CSE 232

# Programming Project 06

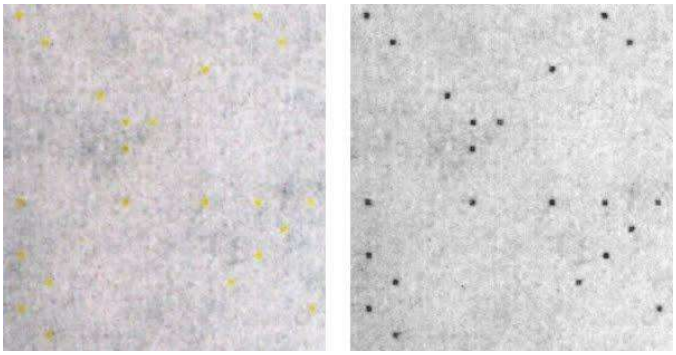
**Background**

**Steganography**

Steganography (https://en.wikipedia.org/wiki/Steganography) is the process of hiding a "secret message" in a text file, image or even sound file. It differs from cryptography in that the overall file/video/audio looks reasonably normal and still conveys information, making it hard to tell that there is a secret hidden inside.

**Color Printer Steganography**

You probably didn't know (I didn't until recently) that most color printers add a "fingerprint" to every page they print. They are not exactly invisible, they are very hard to see, but they are there. That fingerprint encodes information like: time, date and serial number of the printer being used. Every page! They typically come as a small matrix of yellow dots (which are hard to see on white paper) which encode the information. The image below shows the actual yellow dots (highly magnified on the left) and a false-color enhancement to better show them.



The companies are not exactly forthcoming about the encoding for their printers, but at least one has been decoded. The Xerox DocuColor series. We will write a program that can decode these.

**Xerox Docucolor Matrix**

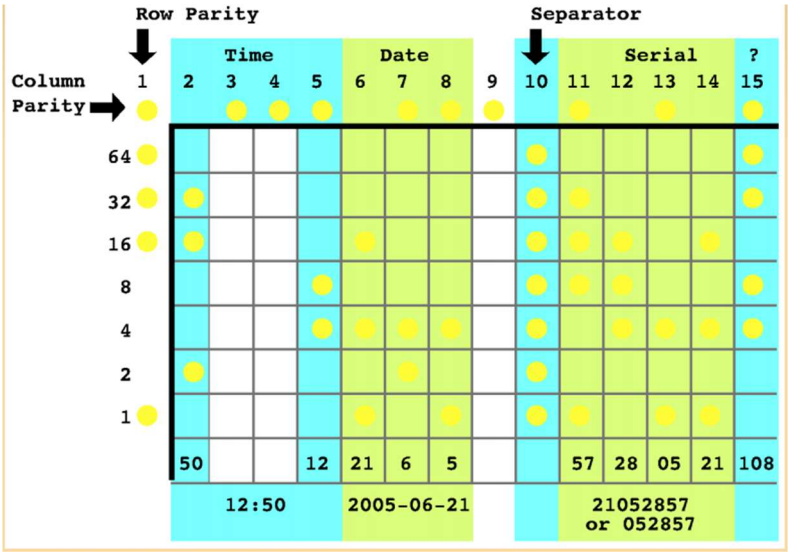
Below is a representations of the DocuColor matrix printed on each sheet, and how they might be interpreted. The fingerprint is a 8 row x 15 column matrix of yellow dots. Note on the **drawing that they number columns starting <u>at index 1</u>** (yuck).

The first row and first column are special. They represent a property called *parity* which we will discuss in a moment. Otherwise:

- columns 2 and 5 represent time (minutes and hours respectively)
- columns 6, 7, 8 represent a date

- columns 11, 12, 13, 14 represent a serial number

Columns 3,4,9,10,15 are ignored (for our purposes).




In the figure, the **two rows at the bottom are not part of the matrix** but are the interpretation of the columns for your benefit. Each row represents a power of two, and so each column represents a decimal number. Look at column 2. There is a dot in the row representing 2, 16 and 32. The sum of those is 50, which represents the minutes in the time. Notice **we do not count the top row**, the parity row, in any calculation. For column 5, there is a dot at 4 and 8, representing 12 hours. Thus the time this page was printed was 12:50. The other interpretations are done in the same manner. Note that, for our purposes, we would interpret all of the serial number columns, so our expectation would be to print the serial number as 21052857 in that order.

**Parity**

Parity is a really easy concept to understand. We typically use parity on binary representations such as a binary string. To determine the parity of a binary string, you count the number of 1's that occur in the string. If the 1's count is even, then the parity for that string is 1, if the 1's count is odd then the parity is 0. Easy! The question is why is that interesting? When you transmit information across a channel that can be noisy (like a wireless connection), you can sometimes lose data. Parity is a really cheap way to check that what got passed is what you intended. That is, if you pass the binary string through some noisy channel along with a parity bit, you can check on the receiving end to see if the parity is still correct. That is, does the number of 1's passed matches the parity that was also passed. If parity does not match the count, something is wrong. If the parity bit is correct, something could still be wrong but if you have multiple parity bits in the information passed, you can have some confidence the that information was correctly passed.

In the image above, let's check. For the column parity in column 2, the parity is not set, representing a parity of 0. If we count the number of 1's in the column there are three dots (three 1's), so the parity is odd. The parity bit accurately reflects the parity of the column. Columns 3

and 4 have no dots. The parity of 0 is 1, so 3 and 4 are set to 1 (a dot). Column 5 has two dots (two 1's), the parity is even, so the parity is 1 (a dot).

Same for the rows. Row 1 has its parity bit set (1). The row has 6 dots, even parity. The bit accurately represents the parity of the row. Row 8 has 5 dots, odd parity, value of 0, no dot. The only weird one is the top left corner. Does that represent the parity of the first column (all the row parities) or the first row (the parity of all the columns). Has to be the parity of the column. Check yourself. 4 dots in the column, even parity. 9 dots in the row, odd parity. The dot is set. Must be for the column.

**Program Specifications**

function `vector<vector<int>> read_file(const string &fname)`

- one argument:
  - a string representing the name of the file to be opened
- the return is a 2D vector

We cannot actually read the dots, so instead we read a file of the following form (this file representing dot patterns used in the image above).

```
101110111010101
100000000100001
110000000110001
110001000111010
000010000111001
000011110101111
010000100100000
100001010110110
```

**No spaces** on a row between each number. My suggestion is to read each line of the file in as a string, convert each character to an integer, and push it back onto the row of the 2D vector. Do it for each row.

**Error Checking**: if you cannot open the filename provided, throw a `runtime_error`

function `vector<int> get_row(const vector<vector<int>> &v, int row)`

- two arguments
  - 2D vector of int (by reference)
  - an int row
- returns a 1D vector containing that row.

No Error Checking.

function `vector<int> get_column(const vector<vector<int>> &v, int col)`

- two arguments
  - 2D vector of int (by reference)
  - an int column
- returns a 1D vector containing that column including the parity bit.

No Error Checking.

function `int col_to_int(const vector<vector<int>> & v, size_t col)`

- two arguments
  - 2D vector of int (by reference)
  - an int column
- returns an integer, the value the column represents (without the parity value)

Should use `get_column`

function `string get_time(const vector<vector<int>> & v)`

- argument is the 2D vector
- return is a string, the time the matrix represents (for the above example, “12:50”)

Should use `col_to_int`

function `string get_date(const vector<vector<int>> & v)`

- argument is the 2D vector
- return is a string, the date the matrix represents (for the above example “06/21/2005”)
- Note: if the month or day has just a single digit, then it must be padded with a 0

Should use `col_to_int`

function `string get_serial(const vector<vector<int>> & v)`

- argument is the 2D vector
- return is a string, the serial number the matrix represents (for the above example 21052857)

Should use `col_to_int`

function `string check_column_parity(vector<vector<int>> & v, int col)`

- two arguments
  - 2D vector of int (by reference)
  - an int col
- returns a **string** with the following information (follow the string format exactly, separated by ":", no spaces, string return as below)
  - column_parity:column_1's_count:true or false if parity of the matrix and the count match.
  - "1:2:true" means the parity bit is set(1) , the 1's count is 2, parity bit and count parity match

Should use `get_col` and `parity`

function `string check_row_parity(vector<vector<int>> & v, int row)`

- two arguments
  - 2D vector of int (by reference)
  - an int row
- returns a **string** with the following information (follow the string format exactly, separated by ":", no spaces, string return as indicated below).
  - row_parity:row_1's_count:true or false if parity of the matrix and the count match.
  - "1:2:true" means: the parity bit is set(1) , the 1's count is 2, parity bit and count parity match

Should use `get_row` and `parity`

**Deliverables**

You will turn in one file: `proj06_functions.cpp`. We provide you only with `proj06_functions.h` , you must write your own main to test your functions. Mimir can test the individual functions without a main program but it's a good idea for **you** to test your own code with a main, perhaps in the manner that we did previously.

Remember to include your section, the date, project number and comments and you ***do not provide*** main.cpp. If you turn in a main with your code Mimir will not be able to grade you.

1. Please be sure to use the specified file names
2. Always a good idea to save a copy of your file in your H: drive on EGR.
3. Submit to Mimir as always. There will be a mix of visible and not-visible cases.

**Assignment Notes**

1. In the header file are two functions that ***we do not test*** on Mimr. That means you are **not required** to do them (but it is suggested)
   a. `int parity(int)`
   b. `string print_vector (const vector<vector<int>> &v)`
   The `parity` function is too easy to test, is the parameter even or odd (though useful to have) and the `print_vector` is really useful but it is difficult to get a good Mimir test up for it (long output can be hard to get right). How you would know you read the vector correctly without printing I don't know, but I guess that's up to you. I'll leave them in put put a comment up about that in the header.
2. You turn in the functions only. To test against your own main you can write a separate file with the main and then compile the two files at the same time. See the lab and videos for examples.
3. I included a `print_vector` function in the proj06_functions.h The intent is for that function print out the 2D vector that you just read in. I won't test that function in Mimir,

but I submit you truly need it if you want to know that you read the data in right. Most of the other tests are going to depend on `read_file` working right. If that doesn't work, very little else will.