- **string:** string [of characters] (text) with length at most 255 chars,
- **boolean:** true or false (having values `true` a `false`).

Arithmetic expressions:

- $+$ Addition,
- $-$ subtraction,
- $*$ multiplication,
- $/$ division (result is a real),
- brackets,
- `div` integral division (with a remainder),
- `mod` remainder (of a division).

Beware of the priority!
Beware, `div` and `mod` has a priority "between" addition and multiplication!

Beware of string-addition!

Example:
$(a + 5) * 17 + (b \text{ mod } c)$

Assignment expression: `:=`

Příklad: `x:= 2*y;`

Relational operators

- $<$ less than (e.g., $a < b$),
- $>$ greater than,
- $>=$ greater or equal,
- $<=$ less or equal,
- $<>$ not equal,
- $=$ equals (are two values the same?).

Conditions

using conditional (boolean) expressions

Syntax (and semantics):

- `if` condition `then` command;
- `if` condition `then` `begin` block of statements `end`;
- `if` condition `then` command `else` command;
  Careful! Before `else` we do *not* place a semicolon!
- `if` condition `then` `begin` block `end` `else` `begin` block `end`;

Příklad:

```
if temperature>25 then
        writeln('Let us go to a pub!');
```

Example

```
if temperature>25 then writeln('Let us go to a
pub!');
```

```
if temperature>25 then
begin
        writeln('Let''s go to a pub!');
end
else
begin
        writeln('Let''s stay at home!');
end;
```

Cycles

- `while` condition `do` command or block;
  Repeat, while `condition` is satisfied (fulfilled).
- `for i:=1 to 10 do` command or block;
  Repeat for each value of the variable starting by the former bound up to the latter one.
- `for i:=100 downto 1 do` command or block;
- `repeat` commands; `until condition;`
  Repeat while the `condition` is **unsatisfied!**

Example:

```
program binary;
var a:integer;
begin
        readln(a);
        while a > 0 do
        begin
                if a mod 2 = 1 then
                        write(1)
                else    write(0);
                a:=a div 2;
        end;
end.
```

Example improved

While programming, it is principal to think on it. Otherwise we tend to perform an **unnecessary computations!**

```
program binary;
var a:integer;
begin
        readln(a);
        while a > 0 do
        begin
                write(a mod 2);
                a:=a div 2;
        end;
end.
```

Example, factorization:

```
program factor;
var a,i:integer;
begin
        i:=2;
        readln(a);
        while i <= a do
        begin   if (a div i)*i = a then
                begin
                        write(i);
                        a:=a div i;
                end
                else    i:=i+1;
        end;
end.
```

Example, the factorization improved:

```
program factor;
var a,i:integer;    repeating:boolean;
begin   i:=2;   repeating:=false;
        readln(a);
        while i <= a do
        begin   if (a div i)*i = a then
                begin   if repeating then
                                write('*')
                        else   repeating:=true;
                        write(i);
                        a:=a div i;
                end  else  i:=i+1;
        end;
end.
```