2. Pre-Arrays

Note that these exercises are in **NO** particular order - try the ones you find easy before attempting more complex ones.

2.1 Planet Trium

On the planet Trium, everyone's name has three letters. Not only this, all the names take the form of non-vowel, vowel, non-vowel.

Exercise 2.1 Write a program that outputs all the valid names and numbers them. The first few should look like :

```
 1 bab
 2 bac
 3 bad
 4 baf
 5 bag
 6 bah
 7 baj
 8 bak
 9 bal
10 bam
11 ban
12 bap
13 baq
14 bar
15 bas
16 bat
17 bav
18 baw
19 bax
20 bay
21 baz
```

```
22 beb
23 bec
24 bed
25 bef
26 beg
```

2.2 Planet Bob

On the planet Bob, everyone's name has three letters. These names either take the form of consonant-vowel-consonant or else vowel-consonant-vowel. For the purposes here, vowels are the letters `{a,e,i,o,u}` and consonants are all other letters. There are two other rules :

1. The first letter and third letters of the name must always be the same.
2. The name is only 'valid' if, when you sum up the values of the three letters ($a = 1, b = 2$ etc.), the sum is prime.

The name "bob" is a valid name: it has the form consonant-vowel-consonant, the first letter and third letters are the same ('b') and the three letters sum to $19(2+15+2)$, which is prime. The name "aba' is **not** valid, since the sum of the three letters is $4(1+2+1)$ which is **not** prime.

Exercise 2.2 Write a program that outputs all the valid names and numbers them. The first few names should look like :

```
 1 aca
 2 aka
 3 aqa
 4 bab
 5 bib
 6 bob
 7 cac
 8 cec
 9 ded
10 did
11 dod
12 dud
13 ece
14 ege
15 eme
16 ese
17 faf
```

2.3 Secret Codes

Write a program that converts a stream of text typed by the user into a 'secret' code. This is achieved by turning every letter 'a' into a 'z', every letter 'b' into a 'y', every letter 'c' into and 'x' and so on.

Exercise 2.3 Write a function whose 'top-line' is :

```
int scode(int a)
```

that takes a character, and returns the secret code for this character. Note that the function

does need to preserve the case of the letter, and that non-letters are returned unaffected.

When the program is run, the following input:

```
The Quick Brown Fox Jumps Over the Lazy Dog !
```

produces the following output :

```
Gsv Jfrxp Yildm Ulc Qfnkh Levi gsv Ozab Wlt !
```

■

2.4 Cash Machine (ATM)

Some cash dispensers only contain £20 notes. When a user types in how much money they'd like to be given, you need to check that the amount requested can be dispensed exactly using only £20 notes. If not, a choice of the two closest (one lower, one higher) amounts is presented.

Exercise 2.4 Write a program that inputs a number from the user and then prompts them for a better choice if it is not correct. For example :

```
How much money would you like ? 175
I can give you 160 or 180, try again.
How much money would you like ? 180
OK, dispensing ...
```

or :

```
How much money would you like ? 25
I can give you 20 or 40, try again.
How much money would you like ? 45
I can give you 40 or 60, try again.
How much money would you like ? 80
OK, dispensing ...
```

In this assessment you may assume the input from the user is "sensible" i.e. is not a negative number etc. ■

2.5 Hailstone Sequence

Hailstones sequences are ones that seem to always return to 1. The number is halved if even, and if odd then the next becomes `3*n+1`. For instance, when we start with the number 6, we get the sequence : `6, 3, 10, 5, 16, 8, 4, 2, 1` that has nine numbers in it. When we start with the number 11, the sequence is longer, containing 15 numbers : `11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1`.

Exercise 2.5 Write a program that :

- displays which initial number (less than 50,000) creates the **longest** hailstone sequence.
- displays which initial number (less than 50,000) leads to the **largest** number appearing in the sequence.

■

2.6 Monte Carlo Π

At :

www http://mathfaculty.fullerton.edu/mathews/n2003/montecarlopimod.html

a square whose sides are of length r, and a quarter-circle, whose radius is of r are drawn.

If you throw random darts at the square, then many, but not all, also hit the circle. A dart landing at position (x,y) only hits the circle if $x^2 + y^2 \le r^2$.

The area of the circle is $\frac{\pi}{4}r^2$, and the area of the square is r^2.

Therefore, a way to approximate π, is to choose random (x,y) pairs inside the square h_a, and count the h_c ones that hit the circle. Then:

$$\pi \approx \frac{4h_c}{h_a} \tag{2.1}$$

Exercise 2.6 Write a program to run this simulation, and display the improving version of the approximation to π. ■

2.7 Leibniz Π

See:

www https://en.wikipedia.org/wiki/Leibniz_formula_for_%CF%80

The Mathematical constant π can be approximated using the formula :

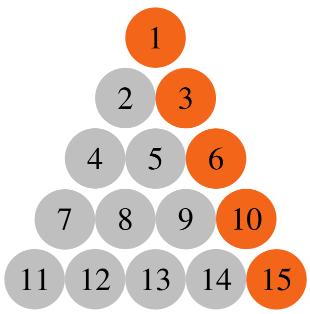
$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \dots$$

Notice the pattern here of alternating $+$ and $-$ signs, and the odd divisors.

Exercise 2.7 Write a program that computes π looping through smaller and smaller fractions of the series above. How many iterations does it take to get π correctly approximated to 9 digits ? ■

2.8 Triangle Numbers

A Triangle number is the sum of numbers from 1 to n. The 5^{th} Triangle number is the sum of numbers $1,2,3,4,5$, that is 15. They also relate to the number of circles you could stack up as equilateral triangles



$T_1 = 1$

$T_2 = 3$

$T_3 = 6$

$T_4 = 10$

$T_5 = 15$

Exercise 2.8 Write a program that prints out the sequence of Triangle numbers, using iteration, computing the next number based upon the previous.

Check these against:

WWW http://oeis.org/A000217

and also by generating the n^{th} Triangle number based on the Equation :

$$T_n = n * (n+1)/2$$

2.9 Higher-Lower

In the game "higher-Lower", a user has to guess a secret number chosen by another. They then repeatedly guess the number, being only told whether their guess was greater, or less than the secret one.

Exercise 2.9 Write a program that selects a random number between 1 and 1000. The user is asked to guess this number. If this guess is correct, the user is told that they have chosen the correct number and the game ends. Otherwise, they are told if their guess was too high or too low. The user has 10 goes to guess correctly before the game ends and they lose.

3. 1D Arrays and Strings

For some of these exercises you'll need to understand command line input to `main()` from the shell, often simply referred to as `argc/argv` in C. Please see:

www `http://www.thegeekstuff.com/2013/01/c-argc-argv/`

for more information about this.

3.1 Neill's Microwave

Last week I purchased a new, state-of-the-art microwave oven. To select how long you wish to cook food for, there are three buttons: one marked "10 minutes", one marked "1 minute" and one marked "10 seconds". To cook something for 90 seconds requires you to press the "1 minute" button, and the "10 seconds" button three times. This is four button presses in total. To cook something for 25 seconds requires three button presses; the "10 second" button needs to be pressed three times and we have to accept a minor overcooking of the food.

Exercise 3.1 Using an array to store the cooking times for the buttons, write a program that, given a required cooking time in seconds, allows the minimum number of button presses to be determined.

Example executions of the program will look like :

```
Type the time required
25
Number of button presses = 3
Type the time required
705
Number of button presses = 7
```

■

3.2 Music Playlisters

Most MP3 players have a "random" or "shuffle" feature. The problem with these is that they can sometimes be **too** random; a particular song could be played twice in succession if the new song to play is truly chosen randomly each time without taking into account what has already been played.

To solve this, many of them randomly order the entire playlist so that each song appears in a random place, but once only. The output might look something this:

```
How many songs required ? 5
4 3 5 1 2
```

or :

```
How many songs required ? 10
1 9 10 2 4 7 3 6 5 8
```

Exercise 3.2 Write a program that gets a number from the user (to represent the number of songs required) and outputs a randomised list. ■

Exercise 3.3 Rewrite Exercise 3.2 so that the program passes an array of integers (e.g. `[1,2,3,4,5,6,7,8,9,10]`) to a function and re-orders them **in-place** (no other arrays are used) and with an algorithm having complexity $O(n)$. ■

3.3 Rule 110

Rather interesting patterns can be created using *Cellular Automata*. Here we will use a simple example, one known as *Rule 110* : The idea is that in a 1D array, cells can be either on ■ or off □ (perhaps represented by the integer values 1 and 0). A new 1D array is created in which we decide upon the state of each cell in the array based on the cell above and its two immediate neighbours.

If the three cells above are all 'on', then the cell is set to 'off' ($111 \rightarrow 0$). If the three cells above are 'on', 'on', 'off' then the new cell is set to 'on' ($110 \rightarrow 1$). The rules, in full, are:

$111 \rightarrow 0$
$110 \rightarrow 1$
$101 \rightarrow 1$
$100 \rightarrow 0$
$011 \rightarrow 1$
$010 \rightarrow 1$
$001 \rightarrow 1$
$000 \rightarrow 0$

You take a 1D array, filled with zeroes or ones, and based on these, you create a new 1D array of zeroes and ones. Any particular cell uses the three cells 'above' it to make the decision about its value. If the first line has all zeroes and a single one in the middle, then the automata evolves as:

Exercise 3.4 Write a program that outputs something similar to the above using plain text, giving the user the option to start with a randomised first line, or a line with a single 'on' in the central location. ■

Exercise 3.5 Rewrite the program above to allow other rules to be displayed - for instance $124, 30$ and 90.

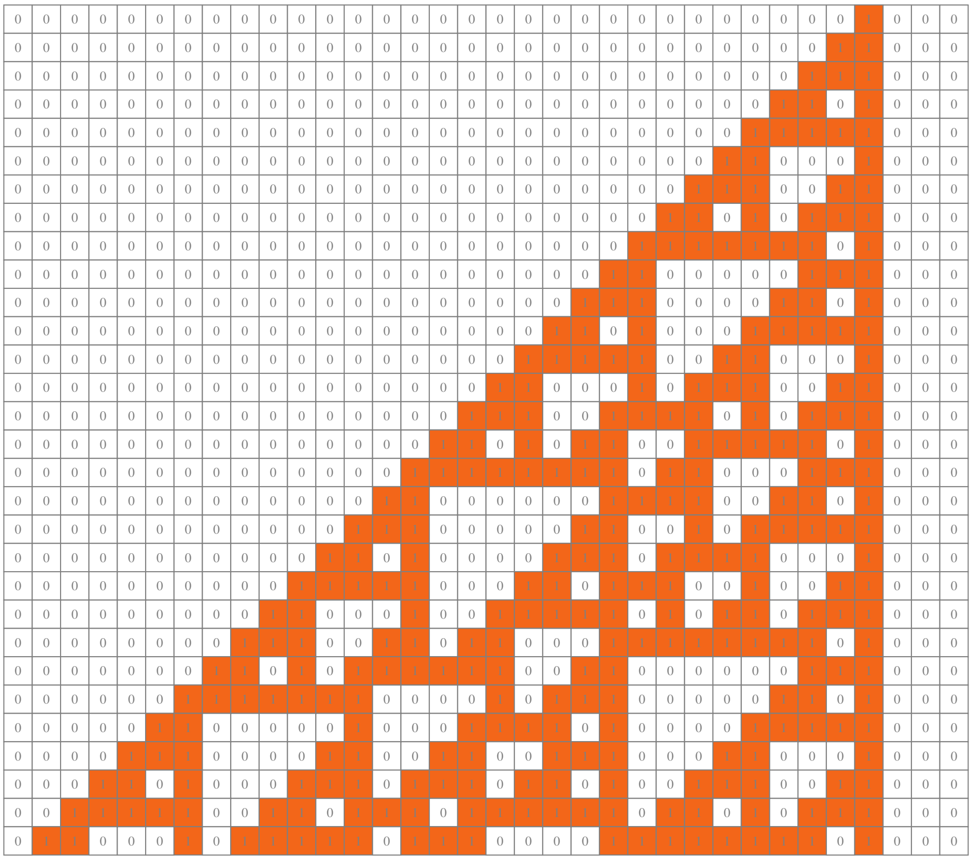


Figure 3.1: 1D cellular automaton using Rule 110. Top line shows initial state, each subsequent line is produced from the line above it. Each cell has a rule to switch it ‘on’ or ’off’ based on the state of the three cells above it in the diagram.

www `http://en.wikipedia.org/wiki/Rule_110`

3.4 Palindromes

From `wikipedia.org` :

> A palindrome is a word, phrase, number or other sequence of units that has the property of reading the same in either direction (the adjustment of punctuation and spaces between words is generally permitted).
>
> The most familiar palindromes, in English at least, are character-by-character: the written characters read the same backwards as forwards. Palindromes may consist of a single word (such as "civic" or "level"), a phrase or sentence ("Neil, a trap! Sid is part alien!", "Was it a rat I saw?") or a longer passage of text ("Sit on a potato pan, Otis."), even a fragmented sentence ("A man, a plan, a canal: Panama!", "No Roman a moron"). Spaces, punctuation and case are usually ignored, even in terms of abbreviation ("Mr. Owl ate my metal worm").

Exercise 3.6 Write a program that prompts a user for a phrase and tells them whether it is a palindrome or not. **Do not** use any of the built-in string-handling functions (`string.h`), such as `strlen()` and `strcmp()`. However, you **may** use the character functions (`ctype.h`), such as `islower()` and `isalpha()`.

Check you program with the following palindromes :

```
"kayak"
"A man, a plan, a canal: Panama!"
"Madam, in Eden I'm Adam,"
"Level, madam, level!"
```

3.5 Int to String

Exercise 3.7 Write a function that converts an integer to a string, so that the following code snippet works correctly:

```
int i;
char s[256];
scanf("%d", &i);
int2string(i,s);
printf("%s\n", s);
```

The integer may be signed (i.e. be positive or negative) and you may assume it is in base-10.

Avoid using any of the built-in string-handling functions to do this (e.g. `itoa()`!) including those in `string.h`.

3.6 Roman Numerals

Adapted from:

WWW http://mathworld.wolfram.com/RomanNumerals.html

> “Roman numerals are a system of numerical notations used by the Romans. They are an additive (and subtractive) system in which letters are used to denote certain "base" numbers, and arbitrary numbers are then denoted using combinations of symbols. Unfortunately, little is known about the origin of the Roman numeral system.
>
> The following table gives the Latin letters used in Roman numerals and the corresponding numerical values they represent :

I	1
V	5
X	10
L	50
C	100
D	500
M	1000

> For example, the number 1732 would be denoted MDCCXXXII in Roman numerals. However, Roman numerals are not a purely additive number system. In particular,

> instead of using four symbols to represent a 4, 40, 9, 90, etc. (i.e., IIII, XXXX, VIIII, LXXXX, etc.), such numbers are instead denoted by preceding the symbol for 5, 50, 10, 100, etc., with a symbol indicating subtraction. For example, 4 is denoted IV, 9 as IX, 40 as XL, etc."

It turns out that every number between 1 and 3999 can be represented as a Roman numeral made up of the following one- and two-letter combinations:

I	1	IV	4
V	5	IX	9
X	10	XL	40
L	50	XC	90
C	100	CD	400
D	500	CM	900
M	1000		

Exercise 3.8 Write a program that reads a roman numeral (in the range `1 - 3999`) and outputs the corresponding valid arabic integer. Amongst others, check that `MCMXCIX` returns `1999`, `MCMLXVII` returns `1967` and that `MCDXCI` returns `1491`.

You should use the following template :

```
#include <stdio.h>

int romanToArabic( char *roman );

int main(int argc, char **argv)
{
    if( argc==2 ){
        printf("The roman numeral %s is equal to %d\n", \
        argv[1], romanToArabic(argv[1]));
    }else{
        printf("ERROR: Incorrect usage, try e.g. %s XXI\n", argv[0]);
    }
    return 0;
}
```

You need to add the function `romanToArabic()`. ■

3.7 Soundex Coding

First applied to the 1880 census, Soundex is a phonetic index, not a strictly alphabetical one. Its key feature is that it codes surnames (last names) based on the way a name sounds rather than on how it is spelled. For example, surnames that sound the same but are spelled differently, like Smith and Smyth, have the same code and are indexed together. The intent was to help researchers find a surname quickly even though it may have received different spellings. If a name like Cook, though, is spelled Koch or Faust is Phaust, a search for a different set of Soundex codes and cards based on the variation of the surname's first letter is necessary.

To use Soundex, researchers must first code the surname of the person or family in which they are interested. Every Soundex code consists of a letter and three numbers, such as B536, representing names such as Bender. The letter is always the first letter of the surname, whether it is a vowel or a consonant.

The detailed description of the algorithm may be found at :

www http://www.highprogrammer.com/alan/numbers/soundex.html

The first letter is simply the first letter in the word. The remaining numbers range from 1 to 6, indicating different categories of sounds created by consanants following the first letter. If the word is too short to generate 3 numbers, 0 is added as needed. If the generated code is longer than 3 numbers, the extra are thrown away.

Code	*Letters Description*
1	*B, F, P, V Labial*
2	*C, G, J, K, Q, S, X, Z Gutterals and sibilants*
3	*D, T Dental*
4	*L Long liquid*
5	*M, N Nasal*
6	*R Short liquid*
SKIP	*A, E, H, I, O, U, W, Y Vowels (and H, W, and Y) are skipped*

There are several special cases when calculating a soundex code:

- *Letters with the same soundex number that are immediately next to each other are discarded. So Pfizer becomes Pizer, Sack becomes Sac, Czar becomes Car, Collins becomes Colins, and Mroczak becomes Mrocak.*
- *If two letters with the same soundex number seperated by "H" or "W", only use the first letter. So Ashcroft is treated as Ashroft.*

Sample Soundex codes:

Word	*Soundex*
Washington	*W252*
Wu	*W000*
DeSmet	*D253*
Gutierrez	*G362*
Pfister	*P236*
Jackson	*J250*
Tymczak	*T522*
Ashcraft	*A261*

Exercise 3.9 Write a program that takes the name entered as `argv[1]` and prints the corresponding soundex code for it. ■

4. 2d Arrays

4.1 The Game of Life

The Game of Life was developed by British mathematician John Horton Conway. In Life, a board represents the world and each cell a single location. A cell may be either empty or inhabited. The game has three simple rules, which relate to the cell's eight nearest neigbours :

1. **Survival** An inhabited cell remains inhabited if exactly 2 or 3 of its neighbouring cells are inhabited.
2. **Death** An inhabited cell becomes uninhabited if fewer than 2, or more than 3 of its neighbours are inhabited.
3. **Birth** An uninhabited cell becomes inhabited if exactly 3 of its neighbours are inhabited.

The next board is derived solely from the current one. The current board remains unchanged while computing the next board. In the simple case shown here, the boards alternate infinitely between these two states.

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

The 1.06 format

A general purpose way of encoding the input board is called the Life 1.06 format :

WWW http://conwaylife.com/wiki/Life_1.06

This format has comments indicated by a hash in the first column, and the first line is always:

```
#Life 1.06
```

Every line specifies an x and y coordinate of a live cell; such files can be quite long. The coordinates specified are relative to the middle of the board, so 01 means the middle row, one cell to the right of the centre.

There are hundreds of interesting patterns stored like this on the above site.

Exercise 4.1 Write a program which is run using the `argc` and `argv` parameters to `main`. The usage is as follows :

```
% life file1.lif 10
```

where `file1.lif` is a file specifying the inital state of the board, and `10` specifys that ten iterations are required.

Display the output to screen every iteration using plain text, you may assume that the board is 150 cells wide and 90 cells tall. ■

Alternative Rules for The Game of Life

The rules for life could also be phrased in a different manner, that is, give birth if there are two neighbours around an empty cell (B2) and allow an 'alive' cell to survive only if surrounded by 2 or 3 cells (S23). Other rules which are *life-like* exist, for instance *34 Life* (B34/S34), *Life Without Death* (B3/S012345678) and HighLife (B36/S23).

www `http://en.wikipedia.org/wiki/Life-like_cellular_automaton`

Exercise 4.2 Write a program that allows the user to input life-like rules e.g. :

```
life B34/S34 lifeboard.lif
```

or

```
life B2/S lifeboard.lif
```

and display generations of boards, beginning with the inital board in the input file. ■

4.2 Life Wars

Inspired by the classic game, *Core Wars*

www `http://en.wikipedia.org/wiki/Core_War`

here we look at a two player version of Conway's Game of Life.

In our game, each of two 'players' submit a Life 1.06 file and cells from these inserted into an empty board. The cells are coded based on which player created them (say '+' and '@'). The game is then run, and the player having most cells left after a fixed number of iterations, over many games, is deemed the winner.

The rules are :

1. The board is 150 cells wide, and 90 cells tall.
2. The board is toroidal; that is, it wraps around from the left edge to the right, and the top to the bottom.
3. Each player can submit a Life 1.06 file that is maximum of 100 lines long, **including** the header line.
4. Since each of the Life 1.06 files describe absolute positions for cells, each player is assigned a random origin for their cells. If there is any collision when attempting to add the cells initially (i.e. both players happen to specify a cell in the same square), then new origins are chosen randomly and the process begun from scratch.

5. The ‘standard’ B3/S23 rules are used.
6. The colour of cells is never taken into account (i.e. cells are still either ‘alive’ or ‘dead’). The sole exception to this is that when a cell is born, it takes the colour of the majority its neighbours.
7. There are 5000 generations played every game.
8. A running count of how many cells each player has left at the end of each game is kept. The board is cleared and the next game randomly restarted.
9. There are 50 games run in total.
10. The player with the highest number of cells over all 50 games is the winner.

It’s easy to extend these rules, of course, to allow for three or more players, but it’s unclear for three players what would happen in the majority vote if there was a draw (i.e. a new cell is born based on three cells around it, one belonging to each player. Other rule variants are also possible (e.g. *Highlife* (B36/S23), but once again, the birth rule for 3 or 6 neighbours would cause majority voting issues for two and three players.

Exercise 4.3 Write a program that accepts two Life 1.06 files and reports which of them is the winner. The first few games might look like:

```
% /lifewars blinkerpuffer2.lif bigglider.lif

        0    50    12  Player 1
        1   370   141  Player 1
        2   437   281  Player 1
        3   450   602  Player 2
        4   540   623  Player 2
        5   991   629  Player 1
        6  1063   674  Player 1
        7  1211   707  Player 1
        8  1263   735  Player 1
        9  1358   758  Player 1
.
.
.
    Player 1 wins by 7857 cells to 2373 cells
```

■

4.3 Wireworld

Wireworld is a cellular automaton due to Brian Silverman, formed from a 2D grid of cells, designed to simulate digital electronics. Each cell can be in one of four states, either ‘empty’, ‘electron head’, ‘electron tail’ or ‘copper’ (or ‘conductor’).

The next generation of the cells follows the rules, where n is the number of electron heads found in the 8-surrounding cells:

- empty $->$ empty
- electron head $->$ electron tail
- electron tail $->$ copper
- copper $->$ electron head if $n == 1$ or $n == 2$
- copper $->$ copper otherwise

See also:

WWW https://en.wikipedia.org/wiki/Wireworld

WWW http://www.heise.ws/fourticklogic.html

Exercise 4.4 Write a program which is run using the `argc` and `argv` parameters to `main`. The usage is as follows :

```
$ wireworld wirefile.txt
```

where `wirefile.txt` is a file specifying the initial state of the board. This file codes empty cells as ‘ ’, heads as ‘H’, tails as ‘t’ and copper as ‘c’. Display the board for 1000 generations using plain text. You may assume that the grid is always 40 cells by 40

■

Make sure all your code is fully ANSI compliant, and fully follows the house-style guidelines. Show that your code has been developed using short, well-tested functions via the use of `assert()` testing.

4.4 ncurses

C has no inherent functionality to allow printing in colour etc. Therefore, a programming library know a `ncurses` was created in 1993 to allow terminals to interpret certain control-codes as colours and other effects.

The library itself is somewhat complex, allowing keyboard and mouse events to be captured and a whole range of simple graphics functionality. On the web page is my ‘wrapper’ for the library, along with a program demonstrating its use. This will only work in unix-style terminals. Note that after you begin ncurses mode (using `Neill_NCURS_Init()`) that you can’t print to stdout or stderr, until you switch it off (using `Neill_NCURS_Done()`).

To compile the code you’ll have to use both my code `neillncurses.c` and also link in the ncurses library. A typical compile might look like

```
gcc yourcode.c neillncurses.c -Wall -Wfloat-equal -Wextra -O2
    -pedantic -ansi -lncurses -lm
```

If you’re running a virtual box you may also need to install the `ncurses` developer files, including `ncurses.h`, using:

```
sudo apt install libncurses-dev
```

Some terminals do not support ncurses, so make sure you are using an ‘xterm’ or equivalent.

Exercise 4.5 Adapt the wireworld code in Exercise 4.4 so that the output is displayed using this library, with tails being red, heads being blue, copper being yellow and background being black. The main loop will update the board, display it, and repeat until a quit event occurs (e.g. a mouse click or the ESC key is pressed).

■

Exercise 4.6 Adapt the life code in Exercise 4.1 so that the output is displayed using this library, with sensible choices made for cell colours. The main loop will update the board, display it, and repeat until a quit event occurs (e.g. a mouse click or the ESC key is pressed). ■

5. Strings, Recursion and SDL

5.1 Anagrams

An anagram is a collection of letters that when unscrambled, using all the leters, make a single word. For instance `magrana` can be rearranged to make the word `anagram`.

Exercise 5.1 Using a file of valid words, allow the user to enter an anagram, and have the answer(s) printed. For instance :

```
 % ./anagram sternaig
angriest
astringe
ganister
gantries
ingrates
rangiest
reasting
stearing
```

■

Exercise 5.2 Using a file of valid words, find all words which are anagrams of each other. Each word should appear in a maximum of one list. Output will look something like :

```
 % ./selfanagram
.
.
7 merits mister miters mitres remits smiter timers
.
.
.
6 noters stoner tenors tensor toners trones
.
.
.
6 opts post pots spot stop tops
```

```
.
.
.
6 restrain retrains strainer terrains trainers transire
.
```

If you wished to create "interesting" anagrams, rather than simply a random jumble of letters, you could combine together two shorter words which are an anagram of a longer one.

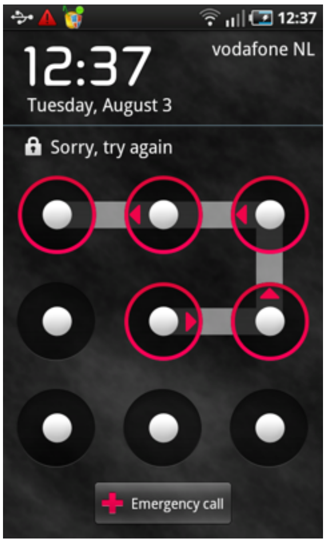
Exercise 5.3 Write a program which uses an exhaustive search of all the possible pairs of short words to make the target word to be computed. For instance, a few of the many pairs that can be used to make an anagram of `compiler` are :

```
 % ./teabreak compiler
LiceRomp
LimeCrop
LimpCore
MileCrop
MoreClip
PermCoil
PromLice
RelicMop
```

The name `Campbell` comes out as `CalmPleb` which is a bit harsh. Can't **ever** remember being called calm ...

5.2 Draw to Unlock

Rather than remembering passwords or passcodes, many mobile devices now allow the user to draw a pattern on the screen to unlock them.



Here we will explore how many unique patterns are available when drawing such patterns to connect "dots", such as shown in the figure. We assume that people put their finger on one "dot" and then only ever move one position left, right, up or down (but never diagonally) at a time. You are not allowed to return to a "dot" once it has been visited once. If we number the first position in our path as 1, the second as 2 and so on, then beginning in the top left-hand corner, some of the possible patterns of 9 moves are :

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

Exercise 5.4 Write a program that computes and outputs all the valid paths. Use **recursion** to achieve this.

- How many different patterns of length 9 are available on a 3×3 grid, if the user begins in the top left corner ?
- How many different patterns of length 9 are available on a 3×3 grid, if the user begins in the middle left ?
- How many different patterns of length 7 are available on a 3×3 grid, if the user begins in the top left corner ?
- How many different patterns of length 25 are available on a 5×5 grid, if the user begins in the top left corner ?

■

5.3 SDL - Intro

Many programming languages have no inherent graphics capabilities. To get windows to appear on the screen, or to draw lines and shapes, you need to make use of an external library. Here we use SDL[1], a cross-platform library providing the user with (amongst other things) such graphical capabilities.

www `https://www.libsdl.org/`

The use of SDL is, unsurprisingly, non-trival, so some simple wrapper files have been created (`neillsdl2.c` and `neillsdl2.h`). These give you some simple functions to initialise a window, draw rectangles, wait for the user to press a key etc.

An example program using this functionality is provided in a file `blocks.c`.

This program initialises a window, then sits in a loop, drawing randomly positioned and coloured squares, until the user presses the mouse or a key.

Exercise 5.5 Using the `Makefile` provided, compile and run this program, using something like : `make -f sdl_makefile`

SDL is already installed on lab machines. At home, if you're using a ubuntu-style linux machine, use: `sudo apt install libsdl2-dev` to install it. ■

5.4 Word Ladders

In this game, made famous by the author Lewis Caroll, and investigated by many Computer Scientists including Donald Knuth, you find missing words to complete a sequence. For instance, you might be asked how go from "WILD" to "TAME" by only changing one character at a time:

```
W I L D
W I L E
T I L E
T A L E
T A M E
```

A useful concept here is that of the `edit distance`. Here, the edit distance is a count of the number of characters which are different between two words. For words of n characters, the edit distance will be in the range $0 \dots n$. For 'aboard' and 'canape' the edit distance is 5. An edit distance of zero means that the words are identical.

In a heavily constrained version of this game we make some simplifying assumptions:

- Words are always four letters long.
- We only seek ladders of five words in total.
- Only one letter is changed at a time.
- A letter is only changed from its initial state, to its target state. This is important, since if you decide to change the second letter then you will always know what it's changing from, to what it's changing to.

So, in the example above, it is enough to give the first word, the last word, and the position of the character which changed on each line. On line one, the fourth letter 'D' was changed to an 'E', on the next line the first character 'W' was changed to a 'T' and so on. The whole ladder can be defined by "WILD", "TAME" and the sequence $4, 1, 2, 3$.

[1] actually, we are using the most recent version SDL2, which is installed on all the lab machines

W I L D
W I L E
T I L E
T A L E
T A M E

Since each letter changes exactly once, the order in which this happens is a *permutation* of the numbers $1,2,3,4$, which we have looked at elsewhere..

We'll also need another function:

```
int edit_distance(char *s, char *t);
```

which returns the number of characters which are different between two strings of the same length. For our strings of length four (excluding the null character) this will be either $0,1,2,3$ or 4. Here, we can check that the first and last word in our search are distance 4 apart.

Exercise 5.6 For the constrained version of the game, given a file of valid four letter words, write a program which when given two words on the command line (`argv[1]` and `argv[2]`) outputs the correct solution, if available. Use an exhaustive search over all 24 permutations until one leads to no invalid words being required. Make sure your program works, with amongst others, the following:

C O L D	P O K E	C U B E
C O R D	P O L E	C U B S
C A R D	P O L L	T U B S
W A R D	M O L L	T U N S
W A R M	M A L L	T O N S

Exercise 5.7 Adapt the program above so that if the first and last words share a letter (the edit distance is less than 4), you can find the word ladder required, as in:

W A S P
W A S H
W I S H
F I S H

For the "full" version of Wordladder, you make no assumptions about the number of words that are needed to make the ladder, although we do assume that all the words in the ladder are the same size.

Exercise 5.8 To achieve this, you could make a list of all the words, and for all words an `edit distance` of 1 away from the initial word, mark these and store their 'parent'. Now, go through this list, and for all words marked, find words which are distance 1 from these, and hence distance 2 from the initial word. Mark these and retain their parent. Be careful you don't use words already marked. If the word ladder is possible, you'll eventually find the

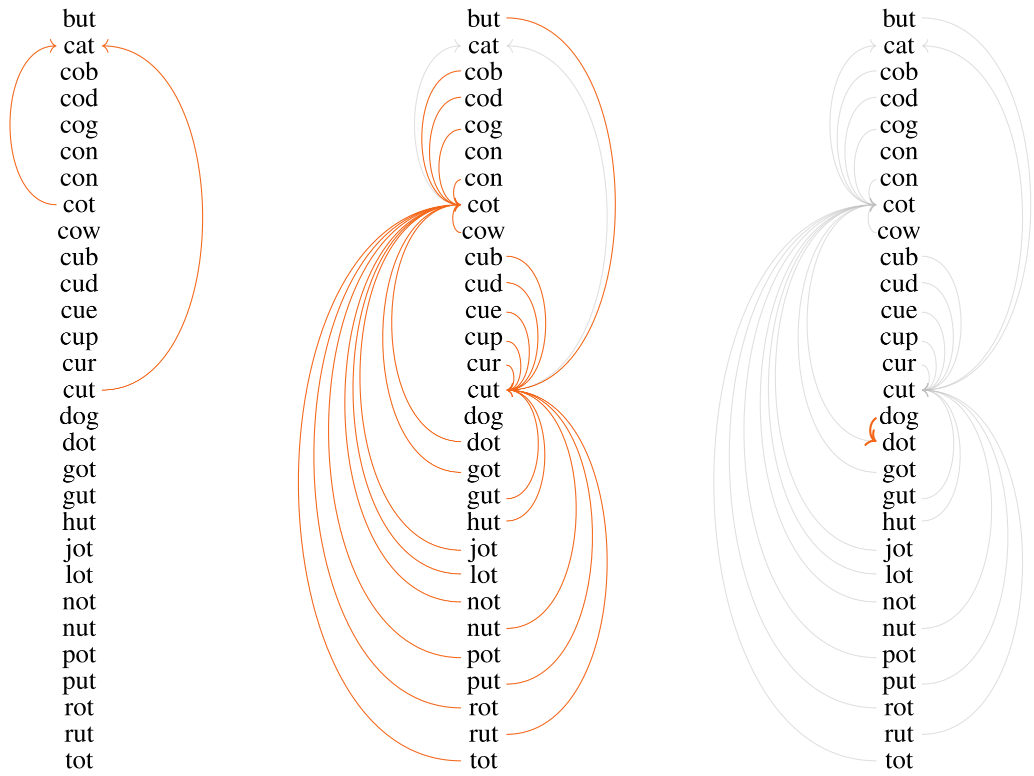


Figure 5.1: Word Ladder from CAT to DOG. (Left) Words which are distance one from CAT are COT and CUT. (Middle) Words which are distance one from CUT and COT, which includes amongst others, DOT. (Right) DOG is distance one from DOT. We now have our route, via the pointers, back to CAT.

solution, and via the record of the parents, have the correct route. This is shown for the word ladder CAT to DOG in Figure 5.1 using a very small subset of the possible three letter words.
■

5.5 The Devil's Dartboard

In the traditional 'pub' game, darts, there are 62 different possible scores : single 1 - 20 (the white and black areas), double 1 - 20 (the outer red and green segments) (i.e. 2, 4, 6, 8 ...), treble 1 - 20 (i.e. 3, 6, 9, 12 ...) (the inner red or green segments), 25 (small green circle) and 50 (the small red inner circle).

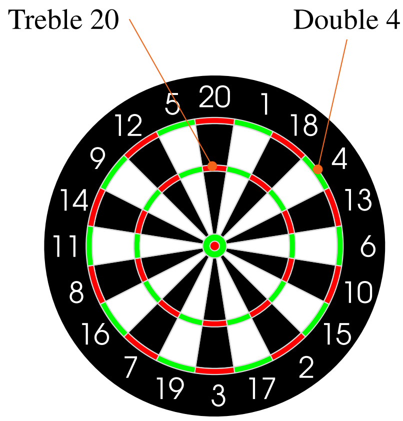
It's not obvious, if you were inventing darts from scratch, how best to lay out the numbers. The London board shown seems to have small numbers near high numbers, so that if you just miss the 20 for example, you'll hit a small number instead.

Here we look at a measure for the 'difficulty' of a dartboard. One approach is to simply sum up the values of adjacent triples, squaring this number. So for the London board shown, this would be: $(20+1+18)^2+(1+18+4)^2+(18+4+13)^2 \ldots (5+20+1)^2 = 20478$

For our purposes a **lower** number is better[2]. For more details see :

www `http://www.mathpages.com/home/kmath025.htm`

[2] It's beyond the scope here to explain why!



Exercise 5.9 Write a program that repeatedly chooses two positions on the board and swaps their numbers. If this leads to a lower cost, keep the board. If not, unswap them. Repeat this *greedy search* 5000000 times, and print out the best board found. Begin with the trivial monotonic sequence. The output may look something like :

```
Total = 19966 :  3 19 11  2 18 12  1 20 10  4 16  8
14  5 13 15  6  7 17  9
```

or

```
Total = 19910 :  3 18 10  5 16  9  8 14 11  4 19  6
7 20  2 13 15  1 17 12
```

The score of 19874 seems to be the lowest possible that may be obtained. ■

5.6 Maze

Escaping from a maze can be done in several ways (ink-blotting, righthand-on-wall etc.) but here we look at recursion.

Exercise 5.10 Write a program to read in a maze typed by a user via the filename passed to `argv[1]`. You can assume the maze will be no larger than 20×20, walls are designated by with a # and the rest are spaces. The entrance can be assumed to be the gap in the wall closest to (but not necessarilty exactly at) the top lefthand corner. The sizes of the maze are given on the first line of the file (width,height). Write a program that finds the route through a maze, read from this file, and prints out the solution (if one exists) using full stops. If the program succeeds it should exit with a status of `0`, or if no route exists it should exit with a status of `1`. ■

```
# # # # # # # # # #
. . # . . . . . . #
# . # . # . # # . #
# . # . # # # # . #
# . # . . . . # . #
# . # . # # # # . #
# . # . . . . # . #
# . # # # # . # . #
# . . . . . . # . .
# # # # # # # # # #
```

A. Appendix : Lab Tests

Examination Rules

This is an exam. Please:

- Wait to be seated by a Lab. supervisor.
- Do not talk to anyone else.
- Do not use a Web browser, unless instructed to do so (e.g. to submit files to SAFE).
- Do not use an email connection.
- Do not use other electronic devices such as laptops, tablets or phones.
- Do not look in other peoples directories.
- Do not use code written in 'groups' with others students.
- **DO** use text books, lecture notes, man pages etc.

If you do not obey the above rules you will be asked to leave.

People arriving late may not be allowed into the Lab.

You have one hour. There is additional time in case a server crashes or some other problem occurs.

Submitting Your Work

As you complete each part, submit it online using the SAFE submission system. Do not submit files that don't work.

For multi-part questions, if you complete Part 1, submit a file called `part1.c`.
If you complete Part 2, submit a file called `part2.c` and so on.

No partial marks are available. The only possible scores for each part are pass (100%) or fail (0%).

Code Style

- Your code must compile without warnings using the gcc flags:

  ```
  -pedantic -Wall -Wextra -Wfloat-equal -ansi -O2
  ```

- Do **NOT** use any global variables.

A.1 Anagrams

Part 1 (60%)

An anagram is a rearrangement of a word, using all the letters. Two words are said to be anagrams if they have the same characters, but in a different order. For instance the words ‘parsley’, ‘players’ and ‘replays’ are all anagrams of each other.

Since you need to rearrange the words, two identical words, by definition, are not anagrams.

Using the following template, fill in the function `anagram()`, so that the program runs successfully :

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>

int anagram(char s1[], char s2[]);

int main(void)
{
   assert(anagram("elvis", "lives") == 1);
   assert(anagram("dreads", "sadder") == 1);
   assert(anagram("replays", "parsley") == 1);
   assert(anagram("listen", "silent") == 1);
   assert(anagram("orchestra", "carthorse") == 1);

   /* Two identical words are not anagrams */
   assert(anagram("elvis", "elvis") == 0);

   assert(anagram("neill", "neil") == 0);
   assert(anagram("neil", "neill") == 0);
   assert(anagram("horse", "short") == 0);

   return 0;
}

int anagram(char s1[], char s2[])
{


}
```

Obviously, your program will need to run for other, unseen but similar, test cases.

Part 2 (40%)

A deranged anagram has two words with the same characters, but the same character does not appear in the same position.

The words ‘elvis’ and ‘lives’ are not a derangement since the ‘s’ is in the same position in both words. However, ‘dreads’ and ‘sadder’ are, since no letter appears in the same position between the two words.

Extend the program above so that the following assertions, inside `main()` are correct:

```
    assert(derange("elvis", "lives") == 0);
    assert(derange("dreads", "sadder") == 1);
    assert(derange("replays", "parsley") == 1);
    assert(derange("listen", "silent") == 0);
    assert(derange("orchestra", "carthorse") == 1);
```

A.2 Isograms

Part 1 (60%)

An isogram is a word that has no repeating letters. For instance the words ‘graciously’, ‘disgraceful’ and ‘productively’ are all isograms. However, the word ‘dazzlingly’ is not (it contains the letters ‘z’ and ‘l’ twice).

Using the following template, fill in the function `isogram()`, so that the program runs successfully :

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>

int isogram(char *s);

int main(void)
{
    assert(isogram("programming") == 0);
    assert(isogram("housewarmings") == 0);
    assert(isogram("abductions") == 1);
    assert(isogram("housewarming") == 1);
    assert(isogram("hydromagnetics") == 1);
    assert(isogram("uncopyrightable") == 1);
    return 0;
}

int isogram(char *s)
{


}
```

Part 2 (40%)

Using the `isogram()` function written above, write a program which finds the *longest* isogram in a file of words. The name of the file is provided via the use of `argv`. On success, the program simply outputs the longest word and its length, nothing else. For example :

```
$ ./parttwo eowl_shuffle.txt
waveringly (10)
```

The file may contain many isograms of (equal) longest length. In this case, outputting any one of them will do.

A.3 Mutating Boards

Part 1 (60%)

Write a **function** that fills up a square board, randomly, with an integer $0 \ldots 9$. Use a:
`#define N 20`
to define the size of the board. Write another function to print the board. The board may look something like:

```
36753562912709360626
18792023759228973612
93194784503610632061
55476569374525474430
78688431492068926649
50487172722610615949
09177115977673656394
81293908850963856115
98481030444476317596
21785741859753883189
64333860488897764303
09254059469224775481
28936802105110850646
25862847240629908131
10340391969338056640
04626756987282996027
61321599149107587048
04296104222055290283
80409196254499360502
94351743146942264128
```

Write a function to 'mutate' the board. Mutating is done like this:

- Choose two random locations which are **horizontally adjacent** (next to each other left-right).
- Swap these two numbers on the board if the left one is greater than the right one, numerically.
- Choose two random locations which are **vertically adjacent** (next to each other up-down).
- Swap these two numbers on the board if the upper one is greater than the lower one, numerically.
- Repeat the above steps (N*N*N) times.

Now print out the board. It should look something like :

```
00000000000001111224
00000001111111233456
00000111112222244456
00001122222333445666
00112222223333555667
01112223333334556678
01112223334445556779
01122333344445556789
01223334444455666789
01223344445556666789
01223344455666667789
01224444456666777889
01234455566677777889
01234555666677788899
01234555666778888899
```

```
122345666777888888999
123455677778888889999
124456777888888899999
344466788899999999999
466788999999999999999
```

Ensure that if you change the size of your array, by changing your `#define` that the program still operates correctly.

Part 2 (40%)

Adapt the code above, so that the algorithm is:

- Choose two numbers at random locations on the board.
- Check that of these two numbers, the one closest to the centre of the array is numerically less than the number furthest away from the centre. If not, swap them.
- Repeat the above steps (N*N*N*N) times.

Once again randomise the array initially, and ensure that after changing your `#define` the program still works correctly.

When $N = 21$, the array may look something like:

```
999998887777788899999
999987666656666788999
998876554444555678899
998665443333444566889
987654333222233456789
876543322112223345678
865443211111112334568
765432111000011234568
765422100000001234567
764321100000001223467
764321100000001123457
764322100000001223467
765422100000001224567
865432110000011234568
865432211111122334568
876543322212223345678
987654333222233456789
988665444333344567889
998876555444455678899
999887666656666789999
999998887777788899999
```