COMSM1201 : Exercises in C

Neill Campbell

Department of Computer Science, University of Bristol

Contents

1. Hello World

Some of the exercises in this Chapter are taken from the book "C by Dissection".

1.1 Lecture Notes Chapter B

Exercise 1.1.1 After you've studied Chapter B (*Hello World!*), compile and run the examples given in the lecture notes. ∎

1.2 Twice the Sum

Here is part of a program that begins by asking the user to input three integers:

```
#include <stdio.h>

int main(void)
{
    int a, b, c;

    printf("Input three integers: ");
```

. . .

Exercise 1.2.1 Complete the program so that when the user executes it and types in 2, 3, and 7, this is what appears on the screen:

```
Input three integers: 2 3 7
Twice the sum of integers plus 7 is 31 !
```
∎

1.3 Letter C

Execute this program so you understand the output:

```
#include <stdio.h>
```

```c
#define HEIGHT 17

int main(void)
{

    int i = 0;

    printf("\n\nIIIIIII\n");
    while(i < HEIGHT){
        printf("  III\n");
        i = i + 1;
    }
    printf("IIIIIII\n\n\n");
    return 0;

}
```

Exercise 1.3.1 Write a similar program that prints a large letter C on the screen (it doesn't need to be curved!). ∎

1.4   Lecture Notes Chapter C

Exercise 1.4.1 After you've studied Chapter C (*Grammar*), compile and run the examples given in the lecture notes. ∎

1.5   ++a++

Study the following code and write down what you think it prints.

```c
int a, b = 0, c = 0;
a = ++b + ++c;
printf("%i %i %i\n", a, b, c);
a = b++ + c++;
printf("%i %i %i\n", a, b, c);
a = ++b + c++;
printf("%i %i %i\n", a, b, c);
a = b-- + --c;
printf("%i %i %i\n", a, b, c);
```

Exercise 1.5.1 Then a program to check your answers. ∎

1.6   Randomness

The function `rand()` returns values in the interval [0, RAND_MAX]. If we declare the variable `median` and initialise it to have the value `RAND_MAX/2`, then `rand()` will return a value that is sometimes larger than `median` and sometimes smaller.

Exercise 1.6.1 Write a program that calls `rand()`, say 500 times, inside a `for` loop, increments the variable `minus_cnt` every time `rand()` returns a value less than `median`. Each time through the `for` loop, print out the value of the difference of `plus_cnt` and `minus_cnt`. You might think that this difference should oscillate near zero. Does it ? ∎

'

1.7 Lecture Notes Chapter D

Exercise 1.7.1 After you've studied Chapter D (*Flow Control*), compile and run the examples given in the lecture notes. ∎

1.8 find_max

Exercise 1.8.1 Write a program that finds the largest number entered by the user. Executing the program will produce something like:

```
How many numbers do you wish to enter ? 5
Enter 5 real numbers: 1.01 −3 2.2 7.0700 5
Maximum value: 7.07
```

∎

1.9 Loving Oddness

Suppose that you hate even integers but love odd ones.

Exercise 1.9.1 Modify the program you wrote for Exercise 1.8.1 so that all variables are of type `int`, even numbers are ignored and only odd integers are processed. Explain all this to the user via appropriate `printf()` statements. ∎

1.10 Linear Congruent Generator

One simple way to generate 'random' numbers is via a Linear Congruent Generator which might look something like this:

```
int seed = 0;
/* Linear Congruential Generator */
for(i=0; i<LOOPS; i++){
    seed = (A*seed + C) % M;
    /* Seed now contains your new random number */
}
```

here, A, C and M are constants defined in the code. All of these pseudo-generators have a period of repetition that is shorter than M. For instance, with A set to 9, C set to 5 and M set to 11 the sequence of numbers is :

```
5
6
4
8
0
5
```

and so repeats after 5 numbers (period equals 5).

Exercise 1.10.1 Adapt the above program so that it prints the period of the LCG, where you've `#defined` the constants A, C and M and `seed` always begins at zero. For the constants described above, the program would output 5. For $A = 7, C = 5$ and $M = 11$ it will output 10. ∎

1.11 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 1.11.1** 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. ∎

1.12 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 1.12.1** 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. ∎

1.13 Triangle Numbers

A Triangle number is the sum of numbers from 1 to n. The 5th 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 1.13.1 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 nth Triangle number based on the Equation :

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

■

1.14 Lecture Notes Chapter E

Exercise 1.14.1 After you've studied Chapter E (*Functions*), compile and run the examples given in the lecture notes. ■

1.15 Hailstone

The next number in a hailstone sequence is $n/2$ if the current number n is even, or $3n + 1$ if the current number is odd.

    So, for instance, if the initial number is 77, then the following sequence is produced:

```
77
232
116
58
29
88
44
22
11
34
```

Exercise 1.15.1 Write a program that, given a number typed by the user, prints out the sequence of *hailstone* numbers. The sequence terminates when it gets to 1. ■

1.16 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.

Note that whilst the sequences tend to be relatively short, hailstone numbers can be extremely large. You may need a storage type larger than an *int* for storing them.

Exercise 1.16.1 Write a program that :
- displays which initial number (less than 10,000,000) creates the **longest** hailstone

> sequence.
>   ■

> **Exercise 1.16.2** Write a program that :
>   • displays which initial number (less than $10,000,000$) leads to the **largest** number
>   appearing in the sequence.
>   ■

1.17 Primes

A prime number can only be exactly divided by itself or 1. The number 17 is prime, but 16 is not
because the numbers 2, 4 and 8 can divide it exactly. (Hint $16\%4 == 0$).

> **Exercise 1.17.1** Write a program that prints out the first n primes, where n is input by the
> user. The first 8 primes are:
>
> ```
> 2
> 3
> 5
> 7
> 11
> 13
> 7
> 19
> ```
>
> What is the 3000th prime ?   ■

1.18 Triangles

A triangle can be equilateral (all three sides have the same length), isosceles (has two equal
length sides), scalene (all the sides have a different length), or right angled where if the three
sides are a, b and c, and c is the longest, then : $c = \sqrt{a^2 + b^2}$

> **Exercise 1.18.1** Write a program so that you can process a number of triples of side lengths
> in a single run of your program using a suitable unlikely input value for the first integer in
> order to terminate the program. e.g. -999.
>   Think hard about the test data for your program to ensure that all possible cases are
> covered and all invalid data results in a sensible error message. Such cases can include sides
> of negative length, and impossible triangles (e.g. one side is longer than the sum of the other
> two).   ■

1.19 Time Flies

> **Exercise 1.19.1** Write a program which allows the user to enter two times in 24-hour clock
> format, and computes the length of time between the two, e.g.:
>
> ```
> Enter two times : 23:00 04:15
> Difference is : 5:15
> ```
>
> or,
>
> ```
> Enter two times : 23:40 22:50
> ```

```
Difference is : 23:10
```

1.20 Dirichlet Distribution

When you first start thinking about prime numbers, it becomes pretty obvious that most of them end with the digit $1, 3, 7$ or 9 (they can't end with two, because they can't be even etc.)

> **Exercise 1.20.1** Write a program that computes what fraction of all primes end with a 3. You can do this by generating a large number of primes, in sequence, and keeping a running count of the number that end with 3, as compared to the total of primes generated. ∎

2. Mathematics & Characters

Note that these exercises are in **NO** particular order - try the ones you find more straightforwards before attempting complex ones. Remember to assert `test()` all of your functions.

2.1 Lecture Notes Chapter F

Exercise 2.1.1 After you've studied Chapter F (*Mathematics & Characters*), compile and run the examples given in the lecture notes. ∎

2.2 Unit Circle

In mathematics, for all (real) x, it is true that:

$$sin^2(x) + cos^2(x) = 1$$

i.e. $sin(x) * sin(x) + cos(x) * cos(x) = 1$.

Exercise 2.2.1 Write a program to demonstrate this for values of x input by the user. ∎

2.3 Monte Carlo Π

At :

www `https://www.geeksforgeeks.org/estimating-value-pi-using-monte-carlo`

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 $x^2 + y^2 \leq r^2$.

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 2.3.1 Write a program to run this simulation, and display the improving version of the approximation to π. ∎

2.4 Leibniz Π

See:

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

The Mathematical constant π can be approximated using the formula :

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

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

Exercise 2.4.1 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 7 decimal places ? ∎

2.5 Irrational numbers

Neill's favourite number (don't ask why!) is e which has the value $2.71828182845904523536\ldots$ This is an example of an irrational number - one that can only ever be *approximated* by the ratio of two integers - a poor approximation of e is 87 divided by 32, that is $\frac{87}{32}$.

Exercise 2.5.1 Write a program that loops through all possible denominators (that is the integer on the bottom, b in the fraction $\frac{a}{b}$) and finds which a and b pair give the best approximation to e. The output of your program might look something like :

```
271801/99990 = 2.71828182818281849364
```

You need only investigate denominators $< 100,000$. #define the number being searched for. Check your code works for other famous constants such as $\pi = 3.14159265358979323846\ldots$:

```
312689/99532 = 3.14159265361893647039
```

∎

2.6 Fibonacci Words Using ϕ

www   `https://en.wikipedia.org/wiki/Fibonacci_word`

A *Fibonacci Word* is simply a particular (infinite) sequence of digits:

0100101001001010010100100101001001010010100100101001010010010100100010...

and shouldn't be confused with the related, but totally different *Fibonacci Sequence* (which we'll see later in the lecture series).

You start by defining S_0 to be the sequence "0", and S_1 to be "01". Each subsequent part of the sequence is made by simply concatenating the previous two, so $S_2 = 010$ and $S_3 = 01001$ and so on.

Perhaps somewhat surprisingly, the nth-digit can be predicted without computing the rest of the sequence, using the equation :

$$2 + \lfloor n\phi \rfloor - \lfloor (n+1)\phi \rfloor \tag{2.2}$$

or, maybe more understandably :

$$2 + \text{floor}(n\phi) - \text{floor}((n+1)\phi) \tag{2.3}$$

Here `floor()` is the function that rounds a number down to its lowest integer part (e.g. 1.815 becomes 1.0), and where ϕ is the golden ratio $= 1.61803398875...$. Note also that the sequence is assumed to begin at the $n = 1$, and not at zero which a programmer might assume!

> **Exercise 2.6.1** Write the function:
>
>   bool fibword_phi(int n)
>
> which returns to n^{th}-digit of the sequence (which begins at $n = 1$) using: an approximation of ϕ, the function `floor()` found in the mathematics library, and Equation 2.3. ∎

[Since ϕ can only be stored approximately on a computer, for larger numbers of n, this equation doesn't hold. Exactly where this happens depends upon how closely you have approximated ϕ. We will elore this later in Exercise 3.8.2.]

2.7 Vowelness

Vowels are the letters *a*, *e*, *i*, *o* and *u*.

> **Exercise 2.7.1** Write a program that reads characters from the keyboard and writes to the screen. Write all vowels as uppercase letters, and all non-vowels as lowercase letters. Do this by using writing a function `isvowel()` that tests whether or not a character is a vowel. ∎

2.8 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 2.8.1** 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
```

2.9 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 2.9.1 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
```

■

2.10   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 2.10.1 Write a program which includes a function whose 'top-line' is :

int secret(**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 !
```

■

2.11   Lecture Notes Chapter G

Exercise 2.11.1 After you've studied Chapter G (*Prettifying*), compile and run the examples given in the lecture notes.   ■

2.12   Roulette

This is an chance for you to practice self-document techniques such as sensible identifier naming, commenting, `typedefs` and enumeration.

Exercise 2.12.1 Write a roulette program. The roulette machine will select a number between 0 and 35 at random. The player can place an odd/even bet, or a bet on a particular number. A winning odd/even bet is paid off at 2 to 1, except that all odd/even bets lose if the roulette selects 0. If the player places a bet on a particular number, and the roulette selects it, the player is paid off at 35 to 1.   ■

3. $1D$ Arrays & Strings

Note that these exercises are in **NO** particular order - try the ones you find more straightforwards before attempting complex ones. Remember to assert `test()` all of your functions.

3.1 Lecture Notes Chapter H

Exercise 3.1.1 After you've studied Chapter H (*1D Arrays*), compile and run the examples given in the lecture notes. ∎

3.2 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 3.2.1 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
```
∎

3.3   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 3.3.1 Write a program that gets a number from the user (to represent the number of songs required) and outputs a randomised list.   ∎

Exercise 3.3.2 Rewrite Exercise 3.3.1 so that the program passes an array of integers (e.g. `[1,2,3,4,5,6,7,8,9,10]`) to a function which shuffles them **in-place** (no other arrays are used) and with an algorithm having complexity $O(n)$.   ∎

3.4   Yahtzee

The game of Yahtzee is a game played with five dice, and you try to obtain certain 'hands'. In a similar way to poker, these hands could include a *Full House* (two dice are the same, and another three are the same), e.g.:

⚃⚃⚁⚃⚁⚀ or ⚁⚀⚁⚀⚁

or another possible hand is *Four-of-a-Kind*, e.g.: ⚁⚁⚀⚁⚁ or ⚄⚄⚀⚄⚄ (but not ⚁⚁⚁⚁⚁ which is *Five-of-a-Kind*)

A little mathematics tells use that the probability of these two hands should be 3.85% and 1.93% respectively.

Exercise 3.4.1 Complete the program `yahtzee.c` (which is in the usual place online), by analysing a large number of random dice rolls, the probabilty of each of these two hands. The five dice of the hand are stored in an array, and to facilitate deciding which hand you've got, a histogram is computed to say how often a ⚀ occurs in the hand, how often a ⚁ occurs and so one. A *Full-House* occurs when both a 2 and a 3 occurs in the histogram; a *Four-of-a-Kind* occurs when there is a 4 somewhere in the histogram.
                                                                                          ∎

3.5   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:
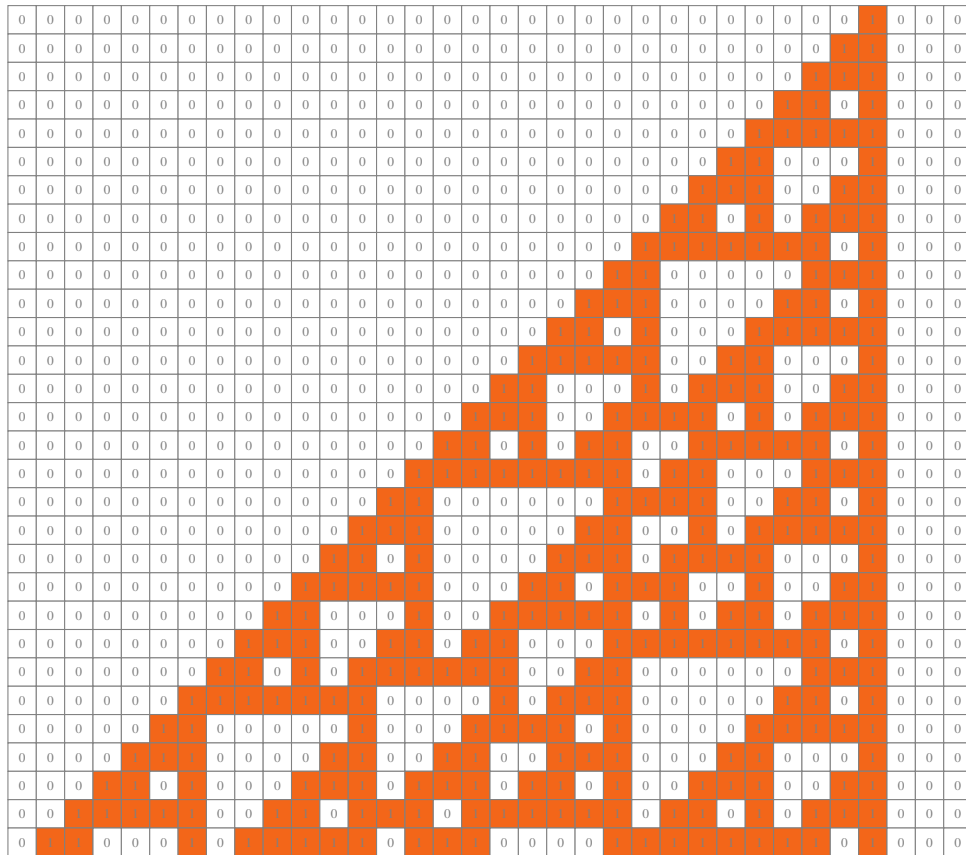
Figure 3.1: 1D cellular automaton using Rule 110. Top line shows initial state, each subsequent line is produced from the line above it. Each cell has a rule to switch it 'on' or 'off' based on the state of the three cells above it in the diagram.

Exercise 3.5.1 Write a program that outputs something similar to Figure 3.1, but 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. Do not use 2D arrays for this - a couple of 1D arrays is sufficient. ∎

Exercise 3.5.2 Rewrite the program above to allow other rules to be displayed - for instance 124, 30 and 90.

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

■

3.6 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, then 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[1]. For more details see :

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

Exercise 3.6.1 Write a program that repeatedly chooses two positions on the board at random and swaps them. 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
```

---

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

```
 7  20   2  13  15   1  17  12
```

Is the score of 19874 the lowest possible that may be obtained via this technique ?   ∎

3.7 Counting Sort

UNDER CONSTRUCTION - Please don't attempt this yet.
We'll have this new exciting exercise ready for you soon!

3.8 Fibonacci Words using Substitution Rules

This follows on from Exercise 2.6.1, where we saw a good approximation of the Fibonacci Word using ϕ.

To be certain of the sequence at very large values of n, you'd need to construct the sequence iteratively. One interesting way of doing this is to use a Substitution Rule. Start with a counter indexing a single digit zero at the beginning of a `bool` array. Now at each step, if the digit is 0, append a 1 and a 0 onto the end of the array. If the digit is a 1 append a 0 onto the end of the array. Either way, move along to the next digit and repeat.

```
Array initially :
0 (index zero contains a zero therefore append 1 0)
010 (index 1 contains a one therefore append 0)
0100 (index 2 contains a zero therefore append 1 0)
010010 (index 3 ...)
```

Exercise 3.8.1 Write the function:

bool fibword_subs(**int** n)

which returns to n^{th}-digit of the sequence (which begins at $n = 1$) using an array of `bool`s, and repeatedly applying the substitution rules described above, until you have a sequence large enough to extract the nth-digit.   ∎

Exercise 3.8.2 Write a program which experiments with the approximation of ϕ from Exercise 2.6.1, to see at what value of n it begins to return incorrect values. These 'correct' values can be obtained from the function written in Exercise 3.8.1.

In my implementation, when I

#define PHI 1.61

the values are wrong from the 12th digit. For:

#define PHI 1.61803

the values are wrong from the 609th digit.

                                                                                       ∎

3.9 Lecture Notes Chapter I

> **Exercise 3.9.1** After you've studied Chapter I (*Strings*), compile and run the examples given in the lecture notes. ∎

3.10 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 3.10.1** 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()`.
>   Test your program with (amongst others) the following palindromes :
> ```
> "kayak"
> "A man, a plan, a canal: Panama!"
> "Madam, in Eden I'm Adam,"
> "Level, madam, level!"
> ```
> ∎

3.11 Int to String

> **Exercise 3.11.1** Write a function that converts an integer to a string, using the skeleton code in the usual place online. works correctly:
>   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. `snprintf()`) including all those in `string.h`. ∎

3.12 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 3.12.1 Write a program that contains a function which is passed a string and returns an integer. The string is a roman numeral in the range `1 - 3999`. Amongst others, `assert()` test that MCMXCIX returns 1999, MCMLXVII returns 1967 and that MCDXCI returns 1491. ∎

3.13 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 3.13.1 Write a program that contains a function which is passed a name as a string, and returns the soundex code for it. Using `assert()` tests, check it works correctly for, amongst others, all the examples above.

∎

3.14 Fibonacci Words using Strings

See Section 2.6.

Exercise 3.14.1 Write the function:

```
bool fibword_str(int n)
```

which returns to n^{th}-digit of the sequence (which begins at $n = 1$) using strings. Previous strings S_{k-1} and S_{k-2} are repeatedly concatenated to create the new S_k, until you have a sequence large enough to extract the nth-digit.

To achieve this, use fixed-size strings that are set to always be large enough.   ■

3.15 Merging Strings

Experience shows that the next exercise is trickier than most - you have been warned!

Exercise 3.15.1 Write the function `strmerge()` which concatenetes `s1` and `s2` into `s3`, discarding any overlapping characters which are common to the end (tail) of `s1` and the beginning (head) of `s2`. To help, I've put some skeleton code in the usual place online.

Hint : Functions such as `strncmp()` and `strcat` might be useful.   ■

3.16 Rot18

ROT18 is an encryption algorithm made of the combination of ROT13 and ROT5. ROT13 replaces each letter in a string with the letter 13 places later in the alphabet, wrapping back to the beginning of the alphabet where necessary e.g. 'a' becomes 'n', 'b' becomes 'o', and 'n' becomes 'a'. ROT5 replaces each digit in a string with the digit 5 greater, wrapping back to 0 where necessary e.g. '0' becomes '5', '1' becomes '6', and '5' becomes '0'.

www   `https://www.boxentriq.com/code-breaking/rot13`

Exercise 3.16.1 Write the function whose "top-line" is:

void rot(**char** str[])

that takes a string as input and encrypts it using the ROT18 cipher. The function must preserve the case of any letters, and any characters that aren't letters or digits must be unaffected.

When the string `"Hello, World!"` is encrypted, it becomes `"Uryyb, Jbeyq!"` and when ecrypted a second time, reverts to the original string.

```
Erzrzore:  fubeg shapgvbaf; ubhfr-fglyr ehyrf :-)
```

■

4. *2D* Arrays

Note that these exercises are in **NO** particular order - try the ones you find more straightforwards before attempting complex ones. Remember to assert `test()` all of your functions.

4.1 Crosswords

Many newspapers in the UK have a puzzles section. One of the most well established such puzzle is the *crossword*.

www `https://en.wikipedia.org/wiki/Crossword`

A crossword is a 2*D* grid of full and empty cells. Clues are given so that the empty squares can be filled-in with answers to *across* or *down* clues.
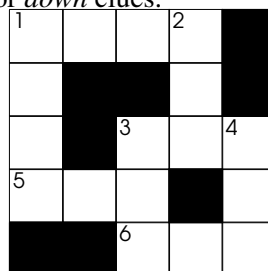
Across 1 Tailless amphibian 3 Join two items together 5 Pull something suddenly 6 Large deciduous tree

Down 1 Shape having one face and size 2 Higher being 3 Period of History 4 Water barrier

Here though, we aren't going to be dealing with solving crosswords, only in understanding something about their structure.

Firstly, let us consider how crosswords are numbered. In the above example the first square is shared by both 1 across and 1 down. Working from the top-left rightwards and **then downwards**, squares that are at the start of a new word are numbered. The second square in the above example has no number because it is already part of 1 across, and there is no room for a down clue. On the fourth square, there is room for a down clue, but not for an across since it is already being used by 1 down.

Actually, the algorithm for deciding on whether a square should be numbered or not is fairly simple:

1. All squares outside (off the grid) of the board are considered full.
2. The pattern:

   shows that we can start a new across clue from the middle square.
3. Similarly, the pattern:

   shows that we can start a new down clue from the middle square.
4. Sometimes a square can be the start of both an across and down clue. In this case, they must share the same number.

Secondly, another important concept in crossword clues is that of the number of *checked* squares. This is the number of squares which are shared by both an across clue and a down clue. In the above crossword, nine squares are shared of the 17 empty ones which rounds to 53%. This is shown below:
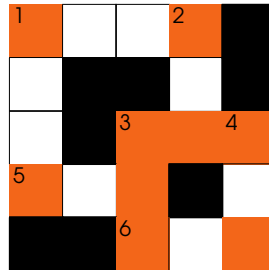
> **Exercise 4.1.1** Complete the file **crossword.c** which, along with my files *crossword.h* and *cwddriver.c*, allows a crossword to be created such that the clue numberings and percentage of checked squares can be found.
>
> My file *crossword.h* contains the function definitions which you'll have to implement in your **crossword.c** file. My file *cwddriver.c* contains the `main()` function to act as a driver to run the code. Your file will contain many other functions as well as those specified, so you'll wish to test them as normal using the `test()` function.
>
> If all of these files are in the same directory, you can compile them using:
>
> ```
> clang crossword.c cwddriver.c -o crossword -Wall -Wextra
> -Wpedantic -std=c99 -Wvla -Wfloat-equal -O3 -Werror -lm
> ```
>
> Do not alter or resubmit *crossword.h* or *cwddriver.c* - my original versions will be used to compile the *crossword.c* file that you adapt. ∎

4.2 Binary Grid Puzzle

Many newspapers in the UK have a puzzles section. One of that has become popular recently is the *Binary Grid*

www   `https://www.binarypuzzle.com/`

A Binary Grid is a *2D* square grid of cells, each of which is either empty, or contains the value 0 or 1. A set of rules governs what is legal in a **valid** board :
- Each cell should contain either a zero or a one.

- No more than two of the same number can appear consecutively in a row or a column.
- There must be an equal number of 0s and 1s in each row and column - this means that the height/width of the puzzle must be even.

Given these rules[1] we can deduce some principles to help us find a solution :

Pairs : If two of the same number (0s or 1s) are in adjacent cells (either horizontally or vertically) then the two surrounding cells on either side must be the other number, otherwise there would be three of the same number in sequence.

OXO : If there are two of the same number with a gap of one empty cell between them (horizontally or vertically), then the opposite number must be in the empty cell (otherwise we'd have three of the same number in sequence).

Counting : If all of a particular number have been found in a row (or column) - that is 3 if the grid is a 6x6, then the remaining 3 unknown cells must all contain the other number.

An example puzzle might look so :

·	·	·	·	·	0
0	·	·	1	1	·
·	·	·	1	·	·
·	·	·	·	·	0
·	·	·	·	·	·
·	·	·	·	·	·

By applying the **Pairs** principle 4 times we get :

·	·	·	0	·	0
0	·	0	1	1	0
·	·	·	1	·	·
·	·	·	0	·	0
·	·	·	·	·	·
·	·	·	·	·	·

The **OXO** principle gives :

·	·	·	0	·	0
0	1	0	1	1	0
·	·	·	1	·	·
·	·	·	0	·	0
·	·	·	·	·	·
·	·	·	·	·	·

Application of the Counting principle can now fill the rightmost column :

---

[1]In the full version of this puzzle, no row or column can be the same - we ignore this constraint here.

.	.	.	0	.	0
0	1	0	1	1	0
.	.	.	1	.	1
.	.	.	0	.	0
.	.	.	.	.	1
.	.	.	.	.	1

Some (but not all) puzzles can be solved by repeatedly applying these three rules. This is what we explore in this assignment.

Exercise 4.2.1 Code for this exercise can be found in

```
https://github.com/csnwc/Exercises-In-C
```

then navigate into `Code/Week4/BinaryGrid`.

Complete the file **bingrid.c** which, along with my files *bingrid.h* and *bingrid_driver.c*, allows Binary Grids to be solved.

My file *bingrid.h* contains the function definitions which you'll have to implement in your **bingrid.c** file. My file *bingrid_driver.c* contains the `main()` function to act as a driver to run the code. Your file will contain many other functions as well as those specified, so you'll wish to test them as normal using the `test()` function.

If all of these files are in the same directory, you can compile them using the `Makefile` given. Do not alter or resubmit *bingrid.h* or *bingrid_driver.c* - my original versions will be used to compile the *bingrid.c* file that you adapt.

4.3   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	0	0	0
0	0	1	0	0		0	0	0	0	0
0	0	1	0	0		0	1	1	1	0
0	0	1	0	0		0	0	0	0	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 :

```
0  -1
```

means the middle row, one cell to the left of the centre.

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

> **Exercise 4.3.1** 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 4.3.2** 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. ∎

4.4 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 4.4.1** 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
> ```

4.5 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 4.5.1 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

∎

4.6 Forest Fire

You can create a very simple model of forest fires using a cellular automaton in the form a $2D$ grid of cells. Each cell can be in one of three states; either 'empty', 'tree', or 'fire'. The next generation of cells follows these rules:

- A 'fire' cell will turn into an 'empty' cell.
- A 'tree' that is within the 8-neighbourhood of a 'fire' cell will itself become 'fire'.
- A 'tree' will burn (due to a lightning strike) 1 time in L.
- An 'empty' space will become a 'tree' (spontaneous growth) 1 time in G.

See also:

www   `https://en.wikipedia.org/wiki/Forest-fire_model`

www   `https://www.aryan.app/randomstuff/forestfire.html`

You can experiment with different values of L and G, but a useful starting point is $G = 250$ and $L = 10 \times G$.

> **Exercise 4.6.1** Write a program which creates an empty $2D$ grid of cells, 80 wide and 30 high. Then, apply the rules above to iterate the simulation, so that the next generation is created from the previous one.
>
> Print out every generation onto the screen, using a space to represent an empty cell, the @ character for a tree, and * for fire cells. Display the board for 1000 generations using plain text. You may assume that the grid is always 80 cells by 30
>
> ■

4.7 Diffusion Limited Aggregation

www `https://en.wikipedia.org/wiki/Diffusion-limited_aggregation`

In its simplest form, DLA occurs on a grid of square cells. The cell at the center of the grid is the location of the seed point, a particle stuck at that square. Now pick a square on the perimeter of the grid and place a wandering particle on that square. At each iteration, this particle moves to one of the four adjacent squares, left, right, above, or below. When a wandering particle arrives at one of the four squares adjacent to the seed, it sticks there forming a cluster of two particles, and another edge particle is released. When a moving particle arrives at one of the squares adjacent to the cluster, it sticks there.
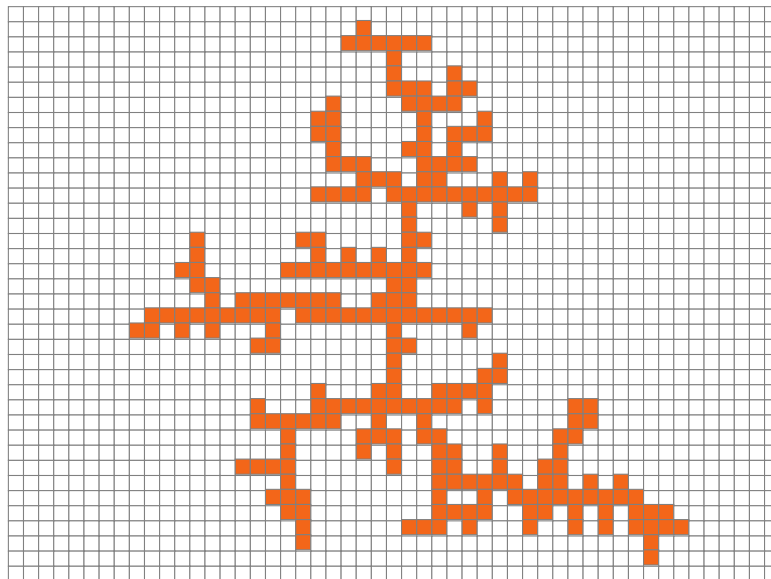
Note that the grid is toroidal - this means that if a particle goes off the top of the grid, it reappears at the bottom. Likewise, if it goes off the edge, it will appear on the other side.

> **Exercise 4.7.1** Using the algorithm outlined above, using a 50×50 board, output each of 250 iterations (here iteration means when each particle has become 'stuck'). The output should be in plain text.   ■

In the basic version of a DLA, a particle 'sticks' to the seed points when it gets adjacent to one. Here we introcude the probability of stickiness, a number between 0 and 1, called p_s. The particle only sticks to a seed point with a probabilty of p_s; if it doesn't stick it continues wandering. This allows for the patterns that emerge to be more hairy and solid:

`http://paulbourke.net/fractals/dla/`

> **Exercise 4.7.2** Implement this stickiness concept by extending the program writen for Exercise 4.7.1, allowing the user to specify a 'stickiness' probablity via the command line using argv[1]. Here a value of 1.0 means that the particle will always stick, and 0.5 means that it will stick one time in 2.   ∎

Here's an example when setting $p_s = 0.25$:
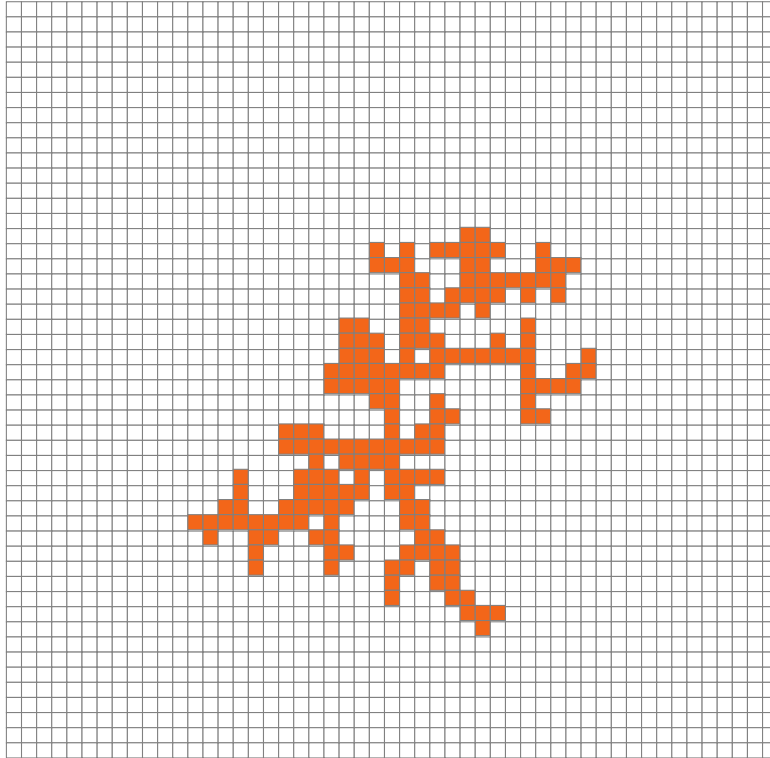
Using different planar walks. e.g. in base-4: `http://demonstrations.wolfram.com/UsingIrrationalSqua`

4.8 Langton's Ant

`http://en.wikipedia.org/wiki/Langton's_ant`

> **Exercise 4.8.1** Write a program that shows the behaviour of Langton's Ant and displays it on the screen using text. The ant will start in the middle, and each cell will have 2 states : '#' (white) or '.' (black).
>
>   At each step, the ant:
>   - At a white square, turn 90° right, flip the color of the square, move forward one unit
>   - At a black square, turn 90° left, flip the color of the square, move forward one unit
>
>   After each step, display the board, and wait for the user to type a character before repeating.   ∎

After 30 or so iterations, the board might look something like :

4.9 Minesweeper

The game *Minesweeper*

www   `https://en.wikipedia.org/wiki/Minesweeper_(video_game)`

is a logic puzzle game, played on a two-dimensional grid of squares. There are 'mines' hidden in the grid, and other, numbered squares, tell you how many mines there are in that square's (eight-count) Moore neighbourhood.

www   `https://en.wikipedia.org/wiki/Moore_neighborhood`

You will know in advance the width and height of the grid, and also the total number of mines in the completed (solved) grid. In our version of the game, we'll be using just two rules to 'solve' the grid by working out the unknown squares.

Rule 1 (**Count the Mines**): If we've discovered all the mines on the board already, then any unknown cell can simply be numbered with the count of mines in its Moore neighbourhood.

In the follwing grid, if we somehow know in advance that the total number of mines in the grid is five, and if these have all been found:

0	1	1	?	0
1	3	X	3	1
1	X	X	X	1
1	3	X	3	1
0	1	1	1	0

then the solution to the unknown square must be:

0	1	1	1	0
1	3	X	3	1
1	X	X	X	1
1	3	X	3	1
0	1	1	1	0

Rule 2 (**Unknowns are Mines**): For a known square having the number n, with u unknown and m known mines in its Moore neighbourhood, if $n = m + u$ and $u > 0$ then all u unknown squares should be made mines.

Applying Rule 2 to this grid:

0	1	1	1	0
1	3	X	3	1
1	X	X	X	1
1	3	?	3	1
0	1	1	1	0

at the middle square on the bottom row (for instance), yields:

0	1	1	1	0
1	3	X	3	1
1	X	X	X	1
1	3	X	3	1
0	1	1	1	0

Repeated application of these rules will allow some (but not all) boards to be solved.

Exercise 4.9.1 Code for this exercise can be found in

```
https://github.com/csnwc/Exercises-In-C
```

then navigate into `Code/Week4/Minesweeper`.
   Complete the file *ms.c* which, along with my files *ms.h* and *drv.c*, allows the puzzles to be solved.
My file *ms.h* contains the function definitions which you'll have to implement in your **ms.c** file. My file *drv.c* contains the `main()` function to act as a driver to run the code. Your file will contain many other functions as well as those specified, so you'll wish to test them as normal using the `test()` function.
If all of these files are in the same directory, you can compile them using the `Makefile` given. Do not alter or resubmit *ms.h* or *drv.c* - my original versions will be used to compile the *ms.c* file that you create. Only submit *ms.c*

5. Files, argc and Graphics

Data for these exercises are available in the github repository *Data*, including lists of English words in *Data/Words*.

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 5.1.1 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
```

An anagram is defined here to be a *different* word, so if a valid word is entered, it itself is not reported, e.g. :

```
% ./anagram pots
tops
stop
opts
spot
post
```

Exercise 5.1.2 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 5.1.3 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   ANSI Escape Sequences

C has no inherent functionality to allow printing in colour etc. Therefore, some terminals allow certain control codes to be printed to move the cursor, change the foreground and background colour, and so on. Note that only some terminals support these control code - if you're using a different one, garbage may appear on the screen instead.

www   `https://en.wikipedia.org/wiki/ANSI_escape_code`

I've created a very simple set of functions which shows a small fraction of this functionality

called `demo_neillsimplescreen.c` which using the functions found in `neillsimplescreen.c`

To build and execute this code, use the `Makefile` provided by typing:

```
$ make demo_neillsimplescreen
gcc demo_neillsimplescreen.c neillsimplescreen.c
-o demo_neillsimplescreen -Wall -Wextra -pedantic
-std=c99 -Wvla -g3 -fsanitize=undefined -fsanitize=address -lm
$ ./demo_neillsimplescreen
```

Exercise 5.2.1 Adapt any of the exercises from Chapter 4, so that output is displayed using the simple functions demonstrated above, rather than plain text. ■

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 `demo_neillsdl2.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 5.3.1 Using the `Makefile` provided, compile and run this program. Now adapt it, so that the colour of the boxes displayed are all (random) shades of red.

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 A Simple Spelling Checker

Here, we reads words one at a time from file and carefully place them in the **correct** part of our data structure. This has a complexty of $O(n^2)$.

For this purpose, a list of valid words (unsorted) is available from `github : Data/Words`.

Exercise 5.4.1 Write a program which, based on a fixed-size **array** of strings, reads the words in one at a time, inserting them into the **correct** part of the array so that the words are alphabetically sorted. The name of the file should be passed as `argv[1]`, and you can assume the array is large enough to hold all words. How long does it take to build the list ? ■

Exercise 5.4.2 Now extend Exercise 5.4.1 so that when the user is prompted for a word, they are told whether this word is present in the array or not. Use a *Linear Search* for this: start at the beginning of the list and work your way through it one word at a time. ■

---

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

Exercise 5.4.3 Extend Exercise 5.4.2 using a *Binary Search* for this.   ∎

Exercise 5.4.4 Write a program which, based on a linked list data structure, reads the words in one at a time, inserting them into the **correct** part of the list so that the words are alphabetically sorted. The name of the file should be passed as `argv[1]`. How long does it take to build the list ?
How long does it take to do a linear search ?   ∎

5.5 Crush It!

Match-3 tile games have become one of the world's most popular games.

www   `https://en.wikipedia.org/wiki/Tile-matching_video_game`

Such games use a rectangular grid (board) containing many tiles, of many different types (often colour, but here we will use letters). Where there are 3 or more tiles of the same type in a line (horizontally or vertically), they are removed. Once all removals that are possible have occurred, the tiles above fall down to fill in the gaps. In our version, a large number of tiles are available above (and hidden from the player) which cannot be matched until they have fallen down into the playing area.

   In our version of the game:
   • Matchable tiles are in the range $(A \ldots Z)$, although it's is common for only a small number (e.g. $A \ldots D$) to be used.
   • The width of the board is always five tiles.
   • The 'playing' height of the board is six. Other tiles can be above this, but won't be matched until they have dropped down into one of the bottom six rows.
   • The maximum number of rows the board ever needs to hold is 20.
   • Matching (that is finding a horizonal or vertical line of the same tile) is done in 'parallel' - if a tile is shared between two matches (e.g. the middle tile in a 3×3 '+' pattern) both of these matches are removed.
   An example of this is shown here:

·	·	·	·	·
B	B	B	D	B
C	D	A	A	C
D	A	A	B	D
A	A	B	C	A
A	B	C	D	A
B	C	A	A	A

·	·	·	·	·
·	·	·	D	B
C	D	A	A	C
D	A	A	B	D
A	A	B	C	·
A	B	C	D	·
B	C	·	·	·

·	·	·	·	·
·	·	·	·	·
C	D	·	D	·
D	A	A	A	·
A	A	A	B	B
A	B	B	C	C
B	C	C	D	D

(Left) Game board in it's initial state. Orange squares show were matches can be made. (Middle) Three matches are made - one for the 3 horizonal 'A' tiles, one for the three vertical 'A' tiles and one for the three horizontal 'B' tiles. (Right) Game board final state after tiles are dropped down. Not there are now more matches that can be made.

Exercise 5.5.1 Here we will write some (but not all) of the functionality necessary for a match-3 tile game. Skeleton code may be found in :

www
```
https://github.com/csnwc/Exercises-In-C
```

then navigate into `Code/Week5/CrushIt`.

Complete the files **crushit.c** and **mydefs.h** which, along with my files *crushit.h* and *driver.c*, implements some important functionality necessary for a game of this type.

My file *crushit.h* contains the function definitions which you'll have to implement in your **crushit.c** file. My file *driver.c* contains the `main()` function to act as a driver to run the code. Your file will contain many other functions as well as those specified, so you'll wish to test them as normal using a `test()` function.

If all of these files are in the same directory, you can compile them using the `Makefile` given. The functions you need to complete include:

`initialise()` - this takes a pointer to the board state, and a string. The string can be a filename, but if this filename can't be opened, it is assumed to be a list of tiles to fill the board with, from the top down. Such a string must contain complete rows of tiles, with no partial rows.

`match()` - this takes a pointer to the board state, and removes all matches of 3 or more tiles in a vertical or horizontal line. Removed tiles are replaces with the '.' (empty tile) character.

`dropblocks()` - this takes a pointer to the board state, and makes any blocks that are above an empty tile fall down until all holes are filled in if it is possible to do so.

`tostring()` - this takes a pointer to the board state, and a string and copies whole rows of the board into the string from the top downwards. The whole board isn't copied since most of the characters at the top are unused (hole) tiles. Therefore, we begin copying at the first row on which a non-hole tile appears.

Hints:

- Do not begin by writing the file handling functionality - this cannot be tested, so use the string initialising option instead.
- To begin with, use you're own, simpler driver file - mine makes sense once everything is working, but may seem complex to begin with.
- Your `crushit.c` file should contain many other sub-functions which are used by the major ones specified. You can put anything in this file, provided it still compiles as specified.
- Do not alter or resubmit *crushit.h*, *Makefile* or *driver.c* - my original versions (or even slightly different ones) will be used to compile the *crushit.c* file that you adapt.

7. Recursion

7.1 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
```

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 $4, 1, 2, 3$.

```
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.

Exercise 7.1.1 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 the 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 7.1.2 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 7.1.3 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 solution, and via the record of the parents, have the correct route. This is shown for the word ladder CAT to DOG in Figure 7.1 using a very small subset of the possible three letter words.
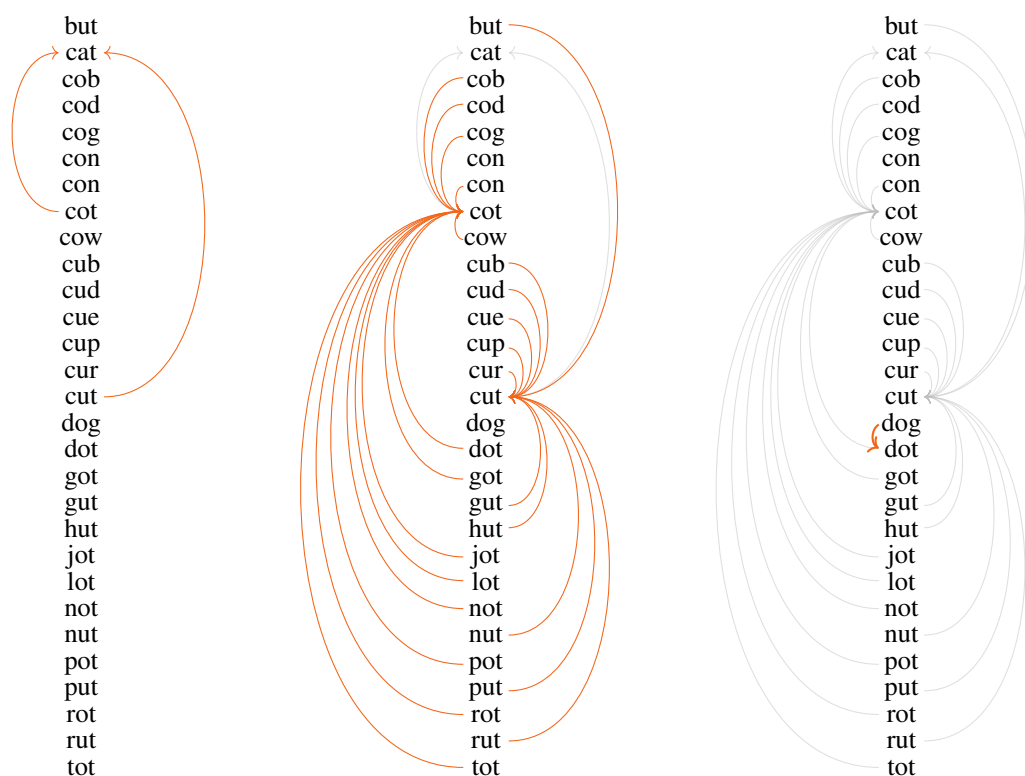
7.2 Maze

This exercise is new for 2023

Figure 7.1: 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.

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

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

> **Exercise 7.2.1** 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 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.
> ∎

It becomes obvious that the walls of every maze (having one unique solution) must consist of two separate sections :

```
A A A A A A A A A A
  . . A           . . A
B . A . B . B B . A
B . A . B B B B . A
B . A         B . A
B . A . B B B B . A
B . A         B . A
B . A A A A . B . A
B . . . . . . B . .
B B B B B B B B B B
```

> **Exercise 7.2.2** Write a program to read in a maze in the same manner as in Exercise 7.2.1, and then display the two sections using the characters A and B.   ∎

7.3 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 7.3.1 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 ?

7.4 Sierpinski Carpet

www `en.wikipedia.org/wiki/Sierpinski_carpet`

The square is cut into 9 congruent subsquares in a 3-by-3 grid, and the central subsquare is removed. The same procedure is then applied recursively to the remaining 8 subsquares, ad infinitum.

www `http://www.evilmadscientist.com/2008/sierpinski-cookies/`

> **Exercise 7.4.1** Write a program that, in plain text, produces a Sierpinski Carpet.   ∎

> **Exercise 7.4.2** Write a program that, using `neillsimplescreen`, produces a Sierpinski Carpet.   ∎
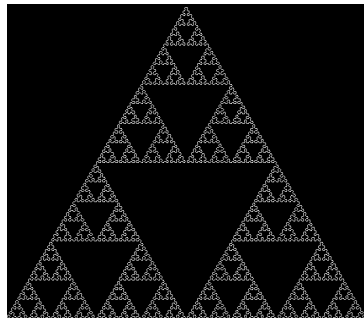
> **Exercise 7.4.3** Write a program that, using SDL, produces a Sierpinski Carpet.   ∎

7.5 Sierpinski Squares

See also :

www   `en.wikipedia.org/wiki/Sierpinski_triangle`

The Sierpinski triangle has the overall shape of an equilateral triangle, recursively subdivded into four smaller triangles :

However, we can approximate it by recursively drawing a square as three smaller squares, as show below :

The recursion should terminate when the squares are too small to draw with any more detail (e.g. one pixel, or one character in size).

> **Exercise 7.5.1** Write a program that, in plain text, produces a Sierpinski Triangle.   ∎

> **Exercise 7.5.2** Write a program that, using `neillsimplescreen`, produces a Sierpinski Triangle.   ∎

> **Exercise 7.5.3** Write a program that, using SDL, produces a Sierpinski Triangle.   ∎

7.6 Prime Factors

www   `en.wikipedia.org/wiki/Prime_factor`

It is well known that any positive integer has a single *unique* prime factorization, e.g.:

210 = 7 × 5 × 3 × 2 (the numbers 7, 5, 3 and 2 are all prime).
117 = 13 × 3 × 3 (the numbers 13 and 3 are all prime).
197 is prime, so has only itself (and 1, which we ignore) as a factor.

Exercise 7.6.1 Write a program that, for any given positive integer input using argv[1], lists the prime factors, e.g.:

```
[campbell@icy]% ./primefacts 210
7 5 3 2

[campbell@icy]% ./primefacts 117
13 3 3
```

■

Exercise 7.6.2 To make the output of the above program briefer, many prefer to show the factors expressed by their power, as in :

$$768 = 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 3$$

could be better expressed as :

$$768 = 2^8 \times 3$$

Write a program to show the factorisation of a number in this more compact style :

```
% ./primefactors 27000
27000 = 1 x 2^3 x 3^3 x 5^3
% ./primefactors 31
31 = 1 x 31
% ./primefactors 38654705664
38654705664 = 1 x 2^32 x 3^2
```
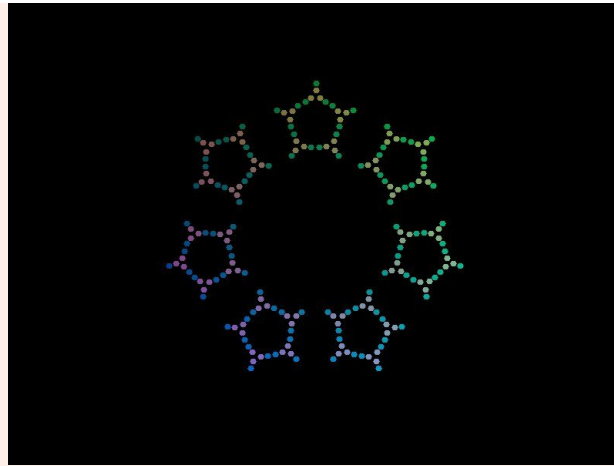
■

Exercise 7.6.3

Experience shows that the next exercise is trickier than most - you have been warned!

For a beautiful visualisation of prime factors, see:

www.datapointed.net/visualizations/math/factorization/animated-diagrams

Adapt the program above to output a pattern similar to the animated display above, using SDL, but only for a single number given via argv[1].

8. Searching Boards

8.1 Conway's Soldiers

The one player game, *Conway's Soldiers* (sometimes known as *Solitaire Army*), is similar to peg solitaire. For this exercise, Conway's board is a 7 (width) × 8 (height) board with tiles on it. The lower half of the board is entirely filled with tiles (pegs), and the upper half is completely empty. A tile can move by jumping another tile, either horizontally or vertically (but never diagonally) onto an empty square. The jumped tile is then removed from the board. A few possible moves are shown below:
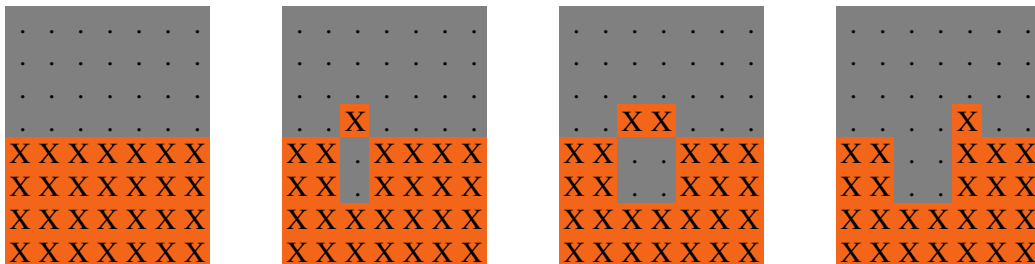
The user enters the location of an empty square they'd like to get a tile into, and the program demonstrates the moves that enables the tile to reach there (or warns them it's impossible). To do this you will use a list of boards. The initial board is put into this list. Each board in the list is, in turn, read from the list and all possible moves from that board added into the list. The next board is taken, and all its resulting boards are added, and so on.

Each structure in the list will contain (amongst other things) a board and a record of its parent board, i.e. the board that it was created from.

> **Exercise 8.1.1** Write a program that:
> - Inputs a target location for a tile to reach (x in argv[1], y in argv[2]).
> - Demonstrates the correct solution (reverse order is fine) using plain text.

Use the algorithm described above and not anything else.

8.2 The 8-Tile Puzzle

The Chinese 8-Tile Puzzle is a 3×3 board, with 8 numbered tiles in it, and a hole into which

neighbouring tiles can move:

After the next move the board could look like: or The problem generally involves the board starting in a random state, and the user returning the board to the 'ordered' "12345678" state.

In this problem, a solution could be found in many different ways; the solution could be recursive, or you could implement a queue to perform a breadth-first seach, or something more complex allowing a depth-first search to measure 'how close' (in some sense) it is to the correct solution.

> **Exercise 8.2.1** Read in a board using `argv[1]`, e.g.:
>
> ```
> $ 8tile "513276 48"
> ```
>
> To do this you will use a list of boards. The initial board is put into this list. Each board in the list is, in turn, read from the list and all possible moves from that board added into the list. The next board is taken, and all its resulting boards are added, and so on. This is, essentially, a queue.
>
> However, one problem with is that repeated boards may be put into the queue and 'cycles' occur. This soon creates an explosively large number of boards (several million). You can solve this by only adding a board into the queue if an identical one does not already exisit in the queue. A linear search is acceptable for this task of identifying duplicates. Each structure in the queue will contain (amongst other things) a board and a record of its parent board, i.e. the board that it was created from.
>
> Be advised that a solution requiring as 'few' as 20 moves may take 10's of minutes to compute. If the search is successful, display the solution to the screen using plain-text.
>
> Use the method described above and only this one. Use a static data structure to acheive this (arrays) and **not** a dynamic method such as linked-lists; a (large) $1D$ array of structures is acceptable. Because this array needs to be so large, it's best to declare it in `main()` using something like:
>
> ```
> static boards[NUMBOARDS];
> ```
>
> ∎

> **Exercise 8.2.2** Repeat Exercise 8.2.1, but use SDL for the ouput rather than plain-text. ∎

> **Exercise 8.2.3** Repeat Exercise 8.2.1, but using a dynamic (linked-list), so that you never have to make any assumptions about the maximum numbers of boards stored. ∎

> **Exercise 8.2.4** Repeat Exercise 8.2.3, but using a 5×5 board instead. ∎

8.3 Happy Bookcases

In a quiet part of our building, there are some rather strange bookcases. They are (like most bookcases) generally happy, but they become unhappy when their books are not arranged

correctly (which, even in a Computer Science Department, is somewhat unusual). After years of dedicated research, a team of scientists led by Simon Lock and Sion Hannuna came to understand the trick to making the bookcases happy again. It turned out that a bookcase is only happy if :

- Each shelf only has books of one colour (or is empty).
- All books of the same colour are on the same shelf.
- The only books that exists are black(K), red(R), green(G), yellow(Y), blue(B), magenta(M), cyan(C) or white(W).

However, to make things worse, there are some complex rules about how books may be re-arranged :

1. You can only move one book at a time.
2. The only book that can move is the rightmost one from each shelf.
3. The book must move to become the rightmost book on its new shelf.
4. You can't put more books on a shelf than its maximum size.

So, for instance, the bookcase below has three shelves, each of which can fit three books; the first contains only one book, the second has three books, and the third shelf has two books on it.

By following the rules, highly-trained librarians can rearrange the books to make the bookcase happy again. One such way of re-arranging the books correctly is shown below :

Here's another example of an unhappy bookcase, and how to reaarange the books to make it happy :

> **Exercise 8.3.1** Write a program that reads in a bookcase definition file (specified on the command line), and shows the 'moves' to make the bookcase happy. Such a file looks something like :
>
> ```
> 4 3 7
> RG.
> GR.
> CY.
> YC.
> ```
>
> The first line has two or three numbers on it; the height of the bookcase (number of shelves), the width (maximum books per shelf) and an **optional** hint as to the minimum number of bookcases involved when 'solving' this bookcase. (This number is meaningless for a bookcase that cannot be made happy.) The number includes the original bookcase, and the final 'happy' one in the count. For the bookcase shown in this file, one possible solution is :
>
>
>
> In the file, an empty space is defined by a full-stop character. You may assume that the

maximum height and width of a bookcase is 9.

The brute-force algorithm for searching over all moves to make the bookcase happy goes like this :

1. You will use a list of bookcases (here list could either be an array, or a linked list).
2. The initial bookcase is put into the front of this list.
3. Take a bookcase from the **front** of the list.
4. For this (parent) bookcase, find the resulting (child) bookcases which can be created from all the valid possible single book moves. Put each of these bookcases into the **end** of the list. There may be as many as $height \times (height - 1)$ of these. If you have found a happy bookcase, stop. Else, go to 3.

To help with printing out the correct moves when a solution has been found, each structure in the list will need to contain (amongst other things) a bookcase and a record of its parent bookcase, i.e. the bookcase that it was created from. For an array, this could simply be which element of the array was the parent, or for a linked list, this will be a pointer.

The program reads the name of the bookcase definition file from `argv[1]`. If it finds a successful way to make the bookcase happy, it prints out the number of bookcases that would be printed in the solution and **nothing else**, or else exactly the phrase 'No Solution?" if none can be found :

```
$ ./bookcase rrggccyy-437.bc
7
$ ./bookcase rrrr-22.bc
No Solution?
$ ./bookcase ccbb-23.bc
1
```

If the 'verbose' flag is used (argv[2]), your program will additionally print out the solution (reverse order is fine) :

```
$ ./bookcase ccbb-23.bc verbose
1

CC.
BB.

$ ./bookcase rgbrmrykwrrr-3521.bc verbose
No Solution?

$ ./bookcase yby-222.bc verbose
2

Y.
BY

YY
B.
```

Your program :

- **Must** use the algorithm detailed above (which is similar to a queue and therefore a breadth-first search). Other search algorithms are possible (e.g. best-first, guided, recursive etc.) but the quality of coding is being assessed, not the quality of the algorithm used!

> • **Should** check for invalid bookcase definition files, and report in a graceful way if there is a problem, aborting with `exit(EXIT_FAILURE)` if so.
> • **May** display the bookcases in colour if you wish - if so use `neillsimplescreen` to do so.
> • **Should not** print anything else out to screen after successfully completing the search, except that which is shown above. Automated checking may be used, and therefore the ouput must be precise.
> • **Should** call the function `test()` to perform any assertion testing etc.
>
> ### Extension
> Basic assignment $= 90\%$. Extension $= 10\%$.
> If you'd like to try an extension, make sure to submit *extension.c* and a brief description in a *extension.txt* file. This could involve a faster search technique, better graphical display, user input or something else of your choosing. The extension will be marked in the same way as the main assignment.

8.4   Roller-Board

The puzzle *Roller-Board* consists of a 2*D* rectangualr grid of cells, each of which is labelled either '0' or '1':

```
0 0 0 0 0
0 1 0 1 0
0 0 1 0 0
0 1 0 1 0
0 0 0 0 0
```

The challenge is to roll one row or column at a time, so that the board is returned to its 'correct' state:

```
1 1 1 1 1
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

having all 1*s* on the top row, and every other cell being a 0. Each 'move' can be either:
• Roll a column one place up - i.e. the cells in this column all move up one, and the cell at the top 'rolls around' and reappears at the bottom of this column.
• Roll a column one place down - i.e. the cells in this column all move down one, and the cell at the bottom 'rolls around' and reappears at the top of this column.
• Roll a row one place left - i.e. the cells in this row all move left one, and the cell on the left 'rolls around' and reappears on the right of this row.
• Roll a row one place right - i.e. the cells in this row all right one, and the cell on the right 'rolls around' and reappears on the left of this row.

To solve a 4×4 board:

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

the best solution might be:

Roll column 1 up :
```
1 1 0 0
0 0 0 0
0 0 1 0
0 0 0 1
```

Roll column 2 down :
```
1 1 0 0
0 0 0 0
0 0 0 0
0 0 1 1
```

Roll column 2 down :
```
1 1 1 0
0 0 0 0
0 0 0 0
0 0 0 1
```

Roll column 3 down :
```
1 1 1 1
0 0 0 0
0 0 0 0
0 0 0 0
```

Here's another example of a board:
```
0 0 0 0
1 1 0 0
1 0 0 0
0 1 0 0
0 0 0 0
```

and how to solve it:
```
0 0 0 0   0 0 0 0   1 0 0 0   1 0 0 0   1 0 0 0   1 1 0 0   1 1 1 0   1 1 1 1
1 1 0 0   1 0 0 1   1 0 0 1   0 0 1 1   0 0 1 1   0 0 1 1   0 0 0 1   0 0 0 0
1 0 0 0   1 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
0 1 0 0   0 1 0 0   0 1 0 0   0 1 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 1 0 0   0 0 0 0   0 0 0 0   0 0 0 0
```

Exercise 8.4.1 Write a program that reads in a roller-board file (specified on the command line), and shows the 'moves' to solve it. Such a file looks something like :

```
5 4
0000
1100
1000
0100
0000
```

The first line has two numbers; the height of the board (number of rows) and then the width (number of columns).

In the remainder of the file, the number of 1s must be equal to the width of the board, and only the characters '0' and '1' are valid.You may assume that the maximum height and width of a board is 6.

The brute-force algorithm for searching over all moves for a solution goes like this :
1. You will use an `alloc()`'d array (list) of boards.
2. Put the initial board into the front of this list, `f=0`.
3. Consider the board at the **front** of the list (index `f`).
4. For this (parent) board, find the resulting (child) boards which can be created from all the possible $(rows + columns) \times 2$ rolls. For each of these child boards:
    - If this board is unique (i.e. it has not been seen before in the list), add it to the end of the list.

- If it has been seen before (a duplicate) ignore it.
- If it is the 'final' board, stop and print the solution.

5. Add one of f. If there are more boards in the list, go to step 3.

To help with printing out the correct moves, when a solution has been found, each structure in the list will need to contain (amongst other things) a board and a record of its parent board, i.e. the board that it was created from. Since you're using an array, this could simply be which element of the array was the parent.

The program reads the name of the board definition file from the command line. If it finds a successful solution, it prints out the number of boards that would be printed in the solution and **nothing else**, or else exactly the phrase 'No Solution?" if none can be found (as might be the case if you simply run out of memory) :

```
$ ./rollerboard 4x4diag.rbd
4 moves
$ ./rollerboard 5x5lhs.rbd
8 moves
```

If the 'verbose' flag is used, your program will print out the solution in the correct order :

```
$ ./rollerboard -v 4x4lr.rbd
0:
0000
1001
1001
0000

1:
1000
1001
0001
0000

2:
0100
1001
0001
0000

3:
1100
0001
0001
0000

4:
1101
0001
0000
0000

5:
1110
0001
0000
```

```
   0000

6:
1111
0000
0000
0000

$ ./rollerboard -v 3x3crn.rbd
0:
000
001
011

1:
000
100
011

2:
100
000
011

3:
110
000
001

4:
111
000
000
```

Your program :
- **Must** use the algorithm detailed above (which is similar to a queue and therefore a breadth-first search). Do not use the other algorithms possible (e.g. best-first, guided, recursive etc.); the quality of your coding is being assessed against others taking the same approach.
- **Must not** use dynamic arrays or linked lists. Since boards cannot be any larger than 6×6, you can create boards of this size, and only use a sub-part of them if the board required is smaller. The list of boards can be a fixed (large) size.
- **Should** check for invalid board definition files, and report in a graceful way if there is a problem, aborting with exit(EXIT_FAILURE) if so.
- **Should not** print anything else out to screen after successfully completing the search, except that which is shown above. Automated checking will be used during marking, and therefore the ouput must be precise.
- **Should** call the function test() to perform any assertion testing etc.
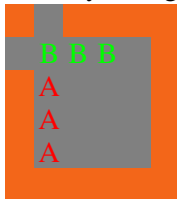
Extension

Basic assignment $= 90\%$. Extension $= 10\%$.

If you'd like to try an extension, make sure to submit *extension.c* and a brief description in a

8.5 Car Park

In a 2-dimesional car park as seen from above, vehicles are allowed to park either vertically or horizontally on a grid :

Here there are two vehicles, both of length 3. One is vertical (aligned north/south) coloured red, and one is horizontal (aligned west/east) coloured green. Empty cells of the gridded car park are shown in gray, and immovable boundaries (bollards) are shown in orange. Vehicles can only move forwards or backwards one square at a time (vertical vehicles move up and down, horizontal vehicles move left and right). These moves cannot be into, or across, another vehicle, nor into a bollard.

Our aim is to get all the vehicles safely out of the car park. During a turn, one vehicle is allowed to move fowards or backwards one square. Once the front of a vehicle touches an empty square on the edge of the car park, it is deemed to have exited the car park, and removed.

To 'solve' the car park above, we would move vehicle B one move left (so that it exits) :

and then vehicle A up one :

and then up one more move so that it too exits :

All vehicles have now exited the car park taking a total of three turns.

Exercise 8.5.1 Write a program that reads in a car park file (specified on the command line), and shows the 'turns' to solve it. The file for the car park above looks like :

```
6x6
#.####
```

```
.BBB.#
#A...#
#A...#
#A...#
#####
```

The first line has two numbers; the height of the car park (number of rows) and then the width (number of columns).

In the remainder of the file, vehicles are shown as a capital letter, gaps as a full-stop and bollards as a hash symbol. Each cars may only lie in the grid vertically or horizontally, and must be of at least length 2. Each vehicle must have a unique uppercase letter, the first of which must be an 'A', the next one be 'B' and so on.

We wil use a brute-force algorithm for searching over all moves for a solution :

1. You will use an array (list) of structures, each one containg the data for one car park. Note that you may choose to store the state of each car park, not as a 2D array, but as something that is easier to manipulate, e.g. an array of vehicles, including their position, orientation and whether they've exited or not. Each approach has pros and cons.
2. Put the initial car park into the front of this list, f=0.
3. Consider the car park at the **front** of the list (index f).
4. For this (parent) car park, find the resulting (child) car parks which can be created from all the possible vehicle moves. For each of these child car parks:
    - If this car park is unique (i.e. it has not been seen before in the list), add it to the end of the list.
    - If it has been seen before (a duplicate) ignore it.
    - If it is the 'final' car park, stop and print the solution.
5. Add one to f. If there are more car parks in the list, go to step 3.

To help with printing out the correct moves, when a solution has been found, each structure in the list will need to contain (amongst other things) a car park and a record of its parent car park, i.e. the car park that it was created from. Since you're using an array, this could simply be which index of the array was the parent.

The program reads the name of the car park definition file from the command line. If it finds a successful solution, it prints out the number of car parks that would be printed in the solution and **nothing else**, or else exactly the phrase 'No Solution?" if none can be found (as might be the case if it is impossible, or you simply run out of memory) :

```
$ ./carpark ../Git/Data/CarPark/6x6_2c_3t.prk
3 moves
$ .car/park ../Git/Data/CarPark/11x9_10c_26t.prk
26 moves
```

If the 'show' flag is used, your program will print out the solution in the correct order :

```
$ ./carpark -show ../Git/Data/CarPark/6x6_2c_3t.prk
#.####
.BBB.#
#A...#
#A...#
#A...#
#####
```

```
#.####
.....#
#A...#
#A...#
#A...#
#####

#.####
.A...#
#A...#
#A...#
#....#
#####

#.####
.....#
#....#
#....#
#....#
#####

3 moves
```

Your program :
- **Must** use the algorithm detailed above (which is similar to a queue and therefore a breadth-first search). Do not use the other algorithms possible (e.g. best-first, guided, recursive etc.); the quality of your coding is being assessed against others taking the same approach.
- **Must not** use dynamic arrays or linked lists. Since car parks cannot be any larger than 20×20, you can create car parks of this size, and only use a sub-part of them if the car park required is smaller. The list of car parks can be a fixed (large) size.
- **Should** check for invalid car park definition files, and report in a graceful way if there is a problem, aborting with exit(EXIT_FAILURE) if so.
- **Should not** print anything else out to screen after successfully completing the search, except that which is shown above. Automated checking will be used during marking, and therefore the ouput must be precise.
- **Should** call the function test() to perform any assertion testing etc.

Extension

Basic assignment = 90%. Extension = 10%.

If you'd like to try an extension, make sure to submit *extension.c* and a brief description in a *extension.txt* file, and an *extension.mak* Makefile, allowing me to build your code using make extension. The extension could involve a faster search technique, better graphical display, user input or something else of your choosing. ∎

8.6 The N-Queens Puzzle

In the game chess, the piece known as the Queen can take other pieces horizontally, vertically or diagonally :
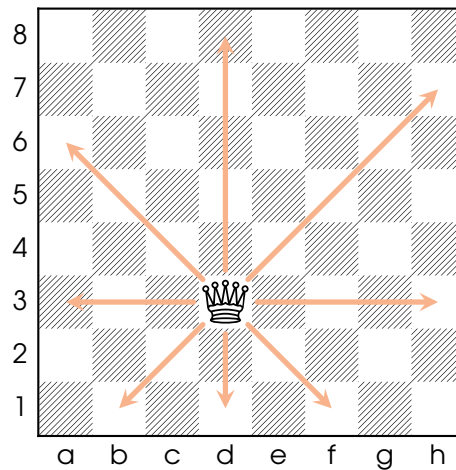
The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal.

www `https://en.wikipedia.org/wiki/Eight_queens_puzzle`

One possible solution for an 8×8 board is :
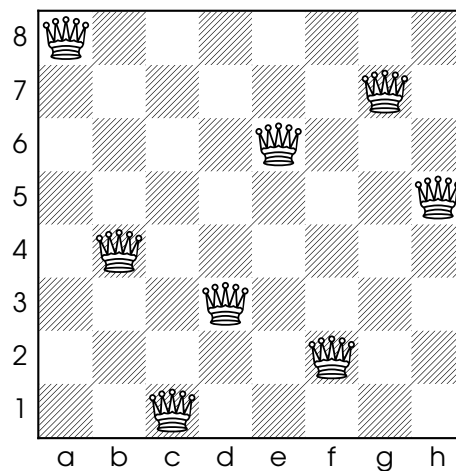
Since there must be one, and one only, queen in each file (column), the above board can be summarized using the queen's position on each rank (height) - the board above would be numbered 84136275.

Since finding all solutions may take a long time, it is normal to allow for board sizes other than 8×8 (but always square) by specifying the size of the board, n.

Exercise 8.6.1 Write a program to find solutions to the n queens problem, using a brute-force algorithm to find all possible solutions. You will use an array (list) of structures, each one containing the data for one board :

1. Put the initial (empty) board into the front of this list, f=0.
2. Consider the board at the **front** of the list (index f).

3. For this (parent) board, find the resulting (child) boards which can be created by placing one queen into the board into a position that doesn't check any other queens. For each of these child boards :
    - If a child is unique (i.e. it has not been seen before in the list), add it to the end of the list.
    - If it has been seen before (a duplicate) ignore it.
    - If it is a solution board, keep a record of the number of them (and possibly) print out a summary of this board.
4. Add one to f. If there are more boards in the list, go to step 2.

We will allow for board sizes other than 8×8, but **never** greater than 10×10, as specified on the command line :

```
$ ./8q 6
4 solutions
```

If the 'verbose' flag is used, your program will also print out summaries of each solution :

```
$ ./8q -verbose 6
362514
246135
531642
415263
4 solutions
```

(Since 10×10 boards may be printed, we'll use the numbers $1 \ldots 9$ and also A).

Your program :

- **Must** use the algorithm detailed above (which is similar to a queue and therefore a breadth-first search). Do not use the other algorithms possible (e.g. best-first, permutations, etc.); the quality of your coding is being assessed against others taking the same approach.
- **Must not** use dynamic arrays or linked lists. Since boards cannot be any larger than 10×10, you can create boards of this size, and only use a sub-part of them if the board required is smaller. The list of boards can be a fixed (large) size.
- **Should** check for invalid board sizes specified, and report in a graceful way if there is a problem, aborting with `exit(EXIT_FAILURE)` if so.
- **Should not** print anything else out to screen after successfully completing the search, except that which is shown above. Automated checking will be used during marking, and therefore the output must be precise.
- **Should** call the function `test()` to perform any assertion testing etc.
- **Should** be submitted by uploading a single `8q.zip` file to Blackboard, which will include (at least) `8q.c` and `Makefile` which allows me to compile your code by typing `make 8q` which creates `8q`, the executable file and can be run using `make run`.

Extension

Basic assignment = 90%. Extension = 10%.

If you'd like to try an extension, make sure to put *extension.c* and an explanation of it in *extension.txt* as part of your submission. Update your `Makefile` so that we can build your code using `make extension` and run it with `make extrun`. The extension could involve a faster search technique, better graphical display, user input or something else of your choosing.

9. ADTs & Data Structures I

9.1 Indexed Arrays

In the usual places are the files `arr.h`, `testarr.c` and a Makefile. My files `general.c` and `general.h` (if you want to use them) are in the ADT section of the github site. You will write `Realloc/realloc.c` and `Realloc/specific.h`. These files will allow you to build, and test, the ADT for a 1D Indexed Array. This simple replacement for C arrays is 'safe' in the sense that if you write the array out-of-bounds, it will be automatically resized correctly (using `realloc()`). The interface to this ADT is in `arr.h` and its implementation will be in `Realloc/realloc.c` Typing `make`, compiles these files together with the test file `testarr.c`. Executing `./testarr` should result in all tests passing correctly.

```
% make run
Basic Array Tests ... Start
Basic Array Tests ... Stop
```

Exercise 9.1.1 Write `Realloc/realloc.c` nd `specific.h` so that the functionality in the driver code `testarr.c` is implemented. ∎

9.2 Packed Boolean Arrays

In C99, every individual variable must have a different address. For this reason, a `bool` must take up a least one byte since this is the smallest individual element uniquely addressable by most hardware.

In some cases it would be good to have a datatype that truly uses one bit of storage for each Boolean. One such example is known as the *packed* Boolean array, where an array of 64 elements takes 8 bytes.

Here we create an ADT for packed Boolean arrays, allowing logical operations to be performed on an entire array, and individual bits to be set and unset.

This will involve manipulation of the bits of the data using the C bitwise logical operators 'xor' (^), 'or' (|) and 'and' (&).

www `https://en.wikipedia.org/wiki/Bitwise_operation`

> **Exercise 9.2.1** Given the files `testboolarr.c` and `boolarr.h`, complete the Boolean Array ADT using a reallocating array of unsigned chars. You will need to write both `realloc.c`, which will define the functions in boolarr.h, and `specific.h`, which will contain your header information. Ignoring the overhead of the main structure, which might store the capacity of the underlying array, and the number of valid bits in use, the array should be around `nbits/8` bytes in length. ∎

9.3 Sets

Sets are an important concept in Computer Science. They enable the storage of elements (members), guaranteeing that no element appears more than once. Operations on sets include initializing them, copying them, inserting an element, returning their size (cardinality), finding if they contain a particular element, removing an element if it exists, and removing one element from a random position (since sets have no particular ordering, this could be the first element). Other set operations include union (combining two sets to include all elements), and intersection (the set containing elements common to both sets).

www   `https://www.mathsisfun.com/sets/sets-introduction.html`

www   `https://en.wikipedia.org/wiki/Set_(mathematics)`

The definition of a Set ADT is given in `set.h`, and a file to test it is given in `testset.c`.

> **Exercise 9.3.1** Write an implementation of the Set ADT which builds on top of the Indexed Array ADT introduced in Exercise 9.1.1 and which compiles againset `testset.c`. ∎

9.4 Towards Polymorphism

Polymorphism is the concept of writing functions (or ADTs), without needing to specify which particular type is being used/stored. To understand the quicksort algorithm, for instance, doesn't really require you to know whether you're using integers, doubles or some other type. C is not very good at dealing with polymorphism - you'd need something like Python, Java or C++ for that. However, it does allow the use of `void*` pointers for us to approximate it.

> **Exercise 9.4.1** Extend the array ADT discussed in Exercise 9.1.1, so that any type can be used - files `varr.h` and `testvarr.c` are available in the usual place - use the Makefile used there, simply swapping `arr` for `varr` at the top. ∎

9.5 Self-Organising Linked Lists

A self-organising linked list (SOLL) improves search efficiency (over an unsorted list) by rearranging the elements in the list each time they are accessed:

www   `https://en.wikipedia.org/wiki/Self-organizing_list`

If no self-organisation is done, a SOLL behaves similarly to a collection. New elements are inserted at the end of the list, and searching is done from the start to the end using pointer-chasing. For efficiencies sake, we keep pointers to both the start and the end of the list. Insertion, therefore, has a $O(1)$ cost.

However, if self-organisation is done via the *move-to-front* (MFT) policy, each time an element is accessed, it is moved to the start of the list. This means that the next time we search for the same element, it will be found more quickly, since it is near the start of the list. Once the element is found, this re-ordering also has a complextity of $O(1)$.

The MFT policy can sometimes be too aggresive; uncommon elements that are searched for will move to the front, potentially displacing elements that are being searched for more frequently. An alternative is the *transpose* policy. When an element is accessed, it is moved one place in the list closer to the start. This once again can be done in constant time, providing we keep a copy of a pointer to the previous element during the search, or alternatively, use a doubly-linked list e.g. each element has both a next and a previous pointer.

> **Exercise 9.5.1** Implement a SOLL using linked lists to implement these different policies and store strings.
> - 55% Write the files `Linked/specific.h` and `Linked/linked.c`. Using the `soll.mak` makefile, you can compile this against the `soll.h` file and one of the test/driver files `testsoll.c`, `build.c` or `uniq.c`.
>   The ADT should implement the standard functions: `soll_init()`, `soll_add()`, `soll_remove()`, `soll_isin()`, `soll_tostring()`, `soll_size()` and `soll_free()`. In addition, write the function `soll_freq()` which reports the frequency of access for a particular element. Therefore, each element (structure) of the list has the overhead of keeping track of the number of times it has been accessed.
> - 30% Show a testing strategy on the above by submitting `testing.txt` where you give details of unit testing, white/black-box testing done on your code. Describe any test-harnesses used. Convince me that every line of your C code has been tested.
> - 15% Show an extension to the project in a direction of your choice via `extension.txt`. It should demonstrate your **understanding** of some aspect of programming or S/W engineering. Make sure it's clear what has been done, why, and how to compile it. ■

9.6 Sudoku

This concerns the automated solving of one of the world's most famous puzzles, Sudoku:

www `https://en.wikipedia.org/wiki/Sudoku`

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

Exercise 9.6.1 Using the simple ADT framework provided and the constraint propagation approach described in:

www

```
https://norvig.com/sudoku.html
```

write a program that can solve "easy" puzzles. The constraint propagation approach cannot completely solve "hard" puzzles. A basic ADT is provided for you in `sudoku.h`, along with a simple test file `testsudoku.c`. Write `Fixed/fixed.c` and `Fixed/specific.h`, so that:

```
% make run
Basic Sudoku Tests ... Start
Basic Sudoku Tests ... Stop
Basic Sudoku Tests ... Start
Basic Sudoku Tests ... Stop
```

works correctly.
The code will hard-wire the grid to be a fixed size $2D$ array, and be capable of reading in the different files present in `Data/Sudoku`. Submit a single `.zip` file which contains your entire submission, including `Fixed/fixed.c`, `Fixed/specific.h` and a `Makefile` (if different to the one provided). In addition, if you wish, you can provide extra material (extra testing etc.). If so provide a `basic.txt` file explaining what you've done. ∎

Exercise 9.6.2 Extend your program so that all grids are solvable, even the "hard" examples given. One typical way to do this is to make a guess for a square, and then backtrack to undo this if it leads to an invalid solution when using constraint propagation. Efficiency of your solution is not a concern here - focus on creating beautifully crafted (and tested) code.
Submit a single `.zip` file which contains your entire submission, including the updated versions of `Fixed/fixed.c` and `Fixed/specific.h` in addition to `solution.txt` which briefly describes the algorithm used. Include a `Makefile` so that `make hard` builds and runs the code.

∎

Exercise 9.6.3 Extend your program in a manner of your own choosing - this could be a faster approach to solving all grids (but maybe at the expense of code readability), exploring Sudokus larger than 9*x*9, or allowing better I/O or user-interaction.
Submit a single `.zip` file which contains your entire submission, including the updated versions of `Fixed/fixed.c` and `Fixed/specific.h` in addition to `extend.txt` which briefly describes what you have done. Include a `Makefile` so that `make extension` builds and runs the code.

∎

9.7 MultiValue Maps

Many data types concern a single value (e.g. a hash table), so that a string (say) acts as both the key (by which we search for the data) and also as the object we need to store (the value). An example of this a spelling checker, where one word is stored (and searched for) at a time. However, sometimesthere is a need to store a value based on a particular key - for instance an associative array in Python allows you to perform operations such as :

```
population["Bristol"] = 536000
```

where a value (the number 536000) is stored using the key (the string "Bristol"). One decision you need to make when designing such a data type is whether multiple values are allowed for the same key; in the above example this would make no sense - Bristol can only have one population size. But if you wanted to store people as the key, with their salary as the value, you might need to use a MultiValue Map (MVM) since people can have more than one job.

Here we write the abstract type for a MultiValueMap that stores key-value pairs, where both the key and the value are strings.

Exercise 9.7.1 The definition of an MVM ADT is given in `mvm.h`, and a file to test it is given in `testmvm.c`. Write `mvm.c`, so that:

```
% make -f mvm_adt.mk
./testmvm
Basic MVM Tests ... Start
Basic MVM Tests ... Stop
```

works correctly. Use a simple linked list for this, inserting new items at the head of the list. Make no changes to any of my files.   ∎

9.8 Rhymes

In the usual place is a dictionary which, for every word, lists the phonemes (a code for a distinct unit of sound) used to pronounce that word in American English. In this file the word itself, and its phonemes, are separated by a '#'. For instance:

```
BOY#B OY1
```

shows that the word `BOY` has two phonemes : `B` and `OY1`.

A simple attempt at finding rhymes for boy would match every word that has `OY1` as its final phoneme. This gives you:

```
POLLOI MCVOY LAFOY ALROY ILLINOIS CROIX DECOY REDEPLOY CLOY
LAVOY MOYE LOYE STOY PLOY KNOY EMPLOY ELROY JOY COY LACROIX
DEVROY ENJOY LOY COYE FOYE MOY DOI BROY TOY LABOY ROI HOY
ROYE NEU CROY SOY YOY MCCOY CHOY GOY ROY BOLSHOI MALLOY JOYE
DESTROY DELACROIX(1) DEBOY MCROY CHOI UNDEREMPLOY FLOY MCKOY
TOYE AHOY BOY OYE SGROI FOIE(1) TROY DEPLOY SAVOY UNEMPLOY
SCHEU WOY BOYE HOYE FOY OI HOI KROY EMPLOY(1) FLOURNOY OIE
MCCLOY ANNOY OY DEJOY
```

Using two phonemes to do the matching is too many, since the only matches are for words that have exactly the same pronunciation:

```
LABOY DEBOY BOY BOYE
```

which are not really rhymes, but homophones. Therefore, using the **correct** number of phonemes will be key to finding 'good' rhyming words.

Here we will use the MutliValue Map written in Exercise 9.7.1 to create two maps. An MVM `map1` stores the word (as the key) and its final n phonemes as a single string (the value). Now `map2` stores the word (value), keyed by its final n phonemes as a single string. Looking up the phonemes associated with a word can be done using the word as a key [1] (via `map1`), and looking up a word given its phonemes can be achieved using `map2`.

---

[1]Strictly speaking, we don't need the *map1* to be capable of storing multiple values, since every word in the dictionary is unique. We'll use it here though for simplicity.

Exercise 9.8.1 Read in the dictionary, and for each line in turn, store the data in the two maps. Now for a requested word to be 'rhymed', search for its phonemes using `map1` and then search for matching words using `map2`. The number of phonemes specified for this rhyming is given via the command line, as are the words to be rhymed:

```
$ ./homophones -n 3 RHYME
RHYME (R AY1 M): PRIME RHYME ANTICRIME(1) CRIME ANTICRIME
GRIME RIME
```

The `-n` flag specifies the number of phonemes to use (you may assume the number associated with it always is always separated from the flag by a space). If no `-n` flag is given then the value 3 is assumed. It only makes sense to use a value of $n \leq$ the number of phonemes in the two words being checked. If you use a value greater than this, the results are undefined.

The list of words to be matched may be found on the command line:

```
$ ./homophones -n 4 CHRISTMAS PROGRAM PASSING
CHRISTMAS (S M AHO S): ISTHMUS CHRISTMAS CHRISTMAS



```

Use the Makefile supplied for this task. Make the output as similar to that shown above as possible.

■

9.9 Advent of Code

Have a look through the archives of the rather wonderful 'Advent of Code' website, which is a good place to begin your lifelong love of recreational programming.

www   `https://adventofcode.com`

and find a simple puzzle to solve.

Exercise 9.9.1 Choose one of the simpler puzzles and solve it.   ■

10. Trees & Hashing

10.1 Depth

The following program builds a binary tree at random:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <time.h>

#define STRSIZE 5000

struct node{
        char c;
        struct node *left;
        struct node *right;
};
typedef struct node Node;

Node *MakeNode(char c);
void InsertRandom(Node *t, Node *n);
char *PrintTree(Node *t);

int main(void)
{

    char c;
    Node *head = MakeNode('A');
    Node *n;

    srand(time(NULL));
    for(c = 'B'; c < 'G'; c++){
        n = MakeNode(c);
        InsertRandom(head, n);
    }
    printf("%s\n", PrintTree(head));
    return 0;
```

```
}

Node *MakeNode(char c)
{

    Node *n = (Node *)calloc(1, sizeof(Node));
    assert(n != NULL);
    n->c = c;
    return n;

}

void InsertRandom(Node *t, Node *n)
{

    if((rand()%2) == 0){ /* Left */
        if(t->left == NULL){
            t->left = n;
        }
        else{
            InsertRandom(t->left, n);
        }
    }
    else{ /* Right */
        if(t->right == NULL){
            t->right = n;
        }
        else{
            InsertRandom(t->right, n);
        }
    }

}

char *PrintTree(Node *t)
{

    char *str;

    assert((str = calloc(STRSIZE, sizeof(char))) != NULL);
    if(t == NULL){
        strcpy(str, "*");
        return str;
    }
    sprintf(str, "%c (%s) (%s)", t->c, PrintTree(t->left), PrintTree(t->right));
    return str;

}
```

Each node of the tree contains one of the characters 'A' ... 'F'. At the end, the tree is printed out in the manner described in the course lectures.

Exercise 10.1.1 Adapt the code so that the maximum depth of the tree is computed using a recursive function. The maximum depth of the tree is the longest path from the root to a leaf. The depth of a tree containing one node is 1.

■

10.2 Two Trees

Adapt the code shown in Exercise 10.1, so that two random trees are generated.

> **Exercise 10.2.1** Write a Boolean function that checks whether the two trees are identical or not. ∎

10.3 Huffman Encoding

Huffman encoding is commonly used for data compression. Based on the frequency of occurence of characters, you build a tree where rare characters appear at the bottom of the tree, and commonly occuring characters are near the top of the tree.

For an example input text file, a Huffman tree might look something like:

```
010 :          00101 (   5 *  125)
' ' :            110 (   3 *  792)
'"' :      111001010 (   9 *   12)
''' :       00100000 (   8 *   15)
'(' :     01100000100 (  11 *    2)
')' :     01100001101 (  11 *    2)
',' :        1001001 (   7 *   39)
'-' :        0010010 (   7 *   31)
'.' :        1001100 (   7 *   40)
'/' :    00100110000 (  11 *    1)
'0' :    11100110010 (  11 *    3)
'1' :       00100010 (   8 *   15)
'3' :    01100000101 (  11 *    2)
'4' :    01100001001 (  11 *    2)
'5' :    11100110011 (  11 *    3)
'6' :    01100001000 (  11 *    2)
'7' :    01100001100 (  11 *    2)
'8' :      001001101 (   9 *    8)
'9' :       10010000 (   8 *   18)
':' :    01100001011 (  11 *    2)
'A' :       00100111 (   8 *   16)
'B' :      111001101 (   9 *   13)
'C' :       10011011 (   8 *   22)
'D' :      111001110 (   9 *   13)
'E' :       10011010 (   8 *   19)
'F' :      111001000 (   9 *   11)
'G' :     0110000000 (  10 *    4)
'H' :     1110011111 (  10 *    7)
'I' :     1110010011 (  10 *    6)
'J' :    11100111101 (  11 *    3)
'K' :     1110010111 (  10 *    6)
'L' :       00100011 (   8 *   15)
'M' :    11100111100 (  11 *    3)
'N' :    01100001010 (  11 *    2)
'O' :    01100000111 (  11 *    2)
'P' :     1110011000 (  10 *    6)
'R' :     0110000111 (  10 *    5)
'S' :       10010001 (   8 *   19)
'T' :     0010011001 (  10 *    4)
'U' :     1110010010 (  10 *    5)
'W' :     0110000001 (  10 *    4)
'a' :           1010 (   4 *  339)
```

```
'b' :         1111110 (    7 *    60)
'c' :          100101 (    6 *    77)
'd' :           01101 (    5 *   143)
'e' :             000 (    3 *   473)
'f' :          100111 (    6 *    84)
'g' :          111000 (    6 *    94)
'h' :           11110 (    5 *   223)
'i' :            0100 (    4 *   266)
'j' :     01100000110 (   11 *     2)
'k' :        00100001 (    8 *    15)
'l' :           10110 (    5 *   176)
'm' :          101111 (    6 *    92)
'n' :            0111 (    4 *   288)
'o' :            0101 (    4 *   269)
'p' :          101110 (    6 *    89)
'q' :     00100110001 (   11 *     2)
'r' :           11101 (    5 *   214)
's' :            0011 (    4 *   260)
't' :            1000 (    4 *   305)
'u' :          111110 (    6 *   108)
'v' :         0110001 (    7 *    37)
'w' :         1111111 (    7 *    60)
'x' :      1110010110 (   10 *     6)
'y' :          011001 (    6 *    72)
2916 bytes
```

Each character is shown, along with its Huffman bit-pattern, the length of the bit-pattern and the frequency of occurrence. At the bottom, the total number of bytes required to compress the file is displayed.

Exercise 10.3.1 Write a program that reads in a file (argv[1]) and, based on the characters it contains, computes the Huffman tree, displaying it as above.   ∎

10.4 Binary Tree Visualisation

The course notes showed a simple way to print out integer binary trees in this form :

```
20(10(5(*)(*))(17(*)(*)))(30(21(*)(*))(*))
```

You could also imagine doing the reverse operation, that is reading in a tree in the form above and displaying it in a 'friendlier' style :

```
20----30
|       |
10-17  21
|
5
```

The tree has left branches vertically down the page and right branches horizontally right. Another example is :

```
17(2(*)(3(*)(4(*)(*))))(6(8(*)(*))(*))
```

which is displayed as:

```
17----6
|     |
2-3-4 8
```

The above examples show the most 'compact' form of displaying the trees, but you can use simplifying assumptions if you wish:

- The integers stored in the tree are always ≥ 0.
- The integers stored in the tree are 5 characters (or less) in length.
- It is just as valid to print the tree in either of these ways :

```
1-6         00001-00006        00001------------00006
| |         |       |          |                    |
2 7         00002  00008        00002                00007
|           |                   |
3-4-5       00003-00004-00005   00003-00004-00005
```

> **Exercise 10.4.1** Write a program that reads in a tree using `argv[1]` and the tree displayed to `stdout` with no other printing if no error has occurred. ∎

> **Exercise 10.4.2** Write a program that reads in a tree using `argv[1]` and displays the tree using SDL. ∎

10.5 Lowest Common Ancestor

> UNDER CONSTRUCTION - Please don't attempt this yet. We'll have this new exciting exercise ready for you soon!

Which node in a tree is the lowest common ancestor of two others ?

10.6 Faster MVMs

The ADT for MVMs used in Exercise 9.7.1 is a simple linked list - insertion is fast, but searching is slow.

> **Exercise 10.6.1** Write a new version of this MVM ADT called `fmvm.c` that implements exactly the same functionality but has a faster search time. The file `fmvm.h` will change very little from `mvm.h`, with maybe only the structures changing, but not the function prototypes. A similar testing file to that used previously, now called `testfmvm.c` should also be written. Note that any ordering of data when using the `mvm_print` and `mvm_multisearch` functions is acceptable, so these can't be tested in exactly the same manner.
>
> By simply changing the #include from <mvm.h> to <fmvm.h> in your `homephones.c` file from Exercise 9.8.1, and compiling it against `fmvm.c`, I can test that program works identically.
>
> Make it clear what you have done to speed up your searching (and how) using comments at the top of `fmvm.h`
>
> Submit : `fmvm.c`, `fmvm.h` and `testfmvm.c` ∎

10.7   Double Hashing

Here we use double hashing, a technique for resolving collisions in a hash table.

Exercise 10.7.1 Use double hashing to create a spelling checker, which reads in a dictionary file from `argv[1]`, and stores the words.
   Make sure the program:
- Use double hashing to achieve this.
- Makes no assumptions about the maximum size of the dictionary files. Choose an initial (prime) array size, created via malloc(). If this gets more than 60% full, creates a new array, roughly twice the size (but still prime). Rehash all the words into this new array from the old one. This may need to be done many times as more and more words are added.
- Uses a hash, and double hash, function of your choosing.
- Once the hash table is built, reads another list of words from `argv[2]` and reports on the *average* number of look-ups required. A *perfect* hash will require exactly 1.0 look-up. Assuming the program works correctly, this number is the only output required from the program.

■

10.8   Separate Chaining

Separate chaining deals with collisions by forming (hopefully small) linked lists out from a base array.

Exercise 10.8.1 Adapt Exercise 10.7.1 so that:
- A linked-list style approach is used.
- No assumptions about the maximum size of the dictionary file is made.
- The same hash function as before is used.
- Once the hash table is built, reads another list of words from `argv[2]` and reports on the *average* number of look-ups required. A *perfect* hash will require exactly 1.0 look-up, on average. Assuming the program works correctly, this number is the only output required from the program.

■

11. ADTs II & Data Structures

11.1 Polymorphic Hashing

Polymorphism is the concept of writing functions (or ADTs), without needing to specify which particular type is being used/stored. To understand the quicksort algorithm, for instance, doesn't really require you to know whether you're using integers, doubles or some other type. C is not very good at dealing with polymorphism - you'd need something like Java or C++ for that. However, it does allow the use of `void*` pointers for us to approximate it.

> **Exercise 11.1.1** Here we write an implementation of a polymorphic associative array using hashing, see `assoc.h`. Both the key & data to be used by the hash function are of unknown types, so we will simply store void pointers to both of these, and the user of the associative array (and **not** the ADT itself) will be responsible for creating and maintaining such memory, and also ensuring it doesn't change when in use by the associative array. In the `testassoc.c` file we show two uses for such a type : a simple string example (to find the longest word in English that is also a valid (but different) word when spelled backwards), and a simple integer example where we keep a record of how many unique random numbers in a range are chosen.
>
> Since we do not know **what** the type of the key is, we need to be careful when comparing or copying keys. Therefore, in the `assoc_init(int keysize)` function, the user has to pass the size of the key used (e.g. `sizeof(int)`) or in the case or strings, the special value 0. Now we can use `memcmp()` and `memcpy()`, or in the case of strings, `strcmp()` and `strcpy()` for dealing with the keys. In the case of data, the ADT only ever needs to return a pointer to the data (not process it) via `assoc_lookup()`, so its size is not important.
>
> Your hash table used to implement the ADT should be resizeable, and you may use open-addressing/double-hashing or separate chaining to deal with collisions. Make no assumptions about the maximum size of the array, and make the initial size of the array small e.g. 16 or 17 (a prime is useful if you're double hashing).. You can use any hash function you wish, but if it's off the internet (etc.) cite the source in a comment..
>
> Submit a single `assoc.zip` file containing your code. Your standard submission will contain the directory structure, including the two files:
> `Realloc/specific.h` and `Realloc/realloc.c`. I will use my `assoc.mk` Makefile to compile your code, so check that it works correctly. ∎

Hint : when you do a resize you cannot simply copy the old table, but must rehash your data,

one entry at a time, into the new table. This is because your hash function is based on the table size, and if the table size has changed you will be hashing keys into different locations.

11.2   Cuckoo Hashing

Exercise 11.2.1 Rewrite your associative array described in Exercise 11.1.1 using cuckoo hashing instead. This involves two tables, and a key is guaranteed to be found (if it has been inserted) in one of its two locations. Different hash functions must be used for each of the two tables. If you do the extension, add :

`Cuckoo/specific.h` and `Cuckoo/cuckoo.c` to your directory structure in `assoc.zip`.   ▪

11.3   Exact and Approximate (Bloom) Dictionaries

A dictionary ADT is given in `dict.h`. This allows words to be inserted, and also to check whether a word is already in the Dictionary or not. Each word in the Dictionary is unique (you don't store the same word multiple times).

Simple Spell Checkers can be made by feeding a 'white-list' of words to the Dictionary and then testing another list against these to find words that are misspelt.

Exercise 11.3.1 Implement the Dictionary using hashing and *Separate Chaining* for collision resolution. Write the files `Exact/specific.h` and `Exact/exact.c` to compile against my `dict.h` file and my two test/driver files `testdict.c` and `spelling.c`. There is no need to ever resize the hashtable, since you know its maximum size upon initialisation.

This creates an *exact* Dictionary since once you've added a word, you're guaranteed to be able to find it again.

▪

A Bloom Filter consists of an array of (*m*) Boolean flags.

www   `https://en.wikipedia.org/wiki/Bloom_filter`

Each word is not stored in the Dictionary, but instead *k* different hashes are computed from it, and each of these used to set a Boolean flag in an array. When searching, compute these hashes, and if **any** of the corresponding flags are zero, you know the word cannot be in the Dictionary. If all the corresponding flags are set, then the word is **probably** in the Dictionary. You can't be certain though, because two different strings might hash to the same hash values, and the table might become congested. Such approaches are therefore known as *approximate*.

Exercise 11.3.2 Use a Bloom Filter to implement the Dictionary. Write the source files `Approx/specific.h` and `Approx/approx.c` to compile against my `dict.h` file. For simplicity, use a traditional Boolean array for the Bloom Filter (and **not** a packed array such as seen in Exercise 9.2.1).

By choosing *m* to be twenty times the number of words, and *k* = 11, the chances of a false match are reduced to (nearly) zero, and the tests `testdict.c` and `spelling.c` should perform in the same way as their 'exact' counterparts.

Inventing *k* different hash functions would be complicated, so instead we create one master hash (using Bernstein for example), and then evolve this iteratively to create the others. You could do this using something like :

```
unsigned long* _hashes(const char* s)
{
```

```
                // You'll need to free this later
                unsigned long* hashes = ncalloc(KHASHES, sizeof(unsigned long));
                // Use Bernstein from Lecture Notes (or other suitable hash)
                unsigned long bh = _hash(s);
                int ln = strlen(s);
                /* If two different strings have the same bh, then
                     we need a separate way to distiguish them when using
                     bh to generate a sequence */
                srand(bh*(ln*s[0] + s[ln−1]));
                unsigned long h2 = bh;
                for (int i=0; i<KHASHES; i++) {
                    h2 = 33 * h2 ^ rand();
                    hashes[i] = h2;
                }
                // Still need to apply modulus to these to fit table size
                return hashes;
            }
```

11.4 Cons, Car and Cdr

When storing real-world data, lists containing other lists are ubiquitous e.g. : `(0 (1 2) 3 4 5)`. In fact many functional (or at least partly functional) languages are based around the idea that all data are, conceptually, nested lists. Examples of such languages include Haskell and, of interest here, Lisp :

www   `https://en.wikipedia.org/wiki/Lisp_(programming_language)`

One of the main operations of these languages is the ability to extract the front (head) or remainder (tail) of the list **very** efficiently. A traditional linked list is not good for this, since it would require work to untangle the head (it has a pointer to the rest of the list) and it's unclear how a list-within-a-list would be stored.

For Lisp, a better structure was developed, which is slightly more complex than a traditional linked list, but which makes the extraction of the head (and remainder) simple and fast. To encode the list `(1 2)`, which is a list containing two atoms, we use :

```
[*|*]--->[*|*]--->NIL
 |        |
 v        v
 1        2
```

Here the main struct (known as the *cons*) has two pointers. The ones going downwards known as the *car* pointers and the ones going horizontally the *cdr*[1] pointers. If we were to have a pointer *p* to the first *cons* of this list, the head of the list is pointed to by `p->car`, and the remainder of the list simply by `p->cdr`.

---

[1]From wiki: Two assembly language macros ... became the primitive operations for decomposing lists: *car* (Contents of the Address part of Register number) and *cdr* (Contents of the Decrement part of Register number).

A more complex list, (0 (1 2) 3 4 5) would be stored as :

```
[*|*]--->[*|*]------------------>[*|*]--->[*|*]--->[*|*]--->NIL
 |        |                       |        |        |
 v        v                       v        v        v
 0       [*|*]--->[*|*]--->NIL    3        4        5
         |        |
         v        v
         1        2
```
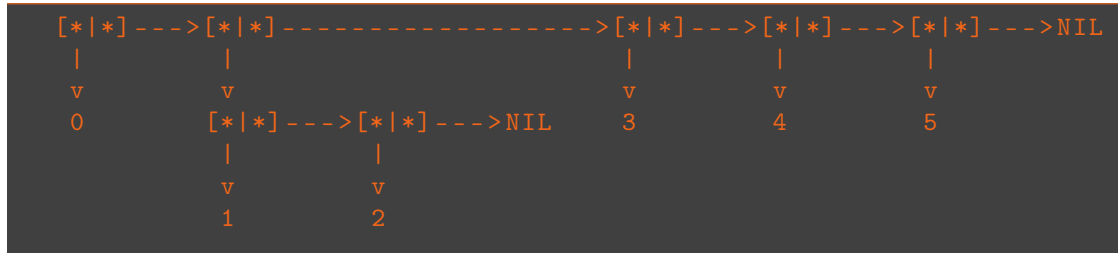
The atomic elements (the integers) are stored as 'leaf' nodes with both *car* and *cdr* pointers set to NULL. The data structure used to store the atom and also *car* and *cdr* pointers is known as a *cons* (short for constructor), due the to process by which we build lists.

So, here, we're interested in recreating this data structure, allowing the user to build lists (the *cons* operation), extract the head and remainder of lists (the *car* and *cdr* operations) and other associated functions such as copying a list or counting the number of elements in it.

> **Exercise 11.4.1** Use a dynamic/linkedlist style approach to implement a car/cdr ADT. Write the source files `Linked/specific.h` and `Linked/linked.c`, so that they compile against my `lisp.h` and `testlisp.c` files and run successfully using my `Makefile`. The basic operations are :
>
> ```
> lisp* lisp_atom(const atomtype a);
> lisp* lisp_cons(const lisp* l1,  const lisp* l2);
> lisp* lisp_car(const lisp* l);
> lisp* lisp_cdr(const lisp* l);
> atomtype lisp_getval(const lisp* l);
> bool lisp_isatomic(const lisp* l);
> lisp* lisp_copy(const lisp* l);
> int lisp_length(const lisp* l);
> void lisp_tostring(const lisp* l, char* str);
> void lisp_free(lisp** l);
> ```
>
> This is worth 90% of the marks. Additional functions you can implement, worth 10%, are :
>
> ```
> lisp* lisp_fromstring(const char* str);
> lisp* lisp_list(const int n, ...);
> void lisp_reduce(void (*func)(lisp* l, atomtype* n), lisp* l, atomtype* acc);
> ```
>
> and are **Exensions** worth 10% of the marks.
> Even if you don't get these extensions (or other functions) to work correctly, make sure the code still compiles by writing 'dummy' functions as placeholders, even if some of the assertions fail.   ∎

11.5 Binary Sparse Arrays

Using dynamic arrays is often a compromise between resizing the array too frequently (e.g. every time the memory required is fractionally too small) or else creating a great deal of 'spare' memory, just in case it is needed later. It is common for such dynamic arrays to double the available memory every time an index is accessed out-of-bounds.

Here we create the Binary Sparse Array (BSA) which aims to be memory efficient for small arrays, releasing new memory at an exponential rate as the array grows. Using a BSA ensures that no data is ever copied (e.g. via `remalloc`) nor too much unused memory created early. The

price we pay for this memory efficient is a slightly more complex calculation for the address of the index required.

Another data structure, the Hashed Array Tree (HAT) has similar goals, but is very different in practice and requires copying of data and resizing.

www `https://en.wikipedia.org/wiki/Hashed_array_tree`

Here, the BSA structure consists of a (fixed-size) row pointer table, each cell of which enables access to a 1D row of data. These rows are of a known size, each of which is increasingly large as we go down the table, as shown in Figure 11.1
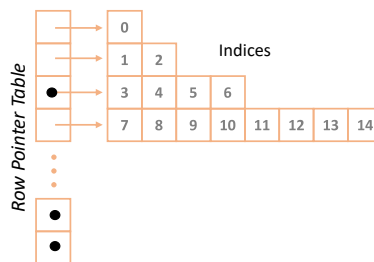
Figure 11.1: The Binary Sparse Array. A (vertical fixed-size) table allows access to each row. The rows are size 1 for the first, size 2 for the second, size 4 for third i.e. for row k it's size is 2^k.

However, the space allocation for each row is only done as-and-when required. After creating an empty BSA, and inserting values into index two and 12, then only rows one and three will have been allocated, as shown in Figure 11.2.

Figure 11.2: A Binary Sparse Array, showing values inserted into index 2 and 12 (shaded cells). Other rows are currently unallocated.

If the only element in a row is deleted, then the entire row is freed up, as shown in Figure 11.3 where the last element in row three is removed (index 12).

Figure 11.3: Deletion in a BSA can result in entire rows being freed up.

Exercise 11.5.1 Write an ADT to implement the `bsa.h` interface given. Write the source files `Alloc/specific.h` and `Alloc/alloc.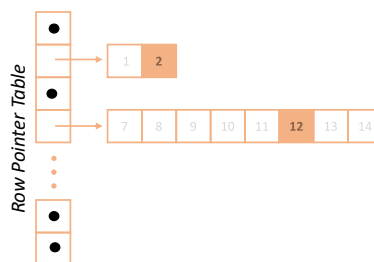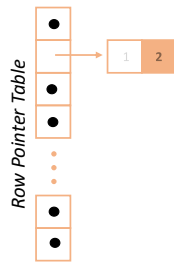c` such that the driver files `driver.c`,`isfactorial.c`, `sieve.c` and `fibmemo.c` work correctly. Ensure that this all works with my `Makefile` which **must be** used **unaltered**. The basic operations are :

```
bsa* bsa_init(void);
bool bsa_set(bsa* b, int indx, int d);
int* bsa_get(bsa* b, int indx);
bool bsa_delete(bsa* b, int indx);
int bsa_maxindex(bsa* b);
bool bsa_free(bsa* b);
```

The function `bsa_set()` allows an integer to be written to a particular index. If it is the first integer to be written into the row, then the space for this will need to be allocated.
`bsa_get()` returns a pointer to the integer stored in an index if it has been set, and `NULL` if it is a cell that is not in use, hasn't been written to, deleted, or past the end the maximum index used.
The functions `bsa_delete()` 'removes' an integer from an index (maybe setting a Boolean flag to show if it's in use or not), and if it was the only cell being used on that row, the entire row is freed.
Additional functionality includes :

```
bool bsa_tostring(bsa* b, char* str);
void bsa_foreach(void (*func)(int* p, int* n), bsa* b, int* acc);
```

`bsa_tostring()` allows a character based version of the BSA to be produced with each row inside {} brackets. The function `bsa_foreach` allows a user-defined function to be passed which is performed on each element in turn.
- The row pointer table is of a fixed size, and never gets deleted until the final call to `bsa_free()`.
- You will never use `realloc` for the BSA - rows are created as required using e.g. `calloc()` of a fixed size, and freed as required.
- Do not use the maths library in this assignment (my Makefile doesn't) - use bit manipulation if required rather than e.g. `log2()`.

- Even if you don't get all the functions to work correctly, make sure the code still compiles by writing 'dummy' functions as placeholders, even if some of the assertions fail.

This is worth 90% **of the marks.**

The manner in which you've been asked to implement the functionality above is **very** specific. However, exactly the same functionality (getting/setting etc.) could be implemented in very different ways (linked lists, 1D dynamic arrays, trees, hashing etc.). As an extension, write a 'rival' version to implement the core functionality of these functions without it necessarily being a BSA. This should still ensure that `fibmemo`, `sieve` and `isfactorial` can be compiled against it. This means you can ignore `bsa_tostring()` since this gives something very specific to the 'shape' of the BSA data structure. Again, use an unchanged version of my Makefile to achieve this. These files will be `Extension/specific.h` and `Extension/extension.c`.

If you do this, submit `extension.txt` which details what you have done, the motivation for it, and how well it works in practice (or not), and comparing it with the original BSA. This discussion (a few hundred words at most) is as (if not more) important than the code itself.

This worth 10% **of the marks.**

12. Parsing Data

12.1 Teletext

In the early 1970s, Phillips Labs began work on transmitting digital information across the television network. The aim was to provide up-to-date news and weather information via a television set. This system was trialled first by the BBC in a system that eventually became known as "Ceefax" and then on other independant British terrestrial stations as "Oracle". A very similar system was implemented on the BBC microcomputer, known as *Mode 7*. An example of

Figure 12.1: An example Ceefax page circa 1983. Taken from http://teletext.mb21.co.uk/gallery/ceefax/main1.shtml

		0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0		Unused/Reserved	Unused/Reserved		0	@	P	-	p
1		Red Alphanumeric	Red Graphics	!	1	A	Q	a	q
2		Green Alphanumeric	Green Graphics	"	2	B	R	b	r
3		Yellow Alphanumeric	Yellow Graphics	£	3	C	S	c	s
4		Blue Alphanumeric	Blue Graphics	$	4	D	T	d	t
5		Magenta Alphanumeric	Magenta Graphics	%	5	E	U	e	u
6		Cyan Alphanumeric	Cyan Graphics	&	6	F	V	f	v
7		White Alphanumeric	White Graphics	'	7	G	W	g	w
8		Unused/Reserved	Unused/Reserved	(	8	H	X	h	x
9		Unused/Reserved	Contiguous Graphics	)	9	I	Y	i	y
A		Unused/Reserved	Separated Graphics	*	:	J	Z	j	z
B		Unused/Reserved	Unused/Reserved	+	;	K	←	k	$1/4$
C		Single Height	Black Background	,	<	L	$1/2$	l	||
D		Double Height	New Background	-	=	M	→	m	$3/4$
E		Unused/Reserved	Hold Graphics	.	>	N	↑	n	÷
F		Unused/Reserved	Release Graphics	/	?	O	#	o	■

Table 12.1: The control codes and characters for alphanumeric mode. Note here (because we're using white paper) foreground is shown in black and background in white. On a teletext screen we use white on a black background.

such a Ceefax screen is shown in Figure 12.1.

This project, inspired by such teletext systems, will allow a 40×25 (1000) character file to be rendered to the screen, using similar control codes. However, some control codes are not implemented, including those to do with flashing or hidden text, and transparent backgrounds. In particular, our definition of the *double height* control code differs from that of the traditional one.

The Control Codes

This section is based to a large extent to Richard Russell's description of Mode 7 on the BBC Micro: `http://www.bbcbasic.co.uk/tccgen/manual/tcgen2.html`.

Coloured Text

By using the control codes $129 - 135$ ($0x81 - 0x87$ in hexadecimal) the rest of the line will have text in the selected foreground colour.

To change the background colour, you issue a foreground colour code first, and then the "New Background" character. All the following line will now have the appropriate background colour. You'll typically then need to choose a new foreground text colour.

Block Graphics

Teletext has a very limited ability to output low-resolution block graphics. These shapes take the place of other characters in the font and are enabled by issuing one of the coloured graphics codes e.g. *red graphics*. At this point the characters available for printing are as displayed in Table 12.2. These new graphics characters are made up of six smaller boxes, known as *sixels*. Each individual sixel has a code, either, $1, 2, 4, 8, 16$ or 64 as shown in Figure 12.2. By adding these values together we can define which of these sixels are 'lit' or not. If we wish the three left-hand ones to be lit we'd use the base code (160) plus $1, 4$ and $16 = 181$ ($0xB5$ in hexadecimal). Therefore issuing the coding *green graphics* and then code 181 puts a green vertical bar on the screen.

	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0	Unused/Reserved	Unused/Reserved	⊞	⊞	@	P	⊞	⊞
1	Red Alphanumeric	Red Graphics	⊞	⊞	A	Q	⊞	⊞
2	Green Alphanumeric	Green Graphics	⊞	⊞	B	R	⊞	⊞
3	Yellow Alphanumeric	Yellow Graphics	⊞	⊞	C	S	⊞	⊞
4	Blue Alphanumeric	Blue Graphics	⊞	⊞	D	T	⊞	⊞
5	Magenta Alphanumeric	Magenta Graphics	⊞	⊞	E	U	⊞	⊞
6	Cyan Alphanumeric	Cyan Graphics	⊞	⊞	F	V	⊞	⊞
7	White Alphanumeric	White Graphics	⊞	⊞	G	W	⊞	⊞
8	Unused/Reserved	Unused/Reserved	⊞	⊞	H	X	⊞	⊞
9	Unused/Reserved	Contiguous Graphics	⊞	⊞	I	Y	⊞	⊞
A	Unused/Reserved	Separated Graphics	⊞	⊞	J	Z	⊞	⊞
B	Unused/Reserved	Unused/Reserved	⊞	⊞	K	←	⊞	⊞
C	Single Height	Black Background	⊞	⊞	L	$^1/_2$	⊞	⊞
D	Double Height	New Background	⊞	⊞	M	→	⊞	⊞
E	Unused/Reserved	Hold Graphics	⊞	⊞	N	↑	⊞	⊞
F	Unused/Reserved	Release Graphics	⊞	⊞	O	#	⊞	⊞

Table 12.2: The control codes and characters for contiguous graphics mode.

	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0	Unused/Reserved	Unused/Reserved	▪	▪	@	P	▪	▪▪
1	Red Alphanumeric	Red Graphics	▪	▪	A	Q	▪	▪▪
2	Green Alphanumeric	Green Graphics	▪	▪	B	R	▪	▪▪
3	Yellow Alphanumeric	Yellow Graphics	▪▪	▪▪	C	S	▪▪	▪▪
4	Blue Alphanumeric	Blue Graphics	▪	▪	D	T	▪	▪▪
5	Magenta Alphanumeric	Magenta Graphics	▪	▪	E	U	▪	▪▪
6	Cyan Alphanumeric	Cyan Graphics	▪	▪	F	V	▪	▪▪
7	White Alphanumeric	White Graphics	▪▪	▪▪	G	W	▪▪	▪▪
8	Unused/Reserved	Unused/Reserved	▪	▪	H	X	▪	▪▪
9	Unused/Reserved	Contiguous Graphics	▪	▪	I	Y	▪	▪▪
A	Unused/Reserved	Separated Graphics	▪	▪	J	Z	▪	▪▪
B	Unused/Reserved	Unused/Reserved	▪▪	▪▪	K	←	▪▪	▪▪
C	Single Height	Black Background	▪▪	▪▪	L	½	▪▪	▪▪
D	Double Height	New Background	▪▪	▪▪	M	→	▪▪	▪▪
E	Unused/Reserved	Hold Graphics	▪▪	▪▪	N	↑	▪▪	▪▪
F	Unused/Reserved	Release Graphics	▪▪	▪▪	O	#	▪▪	▪▪

Table 12.3: The control codes and characters for separated graphics mode.

1	2
4	8
16	**64**

Figure 12.2: Values for computing graphics codes, as added to the base code 160 (0*xA*0 in hexadecimal).

Notice in Table 12.2 that some other characters are still available, particularly all capital letters. This allow simple printing of capitals, even when in graphics mode, and is know as *blast-through Text*.

There is another set of block graphics, as shown in Table 12.3. For these, each sixel is separated from others by thin vertical and horizontal lines. This mode is known as *separated graphics* mode.

Held Graphics

Generally all control codes are displayed as spaces, in the current background colour. In the held graphics mode, control code 158 (0*x9E* in hexadecimal), control codes are instead displayed as a copy of the most recently displayed graphics symbol. This permits a limited range of abrupt display colour changes. The held graphics character is displayed in the same contiguous/separated mode as when it was first displayed. If there has been a change in the text/graphics mode or the normal/double-height mode since the last graphics character was displayed, the held graphics character is cleared and control codes once again display as spaces.

To switch held graphics mode off, use the *release graphics* control code.

Double Height

By using the *double height* control code, characters are displayed as twice their normal size. Since they span two lines, the control codes and characters have to be repeated on the next line too, for them to be correctly displayed. The rule here, is that if a character is to be displayed as double height, the top half of the character is displayed on the first line, and the bottom half on the next line. The bottom half is only displayed as double height if the character vertically above it was also displayed in *double height* mode. The character in question need not be the same one.

Note: here we deviate from other definitions of this control code.

Some General Guidelines

- Characters are considered 7-bit (the 8th bit was typically used for parity checking over the noisy television signal). Therefore any character less than 128(0*x80*) should have 128 added to it. For, example if you read in character 32 (space), it should be 'converted' to character 160.
- Each newline on the Teletext page automatically begins with White text, single height, contiguous graphics, black background, release graphics.
- With the the exception of *hold graphics* (see above), control characters are generally rendered in the same manner as a space would be. If the background is currently red and text colour yellow, say, then the control code would show as an empty red background.

Exercise 12.1.1 Implement a teletext rendering system. The 1000 character input file should be read in using `argv[1]`.

There are many ambiguities as to how various sequences of codes should be rendered. To help with this, several example files have been posted on the unit web page. If there is still doubt, make a best-guess and state your assumptions in the code.

Submit the testing you have undertaken, including examples and a description of your strategies. This should convince us that you have tested every line of code, and that it works correctly. If there are still issues/bugs state them clearly. Also, point out any bugs that you have successfully found using these approaches. Submit a file named `testing.txt`, along with any other files you feel necessary in the appropriate directory.

No particular strategy is mandated. You may wish to explore a couple and briefly discuss strengths and weaknesses.

Undertake an extension of your choosing. Examples of these include:

- A system that allows you to quickly author teletext pages (perhaps using a recursive-descent parser?)
- Automatic image to teletext conversion.
- Automatic (simple) html to teletext conversion (and/or vice-versa).

Remember, that the assessment is based on the quality of your coding, so choose something to demonstrate an aspect of programming or software engineering that you haven't had a chance to use in the main assignment. Submit a file named `extension.txt` outlining, in brief, your contribution.

Hints

- Don't add graphics too early - the code is easier to test and debug with textual output to begin with.
- I advise you to use SDL for your graphics output. The library provided previously contained two functions to deal with printing characters : `Neill_SDL_ReadFont()` and `Neill_SDL_DrawString()`. The font file `m7fixed.fnt` provides the basic characters required here, but not the sixels. By understanding how the font data is rendered, the double height version of the characters should be relatively simple.
- Don't try to do all aspects of this at once - begin with coloured characters only. Add more advanced functionality later.
- Plan how you are going to store your data early in the design process. Does each character need its own data structure? Does each line? Can this be abstracted?

Please create a directory structure, so that I can easily find the different subsections. Your testing strategy will be explained in `testing.txt`, and your extension as `extension.txt`. For the source and extension sections, make sure there's a `Makefile`, so that I can easily build the code.

```
        ------Top Directory------
        |           |           |
        |           |           |
        |           |           |
        |           |           |
      source      testing    extension
        |           |           |
       ...         ...      extension.txt
       ...         ...          ...
     Makefile     Makefile      ...
                testing.txt
```

Bundle all of these up as one **single** `.zip` submission - not one for each subsection. ■

12.2 Guido van Robot

From

 `http://gvr.sourceforge.net/`

Guido van Robot can face in one of four directions, north, east, south, and west. He turns only 90 degrees at a time, so he can't face northeast, for instance. In Guido's world, streets run east-west, and are numbered starting at 1. There are no zero or negative street numbers. Avenues run north-south, and are also numbered starting at 1, with no zero or negative avenue numbers. At the intersection of a street and avenue is a corner. Guido moves from one corner to to the next in a single movement. Because he can only face in one of four directions, when he moves he changes his location by one avenue, or by one street, but not both!
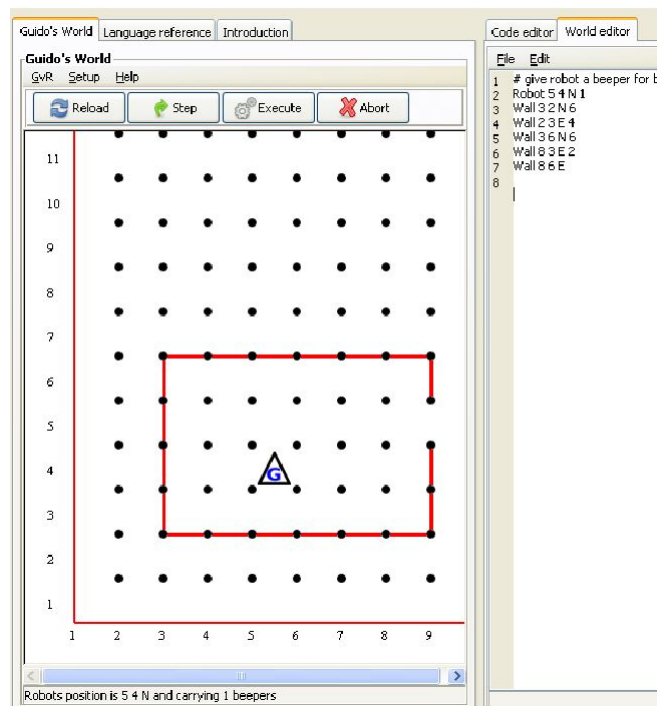
Simple .wld File
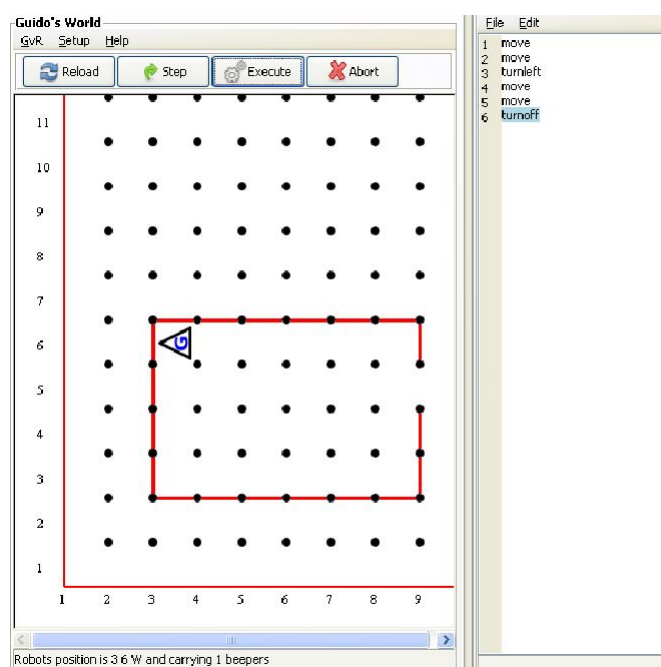
```
Robot 5 4 N 1
Wall 3 2 N 6
Wall 2 3 E 4
Wall 3 6 N 6
```

```
Wall 8 3 E 2
Wall 8 6 E
```


Simple .gvr File

```
move
move
move
move
turnoff
```

Do Loops

```
do 2 :
   putbeeper
   move
turnoff
```

Conditional Loop

```
while front_is_clear :
   putbeeper
   move
turnoff
```

Branching

```
do 13 :
   if front_is_clear :
      putbeeper
      move
   else :
      turnleft
turnoff
```

The Formal Grammar

```
<PROGRAM>    ::= <BLOCK>
<BLOCK>      ::= "turnoff" |
                 "move" <BLOCK> |
                 "turnleft" <BLOCK> |
                 "pickbeeper" <BLOCK> |
                 "putbeeper" <BLOCK> |
                 <DO> <BLOCK> |
                 <WHILE> <BLOCK> |
                 <IF> <BLOCK>
<DO>         ::= "do" <num> ":"
                    <BLOCK>
<WHILE>      ::= "while" <TEST> ":"
                    <BLOCK>
<IF>         ::= "if" <TEST> ":"
                    <BLOCK> |
                 "if" <TEST> ":"
                    <BLOCK>
                 "else" ":"
                    <BLOCK>
<TEST>       ::= <WALL> | <BEEP> | <COMPASS>

<WALL>       ::= "front_is_clear" |
                 "front_is_blocked" |
                 "left_is_clear" |
                 "left_is_blocked" |
                 "right_is_clear" |
                 "right_is_blocked"
<BEEP>       ::= "next_to_a_beeper" |
                 "not_next_to_a_beeper" |
                 "any_beepers_in_beeper_bag" |
```

```
                        "no_beepers_in_beeper_bag"
<COMPASS>    ::= "facing_north" |
                "not_facing_north" |
                "facing_south" |
                "not_facing_south" |
                "facing_east" |
                "not_facing_east" |
                "facing_west" |
                "not_facing_west"
```

This ignores some Guido instructions, e.g. `elseif` and `define`. It also doesn't well explain how to spot the end of a `DO` etc. which is marked by a reduction in indentation. The definition of `.wld` files is so simple a recursive descent parser (and hence grammar) is not required.

Exercise 12.2.1   • (25%) To implement a recursive descent parser - this says whether or not the given `.gvr` and `.wld` follow the formal grammar or not. The input files are specified via `argv[1]` (`.gvr`) and `argv[2]` (`.wld`).
  • (25%) To implement an interpreter, so that the instructions are executed. Printing the world and robot to screen using simple characters is fine, but many will wish to use SDL.
  • (25%) To show a testing strategy on the above - you should give details of white-box and black-box testing done on your code. Describe any test-harnesses used. Give examples of the output of many different programs. Convince me that every line of your C code has been tested.
  • (25%) To show an extension to the project in a direction of your choice. It should demonstrate your understanding of some aspect of programming or S/W engineering. If you extend the formal grammar make sure that you show the new, full grammar. Submit the program(s) and a Makefile so that I can:
  • Compile the parser by typing 'make parse'.
  • Compile the interpreter by typing 'make interp'.
  • Compile the extension by typing 'make extension'.
  • Submit a test strategy report called test.txt. This will include sample outputs, any code written especially for testing etc.
  • Submit an extension report called 'extend.txt'. This is quite brief and explains the extension attempted.
  • You need to be able to load a world file and a `.gvr` and say if they are valid of not.
  • Don't try to write the entire program in one go. Try a cut down version of the grammar first, e.g.:

```
<PROGRAM>    ::= <BLOCK>
<BLOCK>      ::= "turnoff" |
                 "move" <BLOCK> |
                 "turnleft" <BLOCK> |
                 "pickbeeper" <BLOCK> |
                 "putbeeper" <BLOCK>
<DO>         ::= "do" <num> ":"
                     <BLOCK>
```

  • Some issues, such as what happens if you hit a wall are not clear from the formal grammar. In this case, use your common sense, or do what the real program does.

12.3 NLab

- The programming language MATLAB (originally available in the late 1970s, for free) is one of the most widely used scientific langaues in the world.
- One of the most interesting things about MATLAB, is that every single variable is stored as a $2D$ array - even a scaler integer is simply a 1×1 array [1].
- Here, we develop a very simple version of this concept - a language that allows such arrays to be created or read from file, and functions performed on each part of the array, one element at a time.

Examples

```
BEGIN {
    SET $I := 5 ;
    PRINT $I
}
```

```
BEGIN {
    SET $I := 5 ;
    PRINT $I
}
```

sets the variable I to have the value 5, and prints it to the screen:

```
5
```

You can create an array full of ones and add 2 to each cell of the array:

```
BEGIN {
    ONES 6 5 $A
    SET $A := $A 2 B-ADD ;
    PRINT
    PRINT $A
}
```

```
ARRAY:
3 3 3 3 3
3 3 3 3 3
3 3 3 3 3
3 3 3 3 3
3 3 3 3 3
3 3 3 3 3
```

Loops are possible too, here a loop counts from 1 to 10 via the variable I and computes factorials in the variable F. Both variables are scalars (a 1×1 array) :

```
BEGIN {
    SET $F := 1 ;
    LOOP $I 10 {
        SET $F := $F $I B-TIMES ;
        PRINT $F
    }
}
```

---

[1]Actually as the name implies, they are all stored as matrices, but we will ignore the mathematical interprettion here.

```
1
2
6
24
120
720
5040
40320
362880
3628800
```

Such loops (like in C) have counters stored in a variable. Changing this variable inside the loop can affect when the loop ends :

```
# Notice that the loop counter is modified inside the loop
# causing it to count at twice the speed : 2 4 6 8 10
BEGIN {
    LOOP $I 10 {
        SET $I := $I 1 B-ADD ; PRINT $I
    }
}
```

```
2
4
6
8
10
```

As grammar tells you, loops can be nested too :

```
BEGIN {
    SET $A := 0 ;
    LOOP $I 5 {
        LOOP $J 5 {
            SET $A := $I $J B-TIMES ;
            PRINT $A
        }
    }
}
```

```
1
2
3
4
5
2
4
6
8
10
3
6
9
12
15
4
8
12
16
20
5
10
15
20
25
```

The Formal Grammar

```
<PROG> ::= "BEGIN" "{" <INSTRCLIST>


<INSTRCLIST> ::= "}" | <INSTRC> <INSTRCLIST>
<INSTRC> ::= <PRINT> | <SET> | <CREATE> | <LOOP>


# Print array or one-word string to stdout
<PRINT> ::= "PRINT" <VARNAME> | "PRINT" <STRING>


# One of the 26 possible (upper-case) variables
<VARNAME> ::= "$"[A-Z] # e.g. $A, $B, $Z etc.


# Because of the assumption that a program is just a list of words,
strings can't have spaces in them (for simplicity)
<STRING> :: Double-quoted string e.g. "../../doof.arr", "Hello!" etc.


<SET> ::= "SET" <VARNAME> ":=" <POLISHLIST>
<POLISHLIST> ::= <POLISH><POLISHLIST> | ";"


<POLISH> ::= <PUSHDOWN> | <UNARYOP> | <BINARYOP>
<PUSHDOWN> ::== <VARNAME> | <INTEGER>


# A non-negative integer
<INTEGER> ::= [0-9]+ # e.g. 1, 250, 3


# Pop one array, push the result.
# U-NOT : Flip the Boolean values of an array
# U-EIGHTCOUNT : Returns the numbers of true values around each cell in the array in its
# Moore 8-neighbourhood (north, south, west, east, NE, NW, SW, SE).
<UNARYOP> ::= "U-NOT" | "U-EIGHTCOUNT"


# Pop 2 arrays, push the resultant array
# If both arrays are bigger than 1x1, they must be the same size
# If one array is a 1x1 scalar, apply this value to each cell of the other array in turn
# B-TIMES operates on corresponding cells in turn (it is not a full matrix multiplication).
<BINARYOP> :: "B-AND" | "B-OR" | "B-GREATER" | "B-LESS" | "B-ADD" | "B-TIMES" | "B-EQUALS"


# Create an array full of ones, or read from a file
<CREATE> ::= "ONES" <ROWS> <COLS> <VARNAME> | "READ" <FILENAME> <VARNAME>
<ROWS> ::= <INTEGER>
<COLS> ::= <INTEGER>
<FILENAME> ::= <STRING>


# Loop using a variable to count from 1 (!) to <= <INTEGER>
# IF the variable
<LOOP> ::= "LOOP" <VARNAME> <INTEGER> "{" <INSTRCLIST>
```

Exercise 12.3.1   • 30% Implement a recursive descent parser - this will report whether or not a given NLab program follows the formal grammar or not. The input file is specified via `argv[1]` - there is **no** output if the input file is **valid**. Elsewise, a non-zero `exit` is made.

• 30% Extend the parser, so it becomes an interpreter. The instructions are now 'executed'. Do not write a new program for this, simply extend your existing parser.

• 20% Show a testing strategy on the above - you should give details of unit testing, white/black-box testing done on your code. Describe any test-harnesses used. In addition, give examples of the output of many different NLab programs. Convince me that every line of your C code has been tested.

• 20% Show an extension to the project in a direction of your choice. It should demonstrate your **understanding** of some aspect of programming or S/W engineering. If you extend the formal grammar make sure that you show the new, full grammar.

Hints

> Don't try to write the entire program in one go. Try a cut down version of the grammar first, e.g.:

- ```
  <PROG> ::== "BEGIN" { <INSTRCLIST>
  INSTRCLIST ::= "}" | <INSTR> <INSTRCLIST>
  <INSTR> ::= <PRINT> | <SET>
  <PRINT} ::= "PRINT" <VARNAME>
  <SET> ::= <VARNAME> ":=" <POLISHLIST>
  <POLISHLIST> ::= <POLISH> <POLISHLIST> | ";"
  <POLISH> ::= <VARNAME> | <INTEGER>
  ```

- The language is simply a sequence of words (even the semi-colons), so use `fscanf()`.
- Some issues, such as what happens if you use an undefined variable, or if you use a variable before it is set, are not explained by the formal grammar. Use your own common-sense, and explain what you have done.
- Once your parser works, extend it to become an interpreter. DO NOT aim to parse the program first and then interpret it separately. Interpreting and parsing are inseparably bound together.
- Start testing very early - this is a complex beast to test and trying to do it near the end won't work.
- In NLab, all variables are global i.e. they are not local to loops etc.

Submission

Your testing strategy will be explained in `testing.txt`, and your extension as `extension.txt`. For the parser, interpreter and extension sections, make sure there's a `Makefile`, so that I can easily build the code using `make parse`, `make interp` and `make extension`. Submit a single `nlab.zip` file.

12.4 The UNIX awk program

Sometimes handling files containing numerical data in C may be somewhat arduous. A 'simple' program to swap the first and second columns of a file is quite long in C.

For this reason, there is a simple language called awk which allows simple manipulation to be done on a line by line basis. For example:

```
{
print $2, $1;
}
```

swaps the first and second fields in a file. The whole program is applied to every line of the input file in turn.

Other examples of awk would include printing fields in reverse order:

```
{ for (i = NF; i > 0; --i) print $i }
```

Adding up first column, print sum and average

```
{ s += $1 }
END{print "sum is",s," average is",s/NR}
```

Notice the 'special' variables NR and NF. Other special variables include FS which defines how fields are separated (default space and tab). See 'man awk' for more information.

The CAWK formal grammar

This assignment involves writing a recursive descent parser for a simple, cut-down version of awk called 'cawk'. It is based on the following formal grammar:

```
<PROG>        := "{" <ILST> |
                  "{" <ILST> "END" "{" <ILST>
<ILST>        := <INSTR> <ILST> | "}"
<INSTR>       := <LET> | <IF> | <PRINT> | <FOR>
<LET>         := "LET" <USER> "=" <POLISH>
<USER>        : = "$A" | .. | "$Z"
<RD>          := <USER> | "$[" <INDEX> "]" | <SPEC>
<INDEX>       := number | <USER> | <SPEC>
<SPEC>        := "$#" | "$@"
<NUMVAR>      := number | <RD>
<POLISH>      := <OPER><POLISH> | ";"
<OPER>        := <NUMVAR> | <OPERATOR>
<OPERATOR>    := "+" | "-" | "*" | "/"
<IF>          := "IF" "(" <TEST> ")" "{" <ILST>
<TEST>        := <NUMVAR> <COND> <NUMVAR>
<COND>        := "<" | ">" | "==" | "!="
<PRINT>       := "PRINT" <NUMVAR> | "PRINT" string
<FOR>         := "FOR" <USER> "=" <NUMVAR> "TO"
                  <NUMVAR> "STEP" <NUMVAR> "{" <ILST>
```

Note, you may assume that the program consists of a list of words separated by white-space.

```
$# is the number of records.
$@ is the number of fields in the current record.
$[ 0 ] is the entire line
$[ i ] is the ith field of the record
```

Examples of CAWK

```
{
PRINT $[ 2 ]
PRINT $[ 1 ]
PRINT "\n"
}
```

Print fields in reverse order:

```
{
   FOR $I = $@ TO 1 STEP -1 {
      PRINT $[ $I ]
   }
   PRINT "\n"
}
```

> **Exercise 12.4.1** Write a C program to implement the above formal grammar. Your program should read in a cawk program (argv[1]) and expect the data file to be read from standard input (or from argv[2] if specified).
>
> The marks are split as follows:
> - (25%) To implement a recursive descent parser - this says whether or not a given CAWK program follows the formal grammar or not.

- (25%) To implement an interpreter, so that the instructions are executed.
- (25%) To show a testing strategy on the above - you should give details of white-box and black-box testing done on your code. Describe any test-harnesses used. Give examples of the output of many different cawk programs.
- (25%) To show an extension to the project in a direction of your choice. It should demonstrate your understanding of some aspect of programming or S/W engineering. If you extend the formal grammar make sure that you show the new, full grammar.

Submit the program(s) and a Makefile so that I can:
- Compile the parser by typing 'make parse'.
- Compile the interpreter by typing 'make interp'.
- Compile the extension by typing 'make extension'.

In addition:
- Submit a test strategy report called test.txt. This will include sample outputs, any code written especially for testing etc.
- Submit an extension report called 'extend.txt'. This is quite brief and explains the extension attempted.

12.5 Neill's Adventure Language

Some of the very earliest computer games were text-based adventures e.g. Colossal Cave

www   https://en.wikipedia.org/wiki/Colossal_Cave_Adventure

Here we write a simple language (NAL) which allows us to write (simplified) versions of such games, focussing on setting variables and printing. The grammar is as follows:

```
<PROGRAM> := "{" <INSTRS>
<INSTRS> := "}" | <INSTRUCT> <INSTRS>
<INSTRUCT> := <FILE> | <ABORT> | <INPUT> | <IFCOND> | <INC> | <SET> |
              <JUMP> | <PRINT> | <RND>


% Execute the instructions in file, then return here e.g. :
% FILE "test1.nal"
<FILE> := "FILE" <STRCON>


% Halt/abort all execution right now !
<ABORT> := "ABORT"


% Fill a number−variable with a number, or 2 string−variables with string :
% IN2STR ( $C, $ZER ) or INNUM ( %NV )
<INPUT> := "IN2STR" "(" <STRVAR> "," <STRVAR> ")" | "INNUM" "(" <NUMVAR> ")"


% Jump to the nth word in this file (the first word is number zero!)
% Brackets count as one word, "things in quotes" count as one word, e.g. :
% JUMP 5
<JUMP> := "JUMP" <NUMCON>


% Output the value of variable, or constant, to screen with (without a linefeed)
<PRINT> := "PRINT" <VARCON>
<PRINTN> := "PRINTN" <VARCON>


% Set a variable to a random number in the range 0 − 99 e.g. :
```

```
% RND ( %N )
% Number should be seeded via the clock to be different on successive executions
<RND> := "RND" "(" <NUMVAR> ")"

% If condition/test is true, execute INSTRS after brace, else skip braces
<IFCOND> := <IFEQUAL> "{" <INSTRS> | <IFGREATER> "{" <INSTRS>
<IFEQUAL> := "IFEQUAL" "(" <VARCON> "," <VARCON> ")"
<IFGREATER> := "IFGREATER" "(" <VARCON> "," <VARCON> ")"

% Add 1 to a number−variable e.g. :
% INC ( %ABC )
<INC> := "INC" "(" <NUMVAR> ")"

% Set a variable. All variables are GLOBAL, and persist across the use of FILE etc.
% $A = "Hello" or %B = 17.6
<SET> := <VAR> "=" <VARCON>

% Some helpful variable/constant rules
% (Here ROT18 is ROT13 for letters and rot5 for digits)
<VARCON> := <VAR> | <CON>
<VAR> := <STRVAR> | <NUMVAR>
<CON> := <STRCON> | <NUMCON>
<STRVAR> := $[A−Z]+
<NUMVAR> := %[A−Z]+
<STRCON> := A plain−text string in double−quotes, e.g. "HELLO.TXT",
           or a ROT18 string in hashes e.g. #URYYB.GKG#
<NUMCON> := A number e.g. 14.301
```

Note that string constants can be entered via the use of double quotes, or using hashes to encode strings using ROT18

in which characters are encoded/decoded according to:

```
Plain:  ABCDEFGHIJKLMNOPQRSTUVWXYZ
ROT13:  NOPQRSTUVWXYZABCDEFGHIJKLM
Plain:  abcdefghijklmnopqrstuvwxyz
ROT13:  nopqrstuvwxyzabcdefghijklm
Plain:  0123456789
ROT5 :  5678901234
```

the algorithm allows obvious 'spoilers' to be hidden from users browsing through `.nal` files, but is simple to apply.

A simple program showing the use of assignment, conditionals and ROT18 is shown:

```
{
    $A = "Neill"
    %E = 12.4
    IFEQUAL ( $A , #Arvyy# ) {
        PRINT #Uryyb Jbeyq!#
    }
}
```

Here string-variables (`$`) and number-variables (`%`) are initialised.

The use of 'JUMP' is shown next - this allows execution to move a chosen word in the program. Strings inside quotes count as one word, so the following contains six words:

```
{
    PRINT "Warning : Infinite Loop !"
    JUMP 1
}
```

'FILE' allows another file to be executed (as <PROG>) and then execution returns to the original when finished. All variables are shared across files and are global:

```
{
    PRINT "In test4, before"
    FILE "test1.nal"
    PRINT "In test4, after"
}
```

'RND' sets a variable to a number in the range $0 - 99$, while 'INC' adds one to a number-variable.

```
{
    %C = 0
    RND ( %A )
    PRINT %A
    INC ( %C )
    IFGREATER ( %C , 9 ) {
        ABORT
    }
    JUMP 4
}
```

'ABORT' halts execution instantly.

A simple guessing game is shown below:

```
{

    PRINT "I'm thinking of a number (0–99).\nCan you guess it?"
    RND ( %MINE )
    %CNT = 0

    INC ( %CNT )
    PRINT "Type in your guess"
    INNUM ( %GUESS )
    IFGREATER ( %CNT , 7 ) {
        PRINT #Gbb znal gevrf :–(#
        ABORT
    }
    IFGREATER ( %GUESS , %MINE ) {
        PRINT "Too Big ! Try again ... "
        JUMP 10
    }
    IFGREATER ( %MINE , %GUESS ) {
        PRINT "Too Small ! Try again ... "
        JUMP 10
    }
    IFEQUAL ( %MINE , %GUESS ) {
        PRINT #Lbh thrffrq pbeerpgyl, lbh jva :–)#
        PRINTN "Number of goes = "
        PRINT %CNT
        ABORT
    }
```

```
   }
```

'INNUM' gets a number (float) from the user, and the use of 'PRINT' (with a newline after), and 'PRINTN' (without a newline after) is demonstrated.

Exercise 12.5.1

- **(40%)** Implement a parser. The `.nal` file should be read in using `argv[1]`. If the file is parsed correctly, the only output should be:

```
Parsed OK
```

- **(30%)** Implement an interpreter, building on top of the parser in the manner described in the lectures. Do not write a brand new program - interpretation will be done alongside parsing.
- **(20%)** Submit the testing you have undertaken, including examples and a description of your strategies. This should convince us that you have tested every line of code, and that it works correctly. If there are still issues/bugs state them clearly. Also, point out any bugs that you have successfully found using these approaches. Submit a file named `testing.txt`, along with any other files you feel necessary. Due to the recursive nature of this assignment testing is non-trivial - simply submitting many `.nal` files that 'work' is not sufficient. No particular strategy is mandated. You may wish to explore a couple and briefly discuss strengths and weaknesses.
- **(10%)** Undertake an extension of your choosing. Remember, that the assessment is based on the quality of your coding, so choose something to demonstrate an aspect of programming or software engineering that you haven't had a chance to use in the main assignment. Submit a file named `extension.txt` outlining, in brief, your contribution.

Hints

Don't try to write the entire program in one go. Try a cut down version of the grammar first. Build-up from the `01s` example given in lectures.

- Some issues, such as what happens if you use an undefined variable, or if you use a variable before it is set, are not explained by the formal grammar. Use your own common-sense, and explain what you have done.
- Once your parser works, extend it to become an interpreter. DO NOT aim to parse the program first and then interpret it separately. Interpreting and parsing are inseparably bound together.

Submission

Your testing strategy will be explained in `testing.txt`, and your extension as `extension.txt`. For the parser, interpreter and extension sections , make sure there's one `Makefile`, so that I can easily build the code using `make parse`, `make interp` and `make extension`. I've given an example `makefile` in the usual place, but this is an example only - yours may be different. I wrote only one program `nal.c` and built the two different version by setting a `#define` **via the makefile with** `-DINTERP`. Inside the code itself `#ifdef INTERP` and `#endif` are used. Also make sure that basic testing is available using `make testparse` and `make testinterp`.

Place all the files required for your submission in a single `.zip` file called `nal.zip` - this file will not contain other zipped files.

12.6   NUCLEI

The programming language LISP, developed in 1958, is one of the oldest languages still in common use. The langauage is famous for: being fully parenthesised (that is, every instruction is inside its own brackets), having a prefix notation (e.g. functions are written (PLUS 1 2) and not (1 PLUS 2)) and its efficent linked-list Car/Cdr structure for (de-)composing lists.

Here, we develop a very simple langauge inspired by these concepts called NUCLEI (Neill's UnCommon Lisp Expression Interpreter) and a means to parse or interpret the instructions.

The interpreter (but not parser) builds on Exercise 11.4 - you'll need to have your own version of the `linked.c`, `lisp.h` and `specific.h` files.

Examples

Parsing :

```
(
   (SET A '1')
   (PRINT A)
)
```

leads to the output :

```
Parsed OK
```

or with the interpreter :

```
1
```

The `CONS` instruction is used to construct lists :

```
(
    (PRINT (CONS '1' (CONS '2' NIL)))
)
```

```
Parsed OK
```

and when interpreted :

```
(1 2)
```

The `CAR` instruction is used to deconstruct lists :

```
(
   (SET A '(5 (1 2 3))')
   (PRINT (CAR A))
)
```

```
Parsed OK
```

and when interpreted :

```
5
```

Loops are possible too, here a loop counts down from 5 to 1, using the variable `C` as a counter and a Boolean test :

```
(
    (SET C '5')
    (WHILE (LESS '0' C)(
        (PRINT C)
        (SET A (PLUS '-1' C))
        (SET C A))
    )
)
```

```
    Parsed OK
```

and when interpreted :

```
    5
    4
    3
    2
    1
```

The IF is similar; based on a Boolean, one of two possible sets of instructions are taken :

```
(
    (IF (EQUAL '1' '1') ((PRINT "YES"))((GARBAGE)))
)
```

Here the parser fails because it doesn't understand GARBAGE :

```
    Was expecting a Function name ?
```

However, the interpreter never gets to the *false* instruction since the Boolean equates to *true* and so :

```
    YES
```

The Formal Grammar

```
# (N)eill's (U)n(C)ommon (L)isp (E)xpression (I)nterpreter
<PROG> ::= "(" <INSTRCTS>

<INSTRCTS> ::= <INSTRCT> <INSTRCTS> | ")"

<INSTRCT> ::= "(" <FUNC> ")"

<FUNC> ::= <RETFUNC> | <IOFUNC> | <IF> | <LOOP>

<RETFUNC> ::= <LISTFUNC> | <INTFUNC> | <BOOLFUNC>

<LISTFUNC> ::= "CAR" <LIST> | "CDR" <LIST> | "CONS" <LIST> <LIST>

<INTFUNC> ::= "PLUS" <LIST> <LIST> | "LENGTH" <LIST>

<BOOLFUNC> ::= "LESS" <LIST> <LIST> | "GREATER" <LIST> <LIST> | "EQUAL" <LIST> <LIST>

<IOFUNC> ::= <SET> | <PRINT>
<SET> ::= "SET" <VAR> <LIST>
<PRINT> ::= "PRINT" <LIST> | "PRINT" <STRING>

# (IF (CONDITION) ((IF-TRUE-INSTRCTS)) ((IF_FALSE_INSTRCTS)))
<IF> ::= "IF" "(" <BOOLFUNC> ")" "(" <INSTRCTS> "(" <INSTRCTS>
```

```
<LOOP> ::= "WHILE""(" <BOOLFUNC> ")" "(" <INSTRCTS>

<LIST> ::= <VAR> | <LITERAL> | "NIL" | "(" <RETFUNC> ")"

# Variables are just one letter A, B, Z etc.
<VAR> ::= [A-Z]

# For printing only:
<STRING> ::= Double-quoted string constant e.g. "Hello, World!", or "FAILURE ?"

# Since lists appear in bracket, to differentiate them
# from instructions, we use 'single' quotes:
# For the parser, anything in single-quotes is fine - you don't need to "investigate" whether it's valid or not.
<LITERAL> ::= Single-quoted list e.g. '(1)', '(1 2 3 (4 5))', or '2'
```

Exercise 12.6.1   • 30% Implement a recursive descent parser - this will report whether
or not a given NUCLEI program follows the formal grammar or not. The input file is
specified via `argv[1]` - and if the file is valid the output is :

```
Parsed OK
```

Otherwise, a suitbale error message is given and a non-zero `exit` is made.

• 30% Extend the parser, so it becomes an interpreter. The instructions are now 'ex-
ecuted'. Do not write a new program for this, simply extend your existing parser. To
help with this, I've provided a Makefile that does some conditional compilation - it
effectively does a :

```
#define INTERP
```

depending upon whether you're compiling the parser or interpreter version of the code.
In the C file, you can do conditional compilation using the `#ifdef` :

```
#ifdef INTERP
        return Listfunc(s);
#else
        Listfunc(s);
        return;
#endif
```

• 20% Show a testing strategy on the above in `testing.txt` - you should give details
of unit testing, white/black-box testing done on your code, or any test-harnesses used.
Convince me that every line of your C code has been tested, but not just by showing it
running on some NUCLEI files.

• 20% Show an extension to the project in a direction of your choice. It should demon-
strate your **understanding** of some additional aspect of programming or S/W engin-
eering. If you extend the formal grammar make sure that you show the new, full
grammar.

Hints

Don't try to write the entire program in one go. Try a cut down version of the grammar
first, maybe something similar to :

```
<PROG> ::= "(" <INSTRCTS>
<INSTRCTS> ::= <INSTRCT> <INSTRCTS> | ")"
<INSTRCT> ::= "(" <LISTFUNC> ")" | "(" <IOFUNC> ")"
<LISTFUNC> ::= "CAR" <LIST> | "CDR" <LIST> | "CONS" <LIST> <LIST>
<IOFUNC> ::= "SET" <VAR> <LIST> | "PRINT" <VAR>
<LIST> ::= <LITERAL> | "NIL" | "(" <LISTFUNC> ")"
```

```
        <VAR> ::= [A-Z]
        <LITERAL> ::= Single-quoted list e.g. '(1)', '(1 2 3 (4 5))', or '2'
```

- Some issues, such as what happens if you use an undefined variable, or if you use a variable before it is set, are not explained by the formal grammar. Use your own common-sense, and explain what you have done.
- Once your parser works, extend it to become an interpreter. DO NOT aim to parse the program first and then interpret it separately. Interpreting and parsing are inseparably bound together.
- Start testing very early - this is a complex beast to test and trying to do it near the end won't work.
- In NUCLEI, all variables are global i.e. they are not local to loops etc.

Submission

Your testing strategy will be explained in `testing.txt`, and your extension as `extension.txt`. For the parser, interpreter and extension sections, make sure there's a `Makefile`, so that I can easily build the code using `make parse`, `make interp` and `make extension`. Submit a single `nuceli.zip` file, which has all the files required without sub-directories.

■

12.7 Turtle Graphics

History

Many attempts have been made to create programming languages which are intuitive and easy to learn. One of the best of these was *LOGO* which allowed children as young as 3 to learn a computer language. A subset of this language involved a "turtle" which could be driven around the screen using simple instructions.

An Example

At its most basic, the turtle is driven by simple instructions such as FORWARD to move the turtle and leave a trail behind it, and RIGHT which rotates the turtle clockwise :

```
START
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
END
```
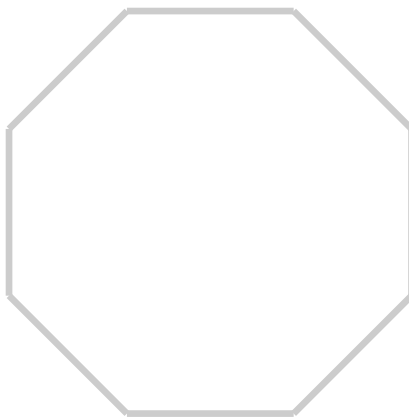
This may be simplified by using a loop to repeat an operation several times. Here our grammar allows a variable to iterate over each item in a list :

```
START
  LOOP A OVER { 1 2 3 4 5 6 7 8 }
    FORWARD 5
    RIGHT 45
  END
END
```

Here the actual values of the variable A weren't really important since the variable wasn't used directly inside the loop itself. Colours can be set by using string constants such as "RED" or "CYAN" :

```
START
  LOOP D OVER { 1 2 3 4 5 6 7 8 }
    LOOP C OVER { "RED" "GREEN" "YELLOW" "BLUE" }
      COLOUR $C
      FORWARD $D
      RIGHT 90
    END
  END
END
```
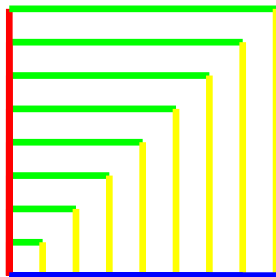

Variables can be set using a postfix notation, e.g. :

```
START
  SET A ( 0 )
  SET B ( $A 1 + )
  SET C ( $B 2 * )
END
```

The Formal Grammar

```
<PROG>    ::= "START" <INSLST>


<INSLST> ::= "END" | <INS> <INSLST>
<INS>     ::= <FWD> | <RGT> | <COL> | <LOOP> | <SET>


<FWD>     ::= "FORWARD" <VARNUM>
<RGT>     ::= "RIGHT" <VARNUM>
<COL>     ::= "COLOUR" <VAR> | "COLOUR" <WORD>
<LOOP>    ::= "LOOP" <LTR> "OVER" <LST> <INSLST>
<SET>     ::= "SET" <LTR> "(" <PFIX>


<VARNUM> ::= <VAR> | <NUM>
% Variables e.g. $A, $B, $Z etc.
<VAR>     ::= $<LTR>
% One Uppercase letter
<LTR>     ::= A, B ... Z
% Any valid double (as defined by scanf("%lf"...)
<NUM>     ::= 10 or -17.99 etc.


% A single word (as defined by scanf("%s"...) with double-quotes around it
% Valid colours include "BLACK", "RED", "GREEN", "BLUE",
% "YELLOW", "CYAN", "MAGENTA", "WHITE"
<WORD>    ::=  "RED", "BLUE", "HELLO!" or  "178"


<LST>     ::= "{" <ITEMS>
<ITEMS>   ::= "}" | <ITEM> <ITEMS>
<ITEM>    ::= <VARNUM> | <WORD>


<PFIX>   ::= ")" | <OP> <PFIX> | <VARNUM> <PFIX>
% A single mathematical operation character
<OP>      ::= + - / *
```

> ### Exercise 12.7.1
>
> **Parser (35%)**
>
> Implement a recursive descent parser - this will report whether or not a given turtle program follows the formal grammar or not. The input file is specified via argv[1] - there is **no** output if the input file is **valid**. Elsewise, a graceful non-zero exit is made. You should use the techniques described in the lecture for this - writing code that carefully follows the grammar. For instance, it won't use a tokenzier.
> All source code will be in the Parse/ sub-directory.
>
> **Interpreter (25%)**
> Extend the parser, so it becomes an interpreter. The instructions are now 'executed'. Do not begin a new program for this, simply copy and then extend your existing parser. Output is to a text file if the users specifies an argv[2], in the form of a 2D array of characters where colours are represented by blac(K), (R)ed, (G)reen, (Y)ellow, (B)lue, (M)agenta, (C)yan and

(W)hite, via a 51 wide × 33 height array of characters e.g.:

```
                        GGGGGY
                       R     Y
                       R     Y
                       R     Y
                       R     Y
                  WWWWWWWWWWW
                   W           W
                   W            W
                    W          W
                    W           W
                     W        W
                     W     W
                      W   W
                      W W
                       WW
                       W
```

If no `argv[2]` is specified, then output is to the screen using ANSI colour characters in same style as that used in Section 5.2, with a 1 second pause after each `FORWARD` instruction.
Note that some files that successfully parse will not successfully interpret (and vice versa) - there are example `.ttl` files provided to demonstrate this.
All interpreter source code will be in the `Interp/` sub-directory.

Postscript (5%)
Extend the output functionality of the interpreter such that if `argv[2]` has a `.ps` extension, then the correct postscript instructions are written to that file.

www   `https://paulbourke.net/dataformats/postscript`

Since PDFs are a more common file format, then at the end, your program should also convert the `.ps` file to a `.pdf` file using a `system()` call in your C code to `ps2pdf` which is installed on the lab machines. My implementation uses the postscript commands `showpage`, `newpath`, `moveto`, `lineto`, `setrgbcolour`, `scale` and `stroke`.
     If you get this to work, add it to the interpreter above - it will be tested using both `.txt` and `.ps` output modes.

Extension (15%)

Extend the project in a direction of your choice. It should demonstrate your **understanding** of some aspect of programming or S/W engineering. If you extend the formal grammar make sure that you give us the new, full grammar, along with the code in the Extension/ sub-directory named grammar.txt, in addition to a 300 word (max) description of what you've achieve in extension.txt. Please do not submit ideas for future work, only functionality you've got working.

Testing (20%)
Show the testing strategy on your code - you should give details of any unit testing or white/black-box testing done on your code. Describe any test-harnesses used. Convince us that every line of your C code has been tested, explaining the process, lessons learned and bugs found. Submit a file testing.txt in the sub-directory Testing/.

Hints
- Understand the noughts and ones example given in the notes.
- Don't try to write the entire program in one go. Try a cut down version of the grammar first, e.g.:

```
<PROG>    ::= "START" <INSLST>

<INSLST> ::= "END" | <INS> <INSLST>
<INS>     ::= <FWD> | <RGT>

<FWD>     ::= "FORWARD" <NUM>
<RGT>     ::= "RIGHT" <NUM>

<NUM>     ::= 10 or -17.99 etc.
```

- The language is simply a sequence of words (even the braces), so use fscanf().
- Some issues, (e.g. drawing out-of-bounds), incorrect postfix expressions cannot explained by the formal grammar. Use your own common-sense to decide how to reconcile these, and explain what you have done.
- Once your parser works, extend it to become an interpreter. DO NOT aim to parse the program first and then interpret it separately. Interpreting and parsing are inseparably bound together. Use our lectures notes for this and not those from other units such as Architecture which take a very slightly different approach.
- For the simple trigonometry required to compute the turtles destination, bear in mind that the cos() and sin() functions in C require parameters in radians and not degrees.
- The turtle always begins in the centre of the screen facing due north (upwards) and having a default white colour. For the postscript/PDF part, since white ink is hard to see on white paper, I've redefined white ink to be slightly grey.
- Start testing very early - this is a complex program to test and trying to do it (or explain it) near the end won't work.

Submission

Adapt the given Makefile if necessary to allow the parser, interpreter and extension to be built. Submit your extension via text files extension.txt in Extension/ along with grammar.txt and your source code.

Your testing strategy will be explained in testing.txt along with any other files you'd like to bring our attention to in Testing/.

Bundle this all up into a single `turtle.zip file` that contains **everything** required for us to compile and understand your code.

∎

A. House Style

A.1 Correctness

These style rules ensure your code is as-correct-as-can-be with the aid of the compiler and other tools:

FLAGS Having no warnings (or errors!) when compiling and executing with the flags:

For array bounds checking, `NULL` pointers being dereferenced etc:

```
-Wall -Wextra -Wfloat-equal -Wvla -pedantic -std=c99
-fsanitize=undefined -fsanitize=address -g3
```

For memory leaks:

```
-Wall -Wextra -Wfloat-equal -Wvla -pedantic -std=c99
-g3
```

then run:

```
valgrind --leak-check=full ./myexec
```

For 'final' production-ready code:

```
-Wall -Wextra -Wfloat-equal -Wvla -pedantic -std=c99
-O3
```

You can use more flags than this, obviously, but these will make sure a few of the essential warnings that commonly indicate the presence of bugs and leaks are checked. These guidelines are meant to be independent of the particular compiler used though. Sometimes it is helpful to use many compilers too, e.g. gcc and clang.

If you have unused variables (for example) in your code, it doesn't matter whether your compiler happened to tell you about it or not - it's still wrong !

BRACE Always brace all functions, `fors`, `whiles`, `if/else` etc. Somewhat contraversial, this ensures that 'extra' lines tagged onto loops are dealt with correctly. For instance:

```
while(i < 10)
    printf("%i\n", i);
    i++;
```

looks like it should print out `i` 10 times, but instead runs infinitely. The programmer probably meant:

```
while(i < 10){
    printf("%i\n", i);
    i++;
}
```

GOTO You do not use any of the keywords `continue`, `goto` or `break`. The one exception is inside `switch`, where `break` is allowed because it is essential ! These keywords usually lead to tangled and complex 'spaghetti' coding style. I often recommend that you rewrite the offending code uing functions, which **can** have multiple `returns` in them.

NAMES Meaningful identifiers. Make sure that functions names and variables having meaningful, but succinct, names.

REPC Repetitive code. If you've cut-and-paste large chunks of code, and made minor changes to it, you've done it wrong. Make it a function, and pass parameters that make the changes required.

```
int inbounds1(int i){
    if(i >=0 && i < MAX){
        return 1;
    }
    else{
        return 0;
    }
}

int inbounds2(int i){
    if(i >=0 && i < LEN){
        return 1;
    }
    else{
        return 0;
    }
}
```

might make more sense as:

```
int inbounds2(int i, int mx){
    if(i >=0 && i < mx){
        return 1;
    }
    else{
        return 0;
    }
}
```

GLOB No global variables. Global variables are declared 'above' `main()`, are in scope of all functions, and can be altered **anywhere** in the code. This makes it rather unclear **which** functions should be reading or writing them. You can make a case for saying that occasionally they could be useful (or better) than the alternatives, but for now, they are banned !

RETV Any functions that returns a value, should have it used:

```
scanf("%i", &i);
```

is incorrect. It returns a value that is ignored. Instead do:

```
        if(scanf("%i", &i) != 1{
            /* PANIC */
```

The only exceptions are `printf` and `putchar` which do return values but which are typically ignored.

MATCH For every `fopen` there should be a matching `fclose`. For every `malloc` there should be a `free`. This helps avoid memory leaks, when your program or functions are later used in a larger project.

STDERR When exiting your program in an error state, make sure that you `fprintf` the error on `stderr` and not `stdout`. Use `exit`, e.g.

```
    if(argc != 2){
        fprintf(stderr, "Usage : %s <filename>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
```

A.2 Prettifying

These rules are about making your code easier to read and having a consistent style in a form that others are expecting to see.

LLEN Line length. Many people use terminal and editors that are of a fixed-width. Having excessively long lines may cause the viewer to scroll to off the screen. Keep lines short, perhaps < 60 characters. However, in a similar way to the **FLEN** rule below, it's really about the complexity of the line that's the issue, not its absolute length. A programmer would generally find:

```
    bool arrcleanse(cell oldarr[HEIGHT][WIDTH], cell newarr[HEIGHT][WIDTH], int h, int w)
```

a great deal easier to read than:

```
    if(a < b && j++ >= szpar(e ? true : false) || h==4){
```

despite it being twice as long.

TABS Don't use tabs to indent your code. Every editor views these differently, so you have no guarantee that I'm seeing the same layout as you do. Use spaces. This also prevents issues when cutting-and-pasting from one source to another.

INDENT Indentation: choose a style for indentation and keep to it. I happen to use 3 spaces, put opening braces for functions on a new line, but at the end of `if`,`else`, `for`, `while` etc, then close them on a new line, underneath the 'i' of the `if`:

```
    int smallest(int a, int b)
    {
        if(a < b){
            return a;
        }
        else{
            return b;
        }
    }
```

You can use any style you like, as long as it's consistent.

MAIN The code should have function prototypes/definitions first, then `main()`, followed by the function implementation. This means the reader always know where to find `main()`, since it will be near the top of the file.

CAPS Constants are `#defined`, and use all CAPITALS. For instance:

```
    #define WEEKS 52
    #define MAX(a,b) (a < b ? b:a)
```

FLEN Short functions. All functions are short. It's quite difficult to put a maximum number
of lines on this, but use 20 as a starting point. Exceptions include a function that simply
prints a list of instructions. There would be no benefit in splitting it into smaller functions.
Short functions are easier to plan, write and test.

I find it more useful to think about how hard the function is to understand, rather than its
length. Therefore, a 30 line, simple function is fine, but an extremely complex and dense
15 line function might need to be split up, or more self-documentation added.

A.3 Readability

Your code should be self-documenting. Comments will be written when there is something
complex to explain, and only read when something has gone catastrophically wrong. In many
cases clever use of coding will avoid the need for them. The compiler never sees them, so cannot
check them. If you change your code, but not your comments, they can be highly misleading.

As Kevlin Henney said :

A common fallacy is to assume authors of incomprehensible code will somehow be
able to express themselves lucidly and clearly in comments.

MAGIC No magic numbers. There should be no inexplicable numbers in your code, such as:

```
if(i < 36){
```

It's probably unclear to the reader where the 36 has come from, or what it means, even if it
is obvious to the programmer at the time of writing the code. Instead, #define them with
a meaningful name. Array overruns are often cured by being consistent with #defines.

BRIEF Comments are brief, and non-trivial. Worthless commenting often looks something
like:

```
// Set the variable i to zero
int i = 0;
```

The programmer extracts no additional information from it. However, for more difficult
edge cases, a comment might be useful.

```
// Have we reached the end of the list ?
if(t1->h == NULL){
```

To prevent lines from becoming too long, it is good practice to put comments above the
line it refers to, not at the end of the same line.

TYPE You should use typedefs, enums and structs to increase readability.

INFIN No loops should be infinite. I'll never ask you to write a program that is meant to run
forever. Therefore statements such as

```
while(1){
```

or

```
for(;;;){
```

are to be avoided.

2DINDEX 2D Arrays in C are indexed [row][col]. Sometimes it may still work correctly,
especially if you've consistently confused the two. Therefore, if you write code that
indexes it [col][row], or [x][y] it will confuse anyone else trying to understand (or
reuse) your code. If you were to sketch a graph using (i,j) you'd almost certainly make *i*
the horizontal axis, and *j* the vertical. Therfore, for any two variables it makes more sense
to write [b][a] or [j][i].