Brainfuck for Dummies

Last updated: 2014-07-21.

1 Preface

Brainfuck is a very small programing language, consisting of only eight statements. The language is "Turing-complete" which means that (theoretically) it can perform any computation that can be done in (for example) Basic or C. I have compiled this document to collect things I found on Brainfuck.

1.1.1 Copyright and Licence

2 A brief history of Brainfuck[1]

Except for its two I/O commands, Brainfuck is a minor variation of the formal programming language P″[2] created by Corrado Böhm in 1964. He was using six symbols equivalent to the respective Brainfuck commands **+**, **−**, <, >, [and]. Böhm provided an explicit program for each of the basic functions that together serve to compute any computable function. So in a very real sense, the first "Brainfuck" programs appear in Böhm's 1964 paper – and they were programs sufficient to prove Turing-completeness.

Urban Müller created Brainfuck in 1993 with the intention of designing a language which could be implemented with the smallest possible compiler, inspired by the 1024-byte compiler for the FALSE programming language. Since then, several Brainfuck compilers have been made which are smaller than 200 bytes.

Since its creation, Brainfuck has inspired many variants.

2.2 The Brainfuck model

The Brainfuck language uses

- a simple machine model consisting of the program and instruction pointer, as well as
- an array of at least 30,000 byte-sized[3] cells initialized to zero;
- a movable data pointer (initialized to point to the leftmost byte of the array); and
- two streams of bytes for input and output (most often connected to a keyboard and a monitor respectively, and using the ASCII character encoding).

Unlike some small languages, the program memory and data memory are separated. No command can make the data cells affect the program cells.

3 The basics: the eight Brainfuck statements

The language consists of eight commands listed below.

Character	Explanation	In other words
>	increment the data pointer.	Move the cursor one cell right.
<	decrement the data pointer	Move the cursor one cell left.
+	increment the byte at the data pointer.	Increase the value of the current cell by 1
-	decrement the byte at the data pointer.	Decrease the value of the current cell by 1
.	output a character, the ASCII value of which being the byte at the data pointer.	Write the ASCII character of the current cell on screen / to file
,	accept one byte of input, storing its value in the byte at the data pointer.	Input a byte and store it in the current cell.
[	if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it forward to the command after the next] command.	"While" the current cell is nonzero, perform the statements between the brackets.
]	if the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, jump it back to the command after the previous [command.	End of a while loop

3.1 Brainfuck programs

A Brainfuck program is a sequence of these commands. Other characters can be placed in between the commands and are ignored. The commands are executed sequentially except at the end of a [] loop; an instruction pointer begins at the first command, and each command it points to is executed, after which it normally moves forward to the next command.
The program terminates when the instruction pointer moves past the last command.

3.2 Exercise

1. Describe each of the eight commands and name the equivalent command in your preferred language (BASIC, Java etc).
2. Imagine the very short brainfuck code "`[]`". What is the behaviour of this program for any value of the current cell?

4 Your first program in Brainfuck

4.1 Read-and-write (cat)

Your first program will take input from the user, and then output that same value on the output, looping until the user enters ASCII value zero. This program is also known as *cat*, named after the unix command. The full syntax of this porgram is:

```
,[.,]
```

There is a variant:

```
,+[-.,+]
```

which does the same, but exits when the user enters Alt-255[4]. In some environments it's hard to enter the ASCII 0 character.

4.1.1 Explanation

This program had five characters: `,` `[` `.` `,` `]`

1. The first character is the comma. This reads one ASCII value from the keyboard.
2. The second character is the open square bracket [. This is the start of the while loop that ends with the closing square bracket]. If the memory cell the program is pointing at contains the value zero at the moment of execution, the while loop will end and the program will continue after the closing square bracket. If there is no further code, the program ends.
3. The third character is the decimal point. This will write the current value of the memory cell to the screen. Note that we requested the user to input a character before the while loop. if we hadn't, the program would have ended right away without printing anything.
4. The fourth character is again the comma. Having printed the output, the user is again asked to input a value.
5. The 5th and last character of this "program" closes the while loop. If at some moment the program finds a memory cell containing 0 at the moment the opening bracket is executed, the program will continue after this closing bracket.

Note the order within the while loop: first a character is printed, then a new character is read. This is because of the while loop. In order for the loop to even start, the user must first have input a character.

5.2 Hello World

The 'Hello World' program is used as a demonstration of a simple program: putting a statement on-screen.
In Brainfuck this is not a simple task. The nature of Brainfuck (each letter must be defined in advance by assigning the appropriate ASCII value to a cell) makes the program rather large.

This is the listing with some comments added. An in-depth explanation will follow after the listing.

```
 +++++ +++++               initialize counter (cell #0) to 10
[                          set the next four cells to 70, 100, 30 and 10
      > +++++ ++           add  7 to cell #1
      > +++++ +++++        add 10 to cell #2
      > +++                add  3 to cell #3
      > +                  add  1 to cell #4
      <<<< -               decrement counter (cell #0)
]

> ++ .                print 'H'  (H = ASC (72))
> + .                 print 'e'  (e = ASC (101))
+++++ ++ .            print 'l'
.                     print 'l'
+++ .                 print 'o'

> ++ .                print ' '

<< +++++ +++++ +++++ .  print 'W'
> .                   print 'o'
+++ .                 print 'r'
----- - .             print 'l'
----- --- .           print 'd'
> + .                 print '!'
> .                   print '\n'
```

For readability, this code has been spread across many lines and blanks and comments have been added. Brainfuck ignores all characters except the eight commands so no special syntax for comments is needed (as long as the comments don't contain the command characters). The code without comments:

```
++++++++++[>+++++++>++++++++++>+++>+<<<<-]
>++.>+.+++++++..+++.>++.
<<+++++++++++++++.>.+++.------.--------.>+.>.
```

5.2.1 Explanation of the Hello World program

The program starts at the first command and also at the first memory location, which will be called a[0]. The next memory location will be called a[1], the one after that a[2], etc.

As you may recall, all memory cells have the value zero at the start of the program.

1. The first line sets a[0] to 10 by simply incrementing the contents of the cell ten times.
2. On the 2nd line the **[** character starts a loop. The loop would end immediately if the value of the current memory cell was zero. The value is ten, so the program continues.
3. The loop sets the values for the four cells:
    a. a[1] = 70 (close to 72, the ASCII code for the character 'H'),
    b. a[2] = 100 (close to 101 or 'e'),
    c. a[3] = 30 (close to 32, the code for space) and
    d. a[4] = 10 (newline).
4. Each pass of the loop, the following things happen:
    a. First the cursor is moved one place to the right (a[1]).
    b. Then the value of the current cell (a[1]) is incremented by seven.
    c. a[2] is incremented by ten
    d. a[3] is incremented by three and lastly
    e. a[4] is incremented by one.
5. The seventh line moves the cursor all the way back to the first position a[0] and then decreases the value of this cell by one. Thus the loop will be executed ten times[5]. After the loop is finished, a[0] is zero.
6. `>++.` then moves the pointer to a[1] which holds 70, adds two to it (72 is the ASCII character code of a capital H), and outputs it with the "." command.
7. The next line moves the array pointer to a[2] and adds one to it, producing 101, a lower-case 'e', which is then output.
8. Since 'l' is the seventh letter after 'e', to output 'll' another seven are added (+++++++) to a[2] and the result is output twice.
9. 'o' is the third letter after 'l', so a[2] is incremented three more times. Then the contents of the cell is written to the screen.
10. For the space and capital letters, different array cells are selected and incremented or decremented as needed.

5.4 Exercise

1. Modify the read-and-write program so it does not overwrite the characters that are written on the screen, but remembers them.
2. Modify the Hello World program to Hello World!
3. Modify the "Hello World"program to write your name on the screen.

6 Simple constructs in Brainfuck

6.1 Find a zero value

To find the first zero value to the right of the current position use the following code:
`[>]`

6.1.1 Explanation

As shown in chapter 3, `[]` loops until the memory cell contains a zero value the moment the program arrives at the [clause. So, starting from the current position, the cursor moves to the right with the command `>` until it arrives at a zero value. At that moment the loop stops.

6.1.2 Variant: seek other values

In order to seek the array for the first occurrence of the value '1' (keeping values inbetween intact), write: `-[+>-]+`
Seeking the value 2 is: `--[++>--]++`, and so on.

6.1.2.1 Explanation

Before the while loop is started, the value in the current cell is decreased by the amount you wish to look for. Then the while loop `[]` is entered. If the current value is zero, we apparently have found the correct value, and the loop is exited. Else the original value is restored and the cursor is moved one position right. This continues until the correct value is found.
After the exit, the original value of the memory cell is restored.

6.1.2.2 Note

If you search for non-zero values and none of the cells contain a zero, you will have created an infinite loop. So you need to make sure that the non-zero value is present somewhere.

6.2 Moving the content of one cell to another

In Brainfuck programming, cells are given a certain (functional) meaning in an algorithm. Because of this, it may be useful to move the value of one cell to another cell. A second reason for moving the contents of a cell can be that the cursor is at that location at the end of a while loop.
The command for moving the contents of a cell to its predecessor is: `[-<+>]`

1. Starting at the current cell, its value is decreased by one.
2. Then the cursor moves back one position.
3. Thirdly, the previous cell is increased by one.
4. Lastly, the cursor moves forward one position.

The loop will automatically end when the starting cell contains the value 0.

Of course, the same 'move' can be applied to a successor instead of a predecessor.

6.3 Adding one cell to another (a = a + b)

The "move" process listed in chapter 6.2 is destructive: in order to add the exact number to the target cell the value of the source cell is decreased to zero. The copy is therefore a two-step process involving an help variable.
The command for adding the contents of a cell to the value of its successor is:

```
[->>+<<]    (a2=a0 and a0=0)
>>[-<+<+>>] (a1=a1+a2; a0=a2;a2=0)
```

using the cell after the successor as temporary storage.
The first part of this command moves the contents of the starting cell to the cell two after that, just like in chapter 6.2. The second part moves the cursor two to the temporary storage and then adds the contents of this cell to the two preceding cells:
A[0] -> A[2]
A[2] -> A[0], A[1].
Which is the same as: A[0] -> A[0], A[1].

6.4 Copying the content of one cell to another (a=b)

If you wish to assign the value of one cell to another cell, this is basically the same as adding b to a, having first emptied a. This gives the following command, similar to that of chapter:

```
>[-]        {a1=0)
>[-]        (a2=0)
<<[->>+<<] (a2=a0; a0=0)
>>[-<+<+>>] (a1=a2; a0=a2; a2=0)
```

When starting a program you could skip the first two lines, since the Brainfuck implementation stipulates that all cells start at zero.

6.5 Substracting one cell from the other (a = a - b)

Subtraction is the same as addition, using only the opposite sign. The substraction code is therefore only one character different from the addition:

```
>>[-]<< [->>+<<] >>[-<-<+>>]
```

6.6 Multiplication (a = a * b)

Multiplication is repeated addition. We already know how to add, now this must be repeated.

We will need two temporary variables:

- one to store a which will be decreased on each run, and
- one to store b.

When the temporary copy of a is reduced to zero, the multiplication is complete.

X[0] is assigned to a,
X[1] is assigned to b,
X[2] is assigned to the temporary a

First clear the temporary cells[6]
`>>[-]>[-]<<`
Move a to X[2]: `[->>+<<]`
While the X[2] > 0, add b to a: `[<`
The inner loop first moves b to a and X[2]: `[+>>+<-]`
Then X[2] is moved back to b: `>[<+>-]`
End the loop by decrementing temporary a. `>-]`

Total code: `>>[-]>[-]<<[->>+<<][<[+>>+<-]>[<+>-]>-]`
This can be easily modified to multiply a = b * c. Note that this snippet does not give a value to a and b. Note also that the maximum result of the multiplication is equal to the maximum value of one cell (255 in the standard implementation).

6.7 If

Brainfuck does not have an 'if' command. This can be solved by using the while loop construction and only executing it once. There are multiple variants of the If problem. The Brainfuck code below has been retrieved from Esolangs wiki; the If-Then-Else code is attributed to Jeffrey Johnston.

6.7.1 If (x <> 0)

This is the easiest If construction. By nature, the while-loop will automatically run if the value of the current cell is not equal to zero. The only thing that need to be done are:

1. Copy x to a temporary value.
2. Start the while loop with the temporary value;
3. At the end of one run, make the temporary value equal to zero.

This code was taken from the Esolangs wiki (A[0] = x, A[1] = temp0, A[2] = temp1):

Make A[1] and A[2] equal to zero `>[-]>[-]`
Copy A[0] to A[2], using A[1] as temporary storage
`<<[>+>+<<-]>[<+>-]`
Execute the While loop if A[1] is not equal to zero. `>[code`
Make sure that the loop only runs once.
`A2[-]]`

6.7.2 If (x <> 1) and other values.

This is the same as in chapter 6.7.1, but A[1] is decreased 1 just before the while loop is entered. If A[1] - 1 = 0, then A[1] is 1 and so was the original value.
The "is not equal to" comparison can thus be extended to other values.

6.7.3 If (x=0) {code}

The If command is slightly harder if the code must only be executed if x=0. By default, the while loop is executed if x<>0. The solution is to use a help variable.

1. This help variable is intialiased at 1 (or any other value <> 0).
2. A while loop checks if x <> 0. If the while loop is executed, the help variable is zeroed.
3. A 2nd while loop will only run if the help variable <> 0.

If the help variable is not equal to zero, then the while loop was not executed, so x is not <> 0, or equal to zero.

6.7.3.1 Code

Initialize the temporary variable A[1], setting it to 1.

```
>[-]+
```

Check the value of x. If it is <>0, then A[1] is assigned 0. The loop automatically ends.

```
<[>-]
```

If A[1] is not 0, then the "then" part was not executed. Execute the "code" part. End the loop by making A[1] equal to 0.

```
<[code A[1]-]
```

6.7.4 If (x = 1) and other values (If-Then-Else)

If the value to be compared against is not equal to zero, some extra code is required to copy x to a temporary variable. In pseudocode it looks like this:

```
temp0[-]
temp1[-]
x[temp0+temp1+x-]temp0[x+temp0-]+
temp1[temp0-temp1[-]]
temp0[
code
temp0-]
```

Initialize A[1] and A[2] to 0:

```
>[-]
>[-]
```

Copy X to A[2]. Assign 1 to A[1].

```
<<[>+>+<<-]]
```

```
<[<+>-]+
```
Reduce A[2] by the value to be compared against, in this example 1:
```
>-
```
If A[2] is not equal to zero, make A[1] equal to 0. Move the cursor one place right and make A[2] equal to zero, thus ending the loop.
```
<[<->[-]]
```
If A[1] is not 0, the "then" part was not executed. Execute the "code" part. End the loop by making A[1] equal to 0.
```
<[code A[1]-]
```

6.7.4.1 Code

```
>[-]>[-] <<[>+>+<<-]]<[<+>-]+>-
<[<->[-]] <[code A[1]-]
```

6.7.5 If-Then-Else

The If-Then-Else clause is is the general version of `if (x=1)`. The following four steps are required:

1. Set a temporary variable to 1.
2. In the If-Then clause, make the variable 0 if the Then part is executed.
3. If the temporary variable is still 1, the Then was not executed, so the Else is executed.
4. Set the temporary variable 0 at the end of the Else clause.

6.8 Comparison

These bits of code have been retrieved from the Esolang website. They can be used in for example IF-statements.

6.8.1 x = (x == y)

This was created by Jeffry Johnston. The outcome of this comparison is 0 if the comparison is False and 1 if it is True.

```
temp0[-]
temp1[-]
x[temp1+x-]+
y[temp1-temp0+y-]
temp0[y+temp0-]
temp1[x-temp1[-]]
```

6.8.2 x = x<>y

This was also submitted by Jeffry Johnston. The outcome of this comparison is 0 if the comparison is False and 1 if it is True.

temp0[-]

```
temp1[-]
x[temp1+x-]
y[temp1-temp0+y-]
temp0[y+temp0-]
temp1[x+temp1[-]]
```

6.8.3 x=x<y

This was submitted by Ian Kelly. The outcome of this comparison is 0 if the comparison is False and 1 if it is True. Of course, if (x<y) is False, then (x>=y) is True, so this code can be used in two ways.

```
temp0[-]
temp1[-] >[-]+ >[-] <<
y[temp0+ temp1+ y-]
temp0[y+ temp0-]
x[temp0+ x-]+
temp1[>-]> [< x- temp0[-] temp1>->]<+<
temp0[temp1- [>-]> [< x- temp0[-]+ temp1>->]<+< temp0-]
```

6.8.4 x=x<=y

This was submitted by Ian Kelly. The outcome of this comparison is 0 if the comparison is False and 1 if it is True. Of course, if (x<y=) is False, then (x>y) is True, so this code can be used in two ways.

```
temp0[-]
temp1[-] >[-]+ >[-] <<
y[temp0+ temp1+ y-]
temp1[y+ temp1-]
x[temp1+ x-]
temp1[>-]> [< x+ temp0[-] temp1>->]<+<
temp0[temp1- [>-]> [< x+ temp0[-]+ temp1>->]<+< temp0-]
```

7 Questions and answers

7.1 What is an esoteric programming language?

An esoteric programming language is (according to Wikipedia) "a programming language designed as a test of the boundaries of computer programming language design, as a proof of concept, or as a joke. There is usually no intention of the language being adopted for mainstream programming."

7.1.1 Is Brainfuck an esoteric programming language?

Certainly. The odds of a game or real-life application or game stating "Software requirements: a Brainfuck compiler" are practically zero. Any Brainfuck program can be converted statement-by-statement in an equivalent C program. Or, as Wikipedia puts it : "As the name suggests, brainfuck programs tend to be difficult to comprehend. This is partly because any mildly complex task requires a long sequence of commands; partly it is because the program's text gives no direct indications of the program's state. These, as well as brainfuck's inefficiency and its limited input/output capabilities, are some of the reasons it is not used for serious programming."

7.2 Is this the smallest programming language possible?

7.2.1 Yes…. but

As far as I can see, yes: this is the smallest programming language possible. It contains the four basic functions:

1. Assignment (+ and -)
2. I/O (, and .)
3. Decision-taking ([and]) and
4. Moving up and down the memory (< and >).

There have been smaller implementations (you may even say there is a special subbranch of esotrics devoted to Brainfuck derivatives), but all of them either:

1. Have severely limited functionality (ie: they require the input to be present in memory before starting), or
2. Operate on a restricted subset of values (Like Bitfuck, that only works on bits), or
3. Are rather unwieldy.

But:

7.2.2 No…. (you may read these subchapters in any order)

There have been efforts by Brainfuck enthousiats to create even smaller languages. Some of these have been documented on Esowiki. An incomplete set of examples:

1. The "Boolfuck" languages is just like Brainfuck, but the memory cells are not bytes but bits. This reduces the assignment operands from two to one: "flipping" the bit is enough (this is the * command). The Bitfuck program can simulate Brainfuck (one byte is eight bits).
2. If you use the Bitfuck option, you can combine the assignment and one of the two "move" instructions. If you define (as <*[7], the total number of instructions is reduced to six. In order to move left and leave the original bit alone, replace < with (>(.
3. One could omit the input- and output if you stipulate that the starting configuration must be put into memory in advance and the memory is the output screen at the same time[8].

This leaves you with four instructions and the same functionality:

1. [for the opening of the WHILE loop;
2.] for the closing of the WHILE loop;
3. (for moving one space to the left and flipping the new bit at the same time;
4. > for moving one space to the right.

There has been a paper written about the URISC: (Ultimate Reduced Instruction Set Computer), that had only one instruction! The one instruction had three parameters: the first parameter was substracted from the second parameter and then the pointer moved to the address in the third parameter if the result was negative. Otherwise, the pointer moved to the next address.

But…

7.3 How do you learn Brainfuck?

Like any other programming language: by reading, learning and practicing. You have already started reading if you arrive at this point. Continue finding resources that teach you about Brainfuck and increase your knowledge. Look at the sample programs here and on the net. Then practice by creating small programs and later bigger programs.

Expert advice is given by Daniel Cristofani on his website:
http://www.hevanet.com/cristofd/brainfuck/intermediate.html

8 Simple Brainfuck programs

8.1 Make a lowercase letter uppercase[9]

8.1.1 Description

This small piece will request the user to enter a key. If the key entered is <Enter> (ASCII 10), the loop does not start. Otherwise, the program will substract 32 from the value and outputs the result. The user can again enter a key. Note that this is an example of an if-then.

8.1.2 Code

```
,----------[----------------------.,----------]
```

8.2 Fibonacci numbers

8.2.1 Description

This program prints *n* (which can be changed by updating the code) Fibonacci numbers. For more about the Fibonacci sequence see Wikipedia. It was written and commented by Max Grosse. The program calculates the first x fibonacci numbers, where x is the number of + signs on the third line, in this case ten.
Note that the output is in ASCII code, which means that it will not be readable for the user. The cells contain Fibonacci value.
In chapter 9 a more complex Fibonaci code is given, which outputs readable numbers.

8.2.2 Code

```
+.>+.
>>>
+++++ +++++
[<<<[->>+<<]>>[-<+<+>>]<<<[->+<]>>[-<<+>>]<.>>>-]
```

8.2.3 Explanation

In this explanation, the code has been split in small pieces.
The following memory cells are used by the program:

- A[0] contains one of the Fibonacci values;
- A[1] contains the other Fibonacci value.
- A[2] is a memory cell for temporary storage
- A[3] is a memory cell for temporary storage
- A[4] contains the number of Fibonacci sequences to be generated.

```
+.>+.
```

1. This first line makes A[0] = 1 and A[1] = 1 and writes both. (Note that they will be printed as Ascii 001, which is not a readable character). A[0] and A[1] thus contain the first two characters of the Fibonacci sequence.

```
>>>
+++++ +++++
```

1. The 2nd and 3rd line place the total number of Fibonacci numbers to be generated in A[4]. In this case ten Fibonacci numbers are generated.

```
[<<<
```

1. The main loop runs as long as A[4] contains a value greater than zero. The cursor moves back three places to A[1].

```
[->>+<<]>>
```

1. Moves A[1] to A[3] by decreasing A[1], and increasing A[3]. The loop ends if A[1] is zero. The cursor is then placed at A[3].

```
[-<+<+>>]
```

Move A[3] to A[1] and A[2] in the same way as just above. The net effect of this and the previous step is that the value of A[1] is copied to A[2]. The loop ends at position A[3], when that cell has value 0.

```
<<<[->+<]>>
```

1. The cursor moves back to A[0] and then adds the value of A[0] to A[1]. This is done by decreasing A[0] one step at the time and adding the same amount to A[1]. When the move is completed, the cursor moves to A[2].

```
[-<<+>>]
```

1. In the same way as in step 5, A[2] is now moved to A[0]. The loop ends at position A[2], when that cell has value 0.

```
<.
```

1. A[1] now contains the sum of the previous two values, which is exactly how Fibonacci's numbers work. The cursor thus moves to A[1] and writes the (ASCII) value.

```
>>>-
]
```

1. Lastly, the loop counter in A[4] is decreased. The loop now ends.

8.3 Sorting

8.3.1 Description

This sorting program was made (and commented) by Daniel Cristofani for a Brainfuck programming contest. The program takes input form the user (ending when the user enters zero) and then outputs the numbers in ascending order on the screen.
To output the numbers in ascending order in memory cells is left as an excersice to the reader.

8.3.2 Code

```
>>,[>>,]<<
[[-<+<]>[>[>>]<[.[-]<[[>>+<<-]<]>>]>]<<]
```

8.3.3 Explanation

The following memory cells are used:

- A[0] and A[1] initially contain zero.
- A[2], A[5] and every third cell thereafter contains the input by user.
- Any intermediate cell contains zero.

Code

```
>>,[>>,]<<
```

1. This inputs numbers (0-255) from the user and creates an array of the following layout:

   ```
   0 0 a 0 b 0 c …. 0 z (0 0)
   ```

   Starting with two zeroes, the user input is alternated with the number zero. This number will be used as a reference number to determine the value the sorting program is comparing against. After the user ends the input (by ending the input with Ascii 00) the cursor is moved two places back. So, the cursor is at the position I've called z.

```
[
[-<+<]>
```

1. The main loop runs while there are still numbers to be processed. The inner loop then starts at the end and works backwards, decreasing all numbers "a" to "z" inclusive and at the same time increasing the "0" values that were alternated with it. Because we started the row with two zeroes, the loop will end. To finish this part, the cursor is moved one spot to the right, so it is one position before 'a' (or memory cell a[1]).

```
[>[>>]<
```

1. The code enter a new loop. It starts by moving to position 'a', and then seeks the

first value zero (which may be anywhere from position 'a' to 'z' inclusive - if the 'a' value is zero the loop immediately ends). If it is found, the loop is exited and the cursor moves one place to the left.
If not, the loop ends at the rightmost part of the array, where it ended with 0 0.
Note the double >> characters: each reference number is skipped.

```
[.[-]<
```

1. If there was a zero in the array, the cell just left of it contains the starting value of the cell in the form of the reference value. These reference cells were increased once for every time the 'real' numbers in the array were lowered. Now this value is written in the output, the cell is emptied (it will be overwritten by other values later) and the cursor moves again one place left. It is now on a previous value (if present), or on a zero of it was at the leftmost place.

```
[[>>+<<-]<]>>
```

1. This loop takes all cells left of the value we just output and moves it two positions to the right, overwriting both the value itself and the reference cell. This loop ends at the leftmost position where a 'reference' number is found.

```
]>
```

1. Ends the loop from step 4 and moves one place right, so it is at the next number to be inspected. If this is also zero, then two cells need to be printed.

```
]<<
```

1. Ends the loop from step 3. Moves to the previous value to be checked.

```
]
```

1. Ends the main loop.

8.3.4 Example of the program execution

The user inputs the following values: 5, 3, 2 and 6.
This is converted to: 0 0 5 0 3 0 2 0 6 0 0 .
The cursor is at the value 6.
The first cycle of the main loop: 0 1 **4** 1 **2** 1 **1** 1 **5** 0 0
This loop ends because after the last "+<", the while encouters a zero value.
In step 3 the cursor is first at the "4", and then "2", "1" and lastly "5", before ending the loop at the rightmost zero. The cursor moves one space to the left, where the value is still zero.
Step 4 does not run this time: the while is entered when the value at the cursor is zero.
Step six and seven move the cursor back to the last number, which is currently 5.

The second run: 0 2 **3** 2 **1** 2 **0** 2 **4** 0 0
This time in step 3 the cursor lands at a 0: the first number to be printed is found. Step 4

prints the number to the left and zeroes this: 0 2 3 2 1 **0** 0 2 4 0 0
Now step 5 moves all values left of the printed cell to the right.

0 0 0 **2** 3 2 1 2 4 0 0

Step 6 checks for other cells that just became zero. If none are found, the cursor ends at the far right of the memory.

The third run: 0 0 0 3 **2** 3 **0** 3 **3** 0 0
Again, one zero value is found, so in the original input row the value three was present. This value is now output (step 4 again). The left values are again moved so a contigious row of nonzero values is present: 0 0 0 0 0 3 2 3 3 0 0

The fourth run: 0 0 0 0 0 4 1 4 2 0 0
No zeroes are found, so the while loop in step 3 runs all the way to the right part of the screen.
The fifth run: 0 0 0 0 0 5 0 5 1 0 0
One zero is found, so '5' is output. The "move" loop will end right away, because there are no more values to be moved to the right.

The sixth and last run outputs six.

8.4 Brainfuck text generator

8.4.1 Description

This program was also retrived from the website of Daniel Cristofani. If you input some text, it will output Brainfuck code which (when used as program code) will output the same text.
This bit of code saves you time when writing your own program and you wish to output text. Note that the BF code itself can later be optimized.

8.4.2 Code

```
+++++[>++++++++++<-],[[>--.++>+<<-]>+.->[<.>-]<<,]
```

8.4.3 Explanation

```
+++++
```

A[0] = 5

```
[>++++++++++<-]
```

Set A[1] to 50, by adding ten to A[1] for every time that A[0] is decreased. In the process, A[0] is cleared.

```
,
```

Requests the user to enter a value.

8.5 Squares

```
++++[>++++++<-]>[<++++++>-]+<+[
   >[>+>+<<-]++>>[<<+>>-]>>>[-]++>[-]+
   >>>+[[-]++++++>>>]<<<[[<++++++++<++>>-]+<.<[>----<-]<]
   <<[>>>>>[>>>[-]+++++++++<[>-<-]+++++++++>[-[<->-]+[<<<]]<[>+<-]>]<<-]<<-
]
```

9 Complex programs

9.1 Powers of two

9.1.1 Description

This program was retrieved from Daniel Cristofani's website. The program will output powers of two (2-4-8-16-….) in decimal format until aborted.

The code starts with the initialisation of the Newline character (ASCII 10) a counter and the initial value. The beginning of the main loop prints the digits on-screen. Memory is set up in the following format:

```
0 10 c d c d c d c d c d
```

where c is a counter and temporary storage and d holds the digits (least significant left, most significant right) of the number generated.

Each digit is multiplies by two and if the original number was 6 or 8 the carry-over is moved right.

9.1.2 Code (with brief explanation)

```
/* Initialisation* Includes the value for 2^0 = 1 and the ASCII code for Newline*/
>++++++++++>>+<+


[
/* Write values on screen /*
    [+++++[>+++++++++<-]
    >.
    <++++++[>--------<-]
   /*Check if there are less significant digits to display; If so display them using
    the same procedure* Repeat until A(0) is reached which contains 0*/
    +<<]
/* Write Newline */
    >.>
   /*Decrease the digit by one before starting the multiplication*/
```

```
    [->
       /* Multiply A(3) by 2; storing the result in A(2) */
       /* First round */
       [<++>-
       /*Second round*/
           [<++>-
              /*Third round*/
              [<++>-
                 /*Fourth round*/
                  [<++>-
                     /*Move 8 A(2) to 2 A(4) and 4(A3) and then*/
                     [<--------->>[-]++
                     /*Move 4 A(3) to 6 A(2) by decreasing A(3) before starting the
                     duplication loop */
                            <-[<++>-]]
                  ]
              ]
       ]
       ]<
       /*Add A(2) to A(3); A(4) to A(5) etc */
       [>+<-]
       /*A(2)=1; A(4)=1; etc */
       +>>]
    <<
]
```

9.1.3 Explanation

The program starts with initializing three variables:

1 A[1] = 10 (Newline)
2 A[2] = 1 (This variable is used as loop counter)
3 A[3] = 1 (This variable contains the digit(s) that are written on the screen.

The "main" while loop starts.

- First the contents of A[3] are increased by 48[10] so the cell contains a value that is readable on the screen. If the number contains multiple digits, each digit will be separately written in A[3] and then put on the screen. Having written the value, 48 is subtracted from A[3]. Finally, A[2] is reset to 1. The 'write' loop ends if A[0] contains the value 0.
- Having written the value, A[2] is lowered by 1. Then four nested multiplication loops are entered, each doubling the value of A[3] and storing the result in A[2], as long as A[3] is greater then zero. The first time the loop ends after one go, so A[2] contains the value 2.
- At the end, A[2] is moved to A[3] and A[2] is reset to 1.

The loop is now complete: Where the last loop started with A[3]=20, the new loop will start with A[3] = 21.

- The 2nd round the value 2 and the newline are printed. Two multiplication loops are executed, so A(2) =4 (22). A(2) is again moved to A(3).
- The third round 4 is printed and four of the multiplication loops are executed. A(2) ends up with value 8.
- The fourth round 8 is printed. When four of the multiplication rounds have been executed, A(2) is 8, and A(3) is 4. Now the innermost loop replaces 8 by 6 in A(2) (since 16 2*8, and the least significant digit of 16 is 6), and adds two to the new memory space A(4). This is later decreased to 1, moved one space to the right: A(5). A(4) is set to 1 by the same code that also reset A(2). The memory now looks like this:
  0 10 1 **6** 1 **1** 0
- The fifth round the cursor starts at A(4) and A(5), where the most significant digit is displayed. These two fields play the same role that A(2) and A(3) play with the least significant digit. The loop thus neatly displays "16" and a Newline.
  Four rounds of multiplying A(3) to A(2) leaves the value "2" in A(3) (2 * 16 = 3**2**). Because four rounds of multiplication have been executed and there is stil a value left in A(3), the value 8 in A(2) is converted into 2 in A(4) and 2 in A(2). A(3) is cleared.
  At the end of the round the value 2 in A(2) and A(4) is added to A(3) and A(5) respectively. Memory now looks like this:
  0 10 1 **2** 1 **3** 0
- 32 is displayed and multiplied to 64 in the sixth round.
- In the seventh round, the 4 (LSD[11] of 64) is converted to 8. The 6 (MSD) in A(3) becomes 2 in A(5) and a new memory cell A(6) is created. The value 1 in A(6) lastly is moved to A(7). The memory now looks like this:
  0 10 1 **8** 1 **2** 1 **1** 0

The program continues forever. Every time the most significant digit is 6 or 8, the part that doubles the value creates a new cell. Otherwise, the digits are simply doubled.

9.1.4 Exercise

Modify the program in such a way that it displays the powers of four.[i]

9.2 Factorials

9.2.1 Description

This program will print (in decimal values) factorials (n!) for each n starting with 0. This program was also retrieved from the Daniel Cristofani's website.
The program runs until aborted by the user.

9.2.2 Code

```
Initialisation
```

```
>++++++++++      A(1)=10
>>>+             A(4)=1
>+               A(5)=1
                 A(8)=1
[>>>+
   [-
      [<<<<<[+<<<<<]>>[[-]>[<<+>+>-]<[>+<-]<[>+<-[>+<-[>
+<-[>+<-[>+<-[>+<-[>+<-[>+<-[>+<-[>[-]>>>>+>+<<<<<<-[>+<-]]]]]]
]]]]]>[<+>-
]+>>>>>]<<<<<[<<<<<]>>>>>>>[>>>>>]++[-<<<<<]>>>>>>-]+>>>>>]<[>+
+<-]<<<<[<[
>+<-]<<<<]>>[->[-]++++++[<+++++++++>-]>>>>]<<<<<[<[>+>+<<-]>.<<<
<<]>.>>>>]
```

9.2.3 Explanation

The format at the start of the main loop is normally

```
0 10 0 0 n t o c b n t o c b...
```

where b is the number whose factorial was output last, stored in binary, least significant digit to the left, using the values 2 and 1 in place of 1 and 0; c is where that number will be copied to, only using 1 and 0; o holds decimal digits of the last factorial output, t are temps that are nonzero as far as n has been set, and n is where the next factorial will be accumulated.

9.3 Fibonacci generator (2)

9.3.1 Description

This program was also retrieved from Daniel Cristofani's website. The program will output Fibonacci numbers until it is aborted. Note the similarity in setup with the previous two items.

9.3.2 Code

```
>++++++++++
>+
>+
[
  [+++++[>+++++++++<-]>.<++++++[>--------<-]+<<<]>.>>[
    [-]<[>+<-]>>[<<+>+>-]<[>+<-[>+<-[>+<-[>+<-[>+<-[>+<-
      [>+<-[>+<-[>+<-[>[-]>+>+<<<-[>+<-]]]]]]]]]]+>>>
    ]<<<
]
```

9.3.3 Explanation

The format is
0 10 a t b a t b ... a t b 0 0 0 0 0 ...
where a and b are (decimal) digits of the two Fibonacci numbers most recently calculated, stored with the most significant digit to the right, and t are temporaries which are nonzero when not in use, to mark how far b extends to the right. The 10 is just used to output a linefeed.

9.4 PI Calculator

9.4.1 Description

This program was written by Felix Nawothnig. According to the author:

- using 8 bit cells causes an overflow after 4 digits
- using 16 bit cells causes an overflow after 537 digits
- using 32 bit cells causes an overflow after several millions of digits

9.4.2 Code

```
>  +++++ +++++ +++++ (15 digits)

[<+>>>>>>>>>++++++++++<<<<<<<-]>+++++[<+++++++++>-]+>>>>>>+[<<++
+[>>[-<]<[>]<-]>>
[>+>]<[<]>]>[[->>>>+<<<<]>>>+++>-]<[<<<<]<<<<<<<<+[->>>>>>>>>>>>
>[<+[->>>>+<<<<]>
>>>>]<<<<[>>>>>[<<<<+>>>>-]<<<<<-[<<++++++++++>>-]>>>[<<[<+<<+>
>>-]<[>+<-]<++<<+
>>>>>>-]<<[-]<<-<[->>+<-[>>>]>[[<+>-]>+>>]<<<<<]>[-]>+<<<-[>>+<
<-]<]<<<<+>>>>>>>
>[-]>[<<<+>>>-]<<++++++++++<[->>+<-[>>>]>[[<+>-]>+>>]<<<<<]>[-]
>+>[<<+<+>>>-]<<<
<+<+>>[-[-[-[-[-[-[-[-<->[-<+<->>]]]]]]]]]]<[+++++[<<<++++++++
+<++++++++>>>>-]<
<<<+<->>>>[>+<<<++++++++++<->>>-]<<<<<[>>+<<-]+<[->-<]>[>>.<<<<[
+.[-]]>>-]>
[>>.<<-]>[-]>[-]>>>
[>>[<<<<<<<<<+>>>>>>>>>-]<<-]]>>[-]<<<[-]<<<<<<<<<]
++++++++++.
```

9.5 Golf Contest #3

9.5.1 Description

The contest was: Write a Brainfuck program that accepts the language L. The language

accepts the set of all input strings consisting of a sequence of a's, followed by a sequence of b's, followed by a sequence of c's, having the same number of a's, b's, and c's. All other strings are rejected.

9.5.2 Code

```
+>+>>+[
    ,-[>++++<[[->]<<]<[>]>]>[                  mod 5 cutdown
        -[<<<+>+>+>-]<<<-[-[-[<[-]>-]]]+++>[<->-] compare and prev=current
        >[-[>]+>]>>>[>>>]+[<<<]<[-<]>+<       increment appropriate stack
    ]>
]+<+<+[>>+>]<<[>+>>]<<[<<<]+<<[>>+>]>>[>>>]    fuse stacks
>+++++[<++++>>++<-]<-[<<++++++>++++>-]<++<<   setup for print
-[-[-[>>.++..++.<--.++>>]]]                    print "accep"
>[.>++++.+++++.-----.--.++>>]                  print "rejec"
<<<++.>.-.>>>.                                 print "ted\n"
```

9.5.3 Explanation

"My basic method was to keep count of the number of 'a', 'b', and 'c' using three interspersed stacks of cells set to 1. Also a flag which starts with a value of 1, but is zorched if any input letter has a smaller ASCII value than the preceding one. This continues until a linefeed is read. I tried a variety of methods for cutting the ASCII values down to size, and for comparing each number with the previous one, but I didn't try all possible combinations and it didn't seem to change the total outcome by more than 15 bytes or so anyway.

The method used in this submission uses a distorted mod-5 operator to cut the ASCII values down, resulting in a value of a=2, b=1, c=0; whereas the previous value is stored in the form a=1, b=2, c=3. Thus, the previous letter had a larger ASCII value if (prev+current) > 3; this is a mildly counterintuitive comparison operation. The overall data layout is
flag prev x x x 0 c b a c b a c b a c b a ... where "x" are cells used during the process of reading the current value and comparing it with the previous.

Once the linefeed is read, my original plan was to merge all three stacks into a single stack, and merge the flag with them; if the last cell of the combined stack has a value of 4, the string is accepted; otherwise, it is rejected. I ended up using a slight variant of that method, as it happens."

9.6 99 bottles of beer

9.6.1 Description

This programming exercise is about looping and based on the marching song of the same name:

> *99 bottles of beer on the wall*
> *99 bottles of beer*

You take one down, you pass it around
98 bottles of beer on the wall

And so on, until there are no bottles left. A program that just prints the above would earn a “D-“ (unless you forgot to put in the loop and just printed the statements, which would earn you an F). More sophisticated solutions include:

- The fact that “one bottle” does not end in “s”.
- If there are 0 bottles left, this should be printed as “no more bottles”
- Some versions even write the numbers instead of just the numbers and
- Lastly, some versions have you “go to the store” and fetch new beer.

There are multiple versions of the Beer song in Brainfuck. This one was created by Aki Rossi (at iki .fi)

9.6.2 Code including explanation

```
#
# Set beer counter to 99
#
>>>>>>>>>
>++++++++++[-<++++++++++>]<-
<<<<<<<<<

#
# Create output registers
#
++++++++++[->++++>++++>++++>++++<<<<]       add 0x28 to all from (1) to (4)
++++++++[->>>++++++++>++++++++<<<<]add 0x40 to all from (3) and (4)
++++[->>>>++++<<<<]                         add 0x10 to (4)
++++++++++                          set (0) to LF
>--------                           set (1) to SP
>++++                               set (2) to comma

>>>>>>>                             go to beer counter (9)
[
        # Verse init
        <<<<
        +++     state 1 in (5)
        >+      state 2 in (6)
        >++     state 3 in (7)
        <<      go to (5)
        [
                ######################
                # N bottles of beer #
                ######################
                >>>>            go to (9)
                [
```

```
        # Print the number in (9)
        # (conversion routine uncommented to save space)
        [->+>+<<]>>[-<<+>>]<[>++++++++++[->>+>+<<<]
        <[>>>[-<<<-[>]>>>>[<[>]>[---------->>]<++++
        ++[-<+++++++++>]<<[<->[->-<]]>->>>[>]+[<]<<[
        ->>>[>]<+[<]<<]<>>]<<]<+>>[->+<<+>]>[-<+>]<
        <<<<]>>[-<<+>>]<<]>[-]>>>>>>[>]<[.[-]<]
        <<<<<<
        # and remain in (10) which is empty
]>+<<   inc (11) and go to (9)
[>]>>   if (9) empty go to (11) else (12)
[
        <<<<<<<                 go to (3)
        ++++++.+.       no
        -------         reset (3)
        >>>>>>>                 go to (11)
        >>              go to (12)
]
<[-]<[-]<               empty (11) and go to (9)
<<<<<<<<.       SP
>>------. b
++++++++++++.       o
>----..         tt
<---.           l
-------.  e
>>>>>>->>>+<<<      dec (9) inc (11)
[>]>>
[       now in (12)
        <<<<<<<<    go to (4)
        -.+             s
        >>>>>>>                 go to (11)
]
>-<<<+<<<<<
<<<.            SP
>>>-----.       o
<+.             f
<<.             SP
>>----.         b
+++..           ee
>+++.           r
<+++>++++++  reset registers

>>              go to (6)

###############
# on the wall #
###############
[
```

```
        <<<<<.          SP
        >>+++++++.-.  on
        <<.             SP
        >>>----. t
        <------.---.    he
        <<.             SP
        >>>+++.                 w
        <----.          a
        ++++++++++..    ll
        ---->+>>        reset and go to (6)
        -               dec (6)
]

#
# comma LF
#
<<<<.
<<.

#####################################
# take one down and pass it around #
#####################################
>>>>>>>             go to (7)
-               dec (7)
[>]>>           if not blank then skip loop
[
        -               dec (9)
        <<<<<           go to (4)
        ----.           t
        <-------.  a
        >---------.     k
        <++++.          e
        <<.             SP
        >>>++++.-.<.  one
        <<.             SP
        >>-.>+.         do
        ++++++++.       w
        ---------.  n
        <<<.            SP
        >>---.>.<+++.  and
        <<.             SP
        >>>++.<---.     pa
        >+++..          ss
        <<<.            SP
        >>++++++++.>+.      it
        <<<.            SP
        >>--------.     a
        >--.---.   ro
```

```
                ++++++.                 u
                -------.  n
                <+++.           d
                ++++>++++++++++    reset registers
                <<.             comma
                <<.             LF
                >>----------    set (2) to excl mark
                >>>>+           inc (6)
                >>>>            go to (10)
            ]
            <<<
            <<              go to (5)
            -               dec (5)
        ]
        >>+             inc (7)
        <<<<<<<.        LF
        >>++++++++++    reset comma
        >>>>>>>         go to beer counter (9)
]
```

Appendix: Sources

Documents about Brainfuck

- http://esolangs.org/wiki/ Brainfuck
  The "Esoteric languages" entry on Brainfuck.
- http://en.wikipedia.org/wiki/ Brainfuck
  The English Wikipedia entry on Brainfuck.
- http://www.lordalcol.com/blog/2008/10/11/ Brainfuck-tutorial
  A tutorial by Nieko Maatjes.
- http://www.nieko.net/projects/brainfuck
- http://cydathria.com/bf/ Brainfuck.html Another tutorial

Sources for BF programs

- http://svn.deepdarc.com/code/bftools/trunk/examples/
- http://www.hevanet.com/cristofd/ Brainfuck/
  progams by Daniel Cristofani. Warning: some programs are very complex.
- http://esoteric.sange.fi/brainfuck/bf-source/prog/

Interpreters.

http://www. Brainfuck.tk	An online interpreter.
http://www.iwriteiam.nl/Ha_BF.html	List of BF resources.
http://kuashio.blogspot.nl/2011/08/visual-brainfuck.html	Visual BF interpreter

---

[1] Retrieved on July 28th, 2011 from the English Wikipedia.
[2] For more about P'', see http://en.wikipedia.org/wiki/P%E2%80%B2%E2%80%B2
[3] Some implementation use an array of cells that do not contain bytes (0-255), but words (0-65535) or Long words (32 bits).
[4] In the standard implementation, Adding one to a cell that contains 255 results in 0.
[5] Like a for-loop.
[6] This may be omitted.
[7] The * is the 'flip' operand from the previous point.
[8] The Commodore-64 computer acutally used this: memory locations 1024-2023 were the screen memory. Anything written in these locations was automatically written on the appropriate place on the screen.
[9] Retrieved from the English Wikipedia
[10] A[2] starts at six and counts down. For each loop of A[2], A[3] is increased by 8
[11] Least Significant Digit

[i] Solution: Build in an IF statement that only displays the value if the last digit is 1, four or six; or only print every other value of the power-2 loop.