The Binary Auditor™

# C++ Programming Exercises

www.Binary-Auditing.com

# Table of Contents

# 1 Common Array Algorithms

## 1.1 Exercise – Acid Level of Coffee

Say that you are interested in computing the average acid level of coffee as served by coffee shops in your home town. You visit many coffee shops and dip your pH meter into samples of coffee. You record your results in a text file such as the following. The first line of the file gives the number of values that follow.

```
13
5.6
6.2
6.0
5.5
5.7
6.1
7.4
5.5
5.5
6.3
6.4
4.0
6.9
```

Unfortunately, your pH meter is known to sometimes produce false readings. So you decide to disregard the reading that is most distant from the average.

Create a text file containing the above or similar data. Now write a program that reads the data into an array. Compute the average of all the data. Now scan through the array to find the value that is farthest (in either direction) from the average. Set this value to -1 to show that it is not to be included. Compute and print the new average.

Here is a run of the program:

```
C:\> CoffeeAverage < CoffeeData.txt
data[ 0 ] = 5.6
data[ 1 ] = 6.2
data[ 2 ] = 6.0
data[ 3 ] = 5.5
data[ 4 ] = 5.7
data[ 5 ] = 6.1
data[ 6 ] = 7.4
data[ 7 ] = 5.5
data[ 8 ] = 5.5
data[ 9 ] = 6.3
data[ 10 ] = 6.4
data[ 11 ] = 4.0
data[ 12 ] = 6.9
average: 5.930769230769231
most distant value: 4.0
new average: 5.6230769230769235
```

## 1.2 Exercise – 64x64 Image 1

Write a program that creates a 64 by 64 image as a text file. The image will consist of eight bands of increasingly higher values. Think of the image as 64 rows of 64 integers each, but actually write out one integer per line. This is for the convenience of the programs the input the image.

The image starts out with 8 rows of zero, so the program writes 8 times 64 zeros (as character '0'). Next, the image has 8 rows of eight, so the program writes 8 times 64 eights (as character '8'). Next, the image has 8 rows of sixteen, and so on. Write one value per line, without spaces or commas.

Use output redirection to send the output to a file. Use this file as input for the image display program (next exercise).

This is a very short program. The body consists of three lines: a double `for` loop with a one line loop body.

## 1.3 Exercise – 64x64 Image 2

It would be nice to display your images by some means other than printing out integers. Ideally, your programs should read and write images using standard image formats. But actual image formats, like *giff, tiff,* and *jpeg* are very complicated. Instead, write a program that displays an image by using rows of characters to represent brightness levels.

Write a program that reads in a file that contains one integer per line. Each integer represents one location in the image. Assume that there are 64 rows and 64 columns in the image. Assume that the integers are in the range 0 to 63.

For each integer, write out a single character depending on its value:

```
0 to 7: space
8 to 15: .
16 to 23: ,
24 to 31: -
32 to 39: +
40 to 47: o
48 to 55: O
above 55: X
```

Don't put extra spaces between characters. Start a new line after each row of 128 characters. This is one place where the `switch` statement is convenient. To use it, divide the input value by 8 to get the integer for the `switch`. Or you can use eight `if-else` statements if you prefer.

# 2 StringBuffers and StringTokenizers

## 2.1 Exercise – Newsletter Mailer

Write a program that creates a "personalized" letter, given a form letter and a person's name. The form letter will be an input file of text (use file redirection as discussed in chapter 23). The person's name will be a command line argument. The file will normal text, but with a * wherever a person's name should be substituted. For example:

```
Dear *,

I have exciting news for you, *!!! For just $49.99 plus postage
and handling you, *, can be the proud owner of a genuine leather
mouse pad!  No more finger strain for you, *, as you surf the web
with style.

Act Soon,

Venture Marketing Corp.
```

Assume the above is in a file *junk.txt*. A run of the program outputs:

```
% JunkGenerator "Occupant" < junk.txt
Dear Occupant,

I have exciting news for you, Occupant!!! For just $49.99 plus postage
and handling you, Occupant, can be the proud owner of a genuine leather
mouse pad!  No more finger strain for you, Occupant, as you surf the web
with style.

Act Soon,

Venture Marketing Corp.

%
```

Write the program so that it will substitute for any number of * on one line, and accepts any number of lines as input. The main program loop will be a `while` loop that continues until the input is `null`. If you wish to avoid the command line argument, ask the user for the occupant name and input it in the usual way.

# 3 File Input and Output

## 3.1 Exercise - File Splitting Program

Sometimes you have a file that you would like to put on a floppy disk but it is too big. Perhaps you want to copy the file from one computer to another and floppy disks are your only means of transfer. It would be useful to split the long file into several short files that each fit on a floppy. After transfer to the other computer the short files can be concatenated into a copy of the original. The command line for this program is:

```
./fileSplit bigFile baseName chunkSize
```

**bigFile** is the name of the big, existing file. Each small file will be named **baseName** with a number appended to the end. For example, if baseName is "chop" then the small files are "chop0", chop1", "chop2", and so on for as many as are needed.

**chunkSize** is the size in bytes of each small file, except for the last one.

For testing, use any file for bigFile, no matter what size, and any size for chunkSize. Use the file concatenation program to put the file together again. See if there are any changes with a file comparison program. (Such as Unix's dif or Microsofts Win's comp).

For preliminary testing, use text files. However, write the program so that it works with any file (use byte-oriented IO). For advanced testing, split an executable file, then reassemble then run it. Or split and reassemble an image file, and view the results.

## 3.2 Exercise - Text File Comparison Program

Write a program that compares two text files line by line. The command line looks like this:

```
./fileComp file1 file2 [limit]
```

Read in a line from each file. Compare the two lines. If they are identical, continue with the next two lines. Otherwise, write out the line number and the two lines, and continue. Frequently when two files differ there are very many lines that are different and you don't want to see them all. The last argument, limit, is optional (that is what the brackets mean). It is an integer that says how many pairs of different lines to write before quitting. If limit is omitted all differing lines are written. Catch the `NumberFormatException` that may be thrown if limit can't be converted.

## 3.3 Exercise - Experiment

Say that your are conducting an experiment to determine the effect of a high fiber diet on cholesterol levels in humans. You have several groups of human subjects. At the beginning of the experiment the cholesterol level of each subject in each group is measured.

Now the experiment runs for one month. Each group consumes a different amount of fiber everyday. At the end of the month you want to see the improvement (if any) in each groups cholesterol levels.

The data for the experiment will be in a text file that looks like the following. Each line of the text file contains a single integer (in character form).

```
number of groups
number of subjects in group 1
group1 subject1 starting cholesterol
group1 subject1 ending cholesterol
group1 subject2 starting cholesterol
group1 subject2 ending cholesterol
 . . . . .
group1 last subject starting cholesterol
group1 last subject ending cholesterol
number of subjects in group 2
group2 subject1 starting cholesterol
group2 subject1 ending cholesterol
 . . . . .
group2 last subject starting cholesterol
group2 last subject ending cholesterol
number of subjects in group 3
group3 subject1 starting cholesterol
group3 subject1 ending cholesterol
 . . . . .
group3 last subject starting cholesterol
group3 last subject ending cholesterol
. . . .
number of subjects in the last group
last group subject1 starting cholesterol
last group subject1 ending cholesterol
last group subject2 starting cholesterol
last group subject2 ending cholesterol
 . . . . .
last group last subject starting cholesterol
last group last subject ending cholesterol
```

For example, the following data file is for three groups. The first group has 2 subjects in it, the second group has 3 subjects in it, and the last group has 1 subject:

```
3
2
200
190
212
210
3
240
220
204
208
256
230
1
202
185
```

Assume that the data is correct (that the counts are correct and all the data is sensible and comes in complete pairs.) Write the program so that there is any number of groups (including zero) and so that a group can have any number of subjects in it (including zero.)

You program should do the following: *for each group* compute and print out the average of the starting cholesterol values, the average of the ending cholesterol values, and the change in the averages. For example, with the above data your program should print out something like this:

```
Group 1  2 subjects
  average starting cholesterol:  206
  average final    cholesterol:  200
  change in        cholesterol:  -6
Group 2  3 subjects
  average starting cholesterol:  233
  average final    cholesterol:  219
  change in        cholesterol:  -14
Group 3  1 subjects
  average starting cholesterol:  202
  average final    cholesterol:  185
  change in        cholesterol:  -17
Done with processing.
```

**Notes:** Use integer arithmetic throughout. If a group has zero subjects in it, report that fact but don't compute the averages (nor print them out.) It would be very helpful for you to plan this program in advance. The main organization of the program will be a counting loop inside of a counting loop.

Use some well though out comments in your program to show how it is organized. Be sure that your indenting also shows how the program is organized. Poor indenting will result in massive loss of points.

Here is another sample input file and its output:

```
4         Group 1  5 subjects
5           average starting cholesterol:  230
230         average final    cholesterol:  220
210         change in        cholesterol:  -10
230       Group 2  3 subjects
215         average starting cholesterol:  210
230         average final    cholesterol:  200
220         change in        cholesterol:  -10
230       Group 3  0 subjects
225       Group 4  2 subjects
230         average starting cholesterol:  205
230         average final    cholesterol:  195
3           change in        cholesterol:  -10
210       Done with processing.
200
210
200
210
200
0
2
200
190
210
200
```

# 4 Writing and Writing Text Files

## 4.1 Exercise - Total Count

Write a program that creates a file containing *TotalCount* random integers (in character format) in the range 0 to *HighValue-1*. Write *PerLine* integers per line. Separate each integer with one space. End each line with the correct line termination for your computer.

The user is prompted for and enters *HighValue*, which should be an integer larger than zero. Then the user is prompted for and enters *PerLine*, which is an integer greater than zero, and *TotalCount*, which also is an integer greater than zero. Finally the user is prompted for and enters the file name.

Construct a *Random* object and use its method *nextInt(int Top)*, which returns an *int* in the range 0..Top-1.

```
C:\Programs>  RandomIntData
Enter HighValue-->100
Enter how many per line-->10
Enter how many integers-->100
Enter Filename-->rdata.dat

C:\Programs>type rdata.dat
4 12 54 10 38 97 40 11 80 16
36 41 67 67 93 58 62 12 50 99
18 42 9 28 45 6 68 72 80 28
86 63 22 17 68 18 59 50 6 50
90 8 68 61 9 24 77 34 62 61
63 8 15 17 67 58 34 56 12 50
43 85 39 77 30 68 89 88 65 68
84 29 42 74 48 55 19 82 95 3
39 27 25 96 41 39 18 84 39 88
82 58 84 90 74 35 24 89 85 92
```

## 4.2 Exercise – Colored Java

Write a program that inputs a Java source code file and outputs a copy of that file with Java keyboards surrounded with HTML tags for bold type. For example this input:

```
public class JavaSource
{
  public static void main ( String[] args )
  {
    if ( args.length == 3 )
      new BigObject();
    else
      System.out.println("Too few arguments.");
  }
}
```

will be transformed into:

```
<b>public</b> <b>class</b> JavaSource
{
  <b>public</b> <b>static</b> <b>void</b> main ( String[] args )
  {
    <b>if</b> ( args.length == 3 )
      <b>new</b> BigObject();
```

```
    <b>else</b>
      System.out.println("Too few arguments.");
  }
}
```

In a browser the code will look like this:

```
public class JavaSource
{
  public static void main ( String[] args )
  {
    if ( args.length == 3 )
      new BigObject();
    else
      System.out.println("Too few arguments.");
  }
}
```

## 4.3 Exercise – Strip HTML Tags

Any text editor, such as Notepad, can be used to create web pages. Unfortunately, these editors usually do not check spelling. Word processors can open a text file and check its spelling. But when a file is sprinkled with HTML tags they all are flagged as errors and the real spelling errors are hard to see. This exercise is to write a utility that strips the HTML tags from a text file.

Write a program that reads in a text file and writes out another text file. The input file may have any number of HTML tags per line. The output file will be a copy of the input file but with spaces substituted for each HTML tag. The program will not check HTML syntax; it looks at the file as a stream of tokens and substitutes spaces for each token that is a tag. For this program, an HTML tag is any token that looks like one of these:

```
<Word>       </Word>
```

Assume that *Word* is a single word (perhaps just one letter or no letters) and that there are no spaces between the left and right angle brackets. With this definition, the following are tags:

```
<p>       </p>     <em>     </em>
<rats>       </1234>     <blockquote>     </>
```

With this definition, the following are NOT tags (although some are with real HTML):

```
< p>       </ p>     <em >     </e m>
<table border cellpadding=5>     <block quote>     < /em>
```

## 4.4 Exercise – Strip HTML Tags 2

Write the program to filter out any tag that looks like one of these:

```
<Word .... >       </Word ... >
```

Now `Word` is a single word that immediately follows the left angle bracket, but may be followed by more text which may include spaces. A tag ends with a right angle bracket, which might or might not be proceeded by a space. Assume that a tag starts and ends on the same line. With this definition, the following are tags

```
<p>       </p>     <em >     </em >
<table border cellpadding=5>     <word another word>     </x y z>
```

Start by setting a flag to *false*. Now look at the input stream of tokens one by one. When a token starts a tag set a Boolean flag to *true*. While the flag is *true* discard tokens until encountering a tag end (either stuff> or >). Set the flag to *false*.

## 4.5 Exercise – Frequency Analysis

**First Part:**

Letters of the alphabet occur in text at different frequencies. Write a program that confirms this phenomenon. Your program will be invoked from the command line like this:

```
  C:\mydir> freqCount avonlea.txt avonlea.rept -all
```

It will then read through the first text file on the command line (in this case "avonlea.txt") accumulating the counts for each letter. When it reaches the end of the file, it will write a report (in this case "avonlea.rept") that displays the total number of alphabetic characters "a-zA-Z" and for each character the number of times it occurred and the relative frequency with which it occurred. In counting characters, regard lower case "a-z" and upper case "A-Z" characters as identical.

You will need an array of 26 **long** integers, one per character. To increment the count for a particular character you will have to convert it into an index in the range 0..25. Do this by first determining which range the character belongs in:"a-z" or "A-Z" and then subtracting 'a' or 'A' from it, as appropriate:

```
      int inx = (int)ch - (int)'A' ;
      count[inx]++ ;
```

Discard characters not in either range without increasing any count.

**Second Part:**

Do the relative frequencies of the initial letters of words differ from the relative frequencies for all letters in a text? Add logic to the program so that it examines only the first character in

each word. Allow the user to chose between the two options with a switch on the command line:

```
C:\mydir> freqCount avonlea.txt avonlea.rept -first

It is often true that handling the an-
noying details makes up the large maj-
ority of the statements in a pro-
gram.
```

So, if the last token in a line (returned by `StringTokenizer`) ends with '-', don't include the first letter of the first token on the next line in the count.

**Testing:**

For testing, create some really simple files that demonstrate that your program is working. For instance:

```
AAAAAAAAAA AAAAAAAAAA AAAAAAAAAA AAAAAAAAAA AAAAAAAAAA
AAAAAAAAAA AAAAAAAAAA AAAAAAAAAA AAAAAAAAAA AAAAAAAAAA
aaaaaaaaaa aaaaaaaaaa aaaaaaaaaa aaaaaaaaaa aaaaaaaaaa
aaaaaaaaaa aaaaaaaaaa aaaaaaaaaa aaaaaaaaaa aaaaaaaaaa
!*$#)#%$#) @##$%))!__ !#4241-432 !_#*%_@*(* !@%#*#.,?+
```

and:

```
AAAAA AAAAA-
BBBBB BBBBB-
CCCCC CCCCC-

DDDDD-DDDDD
EEEEE-EEEEE
```

The first draft of your program will write its count to the monitor for easy debugging. Add text file output later. It is probably wise to write the first part of the program and debug it before moving on to the second option.

Download a text file of a novel of at least 400K bytes from Project Gutenberg.. Use a file that does not use HTML formatting tags (which would confuse the count). Delete the text at the beginning of the file that is not part of the novel (the legalese and documentation). Run both options of the program on the text.

**Example:**

Here is a sample run of my program with the text "Ann of Avonlea" from project Gutenberg.

```
C:\mydir> freqCount avonlea.txt avonlea.rept -all

C:\mydir>type avonlea.rept
Total alphabetical characters:  373267

 A:       31840       8.53%
 B:        5942       1.59%
 C:        7627       2.04%
 D:       17541       4.69%
 E:       45614      12.22%
 F:        7191       1.92%
 G:        7960       2.13%
 H:       22500       6.02%
 I:       25095       6.72%
 J:         733       0.19%
 K:        3443       0.92%
 L:       17534       4.69%
 M:        9324       2.49%
 N:       26516       7.1%
 O:       27344       7.32%
 P:        6083       1.62%
 Q:         275       0.07%
 R:       21285       5.7%
 S:       23398       6.26%
 T:       32579       8.72%
 U:       10720       2.87%
 V:        4201       1.12%
 W:        9063       2.42%
 X:         546       0.14%
 Y:        8745       2.34%
 Z:         168       0.04%
```

## 4.6 Exercise – Simple Encryption

Write an encryption program. Each byte of the source file is altered by reversing each bit. For example,

| input byte | output byte |
|---|---|
| 00110101 | 11001010 |
| 00000000 | 11111111 |
| 10000000 | 01111111 |

This operation is sometimes called a *bit-wise complement*. "Bit-wise" means that each bit is treated independently of all the others. "Complement" is just another word for reversal.

All the bits in an integer can be reversed as follows:

```
int value;
value = ~value;
```

The "~" (tilde) is the *bit-wise complement operator*. Visually it looks like a reversal of up and down. It reverses all the bits in `value`, even though you may be interested in only the low-order byte. But since the operation is bit-wise, the result for the low-order byte is the same no matter how many others are affected.

Encrypt a text file using your program. For amusement purposes, look at the result with Notepad. Now use the program again to encrypt the encrypted file. Look at the result with Notepad. Will you need to write a program to decrypt (decode) files?

This is not a very secure method of encrypting a file.

## 4.7 Exercise – Dumping Files

Write a program that reads in any file byte-by-byte and writes each byte to the monitor using two characters per byte. The eight bits of each byte are divided into two 4-bit groups, for example:

```
01011101 ==> 0101 1101
```

Each 4-bit group is represented with one character according to the following table:

| pattern | character | pattern | character |
|---|---|---|---|
| 0000 | 0 | 1000 | 8 |
| 0001 | 1 | 1001 | 9 |
| 0010 | 2 | 1010 | A |
| 0011 | 3 | 1011 | B |
| 0100 | 4 | 1100 | C |
| 0101 | 5 | 1101 | D |
| 0110 | 6 | 1110 | E |
| 0111 | 7 | 1111 | F |

Even though you are only interested in bytes, it is convenient to do all your bit manipulation using `int` variables. To make all bits in the `int` zero except for the low-order 4 bits, do this:

```
int data, lowFour;
lowFour = data & 0X0000000F ;
```

The "&" is the bit-wise AND operator. Now `lowFour` will contain one of the 16 possible patterns in its low-order 4 bits. To make all bits in the `int` zero except for the other 4 bits of the low-order byte, do this:

```
int data, highFour;
highFour = data & 0X000000F0 ;
```

To shift those four bits right so that they occupy the low-order 4 bit positions, do this:

```
highFour = highFour>>>4 ;
```

Display each byte of the input file using two characters followed by a space. Display 16 bytes per line (fewer on the last line). Here is an example of the program working:

```
C:\Programs>DIR
06/23/00  10:52p                1,859 HexDump.class
06/23/00  10:52p                2,855 HexDump.java
06/23/00  08:23p                   60 ints.dat

C:\Programs> HexDump ints.dat
00 00 00 0C 00 00 03 FF FF FF FF C8 00 01 4B 2D
00 00 00 00 FF FF DB 65 00 00 00 22 00 00 02 8E
FF FF FF E9 FF FF FE DA 00 00 00 12 FF FF FF E7
00 00 00 AD 00 00 00 2D FF FF FF FE
C:\Programs>
```

# 5 File Compression

## 5.1 Exercise – Run-Length Encoding

Write a program which encodes the following string using run-length encoding.

```
VVVNNDDDDDDDWJJJJJJJJJJJJJJJJJJJJJJJJJJJJ
```

a) not using an escape character

b) using the escape character **C**

```
A =  1        H =  8        O = 15        V = 22
B =  2        I =  9        P = 16        W = 23
C =  3        J = 10        Q = 17        X = 24
D =  4        K = 11        R = 18        Y = 25
E =  5        L = 12        S = 19        Z = 26
F =  6        M = 13        T = 20
G =  7        N = 14        U = 21
```

## 5.2 Exercise – Huffman Code

Write a program, Huffman, that has two public methods:

```
string Encode(string filename)
string Decode(string compressedFileContents)
```

The Encode method should read the contents of the file having the given pathname, and return a string, consisting of '0' and '1' characters, corresponding to the Huffman encoding of the contents of the file. The Decode method should be the inverse operation, taking, as its argument, the result of a previous call to Encode, and returning the original contents of the file as a string.

## 5.3 Exercise – Hash Table

Give the contents of the **hash** table when the given **items** are inserted in that order in an initially empty table of size 17 using double hashing. Use the following hash functions:

```
h(k) = (k + 1) mod 17
h2(k) = 8 - (k mod 8)
```

Use the following coding for the alphabet:

```
    A = 1   H =  8   O = 15   V = 22
    B = 2   I =  9   P = 16   W = 23
    C = 3   J = 10   Q = 17   X = 24
    D = 4   K = 11   R = 18   Y = 25
    E = 5   L = 12   S = 19   Z = 26
    F = 6   M = 13   T = 20
    G = 7   N = 14   U = 21
```

# 6 Recursion

## 6.1 Exercise – Prime Number

A **prime number** is an integer that cannot be divided by any integer other than one and itself. For example, 7 is prime because its only divisors are 1 and 7. The integer 8 is not prime because its divisors are 1, 2, 4, and 8.

Another way to define prime is:

```
prime(N)    = prime(N, N-1)

prime(N, 1) = true

prime(N, D) = if D divides N, false
              else prime(N, D-1)
```

For example,

```
prime(4)   = prime(4,3)
prime(4,3) = prime(4,2)
prime(4,2) = false
```

Another example,

```
prime(7)   = prime(7,6)
prime(7,6) = prime(7,5)
prime(7,5) = prime(7,4)
prime(7,4) = prime(7,3)
prime(7,3) = prime(7,2)
prime(7,1) = true
```

Put your method into a class, write a testing class, and test your program. This is because each activation in the activation chain requires some memory, and 12,000 activations uses up all the memory that has been reserved for this use.

# 7 Linked List

## 7.1 Exercise – Simple Linked List 1

Suppose that a linked list is made up of nodes of type

```
typedef struct node* link;
struct node {
   int key;
   link next;
};
```

that you are given a pointer "list" of type link, which points to the first node of the list; and that the last node has NULL as its link.

(a) Write a code fragment to delete the second node of the list. Assume that there are at least two nodes on the list.

(b) Write a code fragment to add a new node with key 17 just after the second node of the list. Assume that there are at least two nodes on the list.

(c) Write an iterative function count() that takes a link as input, and prints out the number of elements in the linked list.

(d) Write an iterative function max() that takes a link as input, and returns the value of the maximum key in the linked list. Assume all keys are positive, and return -1 if the list is empty.

## 7.2 Exercise – Simple Linked List 2

Repeat parts (c) and (d) above, but use a recursive function.

## 7.3 Exercise – Simple Linked List 3

Repeat 2 (c), but assume the linked list is circular, i.e., the last node points to the first node.

## 7.4 Exercise – ADT Sorted List

Implement an ADT (Abstract Data Type) Sorted List using an sorted linked list of integers ordered from smallest to largest. In addition to the operations specified in the ADT, the following public methods:

- a copy constructor
- void Clear(Success) that removes all of the items from a list.

You may include other methods, public or private, that may be useful to you for the problem. To construct the union and intersection, take advantage of the fact that the lists are ordered by "merging" the lists as follows.

**Structure**
The list elements are Strings. The list contains unique elments, i.e., no duplicate elements as defined by the key of the list. The strings are kept in alphabetical order. The list has a special property called the current position -- the position of the next element to be accessed by **getNextItem** during an iteration through the list. Only **reset** and **getNextItem** affect the current position.

**More Information**
Let L1 and L2 are the two lists to be merged and let L3 be the resulting list.

```
e1 = first element of L1
e2 = first element of L2
make L3 empty
while there are elements left in both L1 and L2
    if e1 < e2
        process e1 for L3
        e1 = next element of L1, if any
    else if e1 == e2
        process e1 for L3
        e1 = next element of L1, if any
        e2 = next element of L2, if any
    else // e1 > e2

        process e2 for L3 //
        e2 = next element of L2, if any
if there are elements left in L1
    process the remaining elements of L1, in order, for L3
if there are elements left in L2
    process the remaining elements of L2, in order, for L3
```

The union can be obtained by adding the elements to L3 in each case while the intersection adds only elements that are in both lists (e1 == e2) and can omit processing remaining elements since they are not in the intersection.

## 7.5 Exercise – Scrambled 8-Puzzle

Your program will find solutions to initially scrambled 8-Puzzles (as further defined below). Your program will report a single solution to an initial scrambled 8-Puzzle, the moves made to solve the 8-Puzzle, and the solved 8-Puzzle until the user is tired of running your program and types in q or Q to quit (c or C to continue). You will use a queue to store the unvisited states and a Stack to reconstruct the final solution path.

**Input:** There is no real input to this program, but you can read more about the 8-Puzzle here:

http://en.wikipedia.org/wiki/N-puzzle

**Output:** The program will report the initial scrambled 8-Puzzle, the moves made to solve the 8-Puzzle, and the solved 8-Puzzle for each main loop. At the end of each report the user will be prompted to continue (c or C) or quit (q or Q).

# 8 Sorting

## 8.1 Exercise – Bubble Sort

The first sort that many people learn, because it is so simple, is bubble sort: Keep passing through the file, exchanging adjacent elements that are out of order, continuing until the file is sorted. Bubble sort's prime virtue is that it is easy to implement, but whether it is actually easier to implement than insertion or selection sort is arguable. Bubble sort generally will be slower than the other two methods, but we consider it briefly for the sake of completeness.

Suppose that we always move from right to left through the file. Whenever the minimum element is encountered during the first pass, we exchange it with each of the elements to its left, eventually putting it into position at the left end of the array. Then on the second pass, the second smallest element will be put into position, and so forth. Thus, N passes suffice, and bubble sort operates as a type of selection sort, although it does more work to get each element into position.

For each i from l to r-1, the inner (j) loop puts the minimum element among the elements in a[i], ..., a[r] into a[i] by passing from right to left through the elements, compare–exchanging successive elements. The smallest one moves on all such comparisons, so it "bubbles" to the beginning. As in selection sort, as the index i travels from left to right through the file, the elements to its left are in their final position in the array.

```
static void bubble(ITEM[] a, int l, int r)
  { for (int i = l; i < r; i++)
      for (int j = r; j > i; j--)
        compExch(a, j-1, j);
  }
```

Write an application which takes a list of n numbers as input. Sort this list using the Bubble Sort algorithm and print out the result!

## 8.2 Exercise - Quicksort

The quicksort algorithm's desirable features are that it is in-place (uses only a small auxiliary stack), requires time only proportional to N log N on the average to sort N items, and has an extremely short inner loop. Its drawbacks are that it is not stable, takes about N2 operations in the worst case, and is fragile in the sense that a simple mistake in the implementation can go unnoticed and can cause it to perform badly for some files.

The performance of quicksort is well understood. The algorithm has been subjected to a thorough mathematical analysis, and we can make precise statements about its performance. The analysis has been verified by extensive empirical experience, and the algorithm has been refined to the point where it is the method of choice in a broad variety of practical sorting applications. It is therefore worthwhile for us to look more carefully than for other algorithms at ways of implementing quicksort efficiently. Similar implementation techniques are appropriate for other algorithms; with quicksort, we can use them with confidence, because we know precisely how they will affect performance.

Quicksort is a divide-and-conquer method for sorting. It works by partitioning an array into two parts, then sorting the parts independently. As we shall see, the precise position of the partition depends on the initial order of the elements in the input file.

The crux of the method is the partitioning process, which rearranges the array to make the following three conditions hold:

```
The element a[i] is in its final place in the array for some i.
None of the elements in a[l], ..., a[i-1] is greater than a[i].
None of the elements in a[i+1], ..., a[r] is less than a[i].
```

Write an application which takes a list of n numbers as input. Sort this list using the Quicksort algorithm and print out the result!

## 8.3 Exercise - Sorter

Write a program called `sorter` which behaves as follows:

- If there are no command-line arguments at all when the program is run, the program should print out instructions on its use. There should only be one usage message, and it must follow the standard conventions. Note that the optional command-line arguments must be included as part of the usage message.
- The program will be able to accept up to 32 numbers (integers) on the command line.
- If there are more than 32 numbers on the command line, or no numbers at all, the program should print out the usage message and exit.
- If the optional command-line arguments "`-b`" or "`-q`" are found **anywhere** in the command line, change the behaviour of the program as described below.
- If any of the command-line arguments to the program are not integers or one of the two optional command-line arguments, your program's response is undefined -- it can do anything. (*i.e.* you shouldn't worry about having to handle anything but integer arguments or the two command-line options). The way to deal with this is as follows: for each argument, first check to see if it's one of the command-line options. If so, proceed accordingly. If not, assume it represents an integer and convert it.
- Sort the numbers using either the minimum element sort or the bubble sort algorithm. **Do not use a global array to hold the integers**; use a locally-defined array in `main` and pass the array to the sorting function. Define separate functions for both sorting algorithms.
- Print out the numbers from smallest to largest, one per line.

### The Sample Output

If the above doesn't make sense, this is what your program should look like (bold is what you would type, and % is the Unix prompt):

```
% sorter 5 9 -2 150 -95 23 2 5 80
-95
-2
2
5
5
9
23
80
150
% sorter
usage: sorter [-b] [-q] number1 [number2 ... ] (maximum 32 numbers)
%
```

## 8.4 Exercise – Quicksort

The quicksort algorithm is really a large class of closely related algorithms, all with somewhat different running times. In this assignment you will implement a few variants of quicksort and time them. To make the differences in implementation more obvious, we'll do quicksort on integer keys with no associated data, since that uses a very fast comparison, and the effect of overhead will make a bigger relative difference.

Your task will be to write, document, and debug several variants of the quicksort algorithm, and do timing tests comparing the algorithms.

## 8.5 Exercise – Heapsort

For this assignment you will implement heaps, use them to do two variants of heapsort, and time the results. The times should be compared with quicksort, as in the previous assignment.

Start by creating code for a heap data structure. Once again, we're going to cheat a bit and have only int-valued objects in the heap---not pointers to more complex data objects as one would normally have.

The data structure for the heap should look something like

```
struct _heap
{   int * data;
    int NumHeap;  // number of elements in the heap
    int AllocHeap;// size of data array (NumHeap<=AllocHeap)
}
```

In the comments for your code, be sure to keep track of where the heap property is valid. The heap property is that data(parent)>=data(child). Since we are starting our numbering at 0, this means data[i]>=data[2*i+1] and data[i]>=data[2*i+2]. There are other important properties of the heap to maintain, as discussed in class (like using data[0]...data[NumHeap-1]).

Implement the following operations (using procedures or macros as appropriate)

h = new_heap(int a)
: Allocate a new empty heap of size a and return a pointer to it.

free_heap(struct _heap *h)
: free all the memory used in h. Note: it is a good idea to set h to 0 if you can, to avoid having dangling pointers.

push_heap(struct _heap *h, int x)
: push x onto the heap, maintaining the heap property.

x = top_heap(const struct _heap *h)
: return the largest value in the heap, without changing the heap.

x = pop_heap(struct _heap *h)
: return the largest value in the heap, removing it from the heap, and maintaining the heap property.

h = heapify(int *A, int num)

create a heap object in which data=A, AllocHeap=NumHeap=num, and the array A has been reorganized to have the heap property. Note that I want data to be a pointer to A, not to a

newly allocated copy of A. This means that h doesn't really own data, and so should not be used with free_heap(h). Discuss the advantages and disadvantages of defining heapify in this way.

I actually want you to do two implementations of heapify, one of which builds the heap from zero up, the other from the middle down.

The upward building heap has a loop in which NumHeap grows from 1 to num, with the heap property always holding for 0..NumHeap. This can be implemented as a very simple loop:

```
    h->NumHeap=1;
    while(h->NumHeapNumHeap]);
```

The downward building heap has a loop that counts downward, and maintains a loop invariant that has the heap property holding for [i..num]. The loop here is a bit trickier to write, but still fairly simple.

Build scaffolding to test out your heap manipulation routines. Use assertions as appropriate in your code to make sure that you have robust primitive operations.

Once your heaps are working correctly, write a heapsort routine that heapifies an array, then does repeated pop_heaps to fill the array from the end. You should have two versions of heapsort, corresponding to your two heapify implementations.

Time the heapsorts and compare them to your quicksort and insertion sort implementations. Put the results in a README file.

# 9 Searching

## 9.1 Exercise 1 – Word Search

Write a program that will play the game of Word Search. The game consists of an N x M matrix of characters and a list of words. The object is to find each word in the matrix. The words may appear in any direction (up, down, forward, backward, or any diagonal) but always in a straight line.

### Specifications

Input for this program will come from two sources: a file whose name is specified on the command line and the keyboard (standard input). Information about the word puzzle will be in the file. The words to be found will come from standard input.

Your program should begin by reading in the size of the puzzle. The information should appear as the number of rows (numRows) followed by the number of columns (numCols), both on the first line of the input file. The next numRows lines will each contain the numCols characters for that row of the puzzle. You should allocate a matrix (two-dimensional array of characters) that will hold a 20 by 20 word puzzle. Your program should terminate gracefully if numRows or numCols is greater than 20 or less than 1.

| Input from File | Output | Input from keyboard | Result |
|---|---|---|---|
| 10 8 | h p t n r o c r | turkey | turkey at 4,2 SE |
| hptnrocr | a e e h k c r a | pumpkin | pumpkin not found |
| aeehkcra | m o a i a u a g | pie | pie at 4,5 NW |
| moaiauag | i t m u p c n i | thanks | pie at 4,5 SW |
| itmupcni | r a u i o i b n | potato | pie at 4,5 SE |
| rauioibn | g t e r f p e d | ham | thanks not found |
| gterfped | l o n f k g r i | gravy | potato at 8,2 N |
| lonfkgri | i p u m h e r a | cranberry | ham at 1,1 S |
| ipumhera | p t g r a v y n | indian | ham at 10,6 NW |
| ptgravyn | s q u a s h p i | pilgrim | gravy at 9,3 E |
| squashpi | | corn | cranberry at 1,7 S |
| | | squash | indian at 4,8 S |
| | | thanks | pilgrim at 9,1 N |
| | | stuffing | corn at 1,7 W |
| | | | corn at 4,6 SW |
| | | | squash at 10,1 E |
| | | | thanks not found |
| | | | stuffing at 10,1 NE |

Print the puzzle before reading and trying to look up the words. Blank spaces printed between the individual characters of the puzzle ease the readability of the output because the line spacing is roughly twice the character spacing.

After the puzzle has been read and printed successfully, read input lines from standard input. Each line will contain one word to look for in the puzzle. Your program should print the word followed by the location (row,column coordinate of the first character) followed by the direction used to find the word. The character in the upper left corner is at location 1,1. The direction should be in compass coordinates (N, NE, E, SE, S, SW, W, NW) with N being up and E being right. If a word occurs more than once in the puzzle, you are to print a line for each occurrence. If it doesn't occur at all, you should say so.

You should test the program extensively with your own data that have varying sizes and characteristics.

## Hints for Solution

*Give matrix useful edges to aid word search.* Declare the matrix (a two-dimensional C array) to be 22 by 22 characters so it is big enough for the 20 by 20 possible data characters plus a border all around of '=' (a non-letter). Initialize the matrix (all 22 rows and columns) to the border character. With the border around the edge, you will not need to use any special tests in the code to determine if you have 'run off the edge'. When you run into the border, the characters will not match and you will stop the search.

*Break problem into several smaller problems.* The intermediate assignments will guide you here. Design the solution carefully and use several functions. Each function should do one conceptual action.

*Check for word at a location and direction.* It is **strongly** recommended that you construct a function that takes the puzzle, the word, a location (row,column coordinates), and a direction (the increment in the row,column coordinates to use for looking for the successive characters). It should return a value indicating whether or not the word occurs in the matrix at that location proceeding in that direction. In this assignment, do **NOT** nest loops more than two levels. We will count off at least ten points.

*Examine all directions at a location.* You should also note that given a word and a location, you can generate directions by using two nested for loops each running from -1 to 1. Be careful to skip 0,0.

# 10 Trees

## 10.1 Exercise – Memory Usage

For this assignment, we'll look mainly at memory use, though CPU time will also be of some importance. For this assignment you will implement child-sibling pointers and packed tries and measure how much space each takes up on a given input.

The program is to read in words from a file, treating white-space and punctuation as separators, and build a multiway tree containing the words (each node other than the root contains one letter and a path from the root to a node contains the prefix of all words in the subtree rooted at that node). On input, you should build the tree using the two-pointer/node representation (first-child and next-sibling pointers).

After reading in the words report the number of nodes in the tree (including the root) and the total memory used by the data structure. Be sure to include any overhead incurred. You may have to do your allocation through a wrapper, to keep track of how many calls are made and of what sizes.

You can also use various tools to measure memory use more directly, but these tend to be very operating-system specific. On i686 Linux machines, you may run your program under "valgrind", not free your data structures before exiting your program, and look at the memory leaks reported by valgrind.

Convert the tree to a packed trie and report the memory needed for it. Also report the memory space that would have been needed for an unpacked trie. This can be computed from number of nodes in the tree, but you have to be explicit about your assumptions.

Read another input file, looking up each word in the tree, reporting how many words were in the first file and how many are new. Report the time this operation takes with the child-sibling pointer representation and with the packed trie. Note: this is almost certain to be dominated by the I/O time for the file. How can you make a correct timing estimate for just the lookups?

**Hints**
The child-sibling data structure might look something like

```
struct node
{   struct node* child;
    struct node* right_sibling;
    unsigned char letter;
    char is_word; // really just a bool, but make sure it is one byte
}
```

with the "right_sibling" links forming a sorted linked list of possible children of the parent of the node. The packed-trie data structure might look something like

```
unsigned char *letter_for_branch;
unsigned int *subscript_for_branch;
```

with subscripts into the array identifying nodes. The child of i for letter c is subscript_for_branch[i+c], if letter_for_branch[i+c}==c, and 0 otherwise.

Alternatively, you could use an array Trie of

```
struct node
{    unsigned char letter;
     unsigned int subscript_for_branch : 24;
}
```

The child of Trie[i] for letter c is Trie[i+c].subscript_for_branch if Trie[i+c].letter==c and is 0 otherwise.

There are several ways to store the "is_word" information. Here is one: If you assume a 7-bit alphabet, you can pack the "is_word" bit into the letter_for_branch character, either with explicit masking operations or by using C's "bit field" definitions. With this representation, no node is needed for leaves---the subscript_for_branch can be 0 (or whatever you use to represent the null pointer). Alternatively, you could allow only 23 bits for the subscript_for_branch, and steal a bit for is-word from there.

One problem you might run into when packing the tries is making sure that no two trie nodes start at the same place in the array. If you force every trie node to have a branch to the same character (either an illegal character like 01 or a common character like 'e'), then any packing that avoids conflicts will also avoid starting trie nodes at the same place. You could make a somewhat smaller packed trie, at the cost of somewhat more temporary storage, by keeping a separate bitset of which starting locations have been used, and discarding the bitset when the packed trie is complete. You could also steal yet another bit from the subscript_for_branch field, and use it for marking which indices are starting points.

# 11 Mazes

## 11.1 Exercise – Maze Problem

The goal of this project is to write a program that will automatically generate and solve mazes. Each time the program is run, it will generate and print a new random maze and solution. You will use the disjoint set (*union-find*) data structure, and a *pre-order traversal* algorithm.

Conceptually, to generate a maze, first start with a grid of rooms with walls between them. The grid contains *r* rows and *c* columns for a total of *r*c* rooms. The missing walls at the top left and bottom right of the maze border indicate the start and finish rooms of the maze.

Now, begin removing interior walls to connect adjacent rooms. The difficultly in generating a good maze is in choosing which walls to remove. Walls should be removed to achieve the following maze characteristics:

1. Randomized: To generate unpredictable different mazes, walls should be selected randomly as candidates for removal. For randomization, you should use the rand() function to generate random numbers. Use srand() to initially seed the random number generator with a unique value, e.g. srand((unsigned)time(NULL)).

2. Single solution: There should be only one path between the start room and the finish room. Unnecessarily removing too many walls will make the maze too easy to solve. Therefore, a wall should not be removed if the two rooms on either side of the wall are already connected by some other path. For example, in Figure 2, the wall between a and f should not be removed because walls have previously been removed that create a path between a and f through b,c,d,e. Use the disjoint set data structure to determine room connected-ness.

3. Fully connected: Enough walls must be removed so that every room is reachable from the start room. There must be no rooms or areas that are completely blocked off from the rest of the maze. Figure 3 shows an example generated maze.

**Solving the Maze:**
After generating a maze, your program should then solve the maze using a modified pre-order traversal. The search algorithm will begin at the start room and search for the finish room by traversing wall openings. The search should terminate as soon as the finish room is found. You will output the order in which rooms where visited and indicate the shortest solution path from start to finish.

**Design:**
You will need a data structure to represent the maze of rooms and walls. Since the size of the graph is known at start-up, a 2-dimensional array-based implementation that mimics the room grid structure may work well. To generate the maze, the process should iteratively select a random wall and test whether that wall can be removed from the maze. This should continue until the maze is complete. This process will require two additional data structures. First, you need an efficient way to randomly select walls, such that a single wall is never selected more than once. To do this you will need to maintain a separate **list of walls**. As walls are randomly selected, they should be eliminated from the wall list so they can never be selected again. (How can you directly select and delete items from a list in an efficient way?) This places a tight upper bound on the number of iterations of the wall-removal loop.

Second, you need an efficient way to test if the selected wall is allowed to be removed from the maze (according to rule #2 above). You must use the **disjoint set** data structure for the union and findroot operations on rooms when generating the maze. It is required that you implement the **weighted union rule** and the **path compression** technique for maximum efficiency. Rooms that are connected by some path are in the same set. The *findroot* operation therefore reveals whether two rooms are already connected by some path. When removing a wall, the *union* operation is used to join the two room sets together. The disjoint set data structure enables efficient processing of the union and findroot operations, so that maze generation is fast.

How do you know when the maze is complete (according to rule #3)? The maze generator is done when there is only one set left in the disjoint set data structure, indicating that all rooms are connected. To solve the maze, you will need to modify the pre-order traversal to search the maze. Since the maze data structure does not indicate parent-child relationships you will need a way to prevent your traversal from cycling back on itself. You can use the maze data structure to keep track of various information as you traverse.

**Input:**
The program should be named *maze* and should accept the number of rows r and columns c of rooms in the maze as command-line arguments. If no command line arguments are given, it should default to 20 rows by 20 columns. The following invocation would create a maze that is 10 rows by 20 columns:

```
% maze 10 20
```

**Output:**
The program should print the maze to standard-out 3 times, separated by a blank line, as follows. The first output should simply print the maze. The maze is printed in ASCII using the vertical bar | and dash - characters to represent walls, + for corners, and space character for rooms and removed walls. The start and finish rooms should have exterior openings as shown. The second output should print the maze showing the shortest solution path. The maze should be printed exactly as above except using the hash # character for rooms and wall openings on the shortest solution path.

The third output should print the maze showing the order that the rooms were visited by the algorithm. The maze should be printed exactly as the first except that rooms should be printed with the low-order digit of the visitation order number. The start room is 0. Unvisited rooms should remain a space character. You will need to view the output in a fixed-width font. The program should output <*pre*> as the first line. This will enable you to view your mazes correctly in a web browser by directing standard output to a file on the command line:

```
% maze 4 4 > mymaze.html
```

Following is sample output for the maze:

```
+ +-+-+-+
| | |
+ +-+-+ +
| | | |
+ + +-+ +
| | |
+-+ + + +
| | |
+-+-+-+ +


+#+-+-+-+
|# | |
+#+-+-+ +
|#| | |
+#+ +-+ +
|###|###|
+-+#+#+#+
| ###|#|
+-+-+-+#+


+ +-+-+-+
|0 1 2| |
+ +-+-+ +
|3| | |
+ + +-+ +
|4 5|9 0|
+-+ + + +
|7 6 8|1|
+-+-+-+ +
```

# 12 Graphs

## 12.1 Exercise - Reachability Operation

Write a reachability operation that given two nodes, *i* and *j*, determines whether there exists a path in the graph from *i* to *j*. Write an operation that given node *i*, return a list or set of all the nodes reachable from *i*. Write an operation that computes a reachability matrix so future reachability queries can be answered in constant time. include a test program.

## 12.2 Exercise - Weighted Graph

Implement a version of the graph class (perhaps by using inheritance) that allows for information to be stored at the edges as well as the nodes. A *weighted graph* is a graph with a number associated with each edge. Recall, a *spanning tree* is a tree consisting of all the nodes in a graph and a collection of edges from the graph that connect the nodes to make a tree (no loops). A *minimum spanning tree* of a weighted graph is a spanning tree of a graph such that the sum of all the weights along the edges of the tree is less than or equal to the sum of the weights along the edges of any other spanning tree for that graph. Write an operation to compute the minimum spanning tree of a weighted graph.

## 12.3 Exercise - Shortest Path

A *shortest path* between nodes *i* and *j* in a graph is a path from *i* to *j* such that the sum of the weights of the edges along the path is less than or equal to the sum of the weights along the edges of any other path from *i* to *j*. (In a graph without edge weights, we can pretend each edge has a weight of 1 and then a shortest path from *i* to *j* refers to a path from *i* to *j* with the fewest edges.) Write a program to compute the shortest path between any two nodes in a graph.

# 13 Final Exam 1

A great deal of software is distributed in the form of executable code. The ability to reverse engineer such executables can create opportunities for theft of intellectual property via software piracy, as well as security breaches by allowing attackers to discover vulnerabilities in an application. The process of reverse engineering an executable program typically begins with disassembly, which translates machine code to assembly code. This is then followed by various decompilation steps that aim to recover higher-level abstractions from the assembly code. Most of the work to date on code obfuscation has focused on disrupting or confusing the decompilation phase.

We can introduce disassembly errors by inserting “junk” bytes at selected locations in the instruction stream where the disassembler is likely to expect code. An alternative approach involves partially or fully overlapping instructions. It is not difficult to see that any such junk bytes must satisfy two properties. First, in order to actually confuse the disassembler, the junk bytes must be partial instructions, not complete instructions. Second, in order to preserve program semantics, such partial instructions must be inserted in such a way that they are unreachable at runtime. To this end, define a basic block as a candidate block if it can have such junk bytes inserted before it. In order to ensure that any junk so inserted is unreachable during execution, a candidate basic block cannot have execution fall through into it. In other words, the basic block immediately before a candidate block must end in an unconditional control transfer, e.g., an unconditional jump or a return from a function. Candidate blocks can be identified in a straightforward way by scanning the basic blocks of the program after their final memory layout has been determined.

For example the given C code (which contains a Buffer Overflow vulnerability) can be compiled with the GCC compiler.

```
int main(int argc, char **argv, char **envp) {
     char buf[256];
     strcpy(buf, argv[1]);
     return 0;
}
```

Using the command

```
gcc -S vuln.c
```

we receive the assembly code in a file *vuln.s* before it is finally linked:

```
     .file  "vuln.c"
     .text
.globl main
     .type  main, @function
main:
     pushl  %ebp
     movl   %esp, %ebp
     subl   $264, %esp
     andl   $-16, %esp
     movl   $0, %eax
     addl   $15, %eax
     addl   $15, %eax
     shrl   $4, %eax
     sall   $4, %eax
     subl   %eax, %esp
     movl   12(%ebp), %eax
```

```
     addl   $4, %eax
     movl   (%eax), %eax
     movl   %eax, 4(%esp)
     leal   -256(%ebp), %eax
     movl   %eax, (%esp)
     call   strcpy
     movl   $0, %eax
     leave
     ret
     .size  main, .-main
     .ident "GCC: (GNU) 4.0.3 (Ubuntu 4.0.3-1ubuntu5)"
     .section     .note.GNU-stack,"",@progbits
```

Now consider a simple obfuscation. We take the following code snippet from above

```
main:
     pushl  %ebp
     movl   %esp, %ebp
     subl   $264, %esp
     andl   $-16, %esp
     movl   $0, %eax
```

and then we fill it with simple obfuscating junk code:

```
main:
     pushl  %ebp

     addl   $15, %eax
     subl   $15, %eax

     movl   %esp, %ebp
     subl   $264, %esp

     addl   $15, %eax
     subl   $15, %eax

     andl   $-16, %esp
     movl   $0, %eax
```

# 14 Final Exam 2

Write an application which takes a filename via command line, like:

```
./MyObfuscator myTarget.s
```

Then obfuscate the target by inserting junk code and fake instructions. Do not stay at the simple level like the example given above. Be innovative by inserting more than this. For example insert full junk blocks. Do some research on your own by searching the web. Try to find options for a complicated obfuscation using anti-disassembly or anti-debugging tricks.

The application should then write the obfuscated target into a new file like myTarget.obf.

Try to link the resulting obfuscated file. The most important rule is that the resulting file is still linkable.

Then run the resulting binary. The binary still has to work and your job is that it does not crash due your obfuscation. Examine the resulting binary with a debugger such as GDB.

# 15 Final Exam 3

It is your job now to write a binary watermarking protector. Using the techniques from examination task 1 the goal is to write an application which takes an unlinked input files (such as vuln.s), an watermarking encryption file and additional a text parameter as secret text:

```
./MyWatermarker vuln.s watermarkingenc.txt MySecretWatermarkText
```

The binary watermarking protector has to do the following job process:

For each character of the secret text (MySecretWatermarkText) the protector looks up the encryption code in the encryption table (watermarkingenc.txt). The encryption table can have the following format but you are free to design your own one:

```
[M]
addl $15, %eax
subl $15, %eax

[y]
addl $15, %eax
subl $15, %eax
addl $12, %eax
subl $12, %eax

[S]
subl $13, %eax
addl $13, %eax
subl $14, %eax
addl $14, %eax

[e]
.....
```

Then the protector fills the target file (vuln.s) with the encryption sequence given by the secret text (MySecretWatermarkText).

# 16 Final Exam 4

Finally write an application as watermark checker, which uses the same encryption table as used in examination task 2. The application takes parameters as following:

```
./MyWatermarkChecker myBinary watermarkingenc.txt MySecretWatermarkText
```

If the watermark is found print out a „Watermark OK“ message, if not a „Watermark failed“ message.

Submit your watermark detector as final examination at the Certification-Labs. Provide examples with short explanations to show that your application is working!