# EECS2031

Lab 4

Winter 2017

This lab will continue to cover the basic design, implementation and testing skills required to write a C program. Specifically, reading from text files and calculating a hash function.

## Hashing

Hashing is a mapping usually from a string to an integer, and is usually used to speedup searching. For example consider a class list with 500 student names and grades. Searching for a student mark may means that we have to go through the entire class list. A hashing function maps the students names (string) into some integer. First we start explaining the *ideal* hashing function.

Consider the case of a class list with 500 names. An ideal hashing function will map each student name to a unique number between 1 and 500. If we are searching for a student "John Doe". Instead of searching the list (we assume unsorted) name by name till we find "John Doe" or conclude it is not in the list.

If we have an ideal hashing function, then we construct a hashing table and use the function to map each string to a number *i* such that $1 \leq i \leq 500$ . Then we store the student name and its mark in two arrays *name[i]* and *mark[i]*. If we are looking for "John Doe" grade; instead of searching the entire function for the name "John Doe", we apply the hashing function to "John Doe" and get *i*. Then we go directly to *name[i]*, if it is empty, the student is not in the class, otherwise the student is in the class and his/her mark is in *mark[i]*.

Ideal hashing functions are difficult to come by. For example in the previous case, 2 different student names may be mapped to the same integer. For example "John Doe" and "Jane Doe" may be both mapped to the integer 234. That is called a collision. In that case, the table name[234] has a pointer to a linked list with all the names that are mapped to 234. Still searching is much faster. Instead of going through the 500 names, we hash the name and search all the entries that are mapped to that integer (usually much less than 500).

Another way to use hashing is to map the 500 names to an integer between 1 and 100. For sure there will be collisions, but a good hashing function will spread out the names nicely among the 100 entry table so each entry will have 5 names. Searching 5 names is much faster than 500 names.

One more thing to consider is how many entries in the table are used. For example consider the 500 names and a hashing function that returns an integer between 1 and 100. Sometimes after we complete the file hashing, we find that the names are mapped only to 97 distinct integers (24, 37 and 55 didn't receive any mapping). Like I said before a good hashing function will spread out the names across all the integers, since these 3 entries are wasted space.

**What is a hashing function?** The easiest hashing function it to read the string a character by character and consider each character as an unsigned 8-bit number between 0 and 255. Then we add all the characters modulo some integer *k* resulting in an integer between 0 and *k-1*.

In this lab, we assume the previous hashing function. The hashing function adds the bytes of a string modulo *k*. The size of the hash table is *k*.

## Deliverable:

Write a program to calculate the hashing table of input data. The program reads from the standard input the table size *k*. The program reads the data to be hashed from a text file named "input.txt". Then it calculates the hashing table

After the program reads the size *k*, open and reads the data from "input.txt". Then it displays some statistics ~~and continue to read from the standard input till the end of the file (standard input)~~.

The "input.txt" file consists of records (strings, and may contain spaces) each record is in a separate line. Note that the space is a part of the record to be hashed.

The statistics to be displayed is as follows (each line terminated by a new line).

- The number of entries with collision is xxx (xxx is an integer left justified in 5 digits, note there is a space between is and the first digit), where a collision is a tabel entry that received.
- The number of unused entries is xxx (again xxx is an integer left justified in 5 digits, the unused entries are the entries in the *k*-length table where no strings are mapped).

Submit the program as hash.c

The following are exercise problems, do not submit them.

## Game of Life

In the original game of life, the universe size is infinite. For this lab, we assume that the universe size is finite and can  be descried by 2-D array. The size is an input parameters, so is the initial state.

The game of life is a 0-player game, the game behavior is determined by the initial state, no input is required to play.

The universe (2-D array) evolves in generations. The next generation state is completely depends on the current generation state. Generation 0 is the initial state and is an input to your program.

## Rules:

- The state of any cell in  the next generation is determined by the state of the that cell neighbors in the current generation.
- If a live cell has less that 2 alive neighbors in the current generation, it dies in the next generation (under population).
- If a live cell has more than 3 live neighbors in the current generation, it dies in the next generation (over population).
- If a live cell has 2 or 3 live neighbors in the current generation, it continues to live in the next generation.
- If a dead cell in the current generation has exactly 3 live cells, it becomes live in the  next generation (reproduction).

Some patterns repeat itself after a specific period, some patterns may move to the left/right in a continuous way, some patterns dies off very quickly and every cell  is dead in a small number of generations. After you  complete you code, you may play around with the initial pattern to produce different scenarios.

## Specifications

The input to the program is as follows

two integers, r and c to indicate the size of the array (rows and columns)

one integer that determine the number of generations to calculate and display.

*r* lines each contains *c* integers with values of either 1 or zero to describe the initial state of the game.

The output is a series of 2-D arrays displays where the game is displayed as 2-D array with every cell is either 1 or 0, where 1 means the cell is alive and 0 means the cell is dead.

A separator of 5 dashes (minus sign) followed by a new line after each display.

For example, if the input is

```
5 5

3

0 0 0 0 0

0 0 0 0 0

0 1 1 1 0
```

```
0 0 0 0 0
0 0 0 0 0
```

The output should be

```
0 0 0 0 0
0 0 1 0 0
0 0 1 0 0
0 0 1 0 0
0 0 0 0 0
-----
0 0 0 0 0
0 0 0 0 0
0 1 1 1 0
0 0 0 0 0
0 0 0 0 0
-----
0 0 0 0 0
0 0 1 0 0
0 0 1 0 0
0 0 1 0 0
0 0 0 0 0
-----
```

## Birthday Paradox

The birthday paradox is a problem usually taught in probability classes to show that sometimes what could be intuitive may not be always true.

Consider a room with *n* people, what is the probability that at least tywo of them share the same birthday date (day and month, not year).

Since the year is 365 days, so when there are 20-30 people in a room, the probability that at least two of them having the same birthday is very small, we will see it is not really that small.

The probability that each person has a distinct birthday (no two people share the same birthday) with n people in the room is calculated as follows (see a probability book)

$$p = \frac{365!}{(365-n)!\,365^n}$$

The probability that at least 2 people share the same birthday is *1-p*.

Write a program to calculate

Your program reads *n* from the standard input and calculate the required probability using two different techniques and compare between the two values.

- The first is to calculate the above equation. Note that straight forward calculations may not be feasible (the numbers may be more than what a float can handle). Think of a creative way to do it.
- The second is by simulation. Generate *n* random numbers between 1 and 365. Check if there are at least two numbers with the same values. If yes, then there are at least 2 people with the same birthday, otherwise no two people with the same birthday. Repeat this one thousand (or 2,000) times and calculate the required probability.