CS302 -- Lab 4 -- Enumeration

- **CS302 -- Data Structures and Algorithms II**
- **James S. Plank**
- **This file: http://web.eecs.utk.edu/~jplank/plank/classes/cs302/Labs/Lab4**
- **Lab Directory: /home/jplank/cs302/Labs/Lab4**

What you hand in

You will submit the single file **matrix_enum.cpp**. The TA's will compile that to an executable **matrix_enum** and test it with the grading script.

Introduction, and your enumerations

Back in the late 2000's and early 2010's, I was doing research on *erasure codes*. These are techniques for protecting data in storage systems composed of multiple disks (think RAID). In 2008, I wrote a paper entitled "*A New Minimum Density RAID-6 Code with a Word Size of Eight*." If you dare, you can read the paper at http://web.eecs.utk.edu/~jplank/plank/papers/NCA-2008.html. Read it out loud to your significant other when he/she can't sleep.

Part of this research was performing enumerations of square matrices. The elements in the matrices can be any of three characters: `'.'`, `'X'` and `'E'`. The enumeration is parameterized by two values:

- **W** (which stands for the "word size") is the number of rows in the matrices. It is also the number of columns, because the matrices are square.
- **E** (which stands for "extra non-zero entries") is a number between 0 and W^2-W.

Now, given **W** and **E**, your job is to enumerate all matrices of the following form:

- There are exactly **W** elements with X's. There must be an `'X'` in every row and every column. (BTW, this is called a *Permutation Matrix*).
- There are exactly **E** elements that have E's in them, and they can't be where the X's are.
- The remaining W^2-W-E elements contain `'.'`.

For example, when **W** is 3 and **E** is 1, then you have the following 36 matrices:

XE. .X. ..X	X.E .X. ..X	X.. EX. ..X	X.. .XE ..X	X.. .X. E.X	X.. .X. .EX	XE. ..X .X.	X.E ..X .X.	X.. E.X .X.	X.. .EX .X.	X.. ..X EX.	X.. ..X .XE
EX. X.. ..X	.XE X.. ..X	.X. XE. ..X	.X. X.E ..X	.X. X.. E.X	.X. X.. .EX	EX. ..X X..	.XE ..X X..	.X. E.X X..	.X. .EX X..	.X. ..X XE.	.X. ..X X.E
E.X .X. X..	.EX .X. X..	..X EX. X..	..X .XE X..	..X .X. XE.	..X .X. X.E	E.X X.. .X.	.EX X.. .X.	..X XE. .X.	..X X.E .X.	..X X.. EX.	..X X.. .XE

Your job is to write a program called **matrix_enum**, which takes three command line arguments: **W**, **E** and either an `'x'` or an `'h'`. Your program will enumerate all of the matrices for **W** and **E**, in any order you want, and print them out in one of the following two formats:

- If the last argument is `'x'`, then you'll print the matrices out in the format above. You should print each matrix as **W** lines of **W** characters that are `'.'`, `'X'` or `'E'`. After each matrix, you print a blank line.
- If the last argument is `'h'`, then you'll convert each line of each matrix into an integer, and print that integer in hexadecimal, with no leading 0's and no leading `"0x"`. If element *i* in a row is `'X'` or `'E'`, then you'll set the *i*-th bit of the number to 1. Otherwise, the *i*-th bit is zero. You'll print each integer on its own line, and print a blank line at the end of each matrix.

Here are a few runs:

```
UNIX> matrix_enum 2 0 x
X.
.X

.X
X.

UNIX>
```

```
UNIX> matrix_enum 2 1 x
XE
.X

X.
EX

EX
X.

.X
XE

UNIX>
```

```
UNIX> matrix_enum 2 2 x
XE
EX

EX
XE

UNIX>
```

```
UNIX> matrix_enum 2 0 h
1
2

2
1

UNIX>
```

```
UNIX> matrix_enum 2 1 h
3
2

1
3

3
1

2
3

UNIX>
```

```
UNIX> matrix_enu
3
3

3
3

UNIX>
```

Approach

This is a two-level enumeration. The first enumeration is enumerating the permutation matrices (the X's). You can do this by enumerating permutations of the numbers (0,1,2,...**W**-1). Let's suppose that each permutation is represented by a vector of **W** numbers. Then, the `'X'` in row *i* will be in the column specified by element *i* of the vector. For example, if **W** equals 3, then the permutation (1,2,0) represents the matrix:

```
.X.
..X
X..
```

I don't want you to use **next_permutation()** to implement this part. Use the recursive technique specified in the Enumeration lecture notes, in the section entitled "Using Recursion to Permute." If you use **next_permutation()**, you will lose 10 points on this lab, so when you're developing your code, you may want to start by using **next_permutation()**, and then changing it to use recursion when you've gotten everything else working.

At the second level, you have W^2-W potential locations for the `'E'` characters. Thus, this is an "n choose k" where n is equal to W^2-W and k is equal to **E**. Implement this using the recursive technique specified in the Enumeration lecture notes, in the section entitled "Using Recursion to Perform an "n choose k" Enumeration."

When I did this, I created a vector out of the non-`'X'` matrix elements. I stored these as numbers: (row * **W** + column). I then enumerated all ways to choose **E** of these, using the same technique as in the lecture notes. Continuing the example above, suppose **W** equals three, and my permutation matrix is (1,2,0). Then the potential spots for `'E'` elements are the following (again, using (row * **W** + column)):

(0, 2, 3, 4, 7, 8).

Here is my main class definition. You don't need to use this, but you may find it useful:

```
class Matrix {
  public:
    int W;
    int E;
    int P;                      /* This is 'x' or 'h' */
    vector <int> Perm;          /* Permutation of 0 .. (W-1), for the 'X' elements. */
    vector <int> Non_X;         /* This is the row/col id of each of the non-X elements. */
    vector <int> E_ID;          /* This is the row/col id of the E elements */
    void Print();               /* Print the matrix defined by W, Perm and E_ID */
    void Permute(int index);    /* This is the recursive permuting method. */
    void Choose(int index);     /* This is the recursive n-choose-k method. */
};
```

The Gradescript

I have two techniques for grading these. The first pipes the output of your program through the program **flatten_for_grading.cpp**. This program coalesces the lines for each matrix down to one line, and then prints the lines. Thus, if you pipe the output for this to **sort**, then you'll get all of the matrices, printed on one line, sorted lexicographically

```
UNIX> matrix_enum 2 1 x | flatten_for_grading | sort
.X XE
EX X.
X. EX
XE .X
UNIX> matrix_enum 2 1 h | flatten_for_grading | sort
1 3
2 3
3 1
3 2
UNIX>
```

Your output, after being piped through **flatten_for_grading** and then **sort**, has to match my output (piped through **flatten_for_grading** and **sort**) verbatim. This will be the technique used for gradescripts 0 through 70.

The second technique is to use the program **double_check**. This takes **W**, **E**, and **h**|**x** on the command line and then accepts input of your program piped through **flatten_for_grading**. It then checks every line of input t to see if it is legal for those parameters. This means that for `'x'`:

- Lines have the proper number of characters and are in the proper format.
- Each row has exactly one `'x'` in a different column.
- The number of `'E'` and `'.'` characters is correct.
- Lines are unique.

For `'h'`, the line is converted into a string, and then checked as in `'x'` above. There is a little subtlety here, because a single `'h'` line can correspond to quite a few strings. What **double_check** does is enumerate all of the strings that the `'h'` line corresponds to, and it counts the legal ones. It then allows to to use that string that many times. Let's try an example. Suppose that **W** equals three and **E** equals 2, and you specify the `'h'` matrix (3,3,4). This can correspond to the following legal matrices:

XE. EX. ..X	EX. XE. ..X

So **double_check** will allow you to specify (3,3,4) twice, but not three times.

The **double_check** program prints nothing if its input is legal. If you have a problem, it prints an error message on standard output. For example, here's what happens if you specify (3,3,4) three times in the above example:

```
UNIX> double_check 3 2 h
3 3 4
3 3 4
3 3 4
Bad line 3: Too many lines (3) with these values.
Here are the matrices that correspond to these values:
EX. XE. ..X
XE. EX. ..X
UNIX>
```

Gradescripts 71 through 100 pipe the output of your program through **flatten_for_grading**, and then through **head -n 50000**, and then through **double_check**. This lets me grade for larger values of **W** and **E** without waiting for the giant enumerations to complete.

To help you manage your expectations -- the following table contains how long the gradescripts took on my programs (seconds). Most of this time is in **double_check**, by the way -- please see this Piazza question and answer for more detail on why gradescript 81 takes so long.

1 - 0.08	21 - 3.31	41 - 0.06	61 - 0.29	81 - 11.78
2 - 0.06	22 - 0.76	42 - 2.71	62 - 0.04	82 - 1.00
3 - 0.06	23 - 0.43	43 - 0.13	63 - 0.29	83 - 0.77
4 - 0.06	24 - 0.09	44 - 2.01	64 - 0.35	84 - 1.53
5 - 0.06	25 - 1.38	45 - 0.10	65 - 0.25	85 - 1.52
6 - 0.06	26 - 0.32	46 - 0.06	66 - 0.68	86 - 2.20
7 - 0.07	27 - 1.71	47 - 1.27	67 - 0.06	87 - 0.53
8 - 0.07	28 - 1.81	48 - 0.15	68 - 0.50	88 - 0.52
9 - 0.06	29 - 0.14	49 - 0.22	69 - 0.27	89 - 0.41
10 - 0.06	30 - 3.17	50 - 0.18	70 - 0.22	90 - 2.65
11 - 1.65	31 - 0.34	51 - 0.04	71 - 0.27	91 - 2.40
12 - 0.06	32 - 0.83	52 - 0.04	72 - 1.73	92 - 1.48
13 - 0.81	33 - 0.44	53 - 0.04	73 - 0.42	93 - 1.80
14 - 0.20	34 - 2.77	54 - 0.04	74 - 0.05	94 - 0.78
15 - 4.09	35 - 1.54	55 - 0.40	75 - 1.21	95 - 0.63
16 - 0.37	36 - 0.07	56 - 0.05	76 - 1.17	96 - 1.76
17 - 0.09	37 - 0.07	57 - 0.05	77 - 1.68	97 - 1.17
18 - 1.93	38 - 0.39	58 - 0.05	78 - 1.79	98 - 0.57
19 - 0.66	39 - 2.07	59 - 0.04	79 - 0.49	99 - 0.30
20 - 0.21	40 - 3.38	60 - 0.04	80 - 2.08	100 - 2.38