07.231 Introduction to Software Engineering

—Assignments—

Christian Collberg, Hans Guesgen, Robert 'Buz' Uzgalis

The University of Auckland
Computer Science Department

1994

Rules

Welcome to the 07.231 assignments of 1994!

The assignments make up 50% of your grade. In order to pass the course, you have to pass both the assignments and the test.

When you look through the assignments, you will see that they are worth a total of 1400 points. However, the maximum number of points you can get is 800 points.

This means that you can choose assignments that suit you. For example, you might select two 300-point assignments and a 200-point assignment. Or you might start with a 100-point assignment, continue with two 200-point assignments, and finish with a 300-point assignments. It's up to you.

You can turn in assignments at any time, and as many as you want; however, observe the following deadlines:

Thursday, 24 March 1994
Thursday, 21 April 1994
Thursday, 9 June 1994

Hand in at least one assignment for each deadline! For every deadline you miss, the maximal number of points you can get will decrease by 100.

There will be face-to-face marking during the lecturing weeks. Markers will be available in the lab to mark assignments. Experience has shown, the number of students handing in their assignments increases dramatically as deadlines approach. So if you want to avoid standing in line for hours and talking to markers that have little time for you, don't wait until deadlines are close.

Christian Collberg
Hans Guesgen
Robert 'Buz' Uzgalis

Contents

The University of Auckland
Computer Science Department

07.231 Introduction to Software Engineering 1994
Christian Collberg, Hans Guesgen, Robert 'Buz' Uzgalis

Assignment 1: ADT BitStream (100 points)

1.1 Introduction

Pascal has very limited support for file manipulation. There is, for example, no way to read or write a file one bit at a time. Our task for this assignment will therefore be to "extend" Pascal with an abstract data type `bitstream` which provides functions for reading and writing bits.

1.2 The `bitstream` Interface

The following Pascal module suggests a possible interface to the `bitstream` ADT. Note, however, that additional functions may be required in order to determine the end of the stream of bits, and to convey possible error conditions on to the client module.

```
unit bitstream
interface

type
   bit     = ...;
   bitfile = ...;

function OpenBitFile (
   filename : string): bitfile;

function CreateBitFile (
   filename : string) : bitfile;

procedure CloseBitFile (
   f : bitfile);

function ReadBit (
   f : bitfile) : bit;

procedure WriteBit (
   f : bitfile;
   b : bit);

implementation
   .....
```

1.3 Client Programs

Write two client programs which make use of the `bitstream` package:

countbits Prompt the user for a file name and report back the number of 1's and 0's in the file.

printbits Prompt the user for a file name and print the bits in the file, 80 bits per line.

The University of Auckland
Computer Science Department

07.231 Introduction to Software Engineering 1994
Christian Collberg, Hans Guesgen, Robert 'Buz' Uzgalis

Assignment 2: Order Analysis (100 points)

2.1 Problem

Sometimes the arithmetic operations provided by a programming language are insufficient. For example, it may be necessary to have operations like addition, multiplication, to the power of, etc. on very large numbers. In this assignment, we will assume that a data type `Bignum` is given together with a function `Add` for adding two numbers of type `Bignum` and a Boolean function `GreaterThan` for comparing two numbers of this type. Your task will be to implement some small programs that use `Add` and `GreaterThan` for realizing more complex mathematical functions. You will then run your programs on some test data and measure the runtime. From your measurements you are to derive an order analysis of the programs.

2.2 Programs

All programs should use the data type `Bignum`, which, for the sake of simplicity, we will define here as follows:

```
type Bignum = integer;
```

(You can easily imagine this type definition to be replaced at a later stage by a definition that uses some more sophisticated data structure.)

Assuming the above definition, `Add` and `GreaterThan` can be implemented in the following way:

```
function Add(m : Bignum, n : Bignum) : Bignum;
   begin
      Add = m + n;
   end;

function GreaterThan(m : Bignum, n : Bignum) : boolean;
   begin
      if m > n then
         GreaterThan = true;
      else
         GreaterThan = false;
   end;
```

(Again, these definitions could be replaced at a later stage by definitions based on some more sophisticated data structure.)

The programs you are going to write should use the above definitions. Since most of the programs will use parts of the programs developed at a previous stage, you should implement in a modular way. This enables you to "recycle" most of the programs.

2.2.1 Multiplication

This should be an easy one. Let us assume m and n are two positive numbers of type `Bignum`. Then write a program (regarding the above restriction) which multiplies m with n.

2.2.2 Divided by Two

Let n be a positive number of type `Bignum`. Implement a program that divides n by two. If n is an odd number, the program should output the smallest integer that is greater than $n/2$.

2.2.3 To the Power of Itself

Let n be again a positive number of type `Bignum`. Use parts of the program of section 2.2.1 to write a program that calculates n to the power of n, i.e., n^n.

2.2.4 Two to the Power of

This program realizes a similar function as the one of section 2.2.3. But instead of n^n, this time we will calculate 2^n for a positive number n of type `Bignum`. The program performing this task should do the following: If $n = 1$, then return the value 2 and terminate. Otherwise, test if n is an even or an odd number. If n is an even number, make a recursive call to the function with $n/2$ as argument and multiply the result of the recursive call with itself. If n is an odd number, make a recursive call to the function with $n - 1$ as argument and multiply the result of the recursive call with 2.

2.3 Input

All programs except the one realizing multiplication require one positive number of type `Bignum` as input. The multiplication program requires two numbers as input, but, when looking at your implementation, you will find that only the second number is significant for the runtime of the program. Thus we can assume here without loss of generality that the first number is fixed. So let us assume that m is equal to 42 when testing the multiplication program.[1]

With this assumption, the input to each of your program is one positive number of type `Bignum`. To test your programs, you should randomly generate a sequence of positive numbers. Or, if you prefer, you can design a set of test data by hand.

Usually, twenty to thirty randomly generated numbers will do. Determine by yourself a suitable range for the numbers.

2.4 Evaluating the Programs

Measure the runtime of your program for the test data, and enter your results in the lab book. Then draw a graph for each program, comparing the input size and the runtime of your program (i.e., one axis for the size of the input, the other axis for the runtime).

Look at the graphs and the control structures of your programs, and try to figure out the order of complexity for each of your programs. You probably remember from the lectures that the order of complexity is an expression of the form $O(f(n))$, where f is some standard function and n the input size of the program.

Don't panic at this point if you are wondering what the input size of each of your programs is. It is true that each program takes exactly one integer for each run. But you may view this integer as a code for a sequence of "1"s. When viewing it this way, your program would have to input such a sequence before it could make its computations. So the input size would be the length of the sequence. It is therefore plausible to assume that the input size of your programs is identical with the input itself.

[1] See Douglas Adams' *The Hitch Hiker's Guide to the Galaxy* for an explanation of the meaning of this number.

The University of Auckland
Computer Science Department

07.231 Introduction to Software Engineering 1994

Christian Collberg, Hans Guesgen, Robert 'Buz' Uzgalis

Assignment 3:
DATE-on-LINE (300 points)

3.1 Introduction

You have been employed by the newly formed CompDate Ltd corporation to design and implement a computerized dating service. The company intends to put up a number of DATE-on-LINE terminals where people may register to make themselves available for dating, and check in to see if any suitable prospects have been found.

Here is an example showing how CompDate Ltd changed the lives of Jeremy and Jessica:

Scenario i): Jeremy walks up to DATE-on-LINE terminal for the first time. He enters his personal data (looks, interests, age, address, phone,...), the required data of his date-to-be, plus a PIN-number. In response he receives a short list of possible dates (without their names and addresses) and selects one or more as "possibilities". Jeremy walks off.

Scenario ii): Jessica walks up to DATE-on-LINE terminal, enters her name and PIN-number. Since her last visit to the terminal 3 men have expressed an interest in her (among them our Jeremy from scenario i). She selects him as the most compatible (lucky guy). Walks away.

Scenario iii): Jeremy again approaches terminal, enters PIN, and receives Jessica's name and number.

Scenario iv): A week later Jeremy and Jessica each receive a bill from CompDate Ltd.

3.2 The Program

You should implement a program which performs the functions described in the story of Jeremy and Jessica above. In other words, your program should be able to:

- input personal data from a terminal
- select a small number of people which are compatible with the current client
- protect the personal information from improper use (e.g. by using a PIN-number)
- produce billing information ($5 per compatible match. The bills should be printed in sorted order by name.)

It is furthermore important for the program to have an attractive user interface, since it will be used by people otherwise untrained in computer usage.

The exact matching algorithm is left up to you, but keep the following in mind:

- A poor matching function will make the users disillusioned with CompDate Ltd which may fold and you will lose your job[1].
- Certain items must match exactly (see `wants-gender` below), others need not (`wants-height`).
- A slow matching function will spoil the interactive nature of the program, and, again, your job will be in jeopardy.

3.3 The Data Base

The program should be able to handle large numbers of registered users, at least 10,000. It may therefore be necessary to make use of sophisticated data structures and search algorithms in order to make the program truly interactive (query–response time should be within one second). You are welcome to use your own data, but a large data set will be provided for you. It comes in the form of a text file with the following format:

```
first-name last-name
    street phone PIN
    gender age ethnic-orig height weight
    hair-color eye-color complexion
    music-pref author-pref sport-pref food-pref movie-pref
    smoker drinker car bike
    school major minor
    job pay
    turn-on1 turn-on2 turn-on3 turn-on4
    turn-off1 turn-off2 turn-off3 turn-off4
    wants-gender wants-age wants-ethnic-orig wants-height wants-weight
    wants-hair-color wants-eye-color wants-complexion
    wants-smoker wants-drinker
```

[1] Or, in this context, you will get a poor grade.

Example:

```
Jessica Brown
   25-Wairike-Rd 630-9999 4567
   female 23 european 169 55
   blond grey tan
   new-age Anais-Nin ice-hockey Thai Star-Wars
   no yes Saab no
   Auckland-U French-Litt Art-Hist
   Wendys 8900
   dimples humor muscles guitarists
   computers straight-A-students tramping drugs
   male 33 dont-care  179 65
   bald green dont-care
   no dont-care
```

3.4 Administrivia

While this assignment is worth 300 points, it may be done by two students working together in which case the quality of the resulting product should be considerably higher than if done by one person.

Assignment 4: Improving QuickSort (100–200–300 points)

4.1 Introduction

QuickSort is maybe the fastest internal sorting algorithm based on comparisons, but the standard implementation found in textbooks leaves much room for improvement. In fact, an entire PhD thesis (Sedgewick's) was once written on the improvements and analysis of QuickSort! Your task in this assignment is to implement several different versions of QuickSort, and then examine experimentally how they behave when subjected to input data of various distributions.

4.2 Improvements

We are going to concentrate on three kinds of improvements:

1. Improving QuickSort's worst case behavior.
2. Improving QuickSort's behavior when the subarray to be sorted is small.
3. Reducing the overhead of recursive calls.

QuickSort is $\mathcal{O}(n \log n)$ in the average case, but $\mathcal{O}(n^2)$ in the worst case. The worst case occurs when QuickSort encounters a sorted, or nearly sorted, subarray. To counter this we have to make sure that the Pivot element (the element which decides how the subarray is partitioned) is selected at random. Two of the suggested improvement below deal with this.

For large values of n QuickSort will be faster than any of the simpler $\mathcal{O}(n^2)$ internal sorting algorithms (BubbleSort, SelectionSort, InsertionSort), but for small values of n this may not be the case. It may therefore be preferable to sort small subarrays using one of the simpler sorting algorithms, rather than using QuickSort's recursions.

On some machine architectures procedure calls can be very expensive. If this is the case, it may be worthwhile to recast a recursive program (which uses many procedure calls) into an iterative one.

4.3 Your Program

Your task will be to implement some of the following versions of QuickSort:

BQS Basic Quicksort. Get the code from some standard textbook in computer science.

IQS Same as BQS but use InsertionSort for subvectors of length ≤ 10. There are actually two ways of doing this:

1. Explicitly call InsertionSort on any subvector of length less than 10.
2. Stop the recursion when a subvector is shorter than 10. I.e. at the end of the QuickSort run the vector will consist of a sequence of *unsorted* subvectors of length 10. At this time call InsertionSort on the entire vector.

Which one of these methods should be preferred?

MQS Same as BQS but use Median-of-3 selection of the Pivot element. In other words, select three elements from the current subarray and use the middle one as the Pivot element.

RQS Same as BQS but use randomized selection of the Pivot element. I.e., draw a random number between 1 and the length of the subvector, and use this to select the Pivot element. This version is known as Randomized QuickSort.

IMQS QuickSort with InsertionSort for small subvectors and Median-of-3 selection of the Pivot element.

IRQS Randomized QuickSort with InsertionSort for small subvectors.

TRQ BQS where the second recursive call has been turned into an iteration. This is fairly simple since the call is a case of *tail recursion*.

NRQ A non-recursive version of QuickSort. This will normally require the use of a stack.

4.4 Evaluation

Run every coded version on sample data of the following sizes and distributions:

1. 200 sorted elements.
2. 200 elements in reverse sorted order.
3. 200 random elements.
4. 2000 sorted elements.

5. 2000 elements in reverse sorted order.
6. 2000 random elements.

Measure the time performance for each version and display it as a graph so that you can compare the performance of the various modifications to the algorithm.

4.5 Administrivia

This assignment is worth between 100 and 300 points:

100 points Code and evaluate at least 4 versions of QuickSort.

200 points Code and evaluate at least 6 versions of QuickSort.

300 points Code and evaluate all 8 versions of QuickSort.

Assignment 5: Ackerman's Function (50–100–300 points)

5.1 Problem

Write a program to evaluate Ackerman's function for any arbitrary argument values.

5.2 Description

Ackerman's function is a function of 2 nonnegative integer variables and is defined by the following 3 rules:

$$
\begin{aligned}
A(0,n) &= n+1 \\
A(m,0) &= A(m-1,1) \\
A(m,n) &= A(m-1,A(m,n-1))
\end{aligned}
$$

5.3 Suggestion

Find the value of $A(1,1)$ and $A(2,2)$ by hand. You can then use these to test your program.

5.4 Programs

1. For 50 points, have your program calculate and print the base ten integer value of:

$$A(0,0), A(0,1), A(0,2), \ldots A(0,25)$$
$$A(1,0), A(1,1), A(1,2), \ldots A(1,25)$$
$$A(2,0), A(2,1), A(2,2), \ldots A(2,25)$$

2. For 50 additional points (100 total), have you program calculate and print the base ten integer value of:

$$A(3,0), A(3,1), A(3,2), \ldots A(3,25)$$
$$A(4,0), A(4,1)$$

   in addition to doing 1 above.

3. For 200 additional points (300 total), have your program calculate and print the base ten integer value of $A(4,2)$ in addition to doing 1 and 2 above. You must print the entire number; do not use scientific (mantissa-exponent) notation.

Beware! Ackerman's function really "blows up" for increasing values of m and n (especially m). If you try and get the 100 point answers by using the 50 point methods, you will be unlikely to succeed. Note that just increasing your program's cpu time limit will not solve your problems: Ackerman's function does not yield twice as many answers if given twice as much computation time.

The 3 different problems (and point values) reflect 3 different programming methods (and levels of understanding) needed to obtain the value of Ackerman's function for the different argument ranges.

5.5 Measurement

1. After printing each answer, print how many recursions were necessary to compute the value. Counting $+1$ for every entry and -1 for every return, how "deep" does the recursion go in order to compute the answer? Print this number (the depth).

2. If you are maintaining your own stack, give the same statistics. For recursions count $+1$ any time you push something on the stack. For depth keep track of the maximum extent of the stack. Make a histogram of the number of times the same number goes on the stack in a series (i.e., if you push 1 on the stack 50 times before another value gets put on the stack it counts 1 in the 50 slot).

5.6 Standard Input

No input is required.

5.7 Caveat

Solving the problem for $A(4,2)$ is very difficult. If you want to try it, do so as your last problem.

5.8 Questions

1. Why should A(4,2) be worth so many points?
2. Explain what pattern (if any) Ackerman's function follows.
3. Can you find rules for n's? m's?
4. Have you used recursion? If yes, what seems to be the limiting factor? What goes on the stack? How deep in recursion does the algorithm go? Is there any way of simplifying the stack? What use would this be?
5. Have you avoided recursion by maintaining your own stack? If yes, what seems to be the limiting factor. How do you maintain the stack? Is there some way of reducing the storage you have used? What use would this be? How is the stack used by the algorithm?
6. Prove any modifications or extentions you have made to the formulas stated above. What are the advantages of your new formulas? What effect (if any) do they have on the memory requirements of the program? What effect do they have on running time?
7. Discuss the time/space tradeoff as it applies to programs evaluating Ackerman's function.

5.9 References

Ackerman, Wilhelm (1954), *Solvable Cases of the Decision Problem*, North Holland Press.

Minsky, Marvin (1967), *Computation: Finite and Infinite Machines*, Prentice-Hall, Inc.

The University of Auckland
Computer Science Department

07.231 Introduction to Software Engineering 1994

Christian Collberg, Hans Guesgen, Robert 'Buz' Uzgalis

Assignment 6: File Compression (200–300 points)

6.1 Introduction

In order to save disk space it is convenient to have a *file compression* program which reduces the amount of space necessary to store a particular file. A companion program (a *file decompression* program) is used to restore the file to its original format.

Huffman Coding represent one of many ways to do file compression. It works in five steps:

1. Read (a representative part of) the file to be compressed and determine the relative distribution of the characters in the file. The result will be in the form of a table such as this:

character	frequency
"a"	9%
"b"	2%
...	
"Y"	1%
"Z"	0.5%

2. Write the distribution on to the new, compressed, file.
3. Construct the Huffman code.
4. Read the file to be compressed again, use the Huffman code to compress each character, and write out the result on to the new, compressed, file.
5. Delete the old, uncompressed, file.

6.2 Constructing the Huffman Code

In a normal, uncompressed, text file each character is stored using one 8-bit byte. This is known as a *fixed-length code*. *Variable-length codes* are able to save a considerable amount of space (20% to 90% is typical) by assigning frequent characters short codes and infrequent characters long codes. One example of a variable-length code is the *Morse* code used in telegraphy: the code for `"e"` (the most common letter in the English alphabet) is much shorter than the code for `"z"`.

6.2.1 Prefix Codes

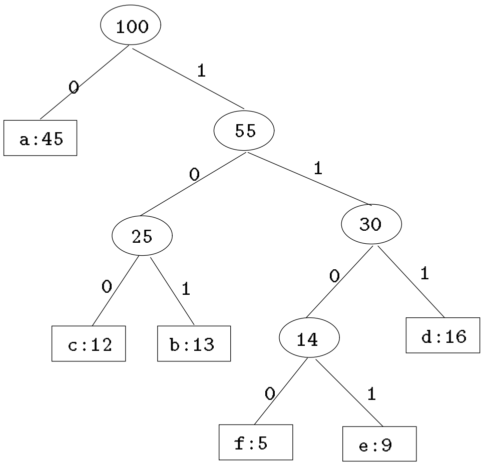
Huffman coding is known as a *prefix code* which means that *no codeword is also the prefix of some other codeword.* To see that a prefix code greatly simplifies the decoding of a compressed file, consider the following Huffman code:

	a	b	c	d	e	f
Frequency	45%	13%	12%	16%	9%	5%
Codeword	0	101	100	111	1101	1100

Let's decode the bitstring `"001011101"`, using this Huffman code. We see from our table that the first `"0"` has to be the letter `"a"`, since no other codeword starts with a `"0"`. The second `"0"` is also an `"a"`, then comes a `"b"` (only the letter `"b"` has a codeword which starts with the bits `"101"`), and finally an `"e"`.

6.2.2 Decoding

To quickly find the character which corresponds to a particular codeword, it is convenient to store the Huffman code in a binary tree where the leaves are the decoded characters. To decode a character we start at the root of the tree and work our way down until a leaf is found, going left for a `"0"` and right for a `"1"`. The decoding tree below corresponds to the coding scheme of our previous example:



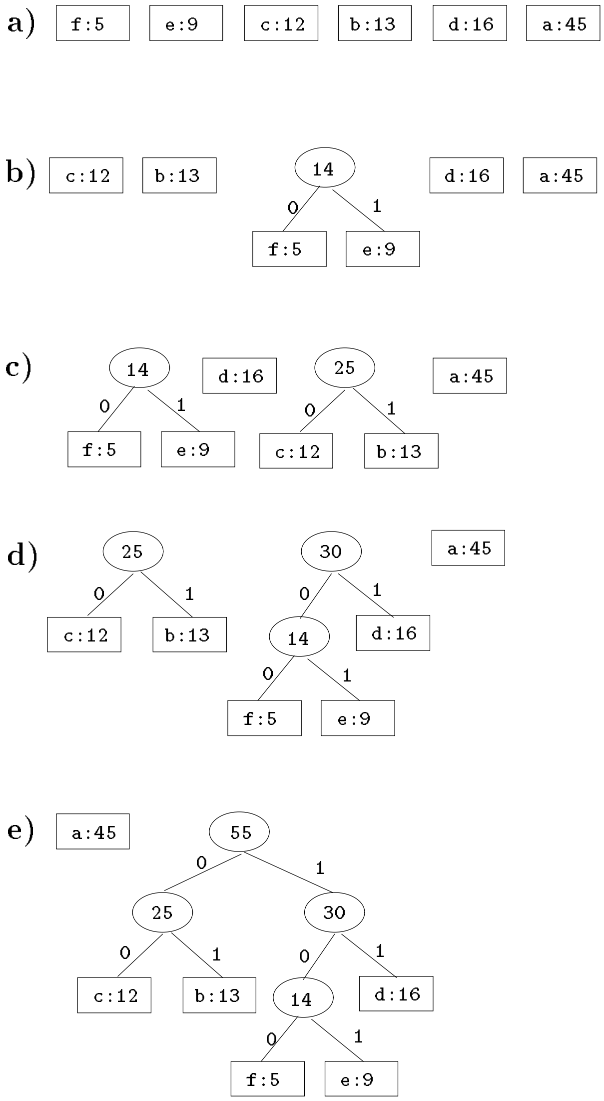
6.2.3 Constructing the Code

The following algorithm (in pseudo-code) builds the tree corresponding to a Huffman code (Q is priority queue of tree nodes, keyed on the relative frequency at which the character representing that node occurs):

```
n := number of characters;
Q := empty priority queue;
for i := 1 to n-1 do
    t := a new tree leaf for the n:th character;
    INSERT (Q, t, the frequency of the n:th character);
end;

for i := 1 to n-1 do
    x := EXTRACT_MIN(Q);
    y := EXTRACT_MIN(Q);
    z := a new tree node with left son x and right son y;
    INSERT (Q, z, the sum of x's and y's frequencies);
end;
```

The algorithm works by building the tree bottom-up, i.e. it assigns characters with low frequencies long codes (they are selected early from the priority queue) by putting them far down in the tree. At each step of the algorithm the two characters with the lowest frequencies are selected and their corresponding tree nodes are merged. Consider the following example (at each step of the algorithm the nodes are shown in the order in which they occur in the priority queue):



6.3 The Programs

You are to construct two programs, `compress` and `uncompress`, which prompt the user for the name of a file and then perform compression and decompression, respectively, using Huffman codes.

6.4 Administrivia

The examples and algorithms in this assignment are taken from *Introduction to Algorithms* by Cormen, Leiserson, and Rivest.

This assignment is worth 200 points. If your implementation of the Huffman code constructing algorithm runs in $\mathcal{O}(n \log n)$ time rather than $\mathcal{O}(n^2)$, you will receive 300 points.