Updates

- 22 XII 2016: first draft
- 30 XII 2016: updates and comments to `accept` and examples. Important updates are green.
- 09 I 2018: use of signals is allowed, **FAQ** added.
- 13 I 2018: updated FAQ; the deadline postponed till **22 I 2018, 23:59**; submission of the solution will be described tommorow. 14 I 2018
- 14 I 2018: task submission.
- 19 I 2018: FAQ update

Automaon

A finite automaton -- a simple state machine which accepts or rejects words. A word is any finite sequence of letters from an alphabet. An alphabet is any finite set. If **w** is a word , then by **|w|** we denote its length, and by **w[i]**, **0 <= i < |w|**, its i-th letter. Moreover, the symbol ε denotes an unique empty word, i.e. a word of length 0.

Formally, an automaton **A** is a tuple $\mathbf{(L, E, U, F, q_I, T)}$, where **L** is an alphabet, **E** is a set of existential states, **U** is a set of universal states, $\mathbf{Q \overset{def}{=} E \cup U}$ is the set of all states of **A**, $\mathbf{F \in Q}$ is the set of final states, $\mathbf{q_I}$ is the unique initial state, and $\mathbf{T: Q \times L \to P(Q)}$ is the transition function.

Intuitively, an automaton recognises a word in the following way: the automaton starts in the initial state and reads the word, letter by letter. When the automaton is in state **q** and sees a letter **a** then it ``chooses'' the next state from the set **T(q,a)** and proceeds. If **q** is universal then every such choice has to lead to acceptance, is **q** is existential then only one has to lead to acceptance. Acceptance, in this context, requires that the above process after reading the last letter of the word ends up in a final state.

More formally: the above process can be described as a tree. A run of the automaton **run** on word **w** of length **d** is a tree of height **d+1**. (For instance, a run on the epty word has unique node -- the root -- at the depth **0**.) The root **ε** of the run is labelled $\mathbf{run[\varepsilon] = q_I}$ and every node **u** in the tree satisfies the following: if node **u** is at depth **i**, is labelled **q** and

- $\mathbf{T(q, w[i]) = \varnothing}$ then **u** is a leaf;
- $\mathbf{T(q, w[i]) \neq \varnothing}$ and the state **q** is existential then **u** has exactly one child labelled with a state belonging to the set **T(q, w[i])**;
- $\mathbf{T(q, w[i]) \neq \varnothing}$ and **q** universal then node **u** has exactly one **p**-labelled child for every state $\mathbf{p \in T(q, w[i])}$.

We say that a run is accepting if the leafs' labels on depth **d** are final states and the labels of the remaining leafs are universal states. A word is accepted by an automaton if there is an accepting run. If there is non then the automaton rejects the word.

The acceptance of a word can also be described as a recursive function. An automaton accepts a word **w** if the following function `bool accept(string w, string r)`

```
// this function accepts the word @w continuing the process from the last node of the path @r
// belonging to some run of the automaton
bool accept(string w, string r):

  d := |r|-1;

  if (d ≥ |w|)
    //we have processed every letter in @w; accept if the last state is accepting
    return (r[d] ∈ F);

  if (r[d] ∈ E)
    // the last state is existential so accept if there
    // is an accepting continuation of the run @r
    return (∃q∈T(r[d],w[d]) accept(w,rq))

  // otherwise, the last state is universal (r[d] ∈ U)
  // and we have to accept every possible continuation
  return (∀q∈T(r[d],w[d]) accept(w,rq))
```

returns **true** when called as **accept(w,q_I)**.

Pozor! Variables **w,r** are words and **rq** is the conctatenation of the word **r** and the letter (also a state) **q**. Intuitively, **r** can be seen as a path in a run.

The task

We require three programs: **validator**, **run** and **tester**.

Proces **validator**

The **validator** is a server accepting, or rejecting, words. A word is accepted if it is accepted by specified automaton. The program **validator** begins by reading from the standard input the description of the automaton and then in an infinite loop waits for the words to verify. When a word is received he runs the program **run** which validates the word and **vaidator** waits for an another word or response from the program **run**. Afer receiving a message from one of the **run** processes **validator** forwords the message to the adequate **tester**.

When a **tester** send an unique stop word **!** the server stops , i.e. he does not accept new words, collects the responses from the **run** processes, forwards the answers, writes a report on **stdout**, and, finally, terminates.

The **validator** report consist in three lines describing the numbers of received queries, sent answers and accepted words:

```
Rcd: x\n
Snt: y\n
Acc: z\n
```

where **x,y,z** respectively are the numbers of received queries, sent answers and accepted words; and a sequence of summaries of the interactions with the programs **tester** from which **validator** received at least one query. A summary for a **tester** with PID **pid** consists in:

```
[PID: pid\n
Rcd: y\n
Acc: z\n]
```

wher `pid,y,z` respectively are: the process' pid, the number of messages received from this process and the number of acceped words sent by this process.

Process `run`

Program `run` receives from `validator` a word to verify and the description of the automaton. Then he begins the verification. When the verification stops, the process sends a message to the server and terminates.

Process `tester`

Queries in a form of words are sent by programs `tester`. A program `tester` in an infinite loop read words form the standard input stream and forwards them to the server. Every word is written in a single line and the line feed symbol `\n` does not belong to any of the words.

When the `tester` receives an answer from the server, he writes on the standard output stream the word he resecived answer for and the decision `A` if the word was accepted and `N` if not.

Ant `teste` terminates when the server terminates of when `tester` receives the `EOF` symbol, i.e. the end of file symbol. When the `tester` terminates it sends no new queries, waits for the remaining answers from the server, and writes on the standard outpu a report.

A Report of a `tester` consist of three lines

```
Snt: x\n
Rcd: y\n
Acc: z\n
```

wher `x,y,z` respectively are the numbers of: queries, received answers, and accepted words sent by this process.

Input and output of the processes

The exact description of the input and the automaton description can be found in Input. The description of reports is in Output.

Misc.

We require the solution to be written in the `C` language, and that the communication between `validator`, `tester`, and `run` is achieved using message queues. The internal communication between the `run` processes can be done using pipes.

The processes of the words verification and query answering should be done in a concurrent manner.

Input

Here we will describe the input. A sequence `[wyr]` denotes that the string `wyr` repeats a finite (greater than or equal to 0) number of times.

Program `validator`

Input of a `validator` consist in the automaton description:

```
N A Q U F\n
q\n
[q]\n
[q a r [p]\n]
```

where

- `N` is the number of lines of the input;
- `A` is the size of the alphabet: the alphabet is the set {a,...,x}, where **'x'-'a' = A-1**;
- `Q` is the number of states: the states are the set **{0,...,Q-1}**;
- `U` is the number of universal states: universal states = **{0, .., U-1}**, existential states = **{U, .., Q-1}**;
- `F` is the number of final states;
- `q,r,p` denotes some states;
- and `a` is a letter of the alphabet
  .

The first line gives the cardinalities of the sets describing the automaton. The second line gives the initial stare, the third line is the list of final states. The remaining lines, of form: `q a r [p]\n` , encode the transition function in the following way: if there is a line `q a` p_1 p_2 `...` p_k then $\mathbf{T(q,a) = \{p_1, p_2, ..., p_k\}}$. If for any pair **(q,a)** there is no line in the encoding then one assumes that **T(q,a) =** ∅.

One can assume that: **0 <= F,U <= Q < 100**, **0 <= A <= 'z'-'a'**.

Program `tester`

A `tester` on the standard input stream receives the following string.

```
[[a-z]\n]
[!\n]
[[a-z]\n]
```

where

- `a-z` are the alphabet letters;
- `!` is the unique terminate the server symbol.

One can assume that the length of the supplied words will not exceed the constant **MAXLEN=1000**.

Output

The `validator`'s output

The `validator` on the standard output writes the following string:

```
Rcd: x\n
Snt: y\n
Acc: z\n
```

```
[PID: pid\n
Rcd: p\n
Acc: q\n]
```

where

- `x` is the number of received queries;
- `y` is the number of sent answers;
- `z` is the number of accepted words;
- `pid` is the `pid` of a `tester` process;
- `p` is the number of queries sent by process of the PID `pid`;
- `q` is the number of the accepted words sent by process of the PID `pid`;

The `tester`'s output

Any program `tester` writes on the standard output a string of the following form:

```
PID: pid\n
[[a-z] A|N\n]
Snt: x\n
Rcd: y\n
Acc: z\n
```

where

- `pid` is the tester's `pid`;
- `a-z` are some letters;
- `x` is the number of sent queries;
- `y` is the number of received answers;
- and `z` is the number of accepted words.

Examples

Example 1

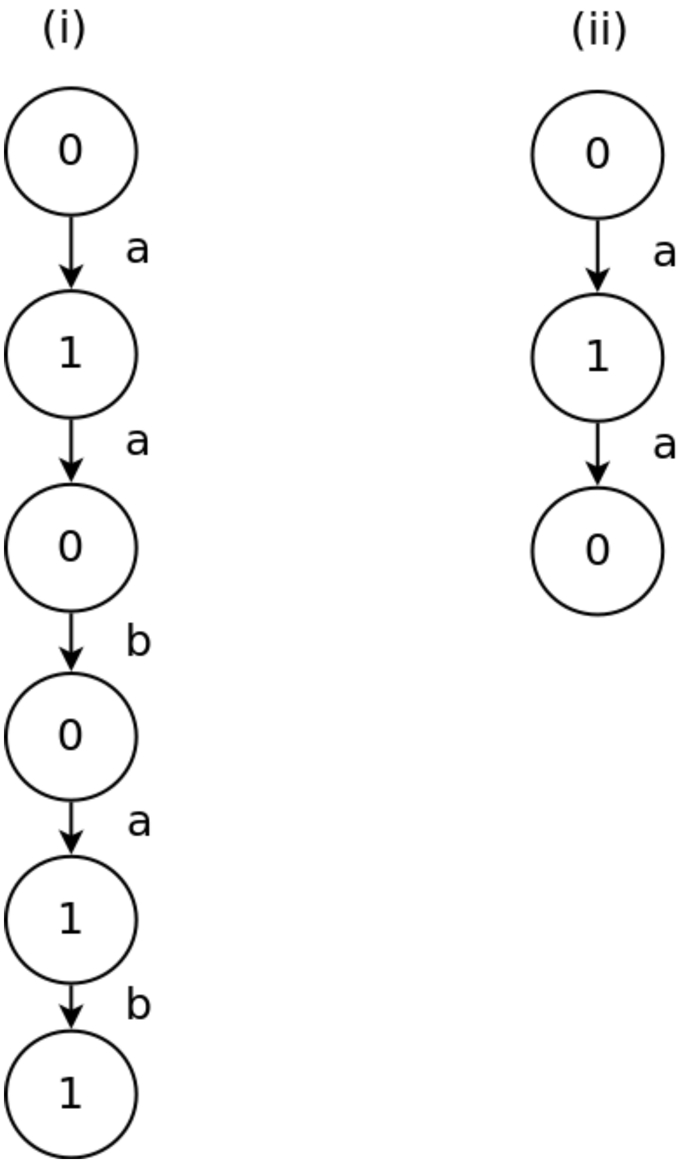
Accept words with an even number of letters `a`.

Input `validator`:
 `7 2 2 0 1`
`0`
`1`
`0 a 1`
`0 b 0`
`1 a 0`
`1 b 1`
	Input `tester`:
	 `a`
	`aa`
	`ab`
	`aabbaba`
	`!`

Output `validator`:
 `Rcd: 4`
`Snt: 4`
`Acc: 2`
`PID: pid`
	Output `tester`:
	 `PID: pid`
	`a A`
	`aa N`
	`aabbaba N`

`Rcd: 4`
`Acc: 2`
	`ab A`
	`Snt: 4`
	`Rcd: 4`
	`Acc: 2`

Runs of this automaton on words (i) `aabab` oraz (ii) `aa`.



Example 2

Accept words such that, after every letter `a` there is a letter `b`.

Input `validator`:	Input `tester`:

`7 2 2 1 1`
`0`
`0`
`0 a 0 1`
`0 b 0`
	`a`
	
	`ab`
	`aabbaba`
	`!`

```
1 a 1
1 b 0
```

Output `validator`:

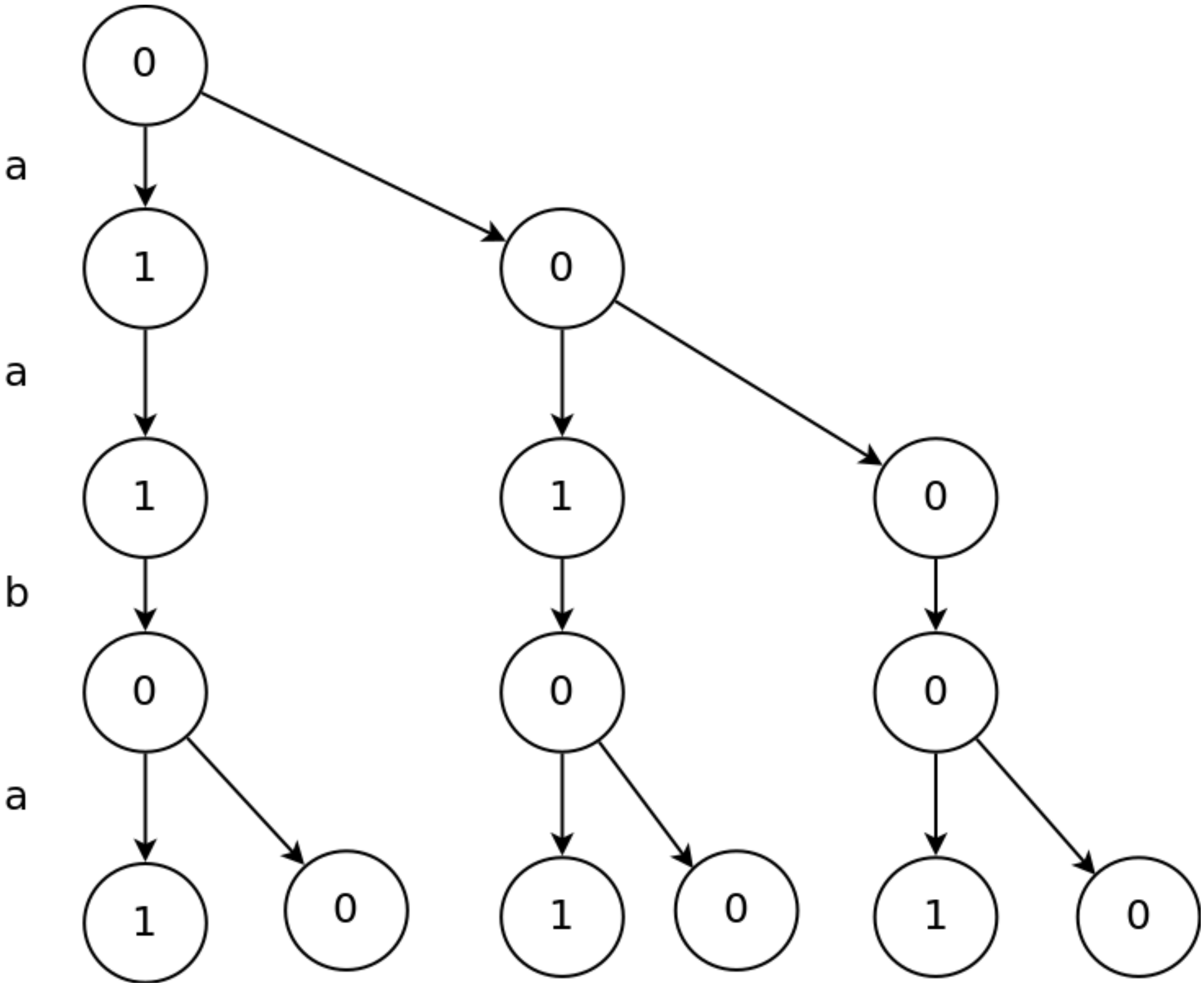
```
Rcd: 4
Snt: 4
Acc: 2
PID: pid
Rcd: 4
Acc: 2
```

Output `tester`:

```
PID: pid
 A
a N
aabbaba N
ab A
Snt: 4
Rcd: 4
Acc: 2
```

Runs

A run on the word `aaba`



Example 3

Accept words such that, after every letter `a` there is a letter `b`, different automaton.

Inputvalidator:

```
6 2 2 2 1
0
0
0 a 0 1
0 b 0
1 a 1
```

Input tester:

```
a

ab
aabbaba
!
```

Output validator:

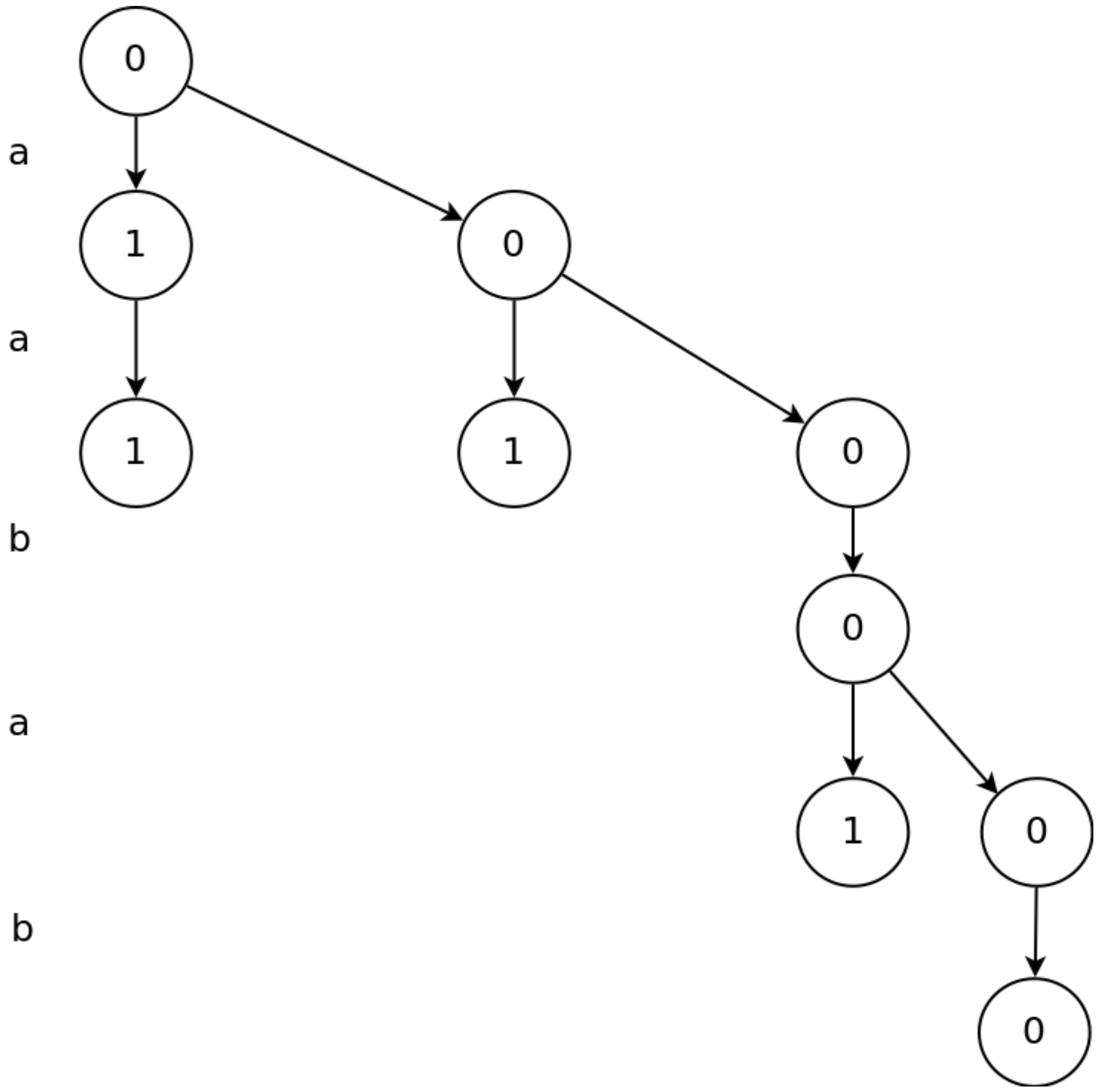
```
Rcd: 4
Snt: 4
Acc: 2
PID: pid
Rcd: 4
Acc: 2
```

Output tester:

```
PID: pid
 A
a N
aabbaba N
ab A
Snt: 4
Rcd: 4
Acc: 2
```

Runs

A run on the word `aabab`



Example 4

Accept words such that, first letter is `a` and (there is an even number of letters `a` or the last letter is `b`.)

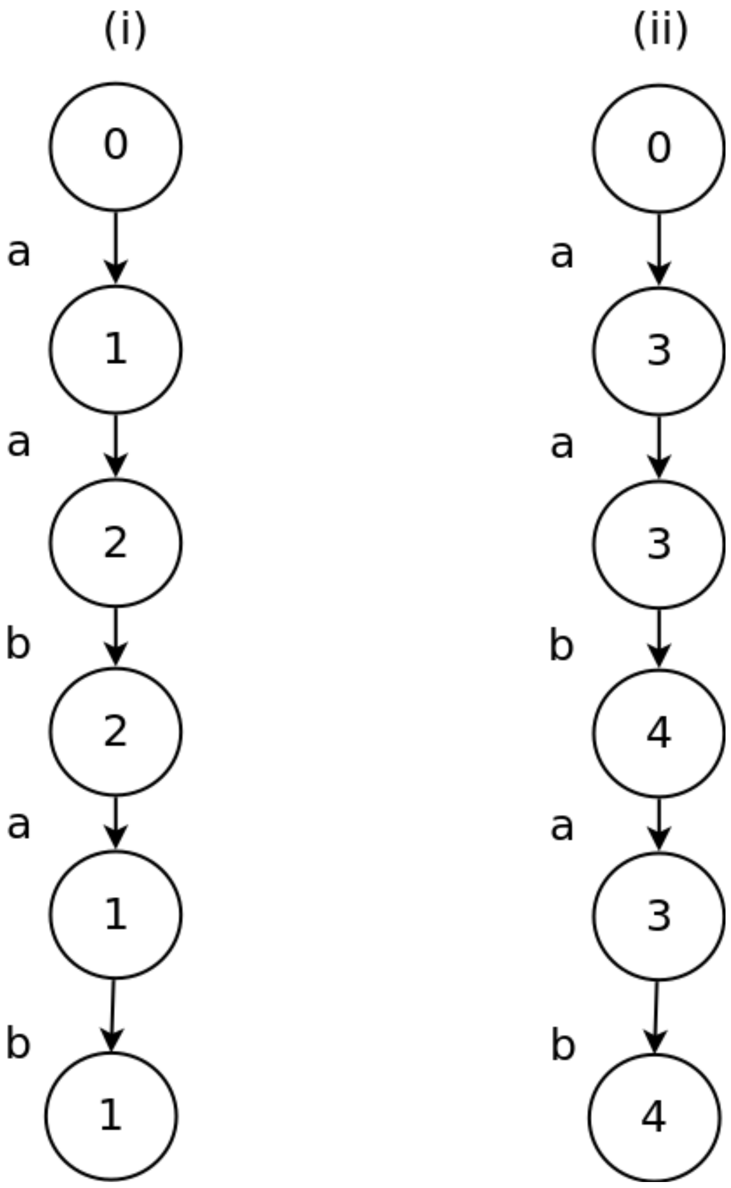
Input `validator`:	Input `tester`:
`12 2 5 0 2`	`ab`
`0`	`a`
`2 4`	`aba`
`0 a 1 3`	`aabbaba`
`1 a 2`	`!`
`1 b 1`	
`2 a 1`	
`2 b 2`	

`3 a 3`
`3 b 4`
`4 a 3`
`4 b 4`	

Output `validator`:
 	
`Rcd: 4`
`Snt: 4`
`Acc: 3`
`PID: pid`
`Rcd: 4`
`Acc: 3`
	Output `tester`:
	
	`PID: pid`
	`ab A`
	`a N`
	`aabbaba A`
	`aba A`
	`Snt: 4`
	`Rcd: 4`
	`Acc: 3`

Runs

Two runs on the word `aabab`, (i) does not accept, (ii) does.



Testing and the solution.

The solution should be contained in an archive **ab123456.zip**, and should consists in the source codes, the makefiles and the readme file. The sequence **ab123456** should be replaced by the student's server alias.

The following sequence of commands:

```
unzip ab123456.zip; cd ab123456; mkdir build; cd build; cmake ..; make
```

should result in the creation of the following files in the directory build `validator`, `run`, `tester`. Those files will be run, e.g., using the following sequence of commands

```
./validator <validator.in
...
./tester <t1.in
...
./tester <t2.in
```

```
...
./myFeed | ./tester
```

Moreover, in the archive there should be a file `README.pdf` describing the solution. The file should contain the full description of the means of communication, the list of used resources, and the list of assumptions (if any).

Deadline

The solution should be sent on ~~**19 January 2018**~~ **22 January 2018, at 23:59** at the latest.

~~The exact means of sending the solution will be provided at a later date.~~ Because of the postponed deadline, we publish verification script named `unpack`, which run as

```
./unpack ab123456
```

in the directory containinig the archive ab123456.zip will: unpack the archive, move the README.pdf file to the appropriate place and, compile the solution.

Warning! The solutions, which will not work with the `unpack` script will receive 0 points.

Any questions to the `unpack` script should be sent before Wednesday the 17th I 2018.

MISC

Few assumptions:

- One can assume that the input will be consistent with the above description. Still, if any system call fails, one needs to react appropriately, clean up the resources and terminate gracefully.
- We require that any parent process should wait for its children to terminate and every system resource .
- The solution should work when run on the computers in the lab.
- Any questions should be sent to Marcin Przybyłko: M.Przybylko 'at' mimuw.edu.pl

Remote access to the lab's computers.

To be alble to remotely access a computer in the lab

- one should know its name, e.g. `yellow01, ..., yellow15`
- the computer has to be turned on.

If the above is satisfied then one can remotely access the machines in the following way:

- use `ssh` to connect with students e.g. `ssh username@students.mimuw.edu.pl`,
- and then use `ssh` to connect to access the computer, e.g.: `ssh yellow11`.

To send file to the server one can use scp.

windows access can be achieved via `putty` i `winscp`.

FAQ

FAQ can be found here.

FAQ

FAQ is in English to avoid maintaining two FAQs.

1. **Q**: What should happen when one of the programs encounters an error in a system call?

   A: If an unexpected error occurs in any system call (e.g. `ENOMEM` in `fork`, but not `EAGAIN` in `mq` with the `O_NONBLOCK` flag set) all programs should terminate gracefully, i.e.:

   - the error should be written out on the stderr stream and the respective process should exit with value `-1`;
   - every acquired system resource (like temporal files, message queues etc.) should be released;
   - all processes should terminate, in particular there should be no zombies.

   To help with the termination, we allow the use of **signals** to message the shut down of the system, even when no error occurred.

   Please recall that some of the system resources are limited. To check some of them, e.g. max user processes, input

   ```
   ulimit -a
   ```

   in terminal.

   Moreover, if one of the programs terminates with an error (with a value that is not `0`) the standard outputs of all programs are ignored. Please take into consideration that we cannot guarantee that the automatic tests do not generate unexpected errors.

2. **Q**: Can I use pipes to pass the automaton and the word from the `validator` process to the newly created `run` process?

   A: Yes, we allow that.

   ## 19/01/2018

3. **Q:** Can the `tester` be divided into two processes? If yes, which pid should be used in the `validator`'s report?

   A: Yes it can create a subprocess, in communication the `pid` of the original process should be used.

4. **Q:** I am using shared memory `shm` to communicate the automaton to the `run` processes. Is this allowed?

A: The quick answer is no. While I agree that this is quite the elegant solution, shared memory was not specified as an allowed form of communication, and since the deadline is in three days, the specification will not change.

On the other hand, I was informed that many of the students use shared memory in their solutions. Therefore, due to the approaching deadline, I shall allow the use of the shared memory with a small penalty (up to -1pt).

5. **Q:** According to the first answer in FAQ, if an error occurres the system should terminate gracefully, but does it have to terminate immidiately?

   A: No, the immediate termination of the system is allowed but not required. The only requirement is to clean up the all unneeded resources ASAP, and release all acquired system resources upon termination. E.g. if one of the testers encounters an error, it can terminate, but the remaining processes are allowed to continue.

6. During this weekend the computers in the room 2044 (brown) will be on-line and can be used for testing.

   On the other hand M.Przybylko may respond to emails with delay.