The Binary Auditor™

# Assembly Language 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:

```
% 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 – Spam 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. 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.

# 4 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.

```
% RandomIntData
Enter HighValue-->100
Enter how many per line-->10
Enter how many integers-->100
Enter Filename-->rdata.dat

% less 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 – 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:

```
% 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:

```
% 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.

```
% freqCount avonlea.txt avonlea.rept -all

% less 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%
```

## 4.4 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. Now use the program again to encrypt the encrypted file. Will you need to write a program to decrypt (decode) files?

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

# 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
```

Write a program to calculate prime number in a recursive way. Keep it tiny and simple!

# 7 Sorting

## 7.1 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!

## 7.2 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 Searching

## 8.1 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 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.

# 9 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 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
```

# 10 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.

# 11 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).

# 12 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!