QUESTIONNAIRE DE

LA FOLLE COURSE INFORMATIQUE

THE MAD PROGRAMMING RACE

PROBLEMS

UNIVERSITÉ DE SHERBROOKE

Département de génie électrique et de génie informatique
Faculté de génie

ÉCOLE POLYTECHNIQUE MONTRÉAL

Département de génie électrique et de génie informatique

UNIVERSITÉ BISHOP'S UNIVERSITY

Computer Science Department

Novembre 2000

A few words on the Mad Programming Race

Dear programmers, once again this year Sherbrooke University presents the 2k edition of "La Folle Course Informatique" also known as "The Mad Programming Race". Even if some aspects of the MPR haven't changed, such as:

- The competition spirit,
- the format: 8 hours of programming,
- the bonus points,
- the teams of 2 to 4 participants with one an only one computer,

There are some new issues. Among others, this year's edition takes into consideration the team's experience and distinguishes the first-time participants from the returning ones. This allows new participants to rank higher. For this reason two price categories are introduced in this year's edition:

1. "Le Grand Prix AbacUS": this is the first price for teams which have at least one member that has already articipated in any of the previous editions of the MPR.
2. "Le Grand Prix ArgUS": this is the first price for teams composed only of members that have never participated in any of the MPR editions.

This year the MPR has gone international with the participation of Rhode's Island University.

We would like to give our sincere thanks to all those who have participated in the elaboration of this edition: those who wrote problems, participated in the elaboration of the questionnaire, wrote the programs that solve the problems, the correctors and everyone in the organizing committee. These people are: Vincent Boisclair, Alex Boisvert, Paule Bolduc, Jonathan Bourque, Jean-Denis Boyer, Charles-Antoine Brunet, Paul-André Chassé, Soumaya Cherkaoui, Daniel Dalle, Jean-Marie Dirand, Simon-Charles DuBerger, Denis Goyette, Jean Goulet, Gaëtan Haché, Johanne Hallée, Jean-Yves Hervé, Ahmed Khoumsi, Yannick Lacroix, Jean-Philippe Matte, Benoit Tessier, Yannick Syam, Lourdes Zubieta. Last but not least are the sponsors !! Our sponsors' list is on the MPR Web page.

Martine Bellaïche, École Polytechnique de Montréal.
Nelly Khouzam, Bishop's University.
Ruben Gonzalez-Rubio, Université de Sherbrooke.

Introduction

Read this preamble with the questionnaire of the *Mad Programming Race*. It contains a summary of the rules and various instructions related to the handing-over of the programs.

The Mad Programming Race

The Mad Programming Race is a programming competition. The participating teams must write programs according to given specifications. Each program that successfully runs earns points for the team and the winning team is the one that accumulates the most points.

For each problem, you must conceive a C, C++ or Java program[1] that respects the specifications. This program will be tested with some testing files, a number of points will be granted to you according to the number of files correctly treated by your program. The number of tests and points corresponding is indicated at the end of each problem. The same sets of testing files be used to evaluate the programs of all the teams.

Bonus points : For a particular problem, bonus points will be given to teams whose programs will successfully pass all the testing files. The amount of the bonus will be determined according to the difficulty of the problem and the time of submission.

The following table indicates how the bonus points will be granted:

Problem number	Bonus at t_0	Bonus at the end of the race t_f
1, 2	30 %	0 %
3, 4, 5, 6, 7, 8	40 %	0 %
9, 10	50 %	0 %

where t_0 is the starting time of the competition and t_f is the end of the competition.

Hence, bonus points decreases in a linear way, as time advances. For example, if after four hours of competition a team hands-over the program for question 10 and that it passes all the tests, it has a bonus of 25 %. If a team hands-over the program for question 10 after six hours of competition and it passes all the tests, it earns a bonus of 12,5 %.

[1]In Sherbrooke we chose these three languages, however, the corrector may allow other compiled languages to be used. If your local organization chooses to use a different language, it is necessary then to consult the particular procedures for the handing-over of programs. In the questionnaire, we refer only to the C, C++ and Java languages.

The bonus is a percentage calculated according to the following formula:

$$B = p \times \frac{t_r}{t_t}$$

where, B is the percentage bonus to be granted, p is the starting percentage at t_0, t_r is time remaining when the program is delivered and t_t is the total time of the race. The granularity of time is that of the marker and is estimated at two seconds.

In a sentence, the quicker the program is handed-over, the larger is the bonus, of course provided that the program passes all the tests.

In eight hours, it is not very probable but not impossible that a team has sufficient time to program all the problems presented. You will have to show judgement by choosing the problems you try to solve.

We tried to present the problems in a uniform way and without ambiguity. Each statement comprises a description and specification of the problem to be solved, as well as examples and files that could be needed for the problem.

At the end of the race, the classification of the teams will be established according to the total of the points accumulated for each handed-over program. In the possibility where two teams would obtain an equal number of points, the first to reach that number of points will be the winning team.

Handing-over programs

You must hand-over only one source file for each problem. This file will have the extension "`.C`" if it must be compiled in C, "`.CPP`" if it is in C++, "`.java`" if it is in Java. The first part of the name will be made up of the number of the problem followed by the number of your team according to the format "`P##_EQ##`". For example, problem 3 of team 9, coded in C++, would bear the name "`P03_EQ09.CPP`" and "`P03_EQ15.CPP`" for the same problem of team 15. In Java, and for problem 3 team 9, the source file "`p03_eq09.java`" must contains a class named "`p03_eq09`" which must contains the `main`. Be careful to write the class name in lower cases since Java is case sensitive: this is very important for the correction. Since you must submit only one file (exactly one for each problem), if you want to define more than one class they must be non-public or inner-class. The name of the file comprises exactly 8 characters before the point. **Files which do not conform to this format will not be considered for correction**.

Each handed-over file will be compiled in order to produce an object program. This program will be run several times with the testing files. The output files will be analyzed in order to check whether they are in conformity with the specifications, and the points will be granted accordingly. A program must compile without any error (warnings will be tolerated). A program which does not compile will not thus be given any point. With the correction, the outputs on the console (like `printf` or others) are also tolerated, but the execution time increases quickly. It is thus advised to avoid them in order not to exceed allocated time. Note that your source code will not be examined so you have total freedom on the programming style you will use.

Note that the correction is automated and is carried out during the race in real time. Unless there is a technical problem with the correction system, you will be able to consult your results on a monitor and this a few moments after you handed-over your program.

The inputs and outputs are always done via ASCII text files. Those will be named according to the number of the problem with the format "`P##.ENT`" for input files and "`P##.SOR`" for output files. The `##` indicates the number of the problem, it varies between `01` and `10`. Their contents and how to use them is clearly defined in the statement of each problem. Moreover, one example is presented for each one of these files.

Take for granted that the input files which will be used to test your programs will follow rigorously the format which is indicated in each problem. The examples of files shown in the text will be provided to you.

It is crucial that the files produced by your programs respect the specifications rigorously since they will be automatically corrected.

When you hand-over your program for its evaluation, you have to copy it in a " deposit box " which will be indicated to you at the time of the race as well as the exact procedure. You will be able to deposit only once your solution to each problem. Once a program is handed-over, no modification will be accepted. If you deposit your program again, the new copy will be ignored.

The rules

- Only one computer will be assigned to each team. A team can use only the computer which is assigned to it. In the event of a breakdown, the team has to wait until an organizer assigns you a new computer.
- A study room near the computer room will be available to the teams.
- You have the right to bring and to use any relevant documentation, as long as it is printed or hand-written. Any material support (other then documents) is prohibited during the race, including portable diskettes and computers. **A team bringing such unauthorized material will be automatically disqualified.**
- The local area network will be cut from the external world and therefore Internet will not be usable.
- In order to avoid annoying accidents, any food or drink will be prohibited in the computer rooms.
- We count on the honesty and the good faith of the participants.

Final Remarks

The Mad Programming Race is organized by a team of voluntary members, which is renewed with each edition. We endeavor to write specifications as clearly as possible.

We wrote programs to the specifications of the problems, we wrote the sets of testing files and program correctors by devoting much effort and time. Even during the race, we check in order to make sure that all is fine and that everything occurs in an equitable way. However, we do not claim perfection! For this reason, we ask you to call upon your "computer-sportsmanship" spirit in order to accept the "official" classifications given at the end of the race. Indeed, it is practically impossible to change the distribution of the prices if changes in the classification occurred. We think that the greatest reward associated with this competition is satisfaction to have made an effort to write programs and hopefully to have learned something. However, we are open to remarks which could improve the future competitions or which inform us with an error.

The "Grand Prix AbacUS" and the "Grand Prix ArgUS" will be allotted after a few days to make sure that all the possible checks were made.

Conventions used in the questionnaire

To indicate the beginning and the end of a file we use the symbols $\triangleright$ and $\triangleleft$ respectively. Of course these symbols do not form part of the file.

For example, the following file contains only one line with the chain `hello`.

$\triangleright$

```
hello
```

$\triangleleft$

Attention! The end of file could be just after the `o` character or at the begining of the next line!

Contents

The Specifications

Specification 1

Encryption

Ruben Gonzalez-Rubio

`File to be produced containing your source code: P01_EQ##.*`

The encryption of messages was almost certainly invented during war to hide important information from the enemy. In principle, only two people will know the contents of a message : the person who encrypts or encodes the message and the person who decodes it.

From The American Heritage Dictionary of the English Language:

encrypt *v.tr.* encrypted, encrypting, encrypts.

1. To put into code or cipher.
2. *Computer Science.* To scramble access codes to (computerized information) so as to prevent unauthorized access.

Problem

Among the many techniques of encoding, there is one based on factors. We will use this latter in this problem.

Above all, it is necessary to agree on an alphabet. To simplify the problem, we will use only the capital letters from “A” to “Z”.

Here is a message:

```
SEE YOU AT THE MAD PROGRAMMING RACE
```

The first thing to do is to remove spaces, such that the message becomes the string:

```
SEEYOUATTHEMADPROGRAMMINGRACE
```

It is then necessary that the message contain a number of letters equal to a multiple of 5. If this is not the case additional character(s) must be added to the end of the string: `Z` if missing 1 character, `ZQ` if missing 2 character, `ZQJ` if missing 3 character, `ZQJX` if missing 4 character.

As this message comprises 29 characters, we add Z. This gives[1]:

```
SEEYOUATTHEMADPROGRAMMINGRACEZ
```

It is here that the factors are introduced. Since we have 30 characters, the factor pairs giving 30 are:

$$\begin{array}{rcr} 1 & \times & 30 \\ 30 & \times & 1 \\ 2 & \times & 15 \\ 15 & \times & 2 \\ 3 & \times & 10 \\ 10 & \times & 3 \\ 5 & \times & 6 \\ 6 & \times & 5 \end{array}$$

We will choose the factors 15 and 2 (15×2) for our encoding. This means that the line of 30 characters will be divided into 15 columns and 2 lines. We will build the 15 columns so that the message may be read downwards starting at the top left corner. Other ways to build the columns will be described later.

```
S E O A T E A P O R M I G A E
E Y U T H M D R G A M N R C Z
```

Here, we introduced spaces to facilitate the reading, but actually the result should be :

```
SEOATEAPORMIGAE
EYUTHMDRGAMNRCZ
```

We concatenate the two lines, the first then the second, giving:

```
SEOATEAPORMIGAEEYUTHMDRGAMNRCZ
```

To finish, we divide the character string into words of 5 letters:

```
SEOAT EAPOR MIGAE EYUTH MDRGA MNRCZ
```

[1] If the message was `YOUATTHEMADPROGRAMMINGRACE` it would become `YOUATTHEMADPROGRAMMINGRACEZQJX` since we add `ZQJX` to have 30 characters (30 is a multiple of 5!).

Illegible, right?

To decipher (or decrypt) the message, one follows the reverse procedure. We remove the spaces :

```
SEOATEAPORMIGAEEYUTHMDRGAMNRCZ
```

We divide the chain into two, of course we know that the factors are 15 and 2 (15×2) :

```
SEOATEAPORMIGAE
EYUTHMDRGAMNRCZ
```

We then read the columns downwards starting once more, from the top left hand corner. The message is almost reconstituted :

```
SEEYOUATTHEMADPROGRAMMINGRACEZ
```

Of course, spaces would have to be added and the last character removed, but this treatment must be carried out by somebody who understands well the language of the message. In the present case, we will be satisfied to simply find all the letters of the message in the correct order. The expected result is therefore :

```
SEEYOUATTHEMADPROGRAMMINGRACEZ
```

If instead of the factor pair 15×2, we had used the 3×10, this would give 10 lines of three columns :

```
SEM
EMM
EAI
YDN
OPG
URR
AOA
TGC
TRE
HAZ
```

Once again, we have built our columns using the rule that the message must be read downwards, one column at a time, starting from the left.

However, there are in general four ways to build columns from a message. Consider our example message encoded using the factor pair 6×5.

Type 0 Vertical, beginning at top left, reading downwards (as previously described)

```
SUERMR
EAMOMA
ETAGIC
YTDRNE
OHPAGZ
```

Type 1 Vertical, beginning at top right, reading downwards

```
RMREUS
AMOMAE
CIGATE
ENRDTY
ZGAPHO
```

Type 2 Vertical, beginning at bottom left, reading upwards

```
OHPAGZ
YTDRNE
ETAGIC
EAMOMA
SUERMR
```

Type 3 Vertical, beginning at bottom right, reading upwards

```
ZGAPHO
ENRDTY
CIGATE
AMOMAE
RMREUS
```

In short, this problem requires that you encrypt or decrypt a message using the factor pair technique.

Input File

The file that you must read is "`P01.ENT`". It is divided into groups of two lines, the groups are separated by a blank line.

The first line contains the command, the second line contains the message. The command consists of four integers. The first integer can take values 0 or 1, (0 for encoding and 1 for decoding). The second integer can take values 0 to 3, indicating the type of column building to be used (i.e., one of the four types defined above). The last two integers represent the factor pair, the first giving the number of columns and the second the number of lines. The four integers are separated by at least one space. The second line contains the message to encrypt or decipher.

Here is an example:

▷

```
0 0 15 2
SEE YOU AT THE MAD PROGRAMMING RACE

1 0 15 2
SEOAT EAPOR MIGAE EYUTH MDRGA MNRCZ
```

◁

Output File

You must write the file "`P01.SOR`" which will contain the messages which have been either encrypted or deciphered. Each message must appear on a new line and the messages must be separated by a blank line. They must appear in the same order as in the input file.

Thus, for the input file described above, the output file should be:

▷

```
SEOAT EAPOR MIGAE EYUTH MDRGA MNRCZ

SEEYOUATTHEMADPROGRAMMINGRACEZ
```

◁

It is guaranteed that the message contains between 16 and 2056 characters. The total number of messages to encrypt or decipher is between 16 and 2056. The number of messages to encrypt may be different from that to decipher.

Marking

The problem will be marked by submitting your program to four different input test files.

A test will be regarded as successful if the program manages to find the solution and correctly write it in the outputfile in the proper format.

Nombre de fichiers réussis	Points accordés
Successful test file	Points
1	80
2	120
3	160
4	200

Maximum execution time for your program : 10 seconds.

Specification 2

Code Breaking

Ruben Gonzalez-Rubio

`File to be produced containing your source code:  P02_EQ##.*`

This problem still treats encoding, but it is about decoding a message. It can be of use to the enemy who wants to decipher the message.

From the Webster's Revised Unabridged Dictionary:

decipher \ De*ci"pher \ *v.t.* [imp. & p. p. Deciphered; p. pr. & vb. n. Deciphering.] [Pref. de- + cipher. Formed in imitation of F. d['e]chiffrer. See Cipher.]

1. To translate from secret characters or ciphers into intelligible terms; as, to decipher a letter written in secret characters.

Problem

We assume that the technique of factors (see specification 1) is the one that is still used. Factor pairs, the addition of the characters and the types are the same as described before. The problem here is to find the type used for encrypting the message, the factors and the message itself.

Let us take as an example the following encrypted message:

```
SEOAT EAPOR MIGAE EYUTH MDRGA MNRCZ
```

Suppose that we know that the word RACE exists in the message, we must find the type, i.e. vertical, beginning at top left hand corner, reading downwards, and the factor pair 15 and 2. We must also find the decoded message.

Input File

The file that you must read is "`P02.ENT` ". It contains two lines. On the first, the encrypted message (a character string) and on the second, a word of the message (a character string).

▷

```
SEOAT EAPOR MIGAE EYUTH MDRGA MNRCZ
PROGRAMMING
```

◁

The encryption type is the same as in problem 1.

It is guaranteed that the word of the message is a representative character string (five or more characters and that it appears only once in the message).

Output File

You must write the file "`P02.SOR` " which will also contain two lines. The first line contains three integers separated by at least a space. The first integer indicates the encryption type, the second the number of columns and the third, the number of lines. The second line contains the decoded message without added spaces.

▷

```
0 15 2
SEEYOUATTHEMADPROGRAMMINGRACEZ
```

◁

In cases where more than one decoded message could result, the output file should contain only one of them.

Marking

The problem will be marked by submitting your program to four different input test files.

A test will be regarded as successful if the program manages to find the solution and correctly write it in the outputfile in the proper format.

Nombre de fichiers réussis	Points accordés
Successful test file	Points
1	80
2	120
3	160
4	200

Maximum execution time for your program : 20 seconds.

Specification 3

Card Shuffling

Alex Boisvert

`File to be produced containing your source code:  P03_EQ##.*`

Most card games rely on chance to provide excitement. Card shuffling is thus a significant operation which determines, indirectly, the outcome of a game.

For some unscrupulous players, card shuffling is an art. Obviously, the one which can put the chances on his side will be the winner.

Problem

You must write a program which simulates the " perfect " shuffling of a package of cards which contains N unique cards.

The shuffling technique is carried out as follows:

One cuts the package to the (m)th card starting from the top to obtain two piles[1]: a upper pile and a lower pile. The shuffling starts by depositing the bottom card of the upper pile, followed by the bottom card of the lower pile. The shuffling continues by depositing a card from each of the two piles, in alternation, until one of the two piles is empty. The remainder of the cards go on the top of the new pile.

The problem is to find the number of consecutive perfect shufflings necessary to bring, at the end, the pack back in its original order.

Example

Let us suppose a package of cards having $N = 5$ cards. One successively cuts it to the 3rd $(m = 3)$ card.

[1]the card m is in the upper pile

Original Pack :

```
1
2
3
4
5
```

After the first shuffling :

```
1
4
2
5
3
```

After the second shuffling :

```
1
5
4
3
2
```

After the 3rd shuffling:

```
1
3
5
2
4
```

And after the 4th shuffling, one finds the original pack, therefore, the answer is 4.

Input File

The file that you must read is "`P03.ENT`". the entry is made up of two values: the number N of unique cards in the pack and the place where the package is cut m. The values are given on two lines.

▷

```
5
3
```

◁

The number of single cards varies between 2 and 501 whereas the cutting position varies between 1 and the number of cards in the package minus 1.

Output File

You must write the file "`P03.SOR`" which will contain only one value, the number of perfect shufflings necessary to bring the pack back in its original order.

▷

```
4
```

◁

Marking

The problem will be marked by submitting your program to four different input test files.

A test will be regarded as successful if the program manages to find the solution and correctly write it in the outputfile in the proper format.

Nombre de fichiers réussis	Points accordés
Successful test file	Points
1	120
2	180
3	240
4	300

Maximum execution time for your program : 10 seconds.

Specification 4

Encryption (Again!)

Jean-Marie Dirand

`File to be produced containing your source code:  P04_EQ##.*`

In 1854, Sir Charles Wheatstone invents an encoding algorithm encoding baptized " Playfair " in honor of his friend Lyon Playfair, Baron of St.Andrews, who popularized and diffused encoding. Its simplicity and its robustness, compared with the simple substitution techniques caused its immediate success in the field of cryptography, in particular by the British during the Boers and First World Wars. It was then used by several armies during the Second world war as an emergency encoding algorithm. When the PT-109 of the Lieutenant John F. Kennedy was sunk by a Japanese ship off the Solomon Islands, JF Kennedy was able to reach, with the survivors of his crew, the shore of Plum Pudding island in enemy territory. He then transmitted from an ally hut a message encrypted using the Playfair algorithm. The enemy was unable to decrypt the message, and a rescue operation was ordered and carried out successfully recovering all the survivors.

Problem

In order to simplify the problem, the alpabet will consist of just the capital letters "A" to "Z".

To encrypt a message with the Playfair algorithm, it is enough to take a keyword and to write it in a table of size 5×5, deleting multiple occurences of letters and combining letters I and J in the same box. In the following example, we use the keyword `MANCHESTER` and write it in table line by line. Alternatively, it could be written differently, for example column by column or in spiral starting from a corner of the table going towards the center. Following the keyword, the remainder of the letters of the alphabet are introduced according to the alphabetical order.

```
M  A  N  C  H
E  S  T  R  B
D  F  G I/J K
L  O  P  Q  U
V  W  X  Y  Z
```

Once this coding table is determined, it serves to prepare the message for encoding, for the example `THISSECRETMESSAGEISENCRYPTED` (note that the letters constituting the message string are capital letters), encoding is done in the following way: Initially letters are gathered two by two; if a pair consists of the same letter it is necessary to insert a `X` between them and to propagate the shift; if the string ends in a group made up of only one letter then it is necessary to complete the pair with a `X` .

```
TH IS SE CR ET ME SX SA GE IS EN CR YP TE DX
```

Now we have to encrypt each pair. For `TH` , we find `T` and `H` in the table, which delimits a rectangle, and then we locate the letters corresponding to the opposite corners of the rectangle. Here is the diagram:

```
.  .  N  .  H
.  .  T  .  B
.  .  .  .  .
.  .  .  .  .
.  .  .  .  .
```

To encrypt the pair `TH` , we start with the first letter and replace it by its counterpart located at the adjacent corner on the same line, then apply the same thing to the second letter: `TH` becomes `BN` . The process of encoding continues with the next pair, and so on.

```
TH IS SE CR ET ME SX SA GE IS EN CR YP TE DX
BN FR
```

For the pair SE, its letters are on the same line. In such a case

```
.  .  .  .  .
E  S  T  .  .
.  .  .  .  .
.  .  .  .  .
.  .  .  .  .
```

we take the letter at the right-hand side of each letter of the pair, thus `SE` becomes `TS` . Note: if there is overflow of letter at the end of the line, we obtain the letter located in first position of the same line.

```
TH IS SE CR ET ME SX SA GE IS EN CR YP TE DX
BN FR TS
```

The following pair `CR` is on the same column. In such a case we take the letter below each letter. Note: if there is overflow at the end of the column, we obtain the letter located in first position of the same column.

```
.  .  .  C  .
.  .  .  R  .
.  .  . I/J .
.  .  .  .  .
.  .  .  .  .
```

Thus `CR` produces `RI` (Note: `I` and `J` being in the same box, by convention the value of the box is `I`). This is the last special case. The process of encoding continues, producing:

```
TH IS SE CR ET ME SX SA GE IS EN CR YP TE DX
BN FR TS RI SR ED TW FS DT FR TM RI XQ RS GV
```

To decode the message, we proceed in the opposite way: If the two letters of a pair are in different columns and lines, we take the letters of the adjacent corners of the rectangle. If they are in the same line, we take the letters on the left of each one of them (modulo on the same line). If they are in the same column, we take the letters located above each one of them (modulo on the same column).

Input File

The file that you must read is "`P04.ENT` ". It is composed of 3 lines and will contain a message to encrypt or to decode.

Here are the contents of the 3 lines for a file:

Line 1: the keyword

Line 2: operation mode (0 for encryption, 1 for decoding)

Line 3 : The text; a sequence of capital letters of unspecified length.

Example of an input file in a case of encryption :

▷

```
MANCHESTER
0
THISSECRETMESSAGEISENCRYPTED
```

◁

Example of an input file in a case of decoding :

▷

```
MANCHESTER
1
BNFRTSRISREDTWFSDTFRTMRIXQRSGV
```

◁

An example of a file is :

▷

```
MANCHESTER
1
BNFRTSRISREDTWFSDTFRTMRIXQRSGV
```

◁

Output File

You must write the file “P04.SOR” which will contain the answer, a coding table followed by two groups of lines of pairs of capital case characters. The first group consists of the pairs of characters derived from the message in the input file, the second group contains the transcoded pairs.

Encryption Case

The coding table on 5 lines; the letters on each line are separated by a space.

```
M A N C H
E S T R B
D F G I K
L O P Q U
V W X Y Z
```

Followed by the lines containing pairs of characters. Each line contains 40 pairs of characters or less for the last line. The pairs of characters are separated by a space.

```
TH IS SE CR ET ME SX SA GE IS EN CR YP TE DX
```

The coded text made up of pairs. Each line contains 40 pairs of characters or less for the last line.

```
BN FR TS RI SR ED TW FS DT FR TM RI XQ RS GV
```

An example of a complete file is :

▷

```
M A N C H
E S T R B
D F G I K
L O P Q U
V W X Y Z
TH IS SE CR ET ME SX SA GE IS EN CR YP TE DX
BN FR TS RI SR ED TW FS DT FR TM RI XQ RS GV
```

◁

Decoding Case

The coding table on 5 lines; the letters on each line are separated by a space.

```
M A N C H
E S T R B
D F G I K
L O P Q U
V W X Y Z
```

Followed by the lines containing pairs of characters. Each line contains 40 pairs of characters or less for the last line. The pairs of characters are separated by a space.

```
BN FR TS RI SR ED TW FS DT FR TM RI XQ RS GV
```

The coded text made up of pairs. Each line contains 40 pairs of the characters or less for the last line.

```
TH IS SE CR ET ME SX SA GE IS EN CR YP TE DX
```

An example of complete file is :

▷

```
M A N C H
E S T R B
D F G I K
L O P Q U
V W X Y Z
BN FR TS RI SR ED TW FS DT FR TM RI XQ RS GV
TH IS SE CR ET ME SX SA GE IS EN CR YP TE DX
```

◁

It is guaranteed that the message contains between 1 and 2056 characters.

Marking

The problem will be marked by submitting your program to four different input test files.

A test will be regarded as successful if the program manages to find the solution and correctly write it in the outputfile in the proper format.

Nombre de fichiers réussis	Points accordés
Successful test file	Points
1	120
2	180
3	240
4	300

maximum execution time for your program : 10 seconds.

Specification 5

Computer Networks

Lourdes Zubieta

`File to be produced containing your source code: P05_EQ##.*`

Suppose someone asks you to connect a number of computers by using the minimum amount of cable in order to minimize the cost.

We consider the computers of the network to be connected as the nodes of a graph. The possible links between two nodes (computers) are the edges of this graph. Each edge e has a corresponding "length" or cost attached $\lambda(e)$.

Figure 5.1 shows a network example. The possible alternatives and the distances between the nodes to be connected are in table 5.1. The number of computers to be connected is: N=7

Start	End	Distance
1	2	15
1	3	40
1	4	30
1	5	65
1	6	60
2	5	45
3	5	30
3	7	25
4	6	20
5	7	35
6	7	15

Table 5.1: The table representing the graph

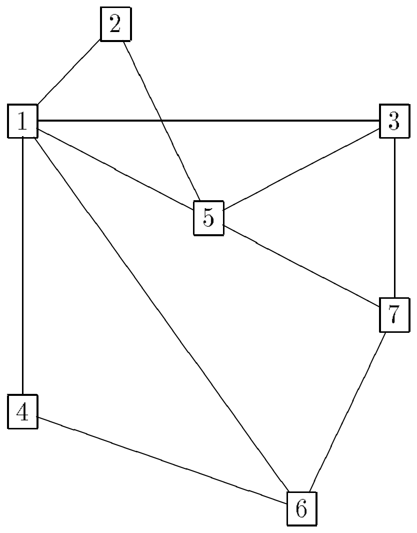


Figure 5.1: A network

Problem

The problem is to determine which edges, among the possibilities given in the file containing the table, should be included so that a minimum amount of cable is used. We must connect **all** the clients with **a** single path. For example, the solution {(1,2),(2,5),(1,3),(3,5),(3,7),(7,6),(6,4)} visits client 5 twice and therefore is not acceptable. Another example, {(1,2), (1,3), (1,4), (1,5), (1,6)} is not good since client 7 is not connected.

Input File

The file you must read is "`P05.ENT`". It contains lines giving the possible edges between nodes and their costs. Each line contains three integers, the first is the source node, the second is the destination node and the third is the cost. The integers are separated by one or more spaces.

▷

```
1 2 15
1 3 40
1 4 30
1 5 65
1 6 60
2 5 45
3 5 30
3 7 25
```

```
4 6 20
5 7 35
6 7 15
```

◁

We can have between 2 and 1000 nodes or computers.

Output File

You must write the file "`P05.SOR`" that will contain the nodes composing the minimal network in the same format (three integers) and on the last line the total cost. The order of the lines is determined by the cost of the corresponding edge, the smallest cost must appear first then in ascending order. If two edges are equal, the lines can appear in the file in any order. The total cost fits in a 32-bit integer.

Here is the output file for the example input file :

▷

```
1 2 15
6 7 15
4 6 20
3 7 25
3 5 30
1 3 40
145
```

◁

If there are several solutions of minimum cost only one must appear in the output file.

Marking

The problem will be marked by submitting your program to four different input test files.

A test will be regarded as successful if the program manages to find the solution and correctly write it in the outputfile in the proper format.

Nombre de fichiers réussis	Points accordés
Successful test file	Points
1	120
2	180
3	240
4	300

Maximum execution time for your program : 10 seconds.

Specification 6

Data Decompression - The Huffman Code

Martine Bellaïche

`File to be produced containing your source code:  P06_EQ##.*`

Problem

In computer science, the technological developments and the users requirements lead to increasingly large amount of data. In order to put the data in a format such that they occupy less memory space, multiple studies were undertaken on the compression algorithms. Once compressed, the data is not any longer directly accessible, and it has to be decompressed so that it becomes understandable. Data compression is vital in order to decrease the size of the files and the transfer time of files through modems.

An algorithm largely used in the data compression is the Huffman algorithm, that generates binary codes of variable lengths. Starting from the original data, a statistic study bearing on the frequency of the characters present in the file is carried out, then the Huffman algorithm assigns to each character a binary code. The more frequent the character is, the shorter (less number of bits) the code is. Moreover, no code associated with any character is a prefix for another.

Our problem is to decompress a file already compressed by the codes generated by the Huffman algorithm. In order to decompress a file, there is a heading at the beginning of the file which contains all information useful for compression: the binary codes, and numbers it bits associated with the characters.

Example

The file to compress is:

```
bafedaffa

Here is is the binary Huffman code for every character of the
file:

b   a  f  e   d  a  f  f  a
010 00 10 011 11 00 10 10 00
```

We gather the bits byte by byte to find the contents of the compressed data.

```
0100 0100        1111 0010        1000
code ascii 68    code ascii 242  code ascii 8
```

Table 6.1: ASCII charcters table and Huffman binary code

Ascii code	a	b	d	e	f
binary code	00	010	11	011	10
number of bits	2	3	2	3	2

Input File

The file you must read is "`P06.ENT`". **Be careful, the file is in binary format. Here we give the explanations necessary to understand it. You will have the real input file on your computer during the Race**.

The binary file has the following structure, byte by byte:

bytes 1 and 2: the number of characters of the original file;

bytes 3 and 4: the number of coded characters;

bytes 5 and 6: the largest number of bits in the binary codes;

In ASCII order and for all the coded characters, we have 2 bytes for the ASCII character, 2 bytes for the Huffman binary code, and 2 bytes for the number of bits in the Huffman binary code;

Then, byte by byte, we have the compressed data.

Here is an example of an input file given in hexadecimal where we deliberately inserted a space between each byte for better presentation. This space is not stored in the file.

```
00 2f 00 11 00 06 00 20 00 00 00 02 00 2c 00 3e 00 06 00 2e 00 1e
00 06 00 61 00 0a 00 04 00 63 00 3f 00 06 00 64 00 1f 00 06 00 65
00 0e 00 04 00 66 00 0c 00 05 00 69 00 02 00 03 00 6c 00 0d 00 05
00 6e 00 0b 00 04 00 6f 00 1a 00 05 00 70 00 1b 00 05 00 72 00 04
00 03 00 73 00 0e 00 05 00 74 00 0c 00 04 00 75 00 1e 00 05 8b ac
be 1d d3 07 f8 ff 5e 8a 7c 26 8c af 61 ba 98 a1 46 f4 af 1e
```

`00 2f`: the number of characters of the original file, that is, in decimal, `47`.

`00 11`: the number of coded characters, that is, in decimal, `17`.

`00 06`: the largest number of bits in the binary codes, that is, in decimal, `6`.

`00 20 00 00 00 02`: ASCII: `32`, code: `0`, bits:2.

`00 2c 00 3e 00 06`: ASCII: `44`, code: `62`, bits:6.

`00 2e 00 1e 00 06`: ASCII: `46`, code: `30`, bits:6.

`00 61 00 0a 00 04`: ASCII: `97`, code: `10`, bits:4.

`00 63 00 3f 00 06`: ASCII: `99`, code: `63`, bits:6.

`00 64 00 1f 00 06`: ASCII: `100`, code: `31`, bits:6.

`00 65 00 0e 00 04`: ASCII: `101`, code: `14`, bits:4.

`00 66 00 0c 00 05`: ASCII: `102`, code: `12`, bits:5.

`00 69 00 02 00 03`: ASCII: `105`, code: `2`, bits:3.

`00 6c 00 0d 00 05`: ASCII: `108`, code: `13`, bits:5.

`00 6e 00 0b 00 04`: ASCII: `110`, code: `11`, bits:4.

`00 6f 00 1a 00 05`: ASCII: `111`, code: `26`, bits:5.

`00 70 00 1b 00 05`: ASCII: `112`, code: `27`, bits:5.

`00 72 00 04 00 03`: ASCII: `114`, code: `4`, bits:3.

`00 73 00 0e 00 05`: ASCII: `115`, code: `14`, bits:5.

`00 74 00 0c 00 04`: ASCII: `116`, code: `12`, bits:4.

`00 75 00 1e 00 05`: ASCII: `117`, code: `30`, bits:5.

`8b ac be 1d d3 07 f8 ff 5e 8a 7c 26 8c af 61 ba 98 a1 46 f4 af 1e`
are the bytes of the compressed file.

Ouput File

You must write the file "`P06.SOR`". In the file, we find the decompressed data in an intelligible format.

Here is the output file for the example above:

```
▷
rien ne sert de courir, il faut partir a point.
◁
```

Marking

The problem will be marked by submitting your program to four different input test files.

A test will be regarded as successful if the program manages to find the solution and correctly write it in the outputfile in the proper format.

Nombre de fichiers réussis	Points accordés
Successful test file	Points
1	160
2	240
3	320
4	400

Maximum execution time for your programme : 10 seconds.

Specification 7

Email Transfer Protocol: SMTP

Martine Bellaïche

`File to be produced containing your source code: P07_EQ##.*`

This problem is inspired by a problem of the 23rd Programming Contest of the ACM.

Problem

We will describe to you the bidirectional communication protocol for transfer of electronic mail SMTP (Simple Mail Transfer Protocol). A user sender sends an electronic mail containing SMTP commands and a message to another user receiver. The SMTP Transmitter receives this mail and establishes a bidirectional communication with the SMTP receiver. Once, the communication established, the SMTP transmitter transmits one command at a same time to the SMTP receiver, which sends back a 3-digit response according to the transmitted command. The goal of the program is to find the dialogue of the commands interchanged between the SMTP transmitter and the SMTP receiver. The SMTP server will play at the same time the role of the transmitter and the receiver. The commands that the SMTP transmitter and SMTP receiver transmit, are defined in tables 7.1 and 7.2. The commands of SMTP transmitter and receiver are case sensitive. In order to write the program, it is important to know the characteristics of SMTP servers. The characteristics of the server `montreal.ca` are :

- it's name: `montreal.ca`.
- the list of its local users. For example: `destin, leduc, bier, lange`. Their email addresses are `destin@montreal.ca, leduc@montreal.ca, bier@montreal.ca` and `lange@montreal.ca`
- the list of the local users who forwarded their addresses to another address where each transferred address is specific to each of the users. For example, `destin destin@paris.fr` means

that the local user `destin` with the server `montreal.ca` forwarded his mail to `destin@paris.fr` and the alias `bier bier@milan.it` means that the local user `bier` forwards to the address `bier@milan.it`.

- aliases correspond to a local or non-local user. If a user sending from an SMTP server sends a message to simply a name, then that the server looks among its aliases for the electronic address. For example, the alias `lele leduc` corresponds to the local user `leduc` and the alias `mama mancini@paris.fr` to the non-local user `mancini@paris.fr`

Table 7.1: SMTP Transmitter Commands

`HELO`	Name	Name of the SMTP transmitter to start the communication
`MAIL FROM`	sender	Electronic Address of the user sender
`RCPT TO`	recipient	Address of the recipient user.
`VRFY`	User alias	Find the alias' complete electronic address
`DATA`		The following lines contain the message consisting of the date, the subject and the text of the message after the command `354` of the receiver. The character `*` in first column indicates the end of the message
`QUIT`		Ends the communication

Table 7.2: Commands of the SMTP receiver

`221`		After a `QUIT` Closing of the connection
`250`	ok	After a `HELO, MAIL FROM` or a `RCPT TO` The command is well received.
`250`	Alias' local electronic Address	After a `VRFY`
`251`	forward to another non-local electronic address	After `VRFY` the SMTP receiver indicates to the SMTP transmitter the forward address, without the SMTP transmitter opening a new connection with the SMTP server of the forward address.
`354`	start email	After the command `DATA`
`550`	no user	After `RCPT TO` means that the destination user on the receiving server is unknown

We will describe to you another SMTP server `paris.fr` in order to show you the dialogue between that it and the server `montreal.ca`.

- it's name: `paris.fr`.
- the list of its local users: `destin, mancini, gagne, kasper`. Their email addresses are `destin@paris.fr`, `mancini@paris.fr`, `gagne@paris.fr` and `kasper@paris.fr`

Example of communication between `paris.fr` and `montreal.ca`. The SMTP server `paris.fr` plays the sender role, and the server `montreal.ca` is the recipient. The sending user is `gagne@paris.fr` who sends a message to the users `lange@montreal.ca`, `leclaire@montreal.ca` and `bier@montreal.ca`.

```
HELO paris.fr
250 ok
MAIL FROM gagne@paris.fr
250 ok
RCTP TO lange@montreal.ca
250 ok
RCPT TO leclaire@montreal.ca
550 no user
RCTP TO bier@montreal.ca
251 forward to bier@milan.it
.......
......
```

Input File

The file that you must read is "`P07.ENT`".

The input file contains initially

- the number of SMTP servers,
- the description of all the SMTP servers,
- the number of messages,
- the messages.

On the line describing the SMTP server, one finds its name, the number of its users, then on each line a reserved word followed by the name of its email users. The various reserved words are:

- `local` user local to the server.
- `forward` local user followed by a forward address.
- `alias` user_name followed by a local or non-local electronic address.

We have at least, the description of the server.

Then, for the lines describing the message, we have:

- the reserved word `from` always followed by the electronic address of the sending user
- the reserved word `to` followed by the number of the destination users and the list of their email addresses. If we have only the name of a local user or of an alias, then the server is the same as the sending user. We have at least one destination user.
- the reserved word `date` followed by the date and time of the message.
- the reserved word `subject` followed by the subject of the message.
- the message text.
- to end the message text, we find the character `*` in the first column and on a line by itself.

The servers of the users' electronic addresses are always valid and defined in the description of the server.

Here is an example of an input file:

```
2
montreal.ca 7
local destin
local leduc
local bier
local lange
forward destin destin@paris.fr
alias lele leduc
alias mama mancini@paris.fr
paris.fr 5
local destin
local mancini
local gagne
local hudon
alias gaga gagne
3
from destin@montreal.ca
to 2 hudon@paris.fr lange@montreal.ca
date 12/05/00
subject un petit bonjour
Comment vas-tu ?
Fait-il beau chez vous ?
*
from lange@montreal.ca
to 1 lele
date 02/08/00
subject confirmation
veuillez confimer
votre date d'arrivÈe
```

```
Paul
*
from bier@montreal.ca
to 3 destin pierre@montreal.ca mama@montreal.ca
date 02/08/00
subject Hello
Donnez nous de vos
nouvelles
Sophie
*
```

Output File

You must write the file "P07.SOR" that will contain the dialog between the SMTP transmitter and receiver.

On each line, we find the SMTP server commands, according to tables 7.1 and 7.2. The commands are case-sensitive.

Here is the output file for the example above:

```
HELO montreal.ca
250 ok
MAIL FROM destin@montreal.ca
250 ok
RCPT TO lange@montreal.ca
250 ok
DATA
354 start email
date 12/05/00
subject un petit bonjour
Comment vas-tu ?
Fait-il beau chez vous ?
QUIT
221
HELO montreal.ca
250 ok
MAIL FROM destin@montreal.ca
250 ok
RCPT TO hudon@paris.fr
250 ok
DATA
354 start email
date 12/05/00
subject un petit bonjour
Comment vas-tu ?
```

```
Fait-il beau chez vous ?
QUIT
221
HELO montreal.ca
250 ok
MAIL FROM lange@montreal.ca
250 ok
VRFY lele
250 leduc@montreal.ca
DATA
354 start email
date 02/08/00
subject confirmation
veuillez confimer
votre date d'arrivÈe
Paul
QUIT
221
HELO montreal.ca
250 ok
MAIL FROM bier@montreal.ca
250 ok
RCPT TO destin@montreal.ca
250 ok
RCPT TO pierre@montreal.ca
550 no user
VRFY mama
251 forward to mancini@paris.fr
DATA
354 start email
date 02/08/00
subject Hello
Donnez nous de vos
nouvelles
Sophie
QUIT
221
```

Marking

The problem will be marked by submitting your program to four different input test files.

A test will be regarded as successful if the program manages to find the solution and correctly write it in the outputfile in the proper format.

Nombre de fichiers réussis	Points accordés
Successful test file	Points
1	160
2	240
3	320
4	400

Maximum execution time for your program : 10 seconds.

Specification 8

Error Detecting Codes

Gaétan Haché

`File to be produced containing your source code:  P08_EQ##.*`

This is no secret: in any data transmission system there is always a possibility that the received message is different from the one that was sent. Moreover, it is impossible to verify with 100% certainty that there were no transmission error. Nevertheless, we know from Shannon's Theorem that it is possible to verify with a certainty of 100% - ϵ, $\epsilon > 0$, whether there were any transmission errors. In other words, it is possible to reduce as much as we want the probability of not detecting errors when some indeed occurred. We call this probability the *probability of not detecting an error* (or simply the error probability) and it is denoted by p_{ndec}. Of course there is a price to pay for reducing p_{ndec}: you must add redundancy to the message. For example, say you are using a binary data transmission channel (telephone line, satellite telecommunication, etc.) with a symmetric error probability of $p = 0.10$. This is not very good (unless you're "talking" or receiving data form a satellite outside the solar system...). This symmetric probability of $p = 0.10$ means that one time out of ten you receive a 0 (respectively a 1) when 1 (respectively 0) was effectively transmitted. You really don't like that so you decide that all messages should be doubled: so to send the message "1" the binary word 11 is sent and the same hold for sending the message "0", that is, 00 is sent. This way if you receive 01 or 10 you are 100% sure that transmission errors occured. But if you receive 00 nothing tells you that 00 was effectively sent so that $p_{\mathrm{ndec}} \neq 0$.

In this problem, instead of calculating p_{ndec}, we will perform an intermediate computation that, once obtained, yields directly p_{ndec} in most solutions used in practice to detect errors.

In order to understand the problem we give a few examples of what we call *error detecting codes.* For those of you who know about this, they are in fact called error correcting code since not only you can detect errors with such codes but you can correct the errors as well.

Example 1: Repeating the message

We consider again the idea we have just seen: to send the message "1" the binary word 11 is sent and the same hold for sending the message "0" that is 00 is sent. To compute p_{ndec} we make the following observation: since the channel is symmetric (i.e. the probability p of error transmission is the same whether a 1 or a 0 is transmitted) the probability of receiving 11 while 00 was sent or receiving 00 while 11 was sent is the same in both cases: this probability is $p_{\text{ndec}} = p^2$, thus if $p = 0.10$ then $p_{\text{ndec}} = 0.01$.

What we have just done here is constructing a error detecting code of *length* 2, that is $C_2 = \{00, 11\}$. A transmission error is detected whenever a received binary word of length 2 is not in C_2.

Exemple 2 : the parity bit

We consider now the code of length 4

$$C_4 = \{0000, 0011, 0101, 0110, 1001, 1010, 1100, 1111\}.$$

Observe that this code is exactly the set of all binary word of length 3 to which a parity bit was added so that the number of 1 in the word is an even number. Observe that the error patterns that transform the word 0000 into another word of C_4 are exactly the same that transform any word of C_4 into another one. An *error pattern* is a binary word indicating in which position an error occurred. For example, the pattern 1010 transforms the word 0000 into 1010 which is also an word of C_4. Similarly, this same pattern transform $0110 \in C_4$ into 1100 which is also in C_4. We also observe that the error patterns that transform a word of C_4 into another word of C_4 are themselves words of C_4. This nice property comes from the fact that the code is *linear*. Hence, without loss of generality we can assume for the calculation of p_{ndec} that the word that was sent is 0000: in this case an error is not detected if any of the other words of C_4 is received. Hence the probability of not detecting an error is the probability of receiving one of 0011, 0101, 0110, 1001, 1010, 1100 ou 1111 which is

$$p_{\text{ndec}} = (1-p)^2p^2+(1-p)p(1-p)p+(1-p)p^2(1-p)+p(1-p)^2p+p(1-p)p(1-p)+p^2(1-p)^2+p^4 = 6p^2(1-p)^2+p^4.$$

Before going any further we need the following definition: we call the *weight* of a binary word the number of 1 in the word. Hence in the code C_4 above there is no word of odd weight, there is 1 word of weight 0, 6 of weight 2 and 1 of weight 4.

Exemple 3 - The double parity bit

Consider C_8 the set of all binary words of length 6 to which we add 2 bits according to the following rule: $(a_1, a_2, a_3, a_4, a_5, a_6, b_1, b_2) \in C_8$ if an only if a_1, a_2, a_3, b_1 has even weight and so has a_4, a_5, a_6, b_2. In other words, b_1 is a parity bit over the 3 first bits and and so is b_2 over the next three bits.

It is easy to verify that C_8 contains no word of odd weight and that it contains exactly 1 word of weight 0, 6 of weight 2, 9 of weight 4 and none of weight 6. One again the code is linear and using the same arguments as we did in the previous example we find:

$$p_{\text{ndec}} = 6p^2(1-p)^6 + 9p^4(1-p)^4.$$

In general, for a code C of length n if we denote $w_C(i)$ the number of word of weight i in the code C then

$$p_{\text{ndec}} = \sum_{i=1}^{n} w_C(i)p^i(1-p)^{n-i}.$$

Hence to compute p_{ndec} it is enough to compute $w_C(i)$ for $i = 1, 2, \dots, n$ where n is the length of the code. This sequence of integer, that is

$$w_C(1), w_C(2), w_C(3), \dots, w_C(n),$$

is called the *weight distribution* of C.

You will agree that the examples we have just seen are "small". In practice the codes that are used are very long. For example, a CD player uses codes of length up to $n = 256$ and the number of words in the code is way over 2^{200}. Obviously it is not practical, if not impossible, to enumerate all the words in a code to check if a received word belongs to the code or not. We use instead *parity check matrices*. For example, the code C_8 above has the following parity check matrix:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{bmatrix}$$

This matrix is used as followed: a word $(c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8)$ is in C_8 if and only if

1. $(1, 1, 1, 0, 0, 0, 1, 0) \cdot (c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8)$ has even weight and
2. $(0, 0, 0, 1, 1, 1, 0, 1) \cdot (c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8)$ also has even weight

where the operation $\cdot$ consists of applying, bit to bit, the "and" defined as usual: by $0 = 0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0$ and $1 = 1 \cdot 1$.

For example,

$$(1, 1, 1, 0, 0, 0, 1, 0) \cdot (0, 1, 1, 1, 1, 0, 0, 0) = (0, 1, 1, 0, 0, 0, 0, 0)$$

and

$$(0, 0, 0, 1, 1, 1, 0, 1) \cdot (0, 1, 1, 1, 1, 0, 0, 0) = (0, 0, 0, 1, 1, 0, 0, 0)$$

are two words of even weight thus $(0, 1, 1, 1, 1, 0, 0, 0) \in C_8$.

On the other hand, since $(1, 1, 1, 0, 0, 0, 1, 0) \cdot (1, 1, 1, 1, 0, 0, 0, 0) = (1, 1, 1, 0, 0, 0, 0, 0)$ has odd weight, the word $(1, 1, 1, 1, 0, 0, 0, 0)$ is not in C_8.

In general, if

$$H = \begin{bmatrix} h_{11} & h_{12} & \cdots & h_{1n} \\ h_{21} & h_{22} & \cdots & h_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ h_{k1} & h_{k2} & \cdots & h_{kn} \end{bmatrix}$$

is the parity check matrix of a code C and $c = (c_1, c_2, \dots, c_n)$ is a binary word of length n then

$$c \in C \iff (h_{j1}, h_{j2}, \dots, h_{jn}) \cdot (c_1, c_2, \dots, c_n) \text{has even weight for } j = 1, 2, \dots k.$$

Problem

Your problem is to write a program that will compute the weight distribution of codes of length 32 which parity check matrix will be given in the input file.

It is very difficult to go through all words of length 32 in less thant one minute (you would need to go through about 71.5 millions in less than a second). Nevertheless, your program should be able to compute the partial weight distribution

$$w_C(0), w_C(1), \ldots w_C(l)$$

for $l \leq 6$. If your program is efficient then it should be able to compute up to $l = 7$ if not $l = 8$.

Input file

The file you must read is "`P08.ENT`".

In this file, the first line contains an integer indicating up to which weight you must compute the partial weight distribution of the code which parity check matrix is given on the following lines. The "n" indicates that the preceding line is the last line of the parity check matrix. After the "n" you will either find the input for another weight distribution calculation or an "F" indication that there is no more computation to perform.

Following is an example of an input file followed by the corresponding output file.

▷

```
5
11111111111111111111111111100000
01111111111111111111111111110000
00111111111111111111111111111000
00011111111111111111111111111100
00001111111111111111111111111110
00000111111111111111111111111111
n
6
10101010101010101010101010101010
11001100110011001100110011001100
11110000111100001111000011110000
11111111000000001111111100000000
11111111111111110000000000000000
11111111111111111111111111111111
n
F
```

◁

Note: there is no blank between 0 and 1 in the line of a parity check matrix.

Output file

You must write a file called "`P08.SOR`" in which each line will contain the partial weight distributions of the codes which parity check matrices were given in the input file.

Here is the output file for the example :

▷

```
1 0 231 110 7325 7700
1 0 0 0 1240 0 27776
```

◁

Marking

The problem will be marked by submitting your program to four different input test files.

A test will be regarded as successful if the program manages to find the solution and correctly write it in the outputfile in the proper format.

Nombre de fichiers réussis	Points accordés
Successful test file	Points
1	160
2	240
3	320
4	400

Maximum execution time for your program : 20 seconds.

Specification 9

Binary Crosswords

Daniel Dalle

`File to be produced containing your source code:  P09_EQ##.*`

One proposes to evaluate an algorithm of correction of error for a system of telecommunications which takes as a starting point the the principle of redundancy that one finds in the cross words.

The message to be transmitted on a channel of communication is appeared as a bit train.

$$\text{message } M \quad \Rightarrow \quad \text{communication system} \quad \Rightarrow \quad \text{message } M'$$

Where the communication system is made up as :

$$M \quad \Rightarrow \quad \text{coding} \quad \Rightarrow \quad \text{transmission} \quad \Rightarrow \quad \text{link} \quad \Rightarrow \quad \text{reception} \quad \Rightarrow \quad \text{decoding} \quad \Rightarrow \quad M'$$

In the communication link, errors can occur that deteriorate the binary message. If there are no errors, $M = M'$. If there is, then $M \neq M'$. However, one can code the message M in order to detect and correct a certain number of errors.

Note: we do not claim to present here a powerful algorithm of error correction, this is merely a programming exercise.

The binary stream to code is divided into packets of N^2 bits, which makes it possible to present them like a square of $N \times N$ bits. The packet is made of bits of the original message placed line after line in the reading order of the characters of a text. To transmit this " square " of bits, the system sends initially the bits line after line and then, it repeats the message column after column. At the end of each line and each column, the system inserts an error detection code of M bits. The detection code is evaluated according to an algorithm described hereafter.

The corresponding system of reception decodes the message according to a reciprocal logic. The error detection code received makes it possible to know if a line or a column contains an error by comparing it with the calculated code.

To illustrate the principle, here is an example of a small size message of 64 bits. The binary message is `< b1 b2 .   .   .  b63 b64 >`. The message is represented in a square of $N \times N$ bits. In the example, N equals 8.

```
 b1  b2  b3  b4  b5  b6  b7  b8

 b9 b10 b11 b12 b13 b14 b15 b16

b17 b18 b19 b20 b21 b22 b23 b24

b25 b26 b27 b28 b29 b30 b31 b32

b33 b34 b35 b36 b37 b38 b39 b40

b41 b42 b43 b44 b45 b46 b47 b48

b49 b50 b51 b52 b53 b54 b55 b56

b57 b58 b59 b60 b61 b62 b63 b64
```

It is equivalent to a crossword grid. In a received message there are uncertainties because several bits may be deteriorated by transmission errors. One adds redundant information to detect and correct these errors.

Considering this error possibility, the message is transmitted twice : first, line by line and then, column by column, while adding after each line and each column an error detection code error of M bits.

The first time, line by line:

```
 b1  b2  b3  b4  b5  b6  b7  b8  < horizontal detection code 1 >

 b9 b10 b11 b12 b13 b14 b15 b16  < horizontal detection code  2 >

b17 b18 b19 b20 b21 b22 b23 b24  < horizontal detection code  3 >

b25 b26 b27 b28 b29 b30 b31 b32  < horizontal detection code  4 >

b33 b34 b35 b36 b37 b38 b39 b40  < horizontal detection code  5 >

b41 b42 b43 b44 b45 b46 b47 b48  < horizontal detection code  6 >

b49 b50 b51 b52 b53 b54 b55 b56  < horizontal detection code  7 >
```

```
b57 b58 b59 b60 b61 b62 b63 b64  < horizontal detection code  8 >
```

The second time, column by column:

```
 b1  b9 b17 b25 b33 b41 b49 b57  < vertical detection code  1 >

 b2 b10 b18 b26 b34 b42 b50 b58  < vertical detection code  2 >

 b3 b11 b19 b27 b35 b43 b51 b59  < vertical detection code  3 >

 b4 b12 b20 b28 b36 b44 b52 b60  < vertical detection code  4 >

 b5 b13 b21 b29 b37 b45 b53 b61  < vertical detection code  5 >

 b6 b14 b22 b30 b38 b46 b54 b62  < vertical detection code  6 >

 b7 b15 b23 b31 b39 b47 b55 b63  < vertical detection code  7 >

 b8 b16 b24 b32 b40 b48 b56 b64  < vertical detection code  8 >
```

The detection codes are described hereafter.

By analogy with crosswords, the horizontal detection codes act as the the horizontal definitions and the vertical codes as the vertical definitions.

We can find an algorithm which exploits this redundancy of information to eliminate the most possible uncertainties and which repeatedly improve the solution just like in crosswords.

If a message is longer than $N \times N$ bits, it is segmented and the process is repeated on several consecutive packets. If the last packet is incomplete, we complement the message with `0`s.

In the problem to be treated, the dimension N of the square is fixed at 32 and the dimension M of the detection codes is 16.

Example : The example given at the end of the problem statement problem corresponds to a message of 1024 bits with a square of 32×32 and detection codes of 16 bits. The original message of the example, shown line by line:

```
10000000000000000000000000000000
11000000000000000000000000000000
11100000000000000000000000000000
11110000000000000000000000000000
11111000000000000000000000000000
11111100000000000000000000000000
11111110000000000000000000000000
11111111000000000000000000000000
11111111100000000000000000000000
```

```
11111111110000000000000000000000
11111111111000000000000000000000
11111111111100000000000000000000
11111111111110000000000000000000
11111111111111000000000000000000
11111111111111100000000000000000
11111111111111110000000000000000
11111111111111111000000000000000
11111111111111111100000000000000
11111111111111111110000000000000
11111111111111111111000000000000
11111111111111111111100000000000
11111111111111111111110000000000
11111111111111111111111000000000
11111111111111111111111100000000
11111111111111111111111110000000
11111111111111111111111111000000
11111111111111111111111111100000
11111111111111111111111111110000
11111111111111111111111111111000
11111111111111111111111111111100
11111111111111111111111111111110
11111111111111111111111111111111
```

This example contains a very regular pattern but the test messages are completely random.

Calculation of the error detection codes: CRC 16 bits CCITT

The error-correcting code CRC corresponds to the remainder of a polynomial division of polynomials whose coefficients are the bits of the message by a constant polynomial. In practice the calculation of the CRC is carried out very simply with single bits memory cells connected as a shift register on which we apply XOR functions at each occurrence of a bit of the message according to the following diagram :

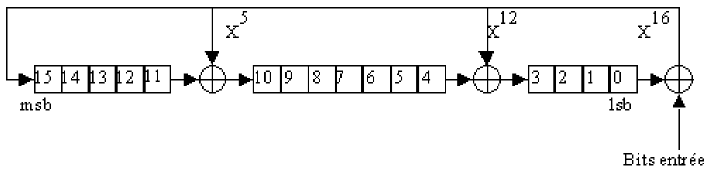


Figure 9.1: Code CRC CCITT 16 bits

The operators $\oplus$ represent exclusif ORs (XOR).

The CRC is initialized to 0. For each bit of the message, the shift register is updated as follows: Example of evolution :

```
     sequence                     CRC
     message       MSb                         LSb
                   0 0 0 0 0   0 0 0 0 0 0   0 0 0 0

        ⇓

bit 1   1          1 0 0 0 0   1 0 0 0 0 0   1 0 0 0
bit 2   0          0 1 0 0 0   0 1 0 0 0 0   0 1 0 0
bit 3   1          1 0 1 0 0   1 0 1 0 0 0   1 0 1 0
...
etc ...
```

At the end of the message, the CRC is appended to the message starting with its LSb (least significant bit) Example :

```
Line of message:      1011101000010011000001000000000
Calculated CRC:       1011 0001 1111 0010
CRC from LSb to MSb   0100 1111 1000 1101
Transmitted code:
message------------------------- CRC-------------
1011101000010011000001000000000 0100111110001101
```

When the CRC is placed in this way, we observe the very useful property that a new CRC2 calculated on the message concatenated with the first CRC1 has a null value. If a bit was incoherent with the CRC1, it would not be the case.

```
The new CRC calculated on the message concatenated with its first CRC:
message------------------------- CRC1------------
message + CRC1---------------------------------- CRC2------------
1011101000010011000001000000000 0100111110001101 0000000000000000
```

Problem

To write a program that decodes the messages coded and altered by errors during the transmission. The program must detect the largest possible number of errors.

- At the input: a file including a coded and deteriorated message. One does not know in advance where the erroneous bits are; they can be either in the message itself or in the detection codes.
- At the output: a file including only the original message decoded in which as much errors as possible are corrected. The program receives only the coded and deteriorated file.

The test is successful if all the errors are corrected. We guarantee that all the test files can be corrected completely.

Input File

The file you must read is “`P09.ENT`”. The files are text files, they contain ASCII characters `0` ou `1`. The characters other than 0 or 1 and the line feeds must be ignored. Only the sequences of bits are considered. The line feeds are present for the legibility of the files only.

The following file represents the example message coded et altered by errors.

▷

```
100000000000000000000011000000001101110100111000
110000000000000100000000000000001011001110100100
111000000000000000000000000000001000010011101010
111100000000000000000000000000001001101101001101
111110000000000000000000000000000001101010001110
111111000000000000000000000000001101000001111111
111111100000000000000000000000000010110100010111
111111110000000000000000000000000100101110100011
111111111000000000000000000000000111000011011001
111110111100010000000000000000000110110101010100
111111111110010000000000000000001110101110010010
111111111111000000000000000000001010100011110001
111111111111100000000000000000000000000101010000
110111111111111000000000000000001101110110010000
111111111111111000000000000000001011001111110000
111111111110111100000000000000001000010011000000
111110111111111110000000000000001001111101011000
111111111111111111000000000000001001001010010100
111111111111111011100000000000001001010001110010
111111111111111111110000100000001001011100000001
111111111111111011111000000000000001111010101000
111111111111111111111100000000001101001001001100
111111111111111111111110000000001011010000001110
111101111111111111111111000000001000011100111111
111111111111111111111111100000000001011010110111
111111111111111111111111110000000101111001110011
111111111111111111111111111000000111101000010001
111111111111111111111111111100000110100000100000
111111111111111111111111111110001110100100101000
111111111111111111111111111110001010100110101100
111111111111111111111111101111101000110111101110
111111111111111111111111111111111101100111001111

111111111111111111111111111111111001100111011111
011111111111111101111111111111110100010011110111
001111111111111111111111111111110010101001101011
000111111111111111111111111111110001110100100101
000011111111111111101111111111110000011010000010
```

```
000001111111111111111111111111111000001101000001
000000111111111111111111111111110100100110110000
000000001111111111111111111111111010010011011000
000000001111111111111111111111111101001001101100
000000000111111111111111111111111110110100110110
000000000011111111111111111111111111010010011011
000000000001111111111111111111110111001001011101
000000000000111111111111111111110011000100111110
000000000000011111111111111111111001100010011111
000000000000001111111111111111110100010001011111
000000000000000111111111111111110010101000111111
000000000010000011111111111111110001110100001111
000000000000000001111110111111110000111010010111
000000000000000000111111111111110000101101011011
000000000000000000011111111111110000010100111101
000000000000000000001111111111110000111011001110
000000000000000000000111111111111000011101100111
000000000000000000000011111101110100101110100011
000000000000000000000001111111110010110111000001
000000000000000000000000111111110001111011110010
000000000000000000000000011111111000111101111000
000000000000000000000000001111111101011110111100
000000000000000000000000000111111110001111011110
000000000000000000000000000011111111000111101111
000000000000000000100000000001110111000011100111
000000000100000000000000000000110011000001100011
000000000000000000000000000000010001000000100001
```

◁

There are 36 altered bits out of 3072 bits which form the complete coded message in this example (an alteredbit is changed from 1 to 0 or reciprocally). The altered bits can be bits of message or the detection codes.

The first part of these files represents the message line by line. At the end of each line the 16-bit error-correcting code is inserted.

The second part repeats this message column by column. At the end of each line of this part, we find the 16-bit error-correcting code of the line which corresponds to a column of the original table.

Output File

The file you must produce is "`P09.SOR`". It contains the decoded message in the same format as the original message with the corrected errors. The algorithm of this example did not leave any residual error.

Here is the output file corresponding to the example input file :

▷

```
10000000000000000000000000000000
11000000000000000000000000000000
11100000000000000000000000000000
11110000000000000000000000000000
11111000000000000000000000000000
11111100000000000000000000000000
11111110000000000000000000000000
11111111000000000000000000000000
11111111100000000000000000000000
11111111110000000000000000000000
11111111111000000000000000000000
11111111111100000000000000000000
11111111111110000000000000000000
11111111111111000000000000000000
11111111111111100000000000000000
11111111111111110000000000000000
11111111111111111000000000000000
11111111111111111100000000000000
11111111111111111110000000000000
11111111111111111111000000000000
11111111111111111111100000000000
11111111111111111111110000000000
11111111111111111111111000000000
11111111111111111111111100000000
11111111111111111111111110000000
11111111111111111111111111000000
11111111111111111111111111100000
11111111111111111111111111110000
11111111111111111111111111111000
11111111111111111111111111111100
11111111111111111111111111111110
11111111111111111111111111111111
```

◁

Marking

The problem will be marked by submitting your program to four different input test files.

A test will be regarded as successful if the program manages to find the solution and correctly write it in the outputfile in the proper format.

Nombre de fichiers réussis	Points accordés
Successful test file	Points
1	240
2	360
3	480
4	600

Maximum execution time for your program : 10 seconds.

Specification 10

The puzzle

Charles-Antoine Brunet

```
   File to be produced containing your source code:  P10_EQ##.*
```

Most of us have played with those small square puzzles where one must move the tiles one at a time in order to reconstitute an image. The image must be reconstituted by moving repeatedly a tile into the adjacent empty cell until the final image is obtained. These puzzles were invented by Sam Loyd in 1878.

Problem

The goal is to move tiles from an initial state or arrangement, until the desired final state is reached . The only allowed moves are a horizontal or vertical displacement of a tile into the free adjacent cell. The figure 10.1 shows some examples of possible moves.

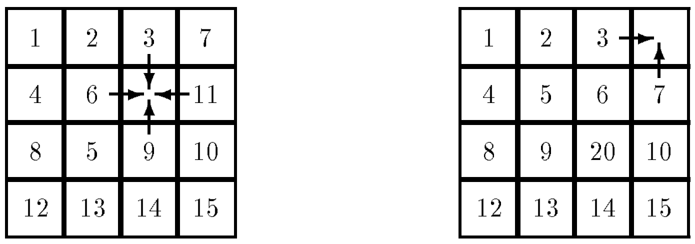


Figure 10.1: Examples of possible moves for a puzzle with dimension $N = 4$.

The puzzle must be square and has only one free cell called the hole. The square is of side N with $N \geq 3$. The number of cells in the puzzle is thus N^2 and the cells are numbered from 0 to $N^2 - 1$ from left to right, top to bottom, as illustrated with the figure 10.2. The tiles are numbered 1 to $N^2 - 1$ and the hole has the number 0. The required final state is always the same one: the tiles above the cells with the same cell number and the hole above the cell 0.

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

Figure 10.2: The numbering of cells for a puzzle with dimension $N = 4$.

Input File

The file you must read is "`P10.ENT`".

The input file gives the dimension of the puzzle (N) on the first line and the initial state on the following line. The initial state indicates which tile is above which cell, by ascending order of cell number. Here is an example of the input file which represents the problem of the figure 10.3.

▷

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

◁

The dimension of the puzzle(N) can be between 3 and 10. The problem is guaranteed to have a solution.

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

Figure 10.3: Example of problem with a puzzle of dimension $N = 4$.

Output File

You must write the file "`P10.SOR` " which will contain your sequence of displacements of the hole required to reach the final state starting from the initial state. The displacement of the hole from a cell to another is only given by the cell destination, the preceding cell being known. The following is an example of an output file that gives the solution to the problem of figure 10.3. It is also the solution to the problem.

▷

```
5 9 10 11 7 3 2 1 0
```

◁

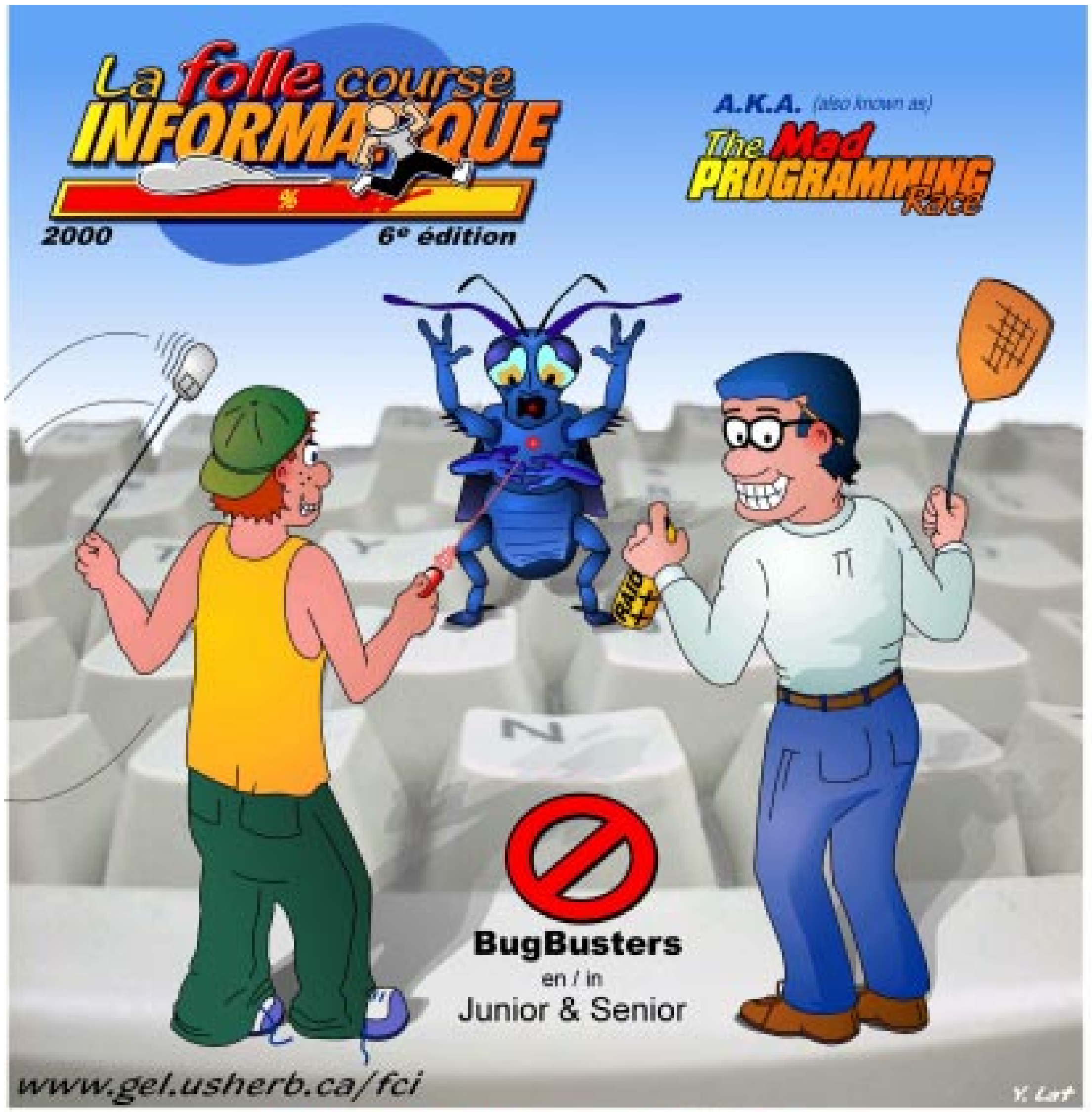
Marking

The problem will be marked by submitting your program to four different input test files.

A test will be regarded as successful if the program manages to find the solution and correctly write it in the outputfile in the proper format.

Nombre de fichiers réussis	Points accordés
Successful test file	Points
1	240
2	360
3	480
4	600

Maximum execution time for your program : 10 seconds.



EICON TECHNOLOGY Connecting People to Information

TRISIGNAL COMMUNICATIONS A DIVISION OF EICON TECHNOLOGY

NATURAL MicroSystems · MOTOROLA · Microsoft · UNIVERSITÉ DE SHERBROOKE

WILEY Publishers Since 1807 · MediaTriX