11th Marathon of Parallel Programming

WSCAD – 2016

October 6th, 2016.

Rules for Local Contest

For all problems, read carefully the input and output session. For all problems, a sequential implementation is given, and it is against the output of those implementations that the output of your programs will be compared to decide if your implementation is correct. You can modify the program in any way you see fit, except when the problem description states otherwise. You must upload a compressed file with your source code, the *Makefile* and an execution script. The script should have the name of the problem. You can submit as many solutions to a problem as you want. Only the last submission will be considered. The *Makefile* must have the rule *all*, which will be used to compile your source code before submit. The execution script runs your solution the way you design it – it will be inspected not to corrupt the target machine. This script may contain PBS directives, but should not call *qsub* or any PBS command.
All *Local Teams* have access to the target machine during the marathon. Your execution may have concurrent process from other teams. Only the judges have access to a non-concurrent environment.
The execution time of your program will be measured running it with *time* program and taking the real CPU time given. Each program will be executed at least three times with the same input and the mean time will be taken into account. The sequential program given will be measured the same way. You will earn points in each problem, corresponding to the division of the sequential time by the time of your program (*speedup*). The team with the most points at the end of the marathon will be declared the winner.

This problem set contains 4 problems; pages are numbered from 1 to 15.

Problem A

String Parsing

String Parsing is an important problem in Computer Science and the central algorithm in the construction of compilers. It is the process of analyzing a string of symbols conforming to the rules of a formal grammar.

A grammar mainly consists of a set of production rules for transforming strings of symbols. To generate a string in the language described by the grammar, one begins with a string consisting of only a single start symbol. The production rules are then applied – in any order – until a string that does not contain neither the start symbol nor designated non-terminal symbols is produced. The language generated by the grammar consists of all distinct strings that can be generated in this manner.

For instance, assume that *a*, *b*, and *c* are terminal symbols – symbols allowed to appear on the input string –, the start, non-terminal symbol is *S*, and we have the following production rules:

$$(1)\ S \rightarrow aSb,\ (2)\ S \rightarrow cc.$$

We, then, start with *S* and can choose a rule to apply to it. If we choose rule 1, we obtain string *aSb*. If we then choose rule 1 again, we replace the *S* symbol within *aSb* and obtain the string *aaSbb*. If we now choose rule 2, we replace *S* with *cc* and obtain the string *aaccbb*, finishing the process.

Formally, a context-free grammar G (the structure we are interested in) is defined as a 4-tuple ⟨N, Σ, P, S⟩ where

- N is a finite set of *non-terminal* symbols, disjoint with the strings possibly formed from G;
- Σ is a finite set of *terminal* symbols, disjoint from N;
- P is a finite set of production rules, each rule of the form

  $$N \rightarrow (\Sigma \cup N)^*,$$

  where * is the Kleene star operator – which denotes "all possible strings over" – and ∪ denotes set union. That is, each production rule maps from one non-terminal symbol to a string of symbols.

The problem at hand is to decide whether a given string of symbols belongs or not to the language generated by the grammar.

Given a context-free grammar G and a string of terminal symbols σ (where $\sigma \in \Sigma^*$) as input, the sequential solution determines if there is a sequence of zero or more production rules $\Rightarrow^*$ such that $S \Rightarrow^* \sigma$. If it does, the algorithm returns that the analysis was successful and the sequence of production rules $\Rightarrow^*$ applied to transform S in σ. If it does not, the algorithm returns that the analysis was unsuccessful.

The technique employed in the implementation is a backtracking search to determine which production rule will be applied at each time. The algorithm starts with an empty stack of production rules Q that will be filled by the rules used to perform the derivations. At each step we compare the current evaluation string π with the input string σ. Initially we set $\pi = S$. If σ belongs to the language generated by G, the algorithm outputs the pair ⟨Q, SUCCESS⟩. Otherwise, it outputs the pair ⟨∅, FAILED⟩:

1. [Eliminate common prefix.] Let φ be the longest common prefix of terminal symbols over π and σ, such that $\pi = \varphi\alpha$ and $\sigma = \varphi\beta$ for any strings of symbols α and β. Then, let $\pi' \leftarrow \alpha$ and $\sigma' \leftarrow \beta$.
2. [Is the initial symbol terminal?] If the first symbol of σ' is a terminal, return ⟨∅, FAILED⟩.
3. [Both empty?] If both φ' and σ' are empty, return ⟨Q, SUCCESS⟩.
4. [Is σ empty?] If σ' is empty, and φ' contains at least one terminal symbol, return ⟨∅, FAILED⟩.
5. [Try all derivations recursively.] At the moment, $\pi' = A\alpha$, where A is a non-terminal symbol and α is any string of symbols. Thus, for every production rule i on the form $A \rightarrow \beta_i$ – where β_i is also any string of symbols – let $Q_i = Q+(A \rightarrow \beta_i)$ and $\pi_i = \beta_i\alpha$. Then, for each i call the algorithm recursively with $\pi = \pi_i$, $Q = Q_i$, and $\sigma = \sigma'$.
6. [Evaluate return.] If any recursive call i returned ⟨Qi, SUCCESS⟩ (where $Q_i! = \varnothing$), return it and terminate.

Input

The first line of the input contains the input string to be analyzed. Each of its symbols is a sequence of at most 256 ASCII characters and the symbols *must* be separated by the ASCII whitespace character, which is never evaluated as a symbol or is part of another symbol.

The second line of the input contains the set of terminal symbols, in the same format as the input string.

The third line of the input contains the set of non-terminal symbols, in the same format as the input string.

The fourth line of the input contains the start symbol, which is always a non-terminal symbol listed on the line above and has the same constraints as each of the input string symbols.

The remaining lines describe the *production rules*, whose format is either "A : α" or a single "-" character. If the rule has format "A : α", then A is a single non-terminal and α is a string of symbols with the same constraints as the input string. Otherwise, if it is a "-", then it signals end-of-input.

The input must be read from the standard input.

Output

If the input string does not belong to the language generated by the grammar, then the program outputs "FAILED" in a single line. Otherwise, the program first prints a list of production rules – in the same format as they appear on the input –, one per line, which are applied, in order, to obtain the derivation from the start symbol to the input string. After that, the program outputs "SUCCESS" in a single line.

The output must be written to the standard output.

Example

The following example is over a grammar that generates three English words whose first letter is "r" and last letter is "d": read, road, and raid. The grammar is $\mathrm{G} = \langle \{S,X,Z\}, \{r,d,o,a,e,i\}, S, \{S \rightarrow rXd,\ S \rightarrow rZd,\ X \rightarrow oa,\ X \rightarrow ea,\ Z \rightarrow ai\} \rangle$.

Input 1

```
road
rdoaei
SXZ
S
S:rXd
S:rZd
X:oa
X:ea
Z:ai
-
```

Output for the input 1

```
S:rXd
X:oa
SUCCESS
```

Input 2

```
raod
rdoaei
SXZ
S
S:rXd
S:rZd
X:oa
X:ea
Z:ai
-
```

Output for the input 2

```
FAILED
```

```
...
Pair <String, String> split_string_delim_pair (String delim, String s) {
    String::size_type pos = s.find (delim) ;
    return {s.substr (0, pos), s.substr (pos + delim.size (), s.size ())} ;
}

void print (String s) {
    std::cout << s << std::endl ;
}

String production_string (Production prod) {
    return
        to_string (left (prod))
            + " "
            + production_delim_string ()
            + " "
            + to_string (right (prod)) ;
}

String path_string (Path deriv) {
    String s ;
    for (Production p : deriv) {
        s += production_string (p) ;
        s += linebreak () ;
    }
    return s ;
}

String not_valid_string () {
    return String ("FAILED") ;
}

String valid_string () {
    return String ("SUCCESS") ;
}

String end_of_productions_string () {
    return String ("-") ;
}

String production_delim_string () {
    return String (":") ;
}

template <typename Stream>
String get_line (Stream& fin) {
    String s ;
    std::getline (fin, s) ;
    return s ;
}

template <typename Container>
Container read_set_of_symbols (String s) {
    return split_string <Set <Symbol>> (s) ;
}

Production read_production (String s) {
    Pair <String, String> ss
        = split_string_delim_pair (production_delim_string (), s) ;
    return
        Production
            ( to_symbol_string (ss.first)
( to_symbol_string (ss.first)
            , to_symbol_string (ss.second) ) ;
}

template <typename Stream>
Set <Production> read_productions (Stream& fin) {
    Set <Production> prod ;
    String s ;
    String end_mark = end_of_productions_string () ;
    while (true) {
        s = get_line (fin) ;
        if (s == end_mark) break ;
        prod = insert (read_production (s), prod) ;
    }
    return prod ;
}

template <typename Stream>
Symbol read_start (Stream& fin) {
    Symbol sym ;
    std::istringstream (get_line (fin)) >> sym ;
    return sym ;
}

template <typename Stream>
Set <Symbol> read_nonterminals (Stream& fin) {
    return read_set_of_symbols <Set <Symbol>> (get_line (fin)) ;
}

template <typename Stream>
Set <Symbol> read_terminals (Stream& fin) {
    return read_set_of_symbols <Set <Symbol>> (get_line (fin)) ;
}

template <typename Stream>
Grammar read_grammar (Stream& fin) {
    Grammar g ;
    g.terminals    = read_terminals (fin)    ;
    g.nonterminals = read_nonterminals (fin) ;
    g.start        = read_start (fin)        ;
    g.productions  = read_productions (fin)  ;
    return g ;
}

template <typename Stream>
SymbolString read_symbol_string (Stream& fin) {
    return to_symbol_string (get_line (fin)) ;
}

template <typename Stream>
Input read_input (Stream& fin) {
    Input in ;
    string  (in) = read_symbol_string (fin) ;
    grammar (in) = read_grammar (fin) ;
    return in ;
}

String productions_string (Set <Production> sp) {
    String s ;
    for (Production p : sp) {
        s += production_string (p) ;
        s += linebreak () ;
```

```
}
    return s ;
}

String make_output_string (Path p, Bool accept) {
    String s
        = accept ? path_string (p) + valid_string () : not_valid_string () ;
    return s ;
}

Bool is_terminal (Grammar g, Symbol s) {
    if (g.terminals.find (s) != std::end (g.terminals)) {
        return true ;
    }
    return false ;
}

Bool is_nonterminal (Grammar g, Symbol s) {
    return ! is_terminal (g, s) ;
}

Bool exists_non_terminal (Grammar g, SymbolString ss) {
    for (Symbol s : ss) if (is_nonterminal (g, s)) {
        return true ;
    }
    return false ;
}

Bool begins_with_terminal (Grammar g, SymbolString ss) {
    if (! ss.empty ()) {
        if (is_terminal (g, * std::begin (ss))) {
            return true ;
        }
    }
    return false ;
}

Symbol first_leftmost_nonterminal (Set <Symbol> n, SymbolString ss) {
    for (Symbol s : ss) {
        if (belongs_to (Symbol (s), n)) return s ;
    }
    return Symbol () ;
}

Bool left_side_equals (Production p, SymbolString l) {
    return p.first == l ;
}

Set <Production> all_productions_from_nonterminal
    ( Set <Production> lp
    , Symbol nt ) {
    Set <Production> x ;
    for (Production p : lp) {
        if (left_side_equals (p, SymbolString {nt})) x = insert (p, x) ;
    }
    return x ;
}

Bool any_empty (Stack s) {
    return s.current.empty () || s.input.empty () ;
}
```

```
SymbolString common_prefix (SymbolString a, SymbolString b) {
    // pre : assert (! empty (a) && ! empty (b)) ;
    auto miss
        = std::mismatch
            ( std::begin (a)
            , std::end   (a)
            , std::begin (b) ) ;
    return SymbolString (std::begin (a), miss.first) ;
}

SymbolString remove_prefix (SymbolString prefix, SymbolString ss) {
    ss.erase
        ( std::begin (ss)
        , std::next
            ( std::begin (ss)
            , std::min (prefix.size (), ss.size ()) ) ) ;
    return ss ;
}

Pair <SymbolString, SymbolString> eliminate_common_prefix (SymbolString a, SymbolString b) {
    SymbolString cpref = common_prefix (a, b) ;
    return
        Pair <SymbolString, SymbolString>
            (remove_prefix (cpref, a), remove_prefix (cpref, b)) ;
}

SymbolString current (Stack s) {
    return s.current ;
}

SymbolString replace_once
        ( SymbolString text
        , SymbolString old_string
        , SymbolString new_string ) {
    auto i
        = std::search
            ( std::begin (text)
            , std::end   (text)
            , std::begin (old_string)
            , std::end   (old_string) ) ;
    i = text.erase (i, std::next (i, old_string.size ())) ;
    text.insert (i, std::begin (new_string), std::end (new_string)) ;
    return text ;
}

Stack reduce (Stack s) {
    if (any_empty (s)) return s ;
    Pair <SymbolString, SymbolString> p
        = eliminate_common_prefix (s.current, s.input) ;
    s.current = p.first ;
    s.input = p.second ;
    return s ;
}

Stack expand (Stack s, Production p) {
    s.current = replace_once (current (s), left (p), right (p)) ;
    s.derivations.push_back (p) ;
    return s ;
}

Set <Stack> expand_all (Stack s, Set <Production> lp) {
    Set <Stack> stacks ;
    for (Production p : lp) {
        stacks = insert (expand (s, p), stacks) ;
    }
    return stacks ;
}

Set <Stack> all_leftmost_derivations_nonempty (Grammar g, Stack s) {
    Set <Production> lp
        = all_productions_from_nonterminal
            ( productions (g)
            , first_leftmost_nonterminal (nonterminals (g), current (s)) ) ;
    return expand_all (s, lp) ;
}

Set <Stack> all_leftmost_derivations (Grammar g, Stack s) {
    if (! exists_non_terminal (g, current (s))) return Set <Stack> () ;
    return all_leftmost_derivations_nonempty (g, s) ;
}

Eval accept_any_of (Grammar g, Set <Stack> ls) {
for (Stack s : ls) {
        Eval e = parse_recursive_descent (g, s) ;
        if (accept (e)) {
            return e ;
        }
    }
    return Eval (Path (), false) ;
}

Eval accept_any_of_all_leftmost_derivations (Grammar g, Stack s) {
Set <Stack> ald = all_leftmost_derivations (g, s) ;
    return accept_any_of (g, ald) ;
}

Eval parse_recursive_descent (Grammar g, Stack s) {
    s = reduce (s) ;
    if (begins_with_terminal (g, s.current)) {
        return Eval (Path (), false) ;
    }
    if (s.input.empty ()) {
        if (s.current.empty ()) {
            return Eval (derivations (s), true) ;
        }
        if (! exists_non_terminal (g, s.current)) {
            return Eval (Path (), false) ;
        }
    }
    return accept_any_of_all_leftmost_derivations (g, s) ;
}

Eval parse_recursive_descent (Grammar g, SymbolString input) {
    return
        parse_recursive_descent
            (g, Stack (SymbolString {g.start}, input, Path ())) ;
}

int main () {
    Input in = read_input (std::cin) ;
    Eval  ev = parse_recursive_descent (grammar (in), string (in)) ;
    print (make_output_string (derivations (ev), accept (ev))) ;
    return 0 ;
}
```

Problem B

Game of Life

The Game of Life is a cellular automaton invented by Cambridge mathematician John Conway. It consists of a collection of cells which, based on a few mathematical rules, can live, die or multiply. Depending on the initial conditions, the cells form various patterns throughout the course of the game. In this version of the game we have a square board (2D array) where cells are updated in each timestep according to the following rules:

- For a space that is populated:
  - Each cell with one or no neighbors dies, as if by solitude.
  - Each cell with four or more neighbors dies, as if by overpopulation.
  - Each cell with two or three neighbors survives.
- For a space that is empty or unpopulated:
  - Each cell with three neighbors becomes populated.

Figure B1 provides an example of the Game of Life in a 11×11 board. Empty cells and populated cells are represented by light and dark gray squares, respectively. Pane (*a*) shows the initial board and pane (*h*) shows the simulation result after 7 timesteps.

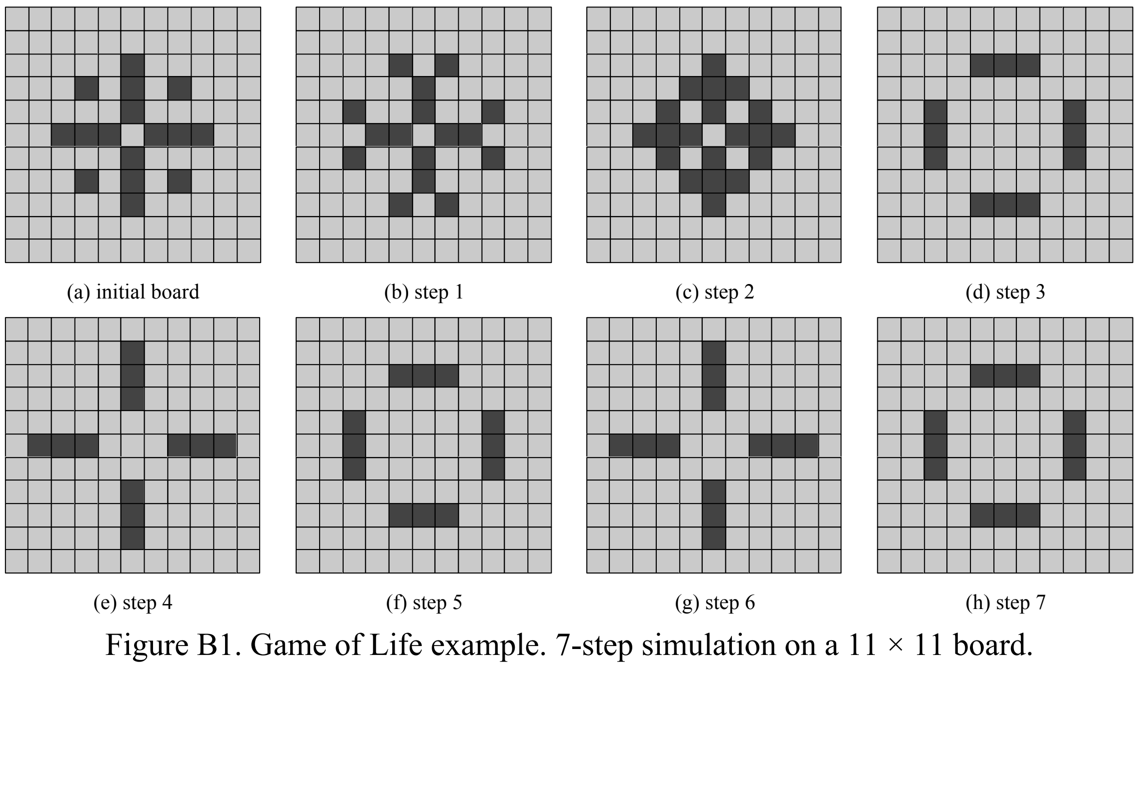


Figure B1. Game of Life example. 7-step simulation on a 11 × 11 board.

Input

The first line informs the board size and the number of steps (both integers), separated by a white space. The following lines represent the cells of the initial board, where a whitespace represent an empty cell and a "x" represent a populated one. Lines in the board are separated by line breaks.

The input must be read from the standard input.

Output

The program will print the last board, i.e., the board after all timesteps, where a whitespace represents an empty cell and a "x" represent a populated one. Lines in the board are separated by line breaks.

The output must be written to the standard output.

Example

Input	Output for the input

`11 7`
` `
` `
`     x`
`   x x x`
`     x`
`  xxx xxx`
`     x`
`   x x x`
`     x`
` `
` `
	` `
	` `
	`    xxx`
	` `
	`  x     x`
	`  x     x`
	`  x     x`
	` `
	`    xxx`
	` `
	` `
	` `

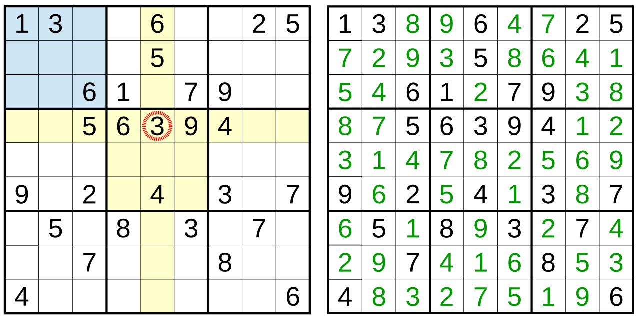
```c
#include <stdio.h>
typedef unsigned char cell_t;

cell_t ** allocate_board (int size) {
    cell_t ** board = (cell_t **) malloc(sizeof(cell_t*)*size);
    int    i;
    for (i=0; i<size; i++)
           board[i] = (cell_t *) malloc(sizeof(cell_t)*size);
    return board;
}

void free_board (cell_t ** board, int size) {
        int     i;
        for (i=0; i<size; i++)
                free(board[i]);
    free(board);
}


/* return the number of on cells adjacent to the i,j cell */
int adjacent_to (cell_t ** board, int size, int i, int j) {
    int    k, l, count=0;

    int sk = (i>0) ? i-1 : i;
    int ek = (i+1 < size) ? i+1 : i;
    int sl = (j>0) ? j-1 : j;
        int el = (j+1 < size) ? j+1 : j;

    for (k=sk; k<=ek; k++)
           for (l=sl; l<=el; l++)
                     count+=board[k][l];
    count-=board[i][j];

    return count;
}

void play (cell_t ** board, cell_t ** newboard, int size) {
    int    i, j, a;
    /* for each cell, apply the rules of Life */
    for (i=0; i<size; i++)
           for (j=0; j<size; j++) {
                     a = adjacent_to (board, size, i, j);
                     if (a == 2) newboard[i][j] = board[i][j];
                     if (a == 3) newboard[i][j] = 1;
                     if (a < 2) newboard[i][j] = 0;
                     if (a > 3) newboard[i][j] = 0;
           }
}

/* print the life board */
void print (cell_t ** board, int size) {
    int    i, j;
    /* for each row */
    for (j=0; j<size; j++) {
           /* print each column position... */
           for (i=0; i<size; i++)
                     printf ("%c", board[i][j] ? 'x' : ' ');
           /* followed by a carriage return */
           printf ("\n");
    }
}
```

```c
/* read a file into the life board */
void read_file (FILE * f, cell_t ** board, int size) {
           int        i, j;
           char       *s = (char *) malloc(size+10);
           char c;
           for (j=0; j<size; j++) {
                      /* get a string */
                      fgets (s, size+10,f);
                      /* copy the string to the life board */
                      for (i=0; i<size; i++)
                      {
                                 //c=fgetc(f);
                                 //putchar(c);
                                 board[i][j] = s[i] == 'x';
                      }
                      //fscanf(f,"\n");
           }
}

int main () {
           int size, steps;
           FILE    *f;
  f = stdin;
           fscanf(f,"%d %d", &size, &steps);
           cell_t ** prev = allocate_board (size);
           read_file (f, prev,size);
           fclose(f);
           cell_t ** next = allocate_board (size);
           cell_t ** tmp;
           int i,j;
           #ifdef DEBUG
           printf("Initial \n");
           print(prev,size);
           printf("----------\n");
           #endif

           for (i=0; i<steps; i++) {
                      play (prev,next,size);
                #ifdef DEBUG
                      printf("%d ----------\n", i);
                      print (next,size);
                      #endif
                      tmp = next;
                      next = prev;
                      prev = tmp;
           }
           print (prev,size);
           free_board(prev,size);
           free_board(next,size);
}
```

Problem C

Sudokount

Sudoku is a combinatorial number placement puzzle. It consists of a 9x9 *grid* (81 cell) divided into 9 3x3 *boxes*. Each cell might be empty or contain a number from 1 to 9. A *unit* is the collection of cells present in a line, column, or box. Therefore, each cell belongs to exactly 3 distinct units. The objective of the game is to fill the empty cells with numbers from 1 to 9 such that every unit contains all the numbers from 1 to 9. In other words, a solution is a fully filled the grid that has no line, column, or box containing the same number more than once. Normally each puzzle has only one solution. Figure C1 shows a Sudoku puzzle with a box (blue) and the three units (yellow) of a specific cell (red) highlighted. Solution numbers for the empty cells are marked in green.



1	3			6			2	5
				5				
		6	1		7	9		
		5	6	3	9	4		
9		2		4		3		7
	5		8		3		7	
		7				8		
4								6

1	3	8	9	6	4	7	2	5
7	2	9	3	5	8	6	4	1
5	4	6	1	2	7	9	3	8
8	7	5	6	3	9	4	1	2
3	1	4	7	8	2	5	6	9
9	6	2	5	4	1	3	8	7
6	5	1	8	9	3	2	7	4
2	9	7	4	1	6	8	5	3
4	8	3	2	7	5	1	9	6

Figure C1. On the left, a Sudoku puzzle. On the right the same puzzle with the solution numbers marked in green.

Solving 9x9 Sudoku puzzles can be a lot of fun. However, for the teams participating in the Marathon of Parallel Programming, these puzzles can quickly become dull since they can be easily solved using brute-force and clever heuristics such as constraint propagation. Therefore, let's make them more fun and create our own version of the puzzle called Sudokount!

Although not strictly necessary, we will assume valid Sudokount grids are squares, i.e., the number of cells in each line is the same number of cells in each column. Let *N* be the number of cells of a line. We will also assume valid puzzles have *N* boxes with *N* cells each and there is no intersection between the cells of each box. Clearly, traditional 9x9 Sudoku puzzles observe all of these properties. However, this definition gives as a nice way to extend grid sizes arbitrarily. Any grid of size *N*x*N* where $N = n^2$ for

n = 3,4,5,..., such as 9x9, 16x16, and 25x25, with cell values ranging from 1 to N is now a valid grid size. We deliberately leave out of the definition n values below 3, since here we are only interested in bigger puzzles. Additionally, Sudokount does not necessarily have only one solution, it might have thousands. Sudokount objective is, given a puzzle, count the number of solutions for that input.

You were given a sequential version of Sudokount which uses Peter Norvig's constraint propagation method[1] to count the number of solutions. Your task is to write a parallel version of Sudokount. Feel free to use any heuristic or method to improve the performance of the sequential version. However, the given sequential version will be used as a reference not only with respect to the performance but also to check the output of your program.

Input

The input contains one puzzle. It consists a list of integers separated by spaces or new lines. The first integer is the puzzle box size (n) followed by n^4 integers with the values of the grid. An empty cell is represented by a 0. Cell values are given from left to right, from top to bottom. The example input below represents the puzzle shown in Figure C1. You may assume $n \leq 8$.

The input must be read from the standard input.

Output

The expected output is one integer: the number of solutions.

The output must be written to the standard output.

[1] See mode details at http://norvig.com/sudoku.html

Example

Input 1

```
3
1 3 0 0 6 0 0 2 5
0 0 0 0 5 0 0 0 0
0 0 6 1 0 7 9 0 0
0 0 5 6 3 9 4 0 0
0 0 0 0 0 0 0 0 0
9 0 2 0 4 0 3 0 7
0 5 0 8 0 3 0 7 0
0 0 7 0 0 0 8 0 0
4 0 0 0 0 0 0 0 6
```

Output for the input 1

```
1
```

Input 2

```
4
0 0 0 15 9 0 16 0 0 6 0 0 13 7 0 0
0 13 6 0 0 15 0 0 7 0 0 4 8 0 0 0
2 0 0 0 0 0 0 13 15 0 10 0 0 3 9 0
7 0 5 0 6 3 0 0 0 12 0 0 0 0 14 11
0 5 0 0 0 11 8 0 0 0 0 15 3 0 0 0
0 6 0 1 10 0 5 0 11 0 2 0 12 14 0 0
0 0 16 0 0 0 0 3 12 0 0 8 0 6 7 0
3 0 0 10 0 6 0 0 0 13 1 0 0 0 0 8
12 0 0 0 0 5 2 0 0 0 8 0 0 15 1 0
0 0 1 9 4 0 0 0 0 15 0 0 7 0 0 0
0 8 2 0 3 0 0 16 0 14 7 0 0 0 0 5
0 7 0 0 0 0 15 6 1 0 0 10 0 12 8 0
8 0 0 13 16 0 0 11 0 0 15 0 0 0 4 7
0 9 0 0 0 0 7 0 0 11 0 16 14 0 0 0
0 11 3 0 15 12 0 0 8 0 0 0 0 10 0 0
1 0 4 0 0 0 3 0 0 2 0 6 16 0 0 15
```

Output for the input 2

```
300064
```

```
static inline int cell_v_get(cell_v *v, int p) {
    return !!((*v).v[(p - 1) / INT_TYPE_SIZE] & (((INT_TYPE)1) << ((p - 1) % INT_TYPE_SIZE))); //!!
otherwise p > 32 breaks the return
}

static inline void cell_v_unset(cell_v *v, int p) {
    (*v).v[(p - 1) / INT_TYPE_SIZE] &= ~(((INT_TYPE)1) << ((p - 1) % INT_TYPE_SIZE));
}

static inline void cell_v_set(cell_v *v, int p) {
    (*v).v[(p - 1) / INT_TYPE_SIZE] |= ((INT_TYPE)1) << ((p -1) % INT_TYPE_SIZE);
}

static inline int cell_v_count(cell_v *v) {
    int acc = 0;
    for (int i = 0; i < CELL_VAL_SIZE; i++)
        acc += __builtin_popcountll((*v).v[i]);
    return acc;
}

static int eliminate (sudoku *s, int i, int j, int d) {
    int k, ii, cont, pos;

    if (!cell_v_get(&s->values[i][j], d))
        return 1;

    cell_v_unset(&s->values[i][j], d);

    int count = cell_v_count(&s->values[i][j]);
    if (count == 0) {
        return 0;
    } else if (count == 1) {
        for (k = 0; k < s->peers_size; k++)
            if (!eliminate(s, s->peers[i][j][k].r, s->peers[i][j][k].c, digit_get(&s->values[i][j])))
                return 0;
    }
    for (k = 0; k < 3; k++) {//row, column, box
        cont = 0;
        pos = 0;
        cell_coord* u = s->unit_list[i][j][k];
        for (ii = 0; ii < s->dim; ii++) {
            if (cell_v_get(&s->values[u[ii].r][u[ii].c], d)) {
                cont++;
                pos = ii;
            }
        }
        if (cont == 0)
            return 0;
        else if (cont == 1) {
            if (!assign(s, u[pos].r, u[pos].c, d))
                return 0;
        }
    }
    return 1;
}
static int assign (sudoku *s, int i, int j, int d) {
    for (int d2 = 1; d2 <= s->dim; d2++)
        if (d2 != d)
            if (!eliminate(s, i, j, d2))
               return 0;
    return 1;
}
```

```
static int search (sudoku *s, int status) {
    int i, j, k;

    if (!status) return status;

    int solved = 1;
    for (i = 0; solved && i < s->dim; i++)
        for (j = 0; j < s->dim; j++)
            if (cell_v_count(&s->values[i][j]) != 1) {
                solved = 0;
                break;
            }
    if (solved) {
        s->sol_count++;
        return SUDOKU_SOLVE_STRATEGY == SUDOKU_SOLVE;
    }

    //ok, there is still some work to be done
    int min = INT_MAX;
    int minI = -1;
    int minJ = -1;
    int ret = 0;

    cell_v **values_bkp = malloc (sizeof (cell_v *) * s->dim);
    for (i = 0; i < s->dim; i++)
        values_bkp[i] = malloc (sizeof (cell_v) * s->dim);

    for (i = 0; i < s->dim; i++)
        for (j = 0; j < s->dim; j++) {
            int used = cell_v_count(&s->values[i][j]);
            if (used > 1 && used < min) {
                min = used;
                minI = i;
                minJ = j;
            }
        }

    for (k = 1; k <= s->dim; k++) {
        if (cell_v_get(&s->values[minI][minJ], k))  {
            for (i = 0; i < s->dim; i++)
                for (j = 0; j < s->dim; j++)
                    values_bkp[i][j] = s->values[i][j];

            if (search (s, assign(s, minI, minJ, k))) {
                ret = 1;
                goto FR_RT;
            } else {
                for (i = 0; i < s->dim; i++)
                    for (j = 0; j < s->dim; j++)
                        s->values[i][j] = values_bkp[i][j];
            }
        }
    }

    FR_RT:
    for (i = 0; i < s->dim; i++)
        free(values_bkp[i]);
    free (values_bkp);

    return ret;
}
```

Problem D

Color Histogram

In image processing and photography, a color histogram is a representation of the distribution of colors in an image. For digital images, a color histogram represents the number of pixels that have colors in each of a fixed list of color ranges, that span the image's color space, the set of all possible colors.[2]

Create a parallel version of an algorithm that generates a color histogram of a digital image.

Input

The input image is a text file using PPM format[3]. The image resolution is up to 4k.

The input must be read from the standard input.

Output

The output contains 64 numbers representing the mean of pixel used in the digital image. Each number is a floating point number with 3 decimal places.

The output must be written to the standard output.

Example

Input	Output for the input

`P6`
`2 2`
`255`
`!!!~~~`
`~~~!!!`
	`0.250 0.000 0.000 0.000 0.000 0.250 0.000 0.000 0.000 0.000`
	`0.000 0.000 0.000 0.000 0.000 0.000 0.250 0.000 0.000 0.000`
	`0.000 0.250 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000`
	`0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000`
	`0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000`
	`0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000`
	`0.000 0.000 0.000 0.000`

[2] Text from Wikipedia: https://en.wikipedia.org/wiki/Color_histogram

[3] More information about this format can be found at http://netpbm.sourceforge.net/doc/ppm.html .

```c
typedef struct {
        unsigned char red, green, blue;
} PPMPixel;

typedef struct {
        int x, y;
        PPMPixel *data;
} PPMImage;

static PPMImage *readPPM() {
        char buff[16];
        PPMImage *img;
        FILE *fp;
        int c, rgb_comp_color;
  fp = stdin;

        if (!fgets(buff, sizeof(buff), fp)) {
                perror("stdin");
                exit(1);
        }

        if (buff[0] != 'P' || buff[1] != '6') {
                fprintf(stderr, "Invalid image format (must be 'P6')\n");
                exit(1);
        }

        img = (PPMImage *) malloc(sizeof(PPMImage));
        if (!img) {
                fprintf(stderr, "Unable to allocate memory\n");
                exit(1);
        }

        c = getc(fp);
        while (c == '#') {
                while (getc(fp) != '\n')
                        ;
                c = getc(fp);
        }

        ungetc(c, fp);
        if (fscanf(fp, "%d %d", &img->x, &img->y) != 2) {
                fprintf(stderr, "Invalid image size (error loading)\n");
                exit(1);
        }

        if (fscanf(fp, "%d", &rgb_comp_color) != 1) {
                fprintf(stderr, "Invalid rgb component (error loading)\n");
                exit(1);
        }

        if (rgb_comp_color != RGB_COMPONENT_COLOR) {
                fprintf(stderr, "Image does not have 8-bits components\n");
                exit(1);
        }

        while (fgetc(fp) != '\n')
                ;
        img->data = (PPMPixel*) malloc(img->x * img->y * sizeof(PPMPixel));
        if (!img) {
                fprintf(stderr, "Unable to allocate memory\n");
                exit(1);
        }

        if (fread(img->data, 3 * img->x, img->y, fp) != img->y) {
                fprintf(stderr, "Error loading image.\n");
                exit(1);
        }

        return img;
}


void Histogram(PPMImage *image, float *h) {

        int i, j,  k, l, x, count;
        int rows, cols;

        float n = image->y * image->x;

        cols = image->x;
        rows = image->y;

        for (i = 0; i < n; i++) {
                image->data[i].red = floor((image->data[i].red * 4) / 256);
                image->data[i].blue = floor((image->data[i].blue * 4) / 256);
                image->data[i].green = floor((image->data[i].green * 4) / 256);
        }

        count = 0;
        x = 0;
        for (j = 0; j <= 3; j++) {
                for (k = 0; k <= 3; k++) {
                        for (l = 0; l <= 3; l++) {
                                for (i = 0; i < n; i++) {
                                        if (image->data[i].red == j && image-
>data[i].green == k && image->data[i].blue == l) {
                                                count++;
                                        }
                                }
                                h[x] = count / n;
                                count = 0;
                                x++;
                        }
                }
        }
}

int main(int argc, char *argv[]) {

        int i;

        PPMImage *image = readPPM();

        float *h = (float*)malloc(sizeof(float) * 64);

        for(i=0; i < 64; i++) h[i] = 0.0;

        Histogram(image, h);

        for (i = 0; i < 64; i++){
                printf("%0.3f ", h[i]);
        }
        printf("\n");
        free(h);
}
```