# 6. Lists, Insertion Sort & more Recursion

## 6.1 A Simple Spelling Checker

Here, an insertion sort involes creating an abstract list data structure, and then reading strings one at a time (possibly from file) and placing them in the **correct** part of the structure. This has a complexty of $O(n^2)$.

For this purpose, a list of valid words (unsorted) is available from the usual place.

**Exercise 6.1** Write a program which, based on a list implemented using **arrays**, 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]`, and you can assume the array is of a fixed-size, and large enough to hold all words. How long does it take to build the list ? ■

**Exercise 6.2** Now extend Exercise 6.1 so that when the user is prompted for a word, they are told whether this word is present in the list or not. Use a binary search to achieve this. How much faster is this than a linear search ? ■

**Exercise 6.3** Now extend Exercise 6.1 so that when the user is prompted for a word, they are told whether this word is present in the list or not. Use an interpolation search to achieve this. How much faster is this than a linear search ? ■

**Exercise 6.4** Write a program which, based on a dynamic 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 ? ■

## 6.2 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.:

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

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
```

```
72 = 2^3 * 3^2
```

**Exercise 6.5** 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 :

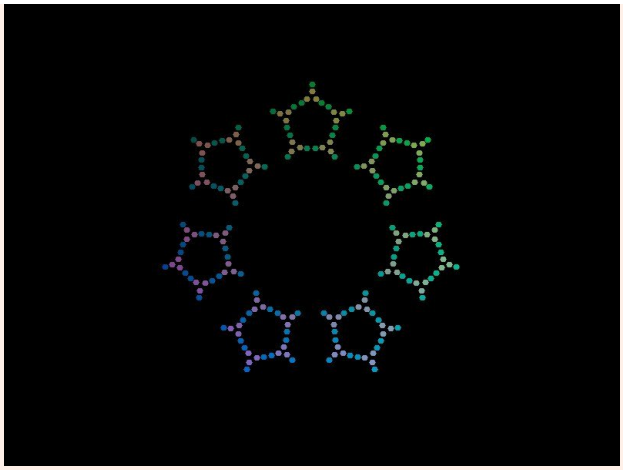
```
 % ./primefactors 27000
27000 = 1 x 2^3 x 3^3 x 5^3
 % ./primefactors 31
31 = 1 x 31
 % ./primefactors 38654705664
38654705664 = 1 x 2^32 x 3^2
```

■

For a beautiful visualisation of prime factors, see:


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

**Exercise 6.6** Adapt the program above to output a pattern similar to the animated display above, using SDL, but only for a single number, not an animation.



## 6.3 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 6.7** Write a program that, in plain text, produces a Sierpinski Carpet.

**Exercise 6.8** Write a program that, using ncurses, produces a Sierpinski Carpet.

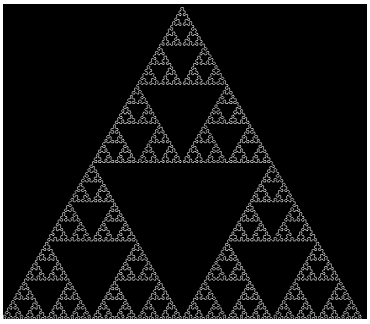
**Exercise 6.9** Write a program that, using SDL, produces a Sierpinski Carpet.

## 6.4 Sierpinski Squares

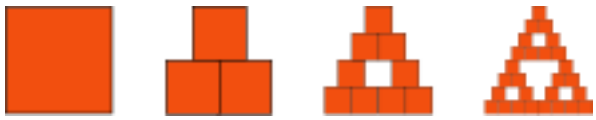
See also :

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 6.10** Write a program that, in plain text, produces a Sierpinski Triangle. ■

**Exercise 6.11** Write a program that, using ncurses, produces a Sierpinski Triangle. ■

**Exercise 6.12** Write a program that, using SDL, produces a Sierpinski Triangle. ■