Instructions: In a number of the questions you are asked to write actual C or MATLAB code. Try to compile and execute the code for each exercise on your computer (see the second problem to get started). If you cannot get the code to work, make sure to get to the end of the exercise using pen and paper. Note that using the online (PDF) version of this problem sheet could be more convenient for cutting and pasting.

Problem 1  Software lifecycle

Write notes contrasting the waterfall model of the software development lifecycle with extreme programming.

Solution to problem 1

Bookwork – really, just for interest’s sake (and to understand the importance of safety-critical software).

Problem 2  Hello world!

The goal of this exercise is to edit, compile, and execute a simple C program. You will use the same procedure later for more complex exercises as well. There are a few options:

- **Use an on-line C compiler.** On-line compilers are suitable for trying out simple programs such as the exercises in this tutorial. However, they are not very representative of a real-world programming environment. There are several such on-line environments (if you are given the option, make sure that you select C or C/C++ for this tutorial). We recommend using C++ Shell:
  - [http://cpp.sh](http://cpp.sh)

  Note also that such tools many not save your code. Therefore, make sure to save a copy on your computer (e.g. by cutting and pasting from a text editor to your browser and viceversa).

- **Install a C compiler on your machine.** This is more complicated, but much closer to a real work environment. Depending on your platform, you can get started here:
  - **Windows.** Install a copy of the Microsoft Visual C integrated development environment (IDE). As Oxford students, you have the right to install a copy of this software for free.
  - **Mac OS X.** Install Xcode from the Mac App Store for free. Xcode is Apple’s IDE, similar to Visual C.
– **Linux.** Install GCC using your Linux distribution package manager. GCC is not an IDE; in this case you will need also a text editor and to use the terminal to compile and run your program (do not worry, it is easy!).

Appendix B contains some instructions on how to get started with these environment; however, *Google is your friend!*

Once you have decided which editor to use, cut and paste the following program:

```c
#include <stdio.h>

int main(int argc, const char * argv[]) {
    printf("Hello, World!\n");
    return 0;
}
```

Make sure you can compile and execute the program, getting the correct output (which one?).

**Solution to problem 2**

The program should print the string **Hello, World!** in the console/terminal.

**Problem 3  Function encapsulation and side-effects**

1. What are function side-effects and why should they be avoided?

2. *Encapsulation* is an important concept in both procedural and object-oriented programming. Explain what is meant by encapsulation, and why it is important in software design and implementation.

3. Consider the mess in the code box below. Determine what the code does, indicate design flaws, and hence tidy it into something sensible.

```c
#include <stdio.h>
int num,res; void r(int s) {
    while (num>=0) { num = num-s; } num=num+s; res=num;}
int main() { num = 10;
r(4); printf("%d\n", res ); }
```

4. Compile and run the messy code as well as your tidier versions. Make sure that you obtain the same result in both cases.

**Solution to problem 3**

1. Function side-effects are changes to the program state made in the execution of a function that are not explicit in the function interface. Examples include the use of a global variable, since the behaviour of the function may vary depending on the value of this variable. Likewise the setting of a global variable, or other hidden" state changes are side-effects. They should be avoided because they violate the encapsulation of the function as providing black-box functionality, making the function harder to understand and difficult to reuse.
2. Encapsulation is the process whereby the interface to a software component, and its implementation details, are explicitly separated. The interface defines the ways in which the software component can interact with other components. If the interface is clearly specified then the component can be re-used with confidence, even if its internal implementation details change.

A well encapsulated component minimises side-effects (i.e. consequences which are not explicitly specified in the interface, e.g. changing the value of a global variable in a function call).

In the context of procedural programming, the software components of primary interest are functions/procedures. The operation and interface of a well designed function are transparent from the function header (prototype) which gives the set of inputs (parameters) and outputs (function return values or in the case of C++, perhaps non-constant reference parameters). This encapsulates the function’s meaning, without revealing internal implementation details. The C and MATLAB standard libraries are good examples of this level of encapsulation. A programmer can use these libraries without needing to know the internal implementation details, and can be confident that the operation is neither dependent on, nor affects the values of, global variables.

3. To begin, let’s tidy up the formatting. It is a little clearer in the box below that function \texttt{r()} is finding the remainder in the integer division \texttt{num/s} using repeated subtraction.

```c
#include <stdio.h>

int num, res;

void r(int s)
{
    while (num >= 0) num = num - s ;
    num = num + s ;
    res = num ;
}

int main()
{
    num = 10 ;
    r(4) ;
    printf("%d\n", res) ;
}
```

The variable \texttt{num} is an implicit input parameter to the function \texttt{r()} and \texttt{res} is being used as an implicit output. In this simple example such side-effects can be readily identified, because the implementation of \texttt{r()} is so tiny. However in more complex code this becomes a real problem. To remove the side-effects we make these relationships explicit in the definition of \texttt{r()}, so that \texttt{num} is passed in (by value) to the function, and the result passed out as a return value. When we do this, the local variable \texttt{g} becomes redundant because \texttt{num} is passed in by value.

```c
#include <stdio.h>
```
Problem 4  Parameters, recursion and the stack

The Chebyshev polynomials of the first kind $T_n(x)$ are defined recursively by

\[
\begin{align*}
T_0(x) &= 1, \\
T_1(x) &= x, \\
T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x).
\end{align*}
\]

The following MATLAB code evaluates $T_n(x)$ for a given order $n$:

```matlab
function ChebyshevExample()
    \text{function Chebyshev polynomials evaluation}
    \text{function val = chebyshev(n,x)}
    \text{if n == 0}
        \text{val = ones(size(x)) ;}
    \text{else}
        \text{if n == 1}
            \text{val = x ;}
        \text{else}
            \text{val = 2 .* x .* chebyshev(n-1,x) - chebyshev(n-2,x) ;}
        \end
    \end
end
```

1. With reference to the code, define and identify the formal and actual parameters of the function `chebyshev` and the function call.

2. For the second function call, show the evolution of the stack and the final value of the vector returned.
Solution to problem 4

1. In the particular case here, the formal parameters are \( n \) and \( x \) (and the return parameter is \( \text{val} \)), appearing in the function header \( \text{function val = chebyshev}(n,x) \). The actual parameters appear in the two calls to \( \text{chebyshev}() \) in the script. In the first call the params are \( a = 4 \) (mapping to formal parameter \( n \)) and \( t = [-10:0.01:10] \) (mapping to formal parameter \( x \)) while in the second they are the value 3 (mapping to \( n \)) and \( t = [-2:1:2] \), mapping to formal parameter \( x \).

2. The first call \( \text{chebyshev}(3,[-2:1:2]) \) pushes the following onto the stack:

   \[
   \begin{array}{c}
   1 \quad \% \text{chebyshev}(3,[-2:1:2]) \\
   2 \quad 3 \\
   3 \quad [-2 -1 0 1 2] \\
   \end{array}
   \]

   and calls \( 2*x*\text{chebyshev}(2,[-2:1:2]) + \text{chebyshev}(1,[-2:1:2]) \). The first of these calls pushes to give a stack:

   \[
   \begin{array}{c}
   1 \quad \% \text{chebyshev}(3,[-2:1:2]) \\
   2 \quad 3 \\
   3 \quad [-2 -1 0 1 2] \\
   4 \quad \% \text{chebyshev}(2,[-2:1:2]) \\
   5 \quad 2 \\
   6 \quad [-2 -1 0 1 2] \\
   \end{array}
   \]

   and runs \( \text{chebyshev}() \), yielding two more calls: \( 2*x*\text{chebyshev}(1,[-2:1:2]) + \text{chebyshev}(0,[-2:1:2]) \). The first of these pushes so the stack is now:

   \[
   \begin{array}{c}
   1 \quad \% \text{chebyshev}(3,[-2:1:2]) \\
   2 \quad 3 \\
   3 \quad [-2 -1 0 1 2] \\
   4 \quad \% \text{chebyshev}(2,[-2:1:2]) \\
   5 \quad 2 \\
   6 \quad [-2 -1 0 1 2] \\
   7 \quad \% \text{chebyshev}(1,[-2:1:2]) \\
   8 \quad 1 \\
   9 \quad [-2 -1 0 1 2] \\
   \end{array}
   \]

   and this call returns \([-2 -1 0 1 2]\) because \( n==1 \), popping the bottom two parameters in the process. Hence the stack is now:

   \[
   \begin{array}{c}
   1 \quad \% \text{chebyshev}(3,[-2:1:2]) \\
   2 \quad 3 \\
   3 \quad [-2 -1 0 1 2] \\
   4 \quad \% \text{chebyshev}(2,[-2:1:2]) \\
   5 \quad 2 \\
   6 \quad [-2 -1 0 1 2] \\
   \end{array}
   \]

The second call \( \text{chebyshev}(0,[-2:1:2]) \) pushes so the stack is:

   \[
   \begin{array}{c}
   1 \quad \% \text{chebyshev}(3,[-2:1:2]) \\
   2 \quad 3 \\
   \end{array}
   \]
and returns \([1 \ 1 \ 1 \ 1 \ 1]\) because \(n=0\) (popping the two bottom parameters in the process).  
\(\text{chebyshev}(3,\[-2:1:2\])\) can now be evaluated as  
\(2.*[-2 -1 0 1 2].*[1 1 1 1 1] = [7 1 -1 1 7]\)  
and its parameters popped from the stack, so the stack is now back to  
\(\% \text{ chebyshev}(3,[-2:1:2])\)
\(3\)
\([-2 -1 0 1 2]\)

We now have  
\(2.*x.*[7 1 -1 1 7] - \text{chebyshev}(1,[-2:1:2])].\)  
The second call pushes so the stack is  
\(\% \text{ chebyshev}(3,[-2:1:2])\)
\(3\)
\([-2 -1 0 1 2]\)
% \text{chebyshev}(1,[-2:1:2])
1
\([-2 -1 0 1 2]\)

and returns \([-2 -1 0 1 2]\) since \(n=1\), popping the parameters as it returns.

Finally we can evaluate  
\(2.*x.*[7 1 -1 1 7] - [-2 -1 0 1 2] = [-26 -1 0 1 26]\)  
and pop the initial parameters from the stack.

**Problem 5  Parameters and recursion**

1. Write a recursive MATLAB function to compute the \(n^{th}\) Fibonacci number, where the first two Fibonacci numbers are defined to be 1 and 1, and each subsequent number is the sum of the previous two.

2. Write a non-recursive function to do the same

3. If your answer to part (i) involves *binary* recursion (*i.e.* the function calls itself twice), consider the cost of this, and devise a recursive algorithm that uses *linear* recursion instead (*i.e.* the function call itself only once).

4. Compare the amount of work required by binary recursion and linear recursion by sketching a trace of the function calls. Is binary recursion doing redundant work?

**Solution to problem 5**
1. The following function computes the Fibonacci numbers using binary recursion:

```matlab
function res = fib2(n)
    if n==1 || n==2
        res = 1;
    else
        res = fib2(n-1) + fib2(n-2);
    end
end
```

2. The following function computes the Fibonacci numbers using a simple iteration:

```matlab
function res = fib_nonrecursive(n)
    if n==1 || n==2
        res = 1;
    else
        a = 1;
        b = 1;
        for i=3:n
            temp = b;
            b = a+b;
            a = temp;
        end
        res = b;
    end
end
```

3. The following function computers the Fibonaccy number using linear recursion:

```matlab
function [res,prev] = fib1(n)
    if n==1
        res = 1;
        prev = 0;
    elseif n==2
        res = 1;
        prev = 1;
    else
        [f1,f2] = fib1(n-1);
        res = f1+f2;
        prev = f1;
    end
end
```

4. The difference between the two recursive implementations can be appreciated by inspecting the call trace:
Note that binary recursion computes the same values over and over again, with exponential complexity (this could be avoided by memoisation).

**Problem 6  Sorting, loop invariants and passing functions as parameters**

The pseudo-code for a sorting function that can sort a set of inputs into ascending order is given below:

```plaintext
% ROUTINE: BubbleSort
% INPUT: A is array of N elements
% OUTPUT: A is sorted array of elements
for i=1 to N-1
    for j=i+1 to N
    end
end
return A
```

1. Determine the order $O(...)$ of the sorting routine.

2. Determine a loop invariant for each loop and hence satisfy yourself that the function will have the desired effect.

3. Explain how the sorting function can be made generic by replacing the comparison $A[i] > A[j]$ with a call to a function GreaterThan, which takes two items and returns 1 if $x$ is less than $y$, and 0 otherwise, and where GreaterThan is a formal parameter to BubbleSort. Write down the likely function prototype for BubbleSort.

4. Write such a comparison function that compares the lengths of two vectors under the 2-norm and show how it can be used with the generic sorting function to sort the rows of a matrix.

5. Write two more comparison functions that use (i) the 1-norm and (ii) the infinity norm.

6. Comment on the advantages of this programming construct in this context.
1. There are two nested loops. The first loops \( N - 1 \) times, and for each value of \( i \) the inner loop executes \( N - i \) times, yielding \( O(N^2) \) as the order of the algorithm.

2. There are two invariants \( P_1(i) \) and \( P_2(i, j) \) for each of the two loops:

\[
\begin{align*}
\% \text{ P1(0) is true here (A)} \\
\text{for } i=1 \text{ to N-1} \quad \% \text{ P2(i, i) is true here (B)} \\
\quad \text{for } j=i+1 \text{ to N} \quad \% \text{ P2(i, j) is true here (C)} \\
\quad \text{if } A[i] > A[j] \text{ then swap } A[i] \text{ and } A[j] \\
\text{end} \quad \% \text{ P1(i) is true here (D)} \\
\end{align*}
\]

Conceptually, the algorithm starts with \( A \) and an empty array \( B \). It then gradually removes the minimal element from \( A \) and appends it to \( B \). Rather than working with two array explicitly, which would be memory inefficient, the code divides \( A \) in two parts and use the swap operation to move the data around.

This intuition can be formalised by means of loop invariants:

(a) The outer invariant is

\[
\text{for } i \geq 1 \quad P_1(i) : \quad A[1] \leq \ldots \leq A[i] \quad \text{and} \quad A[i] \leq A[i], \ldots, A[N];
\]

\[
P_1(0) : \quad \text{empty statement (always true).}
\]

Intuitively \( P_1(i) \) means that the elements \( A[1], \ldots, A[i] \) from 1 to \( i \) are sorted in ascending order and that \( A[i] \) is minimal in the subarray \( A[i], \ldots, A[N] \). In particular, \( P_1(N - 1) \), obtained at the end of the algorithm, corresponds to the whole array being sorted.

(b) The inner invariant is:

\[
P_2(i, j) : \quad P_1(i - 1) \quad \text{and} \quad A[i] \leq A[i], \ldots, A[j];
\]

\[
P_2(i, j) \text{ means that } A[i] \text{ is minimal in } A[i], \ldots, A[j]; \text{ it also means that the outer invariant } P_1(i - 1) \text{ is maintained in the inner loop.}
\]

We prove the correctness of the inner invariant \( P_2(i, j) \) for all \( j \) assuming that the outer invariant \( P_1(i - 1) \) is true from the previous outer iteration. Given \( P_1(i-1), P_2(i, i) \) is trivially true at line 4. We prove the remaining cases by induction on \( j \). This requires showing that, if \( P_2(i, j-1) \) is true, so is \( P_2(i, j) \). The inner loop contains only line 5; according to this line, if \( A[i] \leq A[j] \) nothing is done, otherwise \( A[i] \) and \( A[j] \) are swapped. In the first case, \( A[i] \) is already not greater than \( A[j] \) and, due to the inductive hypothesis \( P_2(i, j - 1) \), it is also not greater than \( A[i + 1], \ldots, A[j - 1] \); hence \( A[i] \) is minimal in \( A[i], \ldots, A[j] \) and \( P_2(i, j) \) is true. In the second case, \( A[j] \) is smaller than \( A[i] \) and the latter is minimal in \( A[i], \ldots, A[j - 1] \) due to \( P_2(i, j - 1) \). As a consequence, \( A[j] \) is minimal in \( A[i], \ldots, A[j] \) and swapping \( A[j] \) and
$A[i]$ makes $A[i]$ the minimal element again. Note also that swapping $A[j]$ with $A[i]$ maintains the validity of $P_1(i - 1)$. Hence $P_2(i, j)$ is true in this case as well.

Now that the inner invariant has been proved, we prove the outer invariant. The proof of correctness is again by induction. Starting from the base case $P_1(0)$, we need to prove that, if $P_1(i - 1)$ is true, so is $P_1(i)$. To this end, note that at line 8 both $P_1(i - 1)$ and $P_2(i, N)$ are satisfied. The former means that the first $i - 1$ elements are sorted and not greater than the last $N - i + 1$ elements. The latter means that, among these $N - i + 1$ elements, the first one is minimal. Combining these two facts, we obtain that the first $i$ elements are sorted and not greater than the remaining $N - i$, proving $P_1(i)$.

3. Pass in a function parameter $GT$ that compares two elements ($GT$ is assumed to be a function handle). For example, in MATLAB this might look like:

```matlab
function A = BubbleSort(A,GT)
    % The function sorts the columns of the array A. N is the number
    % of columns. As a special case, the function can sort a vector
    % of number.
    N = size(A,2);
    for i=1:N-1
        for j=i+1:N
            if (feval(GT,A(:,i),A(:,j)))
                temp = A(:,j);
                A(:,j) = A(:,i);
                A(:,i) = temp;
            end
        end
    end
end
```

The following function demonstrates how to use BubbleSort to sort a list of scalars or a list of vectors. These are coded as columns of a matrix.

```matlab
function BubbleSortExample()
    % Sort a row vector using the standard definition
    % of the greater-than operator
    A = [1 16 20 10 -10];
    B = BubbleSort(A, @gt);
    disp('Before sorting:'); disp(A);
    disp('After sorting:'); disp(B);

    % Sort the columns of a matrix using GreaterThan
    A = [1 2 3 5 0 ;
         2 3 4 5 0 ;
         5 6 7 8 0 ;
         1 2 8 6 0];
    B = BubbleSort(A, @GreaterThan);
    disp('Before sorting:'); disp(A);
```
disp('After sorting:') ; disp(B) ;

function res = GreaterThan(x,y)
res = norm(x) > norm(y) ;

Problem 7  Data structures and interfaces

In this exercise we explore the design and implementation of a list data structure in the C language. The code comprises three files: list.h, containing the list declaration, list.c, containing the list implementation, and main.c, containing a program using the list. The first file, list.h, is as follows:

```c
#ifndef __LIST_H__
#define __LIST_H__

typedef double DataType ;

/* ListElement interface */
typedef struct ListElement_ ListElement ;

ListElement* ListElementGetSuccessor(ListElement *element) ;
DataType ListElementGetValue(ListElement *element) ;

/* List interface */
typedef ListElement List ;

List * ListCreate() ;
void ListDelete(List *list) ;

void ListClear(List *list) ;
int ListIsEmpty(List *list) ;

ListElement * ListGetFirstElement(List *list) ;
ListElement * ListGetLastElement(List *list) ;
ListElement * ListFindElement(List *list, DataType x) ;

void ListInsertAfterElement(ListElement *element, DataType x) ;
void ListRemoveAfterElement(ListElement *element) ;
void ListPrepend(List *list, DataType x) ;
void ListAppend(List *list, DataType x) ;

#endif /* __LIST_H__ */
```

The second file, main.c, is as follows:

```c
#include "list.h"

int main(int argc, char ** argv)
The content of the third file, main.c, is given in Appendix A.

Note that main.c includes the list.h file. Neither list.h nor main.c define the structure ListElement (list.h only declares ListElement to be a structure, but does not provide any detail). In other words, from the viewpoint of main.c, both ListElement and List are opaque data types.

1. How can the program in main.c make use of opaque data types? Are opaque data types a good idea? Why?

2. Compile and execute the program. To do so, note that you need to combine two implementation files, main.c and list.c. This can be done by running the C compiler individually on main.c and list.c to produce the intermediate object files main.o and list.o and then by running the linker to combine the object files into a single executable file. In practice:

   - If you compile the code online using Coding Grounds, simply create the three files list.h, list.c, and main.c and compile and execute as usual.
   - If you use instead an IDE such as Visual C or Xcode, create the three files in the interface. Then, the IDE should be smart enough to correctly understand what needs to be done automatically.
   - If you use the command line, such as GCC in Linux, simply specify the two implementation files; the compiler is smart enough to do all the steps (including linking) automatically:

        > gcc -o list list.c main.c

3. Study carefully the implementation of the list (in list.c) then look at the main() function again. Which ones of the interface function allocate memory on the heap and for which purpose? Which ones free memory? Does the program leak any memory?

4. ListCreate() is used in the main() function to return a pointer to a new list. Intuitively, this corresponds to creating a new instance of a list object. Which part of the program logically owns this object instance? Should the object instance be explicitly disposed of? As an author of the main() function, is it your responsibility to dispose of the object instance? If so, how should this be done?

5. Which part of the program is logically responsible for handling the memory required to hold the list elements?
6. The header file `list.h` contains enough information for the compiler to understand how to use the list interface. However, a user might at best guess the meaning of the interface functions and how they should be used. A clear understanding can only be achieved by studying the implementation file, which defies the purpose of having an interface in the first place (i.e., encapsulation).

The correct solution is to document the interface properly. For example, the documentation for `ListCreate()` might read

```
/**
 * ListCreate - Create a new list.
 * @return - pointer to the new list.
 * @
 * The function ListCreate() allocates and initializes a new empty list and returns a pointer to it. The ownership of the new object is transferred to the caller. The object must be disposed of by calling ListDelete().
 * @
 * Errors: if there is not sufficient memory to allocate the new list, the function returns a NULL pointer.
 */
```

The documentation of the `ListAppend()` function might read

```
/**
 * ListAppend - Append a new element to a list.
 * @param `list` - a pointer to the list.
 * @param `x` - the value of the new list element.
 * @
 * The function ListAppend() creates a new element at the end of the list `list` and initializes its value to `x`. Note that the list retains the ownership of the new element.
 * @
 * Errors: if there is not sufficient memory to allocate the new list element, an exception is raised terminating the program.
 */
```

Write a similar documentation for the `ListInsertAfterElement()` and `ListFindElement()` functions.

**Solution to problem 7**

1. The compiler only requires the function declaration to be able to generate code that calls them. Furthermore, all opaque data types are pointers such that the compiler does not need to know the detail of the pointed objects to manipulate them (for example to pass them as function arguments).

2. Practical work.

3. `ListCreate()` allocates memory to store the list object and `ListAppend` allocates memory to store a new list element. `ListDelete()` deletes the memory used to store the list object; furthermore, by inspecting the code in `main.c`, note that this function first calls `ListClear()`
which removes all elements in the list, including freeing the corresponding memory. Hence, 
\texttt{ListDelete()} deletes all the memory associated to the list.

4. \texttt{ListCreate()} allocates memory to hold the list object; however, it then transfers the own-
ership of this memory to the caller function (in this case \texttt{main()}). Note that this is purely a design choice, or a software pattern – the compiler is completely oblivious to this concept of ownership; instead, it is a concept used by the programmer to understand how the inter-
face is meant to be used. In this case, the \texttt{main()} function should be though of as an \textit{user} of the interface. The fact that \texttt{main()} becomes the owner of the object means that the
it is responsible for deleting the object when the latter is no longer needed, by calling \texttt{ListDelete()}.

5. The list elements are allocated by the \texttt{List} object, for example when the \texttt{ListAppend()} func-
tion is called (\texttt{ListInsertAfterElement()} has a similar effect). Note that the \texttt{ListAppend()} func-
tion \textit{does not} transfer the ownership of the new list element to \texttt{main()} (which is clear because it \textit{does not} return a pointer to the new element). Instead, the list object \textit{retains ownership} of all the list elements. This is why \texttt{ListDelete()} \textit{must} delete any list element in the list before deleting the list itself.

6. For example:

\begin{verbatim}
/** ListInsertAfterElement - Insert a new element after another.
 ** param `element` - a pointer to the reference list element.
 ** param `x` - the value of the new list element.
 **
 ** The function ListInsertAfterElement() creates a new list element
 ** with value `x` and inserts it after `element`. Note that the list
 ** retains the ownership of the new element.
 **
 ** Errors: if there is not sufficient memory to allocate the
 ** new list element, an exception is raised terminating the program.
 **
 ** ListFindElement - Find an element by value.
 ** param `list` - a pointer to the list.
 ** param `x` - the element value to search for.
 **
 ** The function ListFindElement() searches the list `list` for the
 ** first element with value `x` and and returns
 ** a pointer to it. If there is no such element in the list, the
 ** function returns NULL.
 **
\end{verbatim}

\textbf{Problem 8  Simple algorithms on dynamic structures}

With reference to the list program developed in the previous question

1. Add to the file \texttt{main.c} a function \texttt{void print(List* list)} that prints the value of the elements in the list \texttt{list}. Note that the \texttt{DataType} of the values is \texttt{double}, such that the
correct code for printf is "%g" (i.e. printf("%g", x) prints x correctly provided that x is a variable of type double).

2. Using the new print() function, compile and run the following version of the main() function:

```
int main(int argc, char **argv)
{
    List *list;
    ListElement *element;

    list = ListCreate();

    ListAppend(list, 1);
    ListAppend(list, 2);
    ListAppend(list, 3);
    ListAppend(list, 4);
    print(list);

    element = ListFindElement(list, 3);
    ListInsertAfterElement(element, 10);
    print(list);

    ListRemoveAfterElement(ListFindElement(list, 2));
    print(list);

    ListClear(list);
    print(list);

    ListAppend(list, 3);
    print(list);

    ListDelete(list);
    return 0;
}
```

Comment on the generated output.

3. Now write a merge(list3, list1, list2) function merging two lists list1 and list2 into a third list list3 in such a way that, if list1 and list2 are in non-decreasing order, so is list3 (this is the merge step of merge sort). Test your function using the following program:

```
int main(int argc, char **argv)
{
    List *list1, *list2, *list3;

    list1 = ListCreate();
    list2 = ListCreate();
    list3 = ListCreate();
```
ListAppend(list1, 1);
ListAppend(list1, 3);
ListAppend(list1, 4);
ListAppend(list1, 7);

ListAppend(list2, 2);
ListAppend(list2, 5);
ListAppend(list2, 6);

merge(list3, list1, list2);

print(list1);
print(list2);
print(list3);

ListDelete(list3);
ListDelete(list2);
ListDelete(list1);
return 0;
}

Solution to problem 8

1. An implementation is:

```c
#include <stdio.h>

void print(List *list)
{
    ListElement* element = ListGetFirstElement(list);
    printf("(");
    while (element) {
        printf(" %g", ListElementGetValue(element));
        element = ListElementGetSuccessor(element);
    }
    printf(")\n");
}
```

Note that, by using suitable functions of the interface, there is no need to know the implementation details of both List and ListElement, which remain opaque objects.

2. For example:

```c
void merge(List *list3, List *list1, List *list2)
{
    ListElement *element1 = ListGetFirstElement(list1);
    ListElement *element2 = ListGetFirstElement(list2);
```
Problem 9  Algorithms and loop invariants

The method of bisection for finding the roots of a function $f(x)$ is as follows: Starting with two points $x_1$ and $x_2$ such that $x_1 \leq x_2$ in the domain of $f$:

- Compute $f(r)$ where $r = (x_1 + x_2)/2$.
- If $x_2$ is sufficiently close to $x_1$ return $r$ as a root.
- If not, then examine the signs of $f(r)$, $f(x_1)$ and $f(x_2)$. If the signs of $f(r)$ and $f(x_1)$ differ then look for a root in $[x_1, r)$, otherwise look in $(r, x_2]$.

1. Write a MATLAB function to implement it.

2. What are the preconditions and loop invariant required to implement this algorithm as a loop?

3. The loop invariant is insufficient to prove the correctness of this algorithm. One must also show that the algorithm terminates and that it does so in a useful state. What is an upper bound on the number of iterations that the algorithm executes? What can you say about the precision of the estimated root at the end of the algorithm?

4. If the preconditions are not satisfied, what does your implementation do?

Solution to problem 9

1. The following MATLAB function implements `Bisection()`:
function r = Bisection(func, x1, x2)

fx1 = feval(func, x1);
fx2 = feval(func, x2);

% Check the validity of the precondition
assert(x1 <= x2);
assert(fx1*fx2 <= 0);

eps = 1.0e-6;
r = (x1+x2)/2;
fr = feval(func, r);

while (x2 - x1 > eps)
    fx1 = feval(func,x1);
    if (fx1*fr <= 0)
        x2 = r;
    else
        x1 = r;
    end
    r = (x1+x2)/2;
    fr = feval(func, r);
end

The following MATLAB function demonstrates how to use Bisection():

function BisectionExample()

% find a root of cos(x)
x = Bisection(@cos, 0, pi);
fprintf('%.2f is a root of cos(x)\n', x);

% find a root of x^2-3x+1
x = Bisection(@(x) x.^2 - 3*x + 2, 0, 1.5);
fprintf('%.2f is a root of x.^2 - 3*x + 2\n', x);

2. The preconditions are the requirements that the input must satisfy for the algorithm to execute correctly. In this case, the requirements are that $f$ is a continuous function, $x_1 \leq x_2$, and $f(x_1)$ and $f(x_2)$ have opposite signs or are null. Taken together, these conditions guarantee that there is a root in the interval $[x_1, x_2]$. The loop invariant $P$ enforces this condition throughout the execution of the algorithm:

$$P : \quad x_1 \leq r \leq x_2 \quad \text{and} \quad f(x_1)f(x_2) \leq 0.$$  

Proving the validity of the invariant requires showing that $P$ is true at the end of the loop (line 21) given that the invariant $P'$ from the previous iteration is true at the the beginning of the loop (line 12). Note that, in this case, $P$ and $P'$ formally coincide; however, $P'$ is a statement about the variables before the body of the loop is executed (i.e. at the end of the previous iteration), while $P$ after that.
To show that $P$ is true, examine the if statement at lines 16–20: if $f(x_1)f(r) \leq 0$, then setting $x_2 \leftarrow r$ retains the invariant $f(x_1)f(x_2) \leq 0$; otherwise, use the fact that $f(x_1)f(x_2) \leq 0$; since $f(x_1) * f(r) > 0$, we can multiply this quantity to both sides of the inequality, obtaining $f(x_1)^2f(r)f(x_2) \leq 0$. Since $f(x_1)f(r) > 0$, $f(x_1)$ is not zero and $f(x_1)^2$ is strictly positive. Hence this quantity can be simplified, resulting in $f(r)f(x_2) \leq 0$. Thus setting $x_1 \leftarrow r$ retains the invariant $f(x_1)f(x_2) \leq 0$. Finally, line 21 $r \leftarrow (x_1 + x_2)/2 < 2$ restores the property $x_1 \leq r \leq x_2$.

3. The algorithm terminates when when $x_2 - x_1 \leq \epsilon$. Since there is always a root in the interval $[x_1, x_2]$, it means that the point $r$ returned by the algorithm is within a distance $\epsilon/2$ from the root. Note also that the algorithm halves the length of the interval at each iteration. Hence it stops when: $(x_2 - x_1)/2^t \leq \epsilon$, i.e. $t = \lceil \log_2 \frac{x_2 - x_1}{\epsilon} \rceil$.

4. If precondition that $f$ continuous not satisfied then behaviour unpredictable. Other precondition is easily tested before the loop and an error can be flagged.

**Problem 10 Advanced algorithms**

An undirected graph can be represented in MATLAB using a single matrix in which the $(i, j)$th entry in the matrix is the weight of the edge between nodes $i$ and $j$, with a weight of zero meaning no edge exists (we assume that edges must have strictly positive weight).

A spanning tree of a graph is a tree (i.e. every vertex can be reached from every other vertex via one and only one path) containing all the nodes of the graph. A minimal spanning tree of a graph is the spanning tree whose sum of edge weights is minimal.

1. The MATLAB code below finds the minimal spanning tree of the input argument graph using Prim’s algorithm

   [Link to Prim's algorithm](http://en.wikipedia.org/wiki/Prim%27s_algorithm)

   Read and understand the MATLAB code, and add comments where indicated that describe the meaning of each part of the code.

2. Write down the matrix $G$ that represents the graph shown in the figure below.

3. Show an execution trace of the function running on this graph, clearly stating the values of the variables at each starred location in the code.
%% Compute minimal spanning tree of graph G
function T = MinST(G)

%% Add comment here describing V1 and V2
V1 = [1];
V2 = 2:length(G);

%% T is the result, set to have no edges
T = zeros(size(G));

%% max is just a big number bigger than any edge weight in G
max = 100000;

while (~isempty(V2))
    % *** Execution trace: V1, V2, T
    % Add a comment here: what do the following loops do?
    min = max;
    for i=1:length(V1)
        for j=1:length(V2)
            if (G(V1(i),V2(j))>0 && G(V1(i),V2(j))<min)
                % add a comment here: what is the condition
                % and what do these lines do?
                min = G(V1(i),V2(j));
                u = V1(i);
                v = V2(j);
            end
        end
    end
    % *** Execution trace: u, v, min

    % What does the following line do and do the invariants
    % of the loops above ensure it is the right thing?
    T(u,v) = min;

    % explain the following two lines
    V1 = [V1 v];
    V2(V2==v)=[];
end

Solution to problem 10

% Compute minimal spanning tree of graph G
function T = MinST(G)

% V1 is the set of vertices that have been added to the minimal
% spanning tree. V2 is the set of vertices remaining. These two are
% disjoint sets, and at each step we try to pick the edge between the
% two sets that in minimal (a greedy choice)
V1 = [1];
V2 = 2:length(G);

% T is the result, set to have no edges
T = zeros(size(G));

% max is just a big number bigger than any edge weight in G
max = 100000;

while (~isempty(V2))
    % print trace
    V1
    V2
    T

    % min keeps track of the lowest weight edge so far. we start it
    % out as a "big" number; i.e. bigger than any potential edge weight.
    % the two loops consider each pair of vertices with one from V1 and one
    % from V2 and choose the one with the least weight.
    min = max;
    for i=1:length(V1)
        for j=1:length(V2)
            if (G(V1(i),V2(j))>0 && G(V1(i),V2(j))<min)
                % the condition checks that there is an edge (>0) and
                % if its weight is smaller than the best so far. if it beats the
                % current best then we store the weight and save which two vertices
                % are involved in u and v
                min = G(V1(i),V2(j));
                u = V1(i);
                v = V2(j);
            end
        end
    end

    % T is the matrix representing the graph of the minimal spanning tree
    % Here we have decided that u and v are joined in the MST so we add that
    % to T, with its weight min (which equals G(u,v)). the invariant of the
    % loops above is that min,u,v contains the lowest weight edge between the
    % V1(1)...V1(i) and V2(1)...V2(j). The terminating condition is clearly
    % that we have the lowest overall because i is past the end of V1 and
    % likewise j is past the end of V2.
    T(u,v) = min;

    V1 = [V1 v];  % Add v to V1 set of vertices in the MST
    V2(V2==v)=[];  % Remove v from V2, the set of vertices not yet visited
1.

\[ G = \begin{bmatrix} 0 & 1.5 & 3.9 & 4.8; \\ 1.5 & 0 & 1.8 & 0; \\ 3.9 & 1.8 & 0 & 3.2; \\ 4.8 & 0 & 3.2 & 0 \end{bmatrix} \]

2.

\[ V1 = 1 \]
\[ V2 = \\
\begin{bmatrix} 2 & 3 & 4 \end{bmatrix} \]
\[ T = \\
\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \]
\[ u = 1 \]
\[ v = 2 \]
\[ \text{min} = 1.5000 \]
\[ V1 = \\
\begin{bmatrix} 1 & 2 \end{bmatrix} \]
\[ V2 = \\
\begin{bmatrix} 3 & 4 \end{bmatrix} \]
\[ T = \\
\begin{bmatrix} 0.00000 & 1.50000 & 0.00000 & 0.00000 \\ 0.00000 & 0.00000 & 0.00000 & 0.00000 \\ 0.00000 & 0.00000 & 0.00000 & 0.00000 \\ 0.00000 & 0.00000 & 0.00000 & 0.00000 \end{bmatrix} \]
\[ u = 2 \]
\[ v = 3 \]
\[ \text{min} = 1.8000 \]
\[ V1 = \\
\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \]
\[ V_2 = 4 \]
\[ T = \]
\[
\begin{bmatrix}
0.0000 & 1.5000 & 0.0000 & 0.0000 \\
0.0000 & 0.0000 & 1.8000 & 0.0000 \\
0.0000 & 0.0000 & 0.0000 & 0.0000 \\
0.0000 & 0.0000 & 0.0000 & 0.0000 \\
\end{bmatrix}
\]
\[ u = 3 \]
\[ v = 4 \]
\[ \text{min} = 3.2000 \]
\[ \text{ans} = \]
\[
\begin{bmatrix}
0.0000 & 1.5000 & 0.0000 & 0.0000 \\
0.0000 & 0.0000 & 1.8000 & 0.0000 \\
0.0000 & 0.0000 & 0.0000 & 3.2000 \\
0.0000 & 0.0000 & 0.0000 & 0.0000 \\
\end{bmatrix}
\]
#include "list.h"

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

/*------------------------------------------------------------------*/
/* ListElement implementation */
/* *------------------------------------------------------------------*/

typedef struct ListElement_
{
  struct ListElement_ *next ;
  DataType data ;
} ListElement ;

ListElement * ListElementGetSuccessor(ListElement *element)
{
  if (element) {
    return element->next ;
  } else {
    return NULL ;
  }
}

DataType ListElementGetValue(ListElement *element)
{
  assert(element) ;
  return element->data ;
}

/*------------------------------------------------------------------*/
/* List implementation */
/* *------------------------------------------------------------------*/

List * ListCreate()
{
  List* list = malloc(sizeof(List)) ;
  if (list) {
    list->next = NULL ;
  }
  return list ;
}
void ListDelete(List *list) {
    assert(list);
    ListClear(list);
    free(list);
}

void ListClear(List *list) {
    assert(list);
    while (!ListIsEmpty(list)) {
        ListRemoveAfterElement(list);
    }
}

int ListIsEmpty(List *list) {
    return (list->next == NULL);
}

ListElement * ListGetFirstElement(List *list) {
    assert(list);
    return list->next;
}

ListElement * ListGetLastElement(List *list) {
    assert(list);
    ListElement * last = list;
    while (last->next) {
        last = last->next;
    }
    return last;
}

ListElement * ListFindElement(List *list, DataType x) {
    assert(list);
    ListElement * current = ListGetFirstElement(list);
    while (current && current->data != x) {
        current = current->next;
    }
    return current;
}

void ListInsertAfterElement(ListElement *element, DataType x)
{ 
    assert(element); 
    ListElement *newElement = malloc(sizeof(ListElement)); 
    assert(newElement); 
    newElement->next = element->next; 
    newElement->data = x; 
    element->next = newElement; 
}

void ListRemoveAfterElement(ListElement *element) 
{ 
    assert(element); 
    ListElement *elementToRemove = element->next; 
    if (elementToRemove) {
        element->next = elementToRemove->next; 
        free(elementToRemove); 
    } 
}

void ListAppend(List *list, DataType x) 
{ 
    assert(list); 
    ListInsertAfterElement(ListGetLastElement(list), x) ; 
}

B Getting started with compiling and running programs

These instructions should help you getting started with entering, compiling, and running your C programs in Windows, Mac OS X, or Linux. Remember to use Google if you are in trouble, or switch to one of the on-line C compilers as suggested in Problem 2.

B.1 Windows and Visual C

Adapted from https://msdn.microsoft.com/en-us/library/ms235629.aspx:

- Download and install the Visual C IDE. Use Microsoft Dreamspark to download a free student copy using your Oxford credentials (https://www.dreamspark.com).
- In the Visual C++ project types pane, click Win32, and then click Win32 Console Application.
- Type a name for the project (e.g. b16-tutorial). By default, the solution that contains the project has the same name as the project, but you can type a different name. You can also type a different location for the project.
- Click OK to create the project.
- In the Win32 Application Wizard, click Next, select Empty Project, and then click Finish.
• If Solution Explorer is not displayed, on the View menu, click Solution Explorer.

• Add a new source file to the project, as follows.
  – In Solution Explorer, right-click the Source Files folder, point to Add, and then click New Item.
  – In the Code node, click C File (.c), type a name for the file, and then click Add. The .c file appears in the Source Files folder in Solution Explorer, and the file is opened in the Visual Studio editor.
  – In the file in the editor, write the code of your program.
  – Save the file.

• On the Build menu, click Build Solution. The Output window displays information about the compilation progress, for example, the location of the build log and a message that indicates the build status.

• On the Debug menu, click Start without Debugging. A command window is displayed with the output of your program.

B.2 Mac OS X and Xcode

• Install Xcode from the Mac App Store.

• Open Xcode.

• Select the menu File→New→Project.... This will show a window similar to the following:
As a template, choose OS X→Application→Command Line Tool.

- Choose a name for your product” (project) and make sure you choose C as the Language. Do not include spaces in the product name.

- This will start Xcode with the following window. On the left, there is a navigator to find and access the project files. On the right information about the selected item (in this case the product” or executable program). In the middle you have the editor, which you can use to change the item.
• Choose the `main.c` file in the navigator. In the editor, paste your program. Check your spelling and then press the play” symbol (triangle) in the top left. This will compile your program and execute it.

• Provided that there are no errors, Xcode will show the output of your program in the bottom right. If there are errors, select them in the navigator, fix them, and try again.
B.3 Linux and GCC

- The most common Linux C/C++ compiler is GCC. Download and install the GCC compiler package for your platform (the exact instructions depend on the specific Linux distribution; from the command line `apt-get install gcc` might work). Make also sure you install a text editor (e.g. `emacs` or `vi` and learn how to use it).

- Open the terminal application and create a new directory for your project. For example

```
$ mkdir b16-tutorial
$ cd b16-tutorial
```

- Create a file for your program, for example `exercise1.c`:

```
$ edit exercise1.c
```

This will invoke the default editor. Instead of `edit`, you can of course use the editor of your preference.

- Save your program and go back to the terminal. Make sure that the `exercise1.c` file contains your text. To do so, you can for example use the `cat` command:

```
$ cat exercise1.c
```

- Run GCC to compile your program

```
$ gcc -o exercise1 exercise1.c
```

The `-o exercise1` option tells GCC that you want the executable program to be called `exercise1` (without extension).

- If there are errors, find them in your program using the editor, fix them, save, and try again.
• Once the compilation completes, you can execute your program. Go back to the terminal and type

```
> ./exercise1
```

Note the `./` at the beginning: this tells Linux to search for the executable in the current directory. Now your program should run.