Tutorial Letter 101/3/2017

Programming: Data Structures

COS2611

Semesters 1 and 2

School of Computing

This tutorial letter contains important information about your module.

BARCODE

CONTENTS

Page

Dear Student

1 INTRODUCTION

Greetings to you and welcome to COS2611. The contents of this module follow on the first-year programming module, COS1512. You will learn to write more complex C++ programs to implement and use data structures and recursion. You will also learn sorting and searching algorithms and how to do complexity analysis of algorithms.

COS2611 is a practical module in the sense that you constantly have to write, debug and run programs on your computer. There is only one way to learn to program and that is to sit down and write programs.

Because this is a blended online module, you need to use myUnisa to study and complete the learning activities for this course. You need to visit the website on myUnisa for COS2611 frequently. The website for your module is COS2611-17-S1 for the first semester and COS2611-17-S2 for the second semester.

Because this is a blended online module, you need to go online to see your study materials and read what to do for the module. Go to the website here: https://my.unisa.ac.za and login with your student number and password. You will see COS2611-17-S1 (semester 1) or COS2611-17-S2 (semester 2) in the row of modules in the orange blocks across the top of the webpage. Remember to also check in the -more- tab if you cannot find it in the orange blocks. Click on the module you want to open.

In addition, you will receive this tutorial letter and a printed copy of the online study materials for your module. While these printed materials may appear to be different from the online study materials, they are exactly the same and have been copied from the online myUnisa website.

We wish you success on your journey!

1.1 Tutorial matter

When you register, you will receive an inventory letter containing information about your tutorial matter. Some of this tutorial matter may not be available when you register. Tutorial matter that is not available when you register will be posted to you as soon as possible, but is also available on *my*Unisa.

The most important tutorial matter for this module that you will receive is:

- Tutorial letter 101 - This tutorial letter.
- DISK 2017 - A CD-ROM containing the software for this module, as well as some source code for the practical work.
- Tutorial Letter 102 - Contains specific information about what to expect in the examination (the so-called *exam tutorial letter*). It will be sent to you later in the semester.
- After the due dates of the assignments you will receive tutorial letters numbered 201 and 202 containing the solutions to the assignments.
- Solutions to self-assessment assignment will be posted on myUnisa.

2 PURPOSE AND OUTCOMES

2.1 Purpose

Students who successfully complete this module will have the knowledge, skills and competencies to apply Data Structures and Algorithm Analysis knowledge and strategies in solving real-world programming problems, according to industry-approved processes within African, South African and global contexts.

The module is delivered via myUnisa, the Internet, peer group interaction, and community engagement in some of the activities. Your lecturers will interact with students on myUnisa and via email.

2.2 Outcomes

For this module, there are several outcomes that we hope you will be able to accomplish by the end of the course:

- **Specific outcome 1:**

Demonstrate an understanding of algorithm analysis and the Big-oh notation used in algorithm analysis**.**

Assessment Criteria:

- Specify a function to indicate the time and space requirements of a (non-recursive) algorithm in terms of the input size.
- Determine the worst case time and space complexity of non-recursive algorithms.

Range: This would involve performing Big-Oh analysis of short code fragments

- **Specific outcome 2:**

Demonstrate an understanding of the basic properties of pointers and linked lists.

Assessment Criteria:

- Explain the basic concepts of linked lists and pointers.
- Implement and manipulate linked lists using a programming language such as C++.
- Translate problems into solutions using concepts.

Range: Singly Linked Lists only

- **Specific outcome 3:**

Demonstrate an understanding of how to use recursion to solve problems and how to think in terms of recursion.

Assessment Criteria:

- Explain the basic concepts using examples.
- Translate problems into solutions using recursion.

- An executable program is developed to solve a problem using recursion.

- **Specific outcome 4:**

Demonstrate an understanding of Abstract Data Types (ADTs) and how they are stored on computers.

Assessment Criteria:

- Explain the basic concepts using examples.
- ADTs are applied in appropriate contexts.
- Implement and manipulate ADTs using a programming language such as C++.
- Translate problems into solutions using ADTs.
- An executable program that implements ADTs is developed to solve real-world problems, using an IDE (Integrated Development Environment).

Range: Queues, Stacks, Graphs and Trees

NB: **The implementations of the graph algorithms are not examinable.** However, you should be able apply the steps and show how the shortest path may be found [see Self-Assessment Assignment B for example]

- **Specific outcome 5:**

Demonstrate an understanding of search techniques used to retrieve data held in data structures.

Assessment Criteria:

- Explain the basic concepts using examples.
- Analyse search algorithms with respect to space and time complexity.
- Implement search algorithms using a programming language such as C++.
- Translate problems into solutions using search algorithms.

Range: Sequential Search and Binary Search

- **Specific outcome 6:**

Demonstrate an understanding of sort techniques used to sort data held in data structures.

Assessment Criteria:

- Explain the basic concepts using examples.
- Analyse sorting algorithms with respect to space and time complexity.
- Implement sorting algorithms using a programming language such as C++.

- Translate problems into solutions using sorting algorithms.

NB: The implementations of the Quick sort and the Merge sort are not examinable. However, you should be able apply the steps and show how an array list will be sorted using these sorting techniques [see Assignment 02 for examples].

Range: Including but not limited to Selection Sort, Insertion Sort, Quick Sort and Merge Sort

3 LECTURER(S) AND CONTACT DETAILS

3.1 Lecturer(s)

The names and contact details of your COS2611 lecturers are supplied in Tutorial Letter COSALLF/301/4/2017. The contact numbers of the lecturers for COS2611 are also available on the module website for COS2611. In the meantime, if you have any academic query about the module and would like to speak to a lecturer, you may do so in one of the following ways:

- Send an email to the COS2611 email addresses: COS2611-17-S1@unisa.ac.za if you are registered for the first semester or COS2611-17-S2@unisa.ac.za if you are registered for the second semester.
- Always have your student number ready when contacting the university.

3.2 Department

Please note that the School of Computing moved to Florida in 2013. The School of Computing can be contacted telephonically at 011 670 9200 or via e-mail using the e-mail address computing@unisa.ac.za. Should you be unable to reach any of the lecturers for COS2611, please leave a message with one of the secretaries, who can be contacted via the number given above. Remember to include the module code and your student number with the message.

3.3 University

Please note that for general (non-academic) queries, you will find contact details in the brochure *myStudies@Unisa* that you received with your study material. It contains important telephone and fax numbers, addresses and other very useful information.

4 RESOURCES

4.1 Prescribed books

MALIK, D.S., Data structures using C++, International Edition, 2nd Edition, Cengage Learning, 2009 (ISBN: 978 143 904 0232).

We refer to this book, in the tutorial letters, as Malik. You should purchase this prescribed text from one of the official university booksellers.

4.2 Recommended books

The additional books listed below can be consulted if you require additional reading matter on this course. The library usually has only one copy of each of these items. Consequently this material may not be readily available.

1. MALIK, D. S., C++ Programming – Program Design including Data Structures, 4th Edition. Thomson Course Technology, 2009.

2. NYHOFF, L., C++: An Introduction to Data Structures. Prentice-Hall, 1999.

3. PREISS, B. R., Data Structures and Algorithms with Object-Oriented Design Patterns in C++. John Wiley & Sons, Inc. 1999.

4. WEISS M.A., Data structures and problem solving using C++, 2nd edition. Addison Wesley, 2000

4.3 Electronic reserves (e-reserves)

None

4.4 Library services and resources information

For brief information, go to www.unisa.ac.za/brochures/studies

For detailed information, go to the Unisa website at http://www.unisa.ac.za/ and click on **Library**.

For research support and services of personal librarians, go to http://www.unisa.ac.za/Default.asp?Cmd=ViewContent&ContentID=7102.

The library has compiled a number of library guides:

- finding recommended reading in the print collection and e-reserves – http://libguides.unisa.ac.za/request/undergrad
- requesting material – http://libguides.unisa.ac.za/request/request
- postgraduate information services – http://libguides.unisa.ac.za/request/postgrad
- finding, obtaining and using library resources and tools to assist in doing research – http://libguides.unisa.ac.za/Research_Skills
- how to contact the library/finding us on social media/frequently asked questions – http://libguides.unisa.ac.za/ask

5 STUDENT SUPPORT SERVICES

Type here The Student Services Bureau of Unisa provides support for students in general academic matters, such as selecting appropriate modules, developing study skills, adapting to distance education, assistance for students with special needs or general difficulties with studies. See COSALLF/301/4/2017 and *myStudies@Unisa* for contact information.

5.1 E-Tutors

Unisa offers online tutorials (e-tutoring) to students registered for modules at NQF level 5 and 6, this means qualifying first year and second year modules.

Once you have been registered for a qualifying module, you will be allocated to a group of students with whom you will be interacting during the tuition period as well as an e-tutor who will be your tutorial facilitator. Thereafter you will receive an sms informing you about your group, the name of your e-tutor and instructions on how to log onto MyUnisa in order to receive further information on the e-tutoring process. If you login into myUnisa you will notice that your group site has been added there. Your tutor will be able to assist you there. For example, if you were allocated to group 4, the group site will be named COS2611-17-S1-4E. You can use the discussion forum to discuss module content issues with your tutor as well as with students belonging to that group. You will also find the contact details of your tutor. If you have content related problems (that is, the problem with the material in your study guide that you do not understand), please contact your e-tutor.

Online tutorials are conducted by qualified E-Tutors who are appointed by Unisa and are offered free of charge. All you need to be able to participate in e-tutoring is a computer with internet connection. If you live close to a Unisa regional Centre or a Telecentre contracted with Unisa, please feel free to visit any of these to access the internet. E-tutoring takes place on MyUnisa where you are expected to connect with other students in your allocated group. It is the role of the e-tutor to guide you through your study material during this interaction process. For you to get the most out of online tutoring, you need to participate in the online discussions that the e-tutor will be facilitating.

5.2 Free computer and internet access

Unisa has entered into partnerships with establishments (referred to as Telecentres) in various locations across South Africa to enable you (as a Unisa student) free access to computers and the Internet. This access enables you to conduct the following academic related activities: registration; online submission of assignments; engaging in e-tutoring activities and signature courses; etc. Please note that any other activities outside of these are for your own costing e.g. printing, photocopying, etc. For more information on the Telecentre nearest to you, please visit **www.unisa.ac.za/telecentres**.

5.3 Downloading study material and software

One of the requirements for studying at the School of Computing is to have regular access to *myUnisa*. You are therefore expected to download any study material from the Internet that, for whatever reason, is not available on paper in time. You may download it from *myUnisa*. The study material is updated regularly, thus you need to check the COS2611 website at least once a week on *myUnisa*.

The ***software*** should also be downloaded from *myUnisa* under *Additional Resources* for COS2611, at once if you do not receive Disk2017 *immediately* after registration. Please note that it is not necessary to download the full contents of the CD. You need Code::Blocks only.

6 STUDY PLAN

Use your *my Studies @ Unisa* brochure for general time management and planning. For this module we recommend that you use the study programme given below as a starting point. **You will probably need to adapt this schedule, taking into account your other modules together with your personal circumstances.** You are expected to spend at least 8 hours per

week on COS2611. Keep in mind that the skills you will learn in this course are progressive, i.e. you will not be able to understand or do the exercises in the later chapters if you have skipped the earlier ones.

Week	Date S1	Date S2	Activity
1	23 Jan	10 July	Install software, Study Chapter 3
2	30 Jan	17 July	Study Appendix A
3	6 Feb	24 July	Study Chapter 5
4	13 Feb	31 July	Study Chapter 6 **Do Assignment 1 and submit it.** **You must submit this assignment to attain EXAM ADMISSION.**
5	20 Feb	7 Aug	Study Chapter 7
6	27 Feb	14 Aug	Study Chapter 8
7	6 Mar	21 Aug	Study Chapter 9
8	13 Mar	28 Aug	Study Chapter 10
9	20 Mar	4 Sep	Study Chapter 11 Do Assignment 2
10	27 Mar	11 Sep	**Submit Assignment 2**
11	3 April	18 Sep	Study Chapter 12
12	10 Apr	25 Sep	Do Self-Assessment Assignment Do NOT submit.
13-15	17 Apr till exam	2 Oct till exam	Revision
until the examination			Revision. Study all study material, including the solutions to the assignments. Read carefully through the examination tutorial letter.

7 PRACTICAL WORK AND WORK-INTEGRATED LEARNING

7.1 Prescribed compiler

The prescribed C++ compiler for COS2611 is the GNU C++ compiler, provided on the CD-ROM that you should have received when you registered for COS2611. The CD-ROM also contains an Integrated Development Environment (IDE) which can be used to develop your programs.

After inserting the CD-ROM into the CD-ROM drive of your computer, do one of the following:

- Click on **Run...** on the **Start** menu. In the dialog box that appears, type d:index.html and click on the **OK** button.
- Double-click on the **My Computer** icon on your desktop. In the window that appears, double-click on the CD-ROM icon (**D:**); then double-click on the file index.html.
- Load **Windows Explorer** and locate index.html on the CD-ROM drive. Double-click on this file.

After doing any one of the above, the file index.html should be loaded into **Internet Explorer** (or whatever web browser is installed on your computer). Click on the link for COS2611.

Note that the prescribed compiler is the one used for COS1511 and COS1512 and can be downloaded from myUnisa under additional resources.

7.2 Microsoft Visual C++

We are aware that some students have access to other compilers, but it is unfortunately impossible for us to accept and mark assignments prepared with your favourite compiler. The only exception that we make is for Microsoft Visual C++. In other words, we will accept assignments completed by using Microsoft Visual C++. Note that we will not be able to give you any technical support if you use Microsoft Visual C++. You are on your own. If you get stuck, we will simply recommend that you switch to the prescribed compiler.

7.3 Computer Laboratories

If you do not have your own computer facilities, you may use Unisa's computers at the computer laboratories. Copies of the prescribed compiler are available for use at these laboratories at the UNISA regional offices and in Pretoria. Arrangements and regulations regarding the use of the computer laboratories are issued in a separate tutorial letter.

7.4 Errata

7.4.1 Consider the following error that you may come across when compiling code from Malik:

Compiler: Default compiler

```
Executing  make clean
rm -f testUnorderedLinkedList.o  Project1.exe

g++.exe -c testUnorderedLinkedList.cpp -o testUnorderedLinkedList.o -
I"C:/unisa/devcpp/include/c++"  -I"C:/unisa/devcpp/include/c++/mingw32"  -
I"C:/unisa/devcpp/include/c++/backward"  -I"C:/unisa/devcpp/include"

In file included from testUnorderedLinkedList.cpp:3:
UnorderedLinkedList.h: In member function `bool
unorderedLinkedList<Type>::search(const Type&) const':
UnorderedLinkedList.h:45: error: `first' undeclared (first use this function)
UnorderedLinkedList.h:45: error: (Each undeclared identifier is reported only once
for each function it appears in.)

mingw32-make.exe: *** [testUnorderedLinkedList.o] Error 1

Execution terminated
```

Consider the `UnorderedLinkedListType` of Malik. Although `UnorderedLinkedListType` is derived from `linkedListType`, the compiler does not associate the inherited data members such as `first` and `count` as members of the derived class. The problem can be solved by

replacing the all references to `first`, `last` or `count` with **`this->first`**, or **`this->count`** and **`this->last`.**

For examination purposes inserting the `this->` in front of inherited data members is not necessary.
The following paragraph is not examinable and merely included for those students who are interested in knowing why error (a) occurs.

When working with derived classes the compiler does not associate any inherited data members as members of the derived class. The C++ standard says that unqualified names in a template are generally non-dependent and must be looked up when the template is defined. Since the definition of a dependent base class is not known at that time, unqualified names are never resolved to members of the dependent base class. Where names in the template are supposed to refer to base class members or to indirect base classes, they can be made dependent by qualifying them

7.4.2 This type of error occurs when friend functions are used in conjunction with template classes.

```
MyList.h:18: warning: friend declaration  'std::ostream& operator<<(std::ostream&,
const MyList<Type>&)' declares a non-template function
```

You need to insert <> after the `operator<<` in the class definition only. This should get rid of the warning above.

Please note that if you had switched to the compiler (**mingw_gcc3.4.2.exe)** which was supplied to you this year then you will need to add the statements in **bold** below (or similar statements depending on the class type) to any template class that overloads the `ostream` operator.

```
#ifndef H_MyList
#define H_MyList

#include <iostream>
#include <cassert>
using namespace std;

template<class Type>
class MyList;

template<class Type>
ostream& operator<<(ostream&, constlinkedListType<Type>&);


{//rest of the class...}
```

The friend declaration requires this additional syntax - the compiler will not be able to associate the friend function with the class. As the class `MyList` is not fully declared yet. These lines of code known as **incomplete declarations** are used to inform the compiler of the existence of a class or function.

8 ASSESSMENT

8.1 Assessment criteria

To obtain exam admission, **you must submit assignment 1** by **23 February** if you are registered for the first semester and **04 August** if you are registered for the second semester.

- For each assignment you will need a unique assignment number, which you will find in section 8.3.2 with the assignment questions.
- When you submit through *myUnisa* you will be asked to enter this number.
- Please refer to the brochure *myStudies@Unisa* for instructions regarding completing assignments.

Assignment 1 is a multiple choice assignment.

- Students may submit assignments completed on mark-reading sheets **either** by Mobile MCQ submission **or** electronically via myUnisa. **There is no extension to this assignment.**

Assignment 2 is a practical assignment.

- We urge you to do and submit the assignment; otherwise you will find it very difficult in the examination.
- This assignment should be submitted in **PDF format**.

8.2 Assessment plan

This module covers basic data structures and algorithms. The data structures include queues, stacks, linked lists, graphs, trees and binary search trees. You will learn to implement and use them. You will also learn about complexity analysis, recursion and sorting and searching algorithms.

The syllabus covers the following selected topics and chapters of the prescribed textbook:

Algorithm Analysis: After studying **Chapter 1** of *Malik* and Appendix A, you should be able to:

Specify a function to indicate the time and space requirements of a (non-recursive) algorithm in terms of the input size.

- Determine the worst case time and space complexity of non-recursive algorithms.

Pointers: After studying **Chapter 3** you should be able to:

- Declare and manipulate pointer variables.
- Use the new and delete operators to manipulate dynamic variables.
- Use dynamic arrays.
- Distinguish between shallow and deep copies of data.
- Implement classes with pointer data members.

STL: After studying **Chapter 4** you should be able to:

- Apply STL components such as containers, iterators and algorithms.

(Read only – Not Examinable)

Linked Lists: After studying **Chapter 5** you should be able to:

- Explain the basic concepts of linked lists.
- Implement insertion and deletion operations on linked lists.
- Implement and manipulate a linked list.
- Implement and manipulate an ordered list.
- Apply the STL list container.

Recursion: After studying **Chapter 6** you should be able to:

- Understand recursive functions.
- Implement recursive functions to solve problems.

Stacks: After studying **Chapter 7** you should be able to:

- Explain the basic concepts of stacks.
- Implement a stack using an array.
- Apply the STL stack container.

Queues: After studying **Chapter 8** you should be able to:

- Explain the basic concepts of queues.
- Implement a queue using an array.
- Apply the STL queue container.

Search Algorithms: After studying **Chapter 9** you should be able to:

- Implement and analyze the sequential search algorithm.
- Implement and analyze the binary search algorithm.

Sorting Algorithms: After studying **Chapter 10** you should be able to:

- Implement and analyze the selection sort algorithm.
- Implement and analyze the insertion sort algorithm.
- Apply the quick sort algorithm.
- Apply the merge sort algorithm.

Binary Trees: After studying **Chapter 11** you should be able to:

- Explain the basic concepts of binary trees.

- Apply binary tree traversals.
- Implement a binary tree.
- Implement a binary search tree.
- Analyze binary search trees.

Graphs: After studying **Chapter 12** you should be able to:

- Explain the basic concepts of graph theory.
- Know how to represent a graph as an ADT.
- Explain and apply the breadth-first traversal algorithm.
- Explain and apply the shortest path algorithm.

Note that all concepts or sections which are part of the selected chapters above which are not listed above are **not examinable**. You do not have to for instance study hashing which is part of the chapter on sorting.

8.3 Assignment numbers

8.3.1 General assignment numbers

Semester Period	**Assignment Number**
Semester 1	Assignment 01
	Assignment 02
Semester 2	Assignment 01
	Assignment 02

8.3.2 Unique assignment numbers

Semester Period	**Assignment Number**	**Unique Number**
Semester 1	Assignment 01	832599
	Assignment 02	802356
Semester 2	Assignment 01	583432
	Assignment 02	835635

8.4 Assignment due dates

Semester Period	Assignment Number	Due Date
Semester 1	Assignment 01	23 February 2017 No Extension
	Assignment 02	31 March 2017
Semester 2	Assignment 01	04 August 2017 No Extension
	Assignment 02	12 September 2017

8.5 Submission of assignments

Students must submit assignment 1 (completed on a mark-reading sheet) **either** by Mobile MCQ submission on your cell phone **or** electronically via *myUnisa* and assignment 2 (as a .pdf file) via *myUnisa.* No assignments in the wrong format can be accepted.

The marks that you obtain for Assignments 1 and 2 form the semester mark for COS2611. The semester mark forms **20%** of the final mark for the module. Assignment 1 contributes 25% and Assignment 2 contributes 75% towards the semester mark.

Note: The self-assessment assignments A and B do not contribute towards the semester mark.

8.6 The assignments

Assignment 01 [For First Semester Students Only]

Due date:	**23 February 2017 (No Extension)**
Tutorial matter:	**Algorithm Analysis (Chapter 1 + Appendix A)**
Maximum marks:	**30**
Unique assignment number:	**832599**

For each question, you are required to identify the letter of the choice that best completes the statement or answers the question.

For questions 1 to 19 give the algorithmic complexity of code fragment provided

Question 1

```
for (int i = 10000; i > 0; i--)
   sum++;
```

1. O(1)
2. O(n)
3. O(log n)
4. O(n log n)

Question 2

```
int sum = 0;
int i = 20000;
while (i > 0)
{
    sum += i;
    i--;
}
```

1. O(1)
2. O(log n)
3. O(n)
4. O(n log n)

Question 3

```
for (int i = 0; i < 5n; i++)
   sum++;
```

1. O(1)
2. O(5n Log n)
3. O(5n)
4. O(n)

Question 4

```
for (int i = 0; i < n*n; i++)
   sum++;
```

1. O(1)
2. $O(n^2)$
3. O(2n)
4. O(n)

Question 5

```
int sum = 0;
int i = 2;
while (i < 1000)
{
    sum += i;
    i++;
}
```

1. O(1)
2. O(log n)
3. O(n)
4. O(n log n)

Question 6

```
for (int i = 0; i < 2000; i++)
   for (int j = 2; j < n; j++)
        sum++;
```

1. O(n)
2. $O(n^4)$
3. $O(n^3)$
4. O(n log n)

Question 7

```
for (int i = 1; i < n; i*=2)
   for (int j = 0; j < 1000; j++)
        sum++;
```

1. O(1)
2. $O(n^2)$
3. O(n log n)
4. O(log n)

Question 8

```
for (int i = 1; i < n; i*=2)
   for (int j = 0; j <n; j++)
          sum = i + j;
```

1. $O(n^2)$
2. $O(n^3)$
3. O(log n)
4. O(n log n)

Question 9

```
for (int i = 0; i < n; i++)
   sum++;
for (int j = 0; j < n; j++)
   sum++;
```

1. $O(n^2)$
2. O(log n)
3. O(n)
4. O(n log n)

Question 10

```
for (int i = 1; i < n; i*=2)
   sum++;
for (int j = 0; j < n*n; j++)
   sum++;
```

1. $O(n^2)$
2. O(log n)
3. O(n)
4. O(n log n)

Question 11

```
for (int i = n; i > 1; i /=2)
   sum++;
for (int i = 1; i < n; i = i*2)
   sum++;
```

1. O(1)
2. $O(n^2)$
3. O(log n)
4. O(n log n)

Question 12

```
int i;
for (i = 0; i < n*n*n; i++)
   for (int j = 0; j <i; j++)
        sum++;
```

1. O(n)
2. $O(n^4)$
3. $O(n^3)$
4. $O(n^6)$

Question 13

```
int i = 1;
while (i<=n){
   for (j=1; j<10; j++)
        doIt();
   i++;}
```

where `doIt()` has runtime of log n.

1. $O(n^4)$
2. $O(n^2)$
3. O(log n)
4. O(n log n)

Question 14

```
for (int i = 1; i < n; i++);

int j =n;
while (j >0)
{
   sum++;
   j =j/2;
}
```

1. O(n)
2. $O(n^2)$
3. $O(n^3)$
4. O(n log n)

Question 15

```
int i = 1;
while (i<=n){
   for (int j=1; j<10; j++)
        i++;
}

doIt();
```

where `doIt()` has runtime of log n.

1. O(n)
2. $O(n^2)$
3. O(log n)
4. O(n log n)

Question 16

```
for (int i = 1; i <=n/2; i++)
   for (int j = 1; j <= n; j++)
        sum++;
```

1. O(n)
2. $O(n^2)$
3. O(log n)
4. O(n log n)

Question 17

```
for (int i = n; i > 1; i= i/2)
   sum++;
```

1. O(n)
2. $O(n^2)$
3. O(log n)
4. O(n log n)

Question 18

```
int i =1;
int j =1;
while (i < n)
{
    while (j < n)
    {
      j++;
    }


    i++;
    sum++;
}
```

1. O(n)
2. $O(n^2)$
3. O(log n)
4. O(n log n)

Question 19

```
int i =1;
int j =n;
while (i < n)
{
    i++;
    sum++;
}
while (j > 1)
{
   j/=2;
}
```

1. O(n)
2. O(n^2)
3. O(log n)
4. O(n log n)

Question 20

An algorithm takes 20 seconds for an input size of 10. How long will it take for an input size of 100 if the running time is O(n log n)?

1. 8 sec
2. 400 sec
3. 40 sec
4. 200 sec

Question 21

An algorithm takes 5 seconds for an input size of 10. How long will it take for an input size of 20 if the running time is O(n^3)?

1. 5 sec
2. 35 sec
3. 15 sec
4. 40 sec

Question 22

An algorithm takes 2 seconds for an input size of 4. How long will it take for an input size of 8 if the running time is O(2^n)?

1. 8 seconds
2. 32 seconds
3. 16 seconds
4. 64 seconds

Question 23

An algorithm takes 4 seconds for an input size of 100. How large a problem can be solved in 25 seconds if the running time is O(n^2)?

1. input size of 150
2. input size of 200
3. input size of 250
4. input size of 350

Question 24

An algorithm takes 5 seconds for an input size of 500. How large a problem can be solved in 50 seconds if the running time is linear O(n)?

1. input size of 150
2. input size of 250
3. input size of 500
4. input size of 5000

Question 25
Which of the following functions has the slowest growth rate (for n > 1)?

1. n^2
2. $n^3/100$
3. log n
4. n^3

Question 26
Which of the following functions has the fastest growth rate (for n > 1)?

1. n^2
2. n log n
3. n^3
4. log n + 220

Question 27
Which of the following functions is ordered by growth rate from largest to smallest?

1. n log n; 2^n ; n^3; 2^4
2. 2^n; n^3; n log n; 2^4
3. 2^4; n log n; n^3; 2^n
4. 2^n ; 2^4; n^3; n log n;

Question 28
Which of the following functions is ordered by growth rate from smallest to largest?

1. 10 log n; n log n; $2n^2$
2. $2n^2$; n log n; 10 log n
3. n log n; $2n^2$; 10 log n
4. n log n; 10 log n; $2n^2$

Consider the table below when answering questions 29 and 30. The table shows the running time against input size (n) of 2 algorithms

Algorithm	**n = 2**	**n = 6**	**n = 12**
A	10 sec	10 sec	10 sec
B	4 sec	64 sec	4096 sec

Question 29
What is the running complexity of algorithm A?

1. O(1)
2. $O(n^2)$
3. $O(2^n)$
4. O(n log n)

Question 30
What is the running complexity of algorithm B?

1. $O(n^2)$
2. $O(n^2)$
3. $O(2^n)$
4. O(n log n)

Assignment 02 [For First Semester Students Only]

Due date:	**31 March 2017**
Tutorial matter:	Linked Lists (Chapter 5) Recursion (Chapter 6) Stacks (Chapter 7) Queues (Chapter 8) Searching (Chapter 9) Sorting(Chapter 10) Trees(Chapter 11)
Unique assignment number:	**802356**

Question 1: Linked Lists

1.1 Do exercise 4 on page 345 of Malik.

1.2 Do programming exercise 2 on page 349 of Malik.

Question 2: Recursion

2.1 Do exercise 1 on page 390 of Malik.

2.2 Do exercise 4 on page 391 of Malik.

Question 3: Stacks

3.1 The language $L=\{a^n bb^n\}$ where $n \geq 1$, is a language of all words with the following properties:

- The words are made up of strings of a's followed by b's.
- The number of a's is always **equal** to the number of b's plus one b.
- Examples of words that belong to L are

  abb, where n=1;
  aabbb, where n=2;
  aaabbbb, where n=3;
  aaaabbbbb, where n=4.

One way to test if a word w belong to this language is to use a stack to check if the number of a's balances the number of b's. Use the provided header and write a function isInLanguageL that uses a stack to test if any word belongs to L.

```
bool isInLanguageL(string w);
```

Note the following:

- Only words belonging to L should be accepted.
- The word bba does not belong to L.

3.2 The language $L=\{a^n b^n c^n\}$ where $n \geq 1$, is a language of all words with the following properties:

- The words are made up of strings of a's followed by b' followed by c's.
- The number of a's is always **equal** to the number of b's and c's (n is the same).
- Examples of words that belong to L are

  abc, where n=1;
  aabbcc, where n=2;
  aaabbbccc, where n=3;
  aaaabbbbcccc, where n=4.

One way to test if a word w belong to this language is to use two stacks to check if the number of a's balances the number of b's and c's. Use the provided header and write a function isInLanguageL that uses stacks to test if any word belongs to L.

```
bool isInLanguageLc(string w);
```

Hint: Push all a's into the first stack and b's into the second stack. Pop both stacks for every c's that is in the string w.

Question 4: Queues

4.1 Write a function reverseQ that uses a local stack to reverse the contents of a queue. Use the following header:

```
template < class Type >
void reverseQ(queueType< Type >&q )
```

You can make use of any member function of class queueType. Note that this function is not a member of class queueType**.**

4.2 Do question 3.1 using the following header:

```
bool isInLanguageL(queueType<char> & w);
```

Question 5: Searching

Describe how the binary search algorithm searches for 4 in the following list:
5, 6, 8, 12, 15, 21, 25, 31.

Question 6: Sorting

Consider the following sequence of numbers 10, 9, 2, 5, 8, 3, 1, 4, 7, 6

6.1 Sort the list using selection sort. Show the state of the list after each call to the swap procedure.

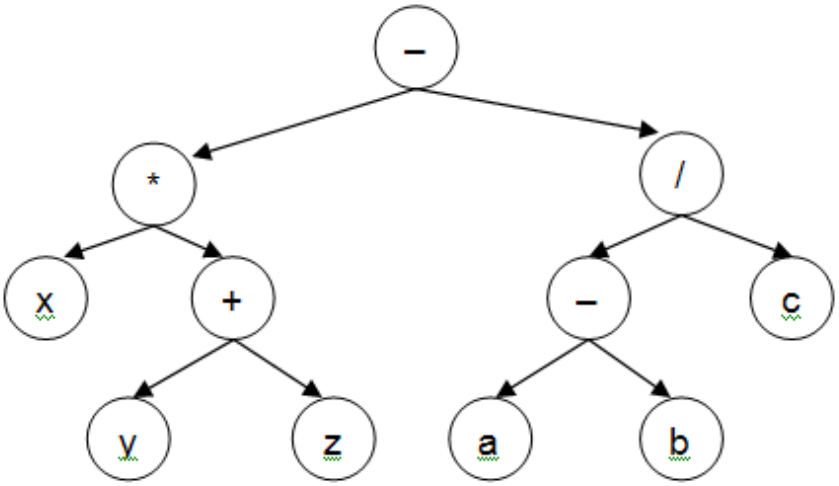
6.2 Sort the list using insertion sort. Show the state of the list after each iteration of the outer for loop of the algorithm.

6.3 Sort the list using quick sort with the middle element as pivot. Show the state of the list after each call to the `partition` procedure.

You are not required to write code for this question. You need to trace through the different sorting algorithms using the given list.

Question 7: Trees

Trees can be used to express formulae. For the tree below, determine the result of the pre-order, in-order, and post-order traversals. The edges showing empty sub-trees have not been included.



Assignment 01 [For Second Semester Students Only]

Due date:	**04 August 2017 (No Extension)**
Tutorial matter:	**Algorithm Analysis (Chapter 1 + Appendix A)**
Maximum marks:	**30**
Unique assignment number:	**583432**

For each question, you are required to identify the letter of the choice that best completes the statement or answers the question.

For questions 1 to 19 give the algorithmic complexity of code fragment provided.

Question 1

```
total = 0;
for (int i = 500; i < 2016; i++)
   total += i;
```

1. O(1)
2. O(log n)
3. O(n)
4. O(n log n)

Question 2

```
int sum = 0;
int i = 2;
while (i < 5)
{
    sum += i;
    i++;
}
```

1. O(1)
2. O(log n)
3. O(n)
4. O(n log n)

Question 3

```
for (int i = 0; i < 5n; i+= 2)
   sum++;
```

1. O(1)
2. O(n)
3. O(5n)
4. O(n log n)

Question 4

```
for (int i = 0; i < 5n*n; i++)
        sum++;
```

1. O(1)
2. O(n)
3. O(6n)
4. $O(n^2)$

Question 5

```
int i = 1;
while (i < n*n)
{
    sum += i;
    i++;
}
```

1. O(1)
2. O(log n)
3. $O(n^3)$
4. $O(n^2)$

Question 6

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < 3000; j++)
        count++;
```

1. O(1)
2. O(n)
3. $O(n^2)$
4. O(n log n)

Question 7

```
for (int i = 0; i < n; i+=2)
   for (int j = 0; j <n; j++)
          sum = i + j;
```

1. $O(n^2)$
2. $O(n^3)$
3. $O(n^4)$
4. O(n log n)

Question 8

```
for (int i = 0; i < n; i+=2)
   for (int j = n; j >1; j/=2)
          sum = i + j;
```

1. $O(n^2)$
2. $O(n^3)$
3. O(log n)
4. O(n log n)

Question 9

```
for (int i = 0; i < n; i++)
   sum++;
for (int j = 1; j < n; j*=2)
   sum++;
```

1. $O(n^2)$
2. O(log n)
3. O(n)
4. O(n log n)

Question 10

```
for (int i = 1; i < n; i*=2)
   sum++;
for (int j = 0; j < n; j++)
   sum++;
```

1. $O(n^2)$
2. O(log n)
3. O(n)
4. O(n log n)

Question 11

```
for (int i = 0; i < 60000; i*=2)
   sum++;
```

1. O(1)
2. $O(2^n)$
3. O(n)
4. $O(n^2)$

Question 12

```
int i;
for (i = 0; i < n*n; i++)
   for (int j = 0; j <i; j++)
        sum++;
```

1. O(1)
2. O(n)
3. $O(n^2)$
4. $O(n^4)$

Question 13

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
        total++;
for (int k = 0; k < n*n*n; k++)
         total--;
```

1. $O(3^n)$
2. $O(n^3)$
3. $O(3n^2)$
4. O(3 log n)

Question 14

```
int j;int i = 1;
while (I <= n){
   for (j = 1; j < 10; j++)
     sum +=j;

i*=2;}
doIt();
```

where `doIt()` has runtime of n^2.

1. O(n)
2. $O(n^2)$
3. O(log n)
4. O(n log n)

Question 15

```
int i;
for (int i = 1; i < n; i++)
   for (int j=i; j > 0; j/=2)
        sum++;
```

1. O(n log n)
2. O(n)
3. $O(n^2)$
4. O(log n)

Question 16

```
int i = 1;
while (i<=n){
   for (int j=1; j<10; j++)
        i++;
doIt();
}
```

where `doIt()` has runtime of n.

1. $O(n^2)$
2. $O(n^4)$
3. $O(2^n)$
4. $O(4^n)$

Question 17

```
for (int i = n; i > 1; i=i/2)
   sum++;
```

1. O(log n)
2. O(n/2)
3. O(n)
4. $O(2^n)$

Question 18

```
int i = 1;
int j = n;
while (i<n)
{
    while (j>0)
    {
      j--;
    }

    i++;
    sum++;
}
```

1. O(n)
2. $O(n^2)$
3. O(log n)
4. O(n log n)

Question 19

```
int i =1;
int j =n;
while (i<n^2)
{
    i++;
    sum++;
}
while (j>0)
{
    j/=2;
}
```

1. O(n)
2. $O(n^2)$
3. O(log n)
4. O(n log n)

Question 20

An algorithm takes 5 seconds for an input size of 20. How long will it take for an input size of 50 if the running time is O(n log n)?

1. 4 seconds
2. 12 seconds
3. 16.32 seconds
4. 18.24 seconds

Question 21

An algorithm takes 2 seconds for an input size of 10. How long will it take for an input size of 11 if the running time is $O(2^n)$?

1. 2 seconds
2. 30 seconds
3. 4 seconds
4. 64 seconds

Question 22

An algorithm takes 10 seconds for an input size of 1000. How large a problem can be solved in 80 seconds if the running time is cubic $O(n^3)$?

1. 800
2. 2000
3. 1250
4. 30000

Question 23

An algorithm takes 5 seconds for an input size of 500. How large a problem can be solved in 40 seconds if the running time is linear O(n)?

1. 4
2. 40
3. 400
4. 4000

Question 24
An algorithm takes 2 seconds for an input size of 40. How large a problem can be solved in 100 seconds if the running time is quadratic $O(n^2)$?

1. 282.84
2. 8000
3. 200
4. 40^2

Question 25
Which of the following functions has the fastest growth rate (for n > 1)?

1. n^3
2. n log n
3. log n
4. n^2

Question 26
Which of the following functions has the slowest growth rate (for n > 1)?

1. n log n
2. n^2
3. 2^n
4. log n + 1

Question 27
Which of the following functions is ordered by growth rate from smallest to largest?

1. 10 log n; n log n; $2n^2$
2. $2n^2$; n log n; 10 log n
3. n log n; $2n^2$; 10 log n
4. n log n; 10 log n; $2n^2$

Question 28
Which of the following functions is ordered by growth rate from largest to smallest?

1. n log n; 2^n ; n^3; 2^4
2. 2^n; n^3; n log n; 2^4
3. 2^4; n log n; n^3; 2^n
4. 2^n ; 2^4; n^3; n log n;

Consider the table below when answering questions 29 and 30. The table shows the running time against input size (n) of 2 algorithms.

Algorithm	**n=4**	**n=8**	**n=16**
A	16 sec	64 sec	256 sec
B	64 sec	512 sec	4096 sec

Question 29
What is the running complexity of algorithm A?

1. n
2. n^2
3. 2^n
4. n log n

Question 30
What is the running complexity of algorithm B?

1. O(n)
2. $O(n^2)$
3. $O(n^3)$
4. O(log n)

Assignment 02 [For Second Semester Students Only]

Due date:	**12 September 2017**
Tutorial matter:	Linked Lists (Chapter 5) Recursion (Chapter 6) Stacks (Chapter 7) Queues (Chapter 8) Sorting(Chapter 10) Trees(Chapter 11)
Unique assignment number:	**835635**

Question 1: Linked lists

1.1 Do exercise 5 on page 345 of Malik.

1.2 Do programming exercise 3 on page 349 of Malik.

Question 2 Recursion

2.1 Do exercise 2 on page 390 of Malik.

2.2 Do exercise 5 on page 391 of Malik.

Question 3: Stacks

3.1 The language $L=\{a^n b^{n-1}\}$ where $n \geq 1$, is a language of all words with the following properties:

- The words consist of strings of a's followed by b's.
- The number of b's is **one less** than the number of a's.
- Examples of words that belong to L are

  a, where n=1;
  aab, where n=2;
  aaabb, where n=3;
  aaaabbb, where n=4.

One way to test if a word w belong to this language L is to use a stack to check if the number of a's balances the number of b's. Use the following header and write a function `isInLanguageL2` that uses a stack to test if any word belongs to L. If w belongs to L then the `isInLanguageL2` should return `true` otherwise `isInLanguageL2` should return `false`.

```
bool isInLanguageL2(string w);
```

Note the following:

- Only words belonging to L should be accepted.
- The word bba does not belong to L.

3.2 The language $L=\{a^n b^n c^{n+1}\}$ where $n \geq 1$, is a language of all words with the following properties:

- The words are made up of strings of a's followed by b' followed by c's.
- The number of a's is always **equal** to the number of b's.
- There is an **extra** c for every number of a's and b's.
- Examples of words that belong to L are

  abcc, where n=1;
  aabbccc, where n=2;
  aaabbbcccc, where n=3;
  aaaabbbbccccc, where n=4.

One way to test if a word w belong to this language is to use two stacks to check if the number of a's balances the number of b's and c's. Use the provided header and write a function isInLanguageLc that uses stacks to test if any word belongs to L.

```
bool isInLanguageLc(string w);
```

Hint: Push all a's into the first stack and b's into the second stack. Pop both stacks for every c's that is in the string w. When both stacks are empty, there should be only one c left.

Question 4: Queues

4.1 Do programming exercise 4 on page 495 of Malik.

4.2 Do question 3.1 using the following header:

```
bool isInLanguageL(queueType<char> & w);
```

Question 5: Searching

Describe how the binary search algorithm searches for 6 in the following list:
2, 6, 7, 12, 15, 21, 25, 31, 32, 36.

Question 6: Sorting

Consider the following sequence of numbers 10, 2, 9, 1, 8, 4, 7, 5, 3, 6.

6.1 Sort the list using selection sort. Show the state of the list after each call to the swap procedure.

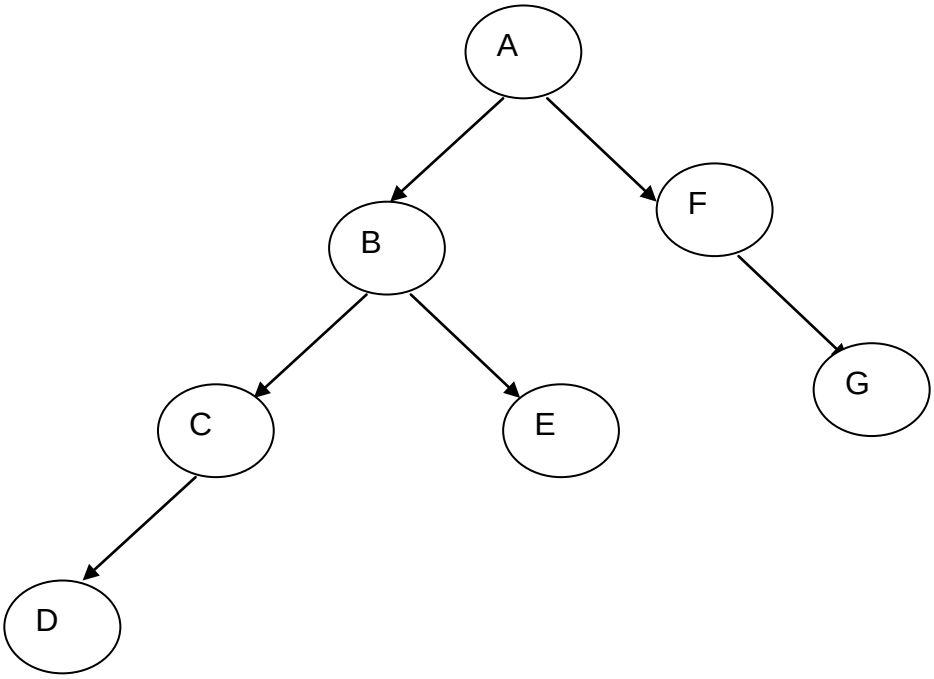
6.2 Sort the list using insertion sort. Show the state of the list after each iteration of the outer for loop of the algorithm.

6.3 Sort the list using quick sort with the middle element as pivot. Show the state of the list after each call to the `partition` procedure.

You are not required to write code for this question. You need to trace through the different sorting algorithms using the given list.

Question 7: Trees

List the node of this binary tree an *in-order*, *pre-order* and *post-order* sequence.



Self-Assessment Assignment A [Both Semesters]

Tutorial matter: Stacks, Queues, Searching, Binary Trees and Graphs

Question 1
Do exercises 7 and 8 on page 199 of *Malik*

Question 2
Do exercises 2, 3, 4, 5 and 6 on page 344-345 of *Malik*

Question 3
Do programming exercise 6 on page 350 of *Malik*

Question 4
Do exercise 8 on page 388 of *Malik*

Question 5
Do programming exercise 9 on page 391 of *Malik*

Question 6
Write a program that reads a sequence of integers from `cin` and writes yes on `cout` if that sequence of integers is a palindrome and writes *no* if it is not. Recall that a palindrome is a sequence that is the same when read from left to right or from right to left. So the sequence 44 12 64 32 32 64 12 44 is a palindrome, while the sequence 44 12 64 32 32 65 12 44 is not.

Here's a catch: You may only use Stacks or Queues as defined above, in addition to simple scalar variables (`int`, `bool`, etc.). You may not use arrays, linked lists, trees or other data structures.

Question 7
Quicksort is claimed to have an expected running time of O(n log n), but it could be as slow as $O(n^2)$.

7.1 Briefly explain why Quicksort could use $O(n^2)$ time instead of always running in time O(n log n).

7.2 How can you fix Quicksort so the expected time is O(n log n), if it can be done? You should give a specific suggestion (don't just say something like "be clever and careful". Explain why your solution will change the expected time to O(n log n).

Question 8
Implement function *smallest* which returns the smallest value found in the Binary Search tree whose root is given as the function argument. For full credit, your function should **visit only those nodes in the tree** necessary to find the smallest one, i.e., it should not visit a node if it does not have to. You should assume the tree has at least one node (i.e., you do not have to figure out what to do if you are asked to find the smallest element in an empty tree).

Self-Assessment Assignment B [Both Semesters]

Tutorial matter: Stacks, Queues, Searching, Binary Trees and Graphs

Question 1: Stacks

Do programming exercises 6 and 9 on page 391 of *Malik*

Question 2: Queues

Do programming exercise 5 on page 495 of *Malik*

Question 3: Searching

Do exercise 4 on page 528 of *Malik*

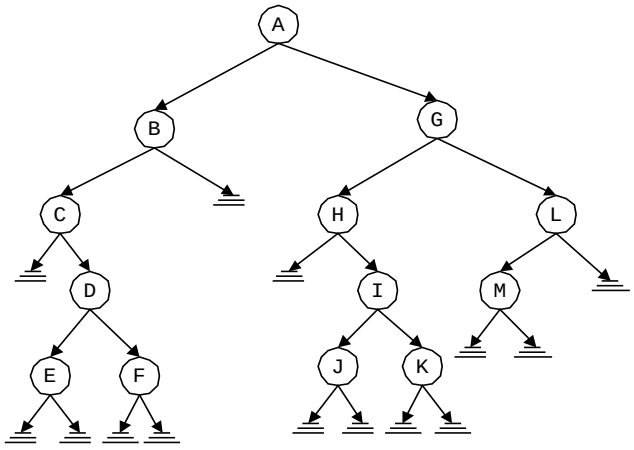
Question 3: Searching

Do programming exercises 1, 2 and 3 on page 530 of *Malik*

Question 5: Binary search trees

Give the

5.1 preorder,
5.2 inorder,
5.3 and postorder
sequences of the following binary search tree.



Question 2: Binary trees

A binary tree has ten nodes. The inorder and preorder traversal of the tree are shown below. Draw the tree.

Preorder : JCBADEFIGH
Inorder : ABCEDJFGIH

Question 3: Binary search trees

3.1 Show the result of inserting 5, 3, 8, 1, 9, 4, 2, 7, 6 into an initially empty binary search tree.

3.2 Show the result of deleting the root from the tree.

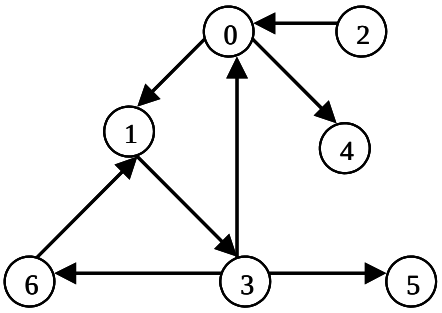
Question 4: Binary search tree

Write a definition of the function `nodeCount` that returns the number of nodes in a binary tree. Add this function to the class `BinaryTreeType` and create a program to test this function.

Question 5: Binary search trees

The `Findkth`() function returns the k^{th} highest element in a binary search tree. Write an **iterative** version of `Findkth`().

Question 6: Graphs



6.1 Find the adjacency matrix of the graph.

6.2 Draw the adjacency list of the graph.

6.3 List the nodes of the graph in breath-first traversal, starting from node 0.

8.7 The examination

There will be a written examination for this module. The examination mark will be converted to a mark out of **80** to which your semester mark out of **20** will be added to get your final percentage for the examination.

The table below is an example of how the **final mark** is calculated if you have submitted all the assignments for the module and written the final exam.

Assignment	Maximum Marks	Mark achieved	Percentage	Weight	Semester mark contribution
1	30	20	67%	0.25	16.8%
2	50	35	70%	0.75	52.5%
Total marks achieved		55	Total semester mark		69.3%
Final mark for the module (Exam mark is 65%) = (Semester mark x 20/100) + (Exam mark x 80/100) = (69 x 0.2) + (65 x 0.8) = 14 + 52 = 66					Final Mark 66

9 IN CLOSING

Do not hesitate to contact your e-tutor by email if you are experiencing problems with the content of this tutorial letter or any aspect of the module. If your e-tutor is unable to assist you, he will contact the lecturer and provide feedback to you.

I wish you a fascinating and satisfying journey through the learning material and trust that you will complete the module successfully.

Enjoy the journey!

10 APPENDIX A: INTRODUCTION TO ALGORITHMS ANALYSIS

1. Introduction

An ***algorithm*** is a set of instructions to be followed (usually by a computer) to solve a computational problem. By computational problem, we mean a problem for which a solution has been designed and structured in the form of an algorithm which could be implemented using a suitable programming language, and processed by a given computer.

Generally there can be more than one algorithm to solve a given computational problem and these algorithms can be implemented using different programming languages on different platforms. In order to cross-compare different algorithms, we need to analyze each of them in terms of their ***space*** and ***time complexities***. *Space complexity* refers to the theoretical evaluation of how much space (memory load) is required by the algorithm if processed by the computer, and *time complexity* refers to the time (processing time) taken by the algorithm to complete (i.e. to solve the problem).

The aim of ***algorithm analysis*** is thus to assess the efficiency of an algorithm. Algorithm efficiency entails not only the time required by the computer to process the corresponding program, but also the memory resources required during its execution. However, a formal algorithm analysis is usually performed theoretically without taking into account the underlying architecture on which the corresponding program will be executed. This kind of formalism helps evaluate the performance of the algorithm at design time before taking into account programming language and hardware considerations.

Algorithm analysis is therefore a means by which the programmer may evaluate various algorithms aiming at solving the same problem in order to choose the optimal solution. An algorithm is said to be **optimal** if it is processed ***faster*** (*time complexity*) compared to its counterparts, and utilizes fewer memory resources (*space complexity*) compared to the others.

Once we have an algorithm that correctly solves a problem, both its time and space complexities should be evaluated in order to capture its efficiency compared to other algorithms designed to solve the same problem.

This tutorial letter focuses on how to estimate the time and space complexities (requirements) of an algorithm.

2. Execution time of algorithms

One approach to compare the time efficiency of two algorithms that solve the same problem, is to implement these algorithms in a programming language and run them to compare their time requirements. The difficulties with this approach are that it is dependent upon the computer used, the programming language, the programmer's style and the test data used. When we analyze algorithms, we should employ mathematical techniques that analyze algorithms independently of *specific implementations, computers, or data.*

To analyze algorithms we first count the number of **basic operations** in a particular solution to assess its efficiency. Then, we will express the efficiency of algorithms using growth functions. ***A basic operation*** is an operation that takes one time unit to execute and is independent from the programming language used. One unit time is simply the theoretical time taken by a basic operation to complete on a given computer. In practice this time may vary from one computer to another, according to the performance of the processor.

An algorithm usually consists of basic operations that can be executed by the computer. Basic operations

include among others:

- Assignment operations (e.g.:`today = "Monday"`)
- Arithmetic operations (e.g.:`salary = hoursWorked*hourlyRate`)
- Comparison operations (e.g.:`age == 20`)

2.1. Some simple examples

Example 1: A single operation

```
count = count + 1;
```

takes one unit of time to complete. Its evaluation depends on the unit time (in micro/milliseconds) it takes the computer to execute a basic operation.

If for a given computer, a unit time is for example 1 micro-second, then approximately 1 micro-second is required to complete the operation.

Example 2: A sequence of operations:

```
count = count + 1;(1)     (Cost = 1 unit time)
sum = sum + count;(2)     (Cost = 1 unit time)
```

Total cost = 1 + 1 = 2

Instructions (1) and (2) are both basic operations. As for the first example, it will take approximately 2 unit-times to complete the sequence.

Example 3: A Simple If-Statement

	Operation	**Cost**
`if (n < 0)`	compare	1
`absval = -n`	Assignment	1
`else`		
`absval = n;`	Assignment	1

Total cost = 1+ 1 = 2

For this simple conditional statement, only one branch is taken after the condition has been evaluated.

Example 4: A Simple Loop

	Cost
`i = 1;`	1
`sum = 0;`	1
`while (i <= n) {`	n+1
`  i = i + 1;`	n
`  sum = sum + i;`	n
`}`	

Total Cost = 1 + 1 + (n+1) + n + n
The time required for this algorithm is 3n + 3

2.2. Remarks

(a) Loops: The running time of a loop is at most the running time of the statements inside that loop times the number of iterations.

(b) Nested Loops: The running time of a nested loop containing a statement in the inner most loop is the running time of the statements multiplied by the product of the size of all loops.

(c) Consecutive Statements: The running time is the sum of the running time of each statement.

(d) If/Else: The running time is that of the test instruction plus the larger of the running times of both branches' instructions

In many cases we can isolate a specific operation fundamental to the analysis of the algorithm, and then ignore other basic operations (that have a negligible influence on the overall time complexity, or are the same for all algorithms being considered to solve a particular problem). Examples of such ignorable operations are initialization and incrementation of loop control variables. The chosen basic operation may, for instance, be a very expensive operation compared to the others, or it may be the only operation that really causes a growth of time required. So then we only count the chosen basic operations. Of course we need to be careful to make the right choice of basic operations. In our simple loop of *example 4*, we can safely choose `sum = sum + i` as our basic operation, and use only this statement in our running time calculations.

3. Order-of-Magnitude Analysis and Big-O Notation

Normally, we measure an algorithm's time requirement as a function of its *problem size*. Problem size depends on the application. For example in a sorting algorithm the problem size will be the number of elements we want to sort. The problem size for an algorithm that calculates the n^{th} prime number will be determined by the value of *n*.

If the problem size of algorithm **A** (an arbitrary algorithm) is *n* then we can say, for example, that Algorithm **A** requires $5*n^2$ time units to solve a problem of size *n*. The most important thing to learn is how quickly an algorithm's time requirement grows as a function of the problem size. Algorithm **A** requires a running time that is proportional to $n^{2.}$. An algorithm's proportional time requirement is known as ***growth rate***. We can compare the efficiency of two algorithms by comparing their growth rates.

If Algorithm **A** requires time proportional to **f(n)**, Algorithm **A** is said to be **order f(n),** and it is denoted as **O(f(n)).** The **function f(n)** is called the algorithm's **growth-rate function**. Since the capital O is used in the notation, this notation is known as the **Big O notation**.
If Algorithm **A** requires time proportional to $\mathbf{n^2}$, it is $\mathbf{O(n^2)}$.

Consider the formal definition of the Big-O function, which is also given on page 15 of Malik (2003):

3.1. Definition:

Let *f* and *g* be nonnegative real-valued functions in the input size *n*. We say that *f*(*n*) is Big-O of *g*(*n*), written *f*(*n*) = O(*g*(*n*)), if there exists positive constants *c* and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

Example:
An algorithm requires $n^2–3*n+10$ seconds to solve a problem size of n. If constants c and n_0 exist such that

$c*n^2 > n^2–3*n+10$ for all $n \geq n_0$.

the algorithm is order n^2(In fact, c is 3 and n_0 is 2)

$3*n^2 > n^2–3*n+10$ for all $n \geq 2$.

Thus, the algorithm requires no more than $c*n^2$ time units for $n \geq n_0$,
So it is O**(n^2)**

3.2. Some useful properties of Growth-Rate Functions:

- We can ignore constants in an algorithm's growth-rate function. That is, if $f(n)$ is $O(cg(n))$ for any constant $c>0$, then $f(n)$ is $O(g(n))$.
  e.g.: If an algorithm is $O(5n^3)$, it is also $O(n^3)$.

- We can ignore low-order terms in an algorithm's growth-rate function. That is, if $f(n)$ is $O(an^x+bn^y+cn^z)$ for the constants $a,b,c>0$, and $x>y>z>0$ then $f(n)$ is $O(an^x)$, which is also $O(n^x)$.

e.g.: if an algorithm is $O(n^3+4n^2+3n)$, it is also $O(n^3)$.

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then $f_1(n)\, f_2(n)$ is $O(g_1(n)\, g_2(n))$ and $f_1(n) + f_2(n)$ is $O(g_1(n)+ g_2(n))$.
  e.g.: If an algorithm is $O(n^3)* O(4n^2)$, it is also $O(n^3 *4n^2)$ which is $O(4n^5)$ therefore it is $O(n^5)$.If an algorithm is $O(n^3) + O(4n^2)$, it is also $O(n^3 +4n^2)$ therefore it is $O(n^3)$.

3.3. Calculation Examples

Example 1:
If an algorithm takes 1 second to run with a problem size of 8, what is the time requirement (approximately) for that same algorithm with a problem size of 16?

If its order is:
O(1) T(n) = 1 second

If its order is:
$O(\log_2 n)$ $T(n) = (1*\log_2 16) / \log_2 8 = 4/3$ seconds

If its order is:
O(n) $T(n) = (1*16) / 8 = 2$ seconds

If its order is:
$O(n*\log_2 n)$ $T(n) = (1*16*\log_2 16) / 8*\log_2 8 = 8/3$ seconds

If its order is:
$O(n^2)$ $T(n) = (1*16^2) / 8^2 = 4$ seconds

If its order is:
$O(n^3)$ $T(n) = (1*16^3) / 8^3 = 8$ seconds

If its order is:
$O(2^n)$ $T(n) = (1*2^{16}) / 2^8 = 2^8$ seconds = 256 seconds

Example 2:
In the following code fragment,

```
i =1
sum = 0;
while (i <= n) {
     i = i + 1;
     sum = sum + i;
}
```

the basic operation is sum = sum + i, which has a cost of 1 unit-time and is executed *n* times.

T(n)= n

So, the growth-rate function for this algorithm is **O(n)**

Example 2:
In the following code fragment,

```
i=1;
sum = 0;
while (i <= n) {
     j=1;
     while (j <= n) {
         sum = sum + i;
         j = j + 1;
}
i = i +1;
}
```

the basic operation is `sum = sum + i`, which has a cost of 1 unit- time and is executed *n*n* times.

T(n) = n*n

So, the growth-rate function for this algorithm is **O(n^2)**

Example 3:
In the following code fragment,

```
for (i=1; i<=n; i++)
  for (j=1; j<=i; j++)
for (k=1; k<=j; k++)
       x=x+1;
```

the basic operation is x=x+1, which has a cost of 1 and is executed n^3 times

T(n) = n^3

So, the growth-rate function for this algorithm is **O(n^3)**

4. What to analyze?

Consider a simple linear search where we are searching for a specific value (`num`) in an array of size *n*:

```
inti=0;
int place=0;
while (num != arr[i] &&i<n)
     i++;
if (i<n)
     place=i;
```

Say we choose the comparison `num != arr[i]`as the basic operation. If `num` happens to be equal to the first element in the array, then the comparison will be made once only. If `num` is not equal to any element of the array, the comparison will be performed *n+1* times. On average, if we know that `num` occurs exactly once in the array, the comparison will be performed *n/2* times. As we can see an algorithm can require different times to solve different problems of the same size.

In general there are three types of algorithm analysis:

Worst-Case Analysis – This is the maximum amount of time that an algorithm requires to solve a problem of size *n*. This gives an upper bound for the time complexity of an algorithm. In the worst case the linear search is *O(n)*. This is achieved when the search item is not in the list or it is the last item in the list.

Best-Case Analysis –This is the minimum amount of time that an algorithm requires to solve a problem of size *n*. In our search the best case is when the search item is the first item in the list. The best case time complexity is then *O(1)*.

Average-Case Analysis –This is the average amount of time that an algorithm requires to solve a problem of size *n*. The linear search is *O(n)* on average. We only focus in this course on Worst-Case analysis of algorithms.

5. Space complexity

The analysis techniques used to measure space requirements are similar to those used to measure time requirements. The asymptotic analysis of the growth rate in the time requirement of an algorithm as a function of the input size, also applies to the measurement of the space requirement of an algorithm. However, time requirements are measured in terms of basic operations on the data structure performed by the algorithm, whereas space requirements are determined by the data structure itself.

If we have an array with n integers and we want to keep the entire array in memory, the space requirements will be O(*n*). If we have a two-dimensional array, then it will be O(n^2). However, if the two-dimensional array is not always filled up, and if we can find a way of only setting aside the memory positions as and when we need them, we may save a lot on memory. In this module you'll study various data structures to accomplish such memory savings and various algorithms that could accomplish the same task, but with different time complexities. As you get acquainted with these data structures, you should also learn how to determine the space complexity of each structure.

In the third-year level *Artificial Intelligence* module, COS3751, you will also encounter algorithms that operate on implicit data structures, called search spaces. These data structures are really huge and space

complexity becomes an important issue when using them, even with today's cheap memory. Search spaces are expanded dynamically; they are never generated and kept in memory in totality. Different orders of expansions yield different space complexities.

6. Big-O Examples

```
//Code Fragment #1
for(int i = 0; i < 2n; i++)
     sum++;
```

The problem size here is the value of `n`. Obviously, as the value of `n` gets bigger, the fragment will take longer to run. Rather than trying to calculate exactly how long it will take to run for a particular value of `n`, we are interested in how quickly the running time of this fragment increases as `n` increases. The amount of time a program takes to run depends on how many basic instructions must be executed. To obtain an estimate of this, we can examine the source code and count how many assignment, increment, or conditional tests the fragment would perform. There are four expressions in Fragment#1:

(a) the initialization `i = 0`
    is performed ***1*** time regardless of *n*
(b) the test `i< 2n`
    is performed ***2n*** + ***1*** times – one more time for the final test when `i = 2n`
(c) the increment `i++`
    is performed ***2n*** times
(d) the increment `sum++`
    is performed ***2n*** times.

So for an input of size ***n***, the number of basic operations this fragment executes is:
6n + 2
The constants in the formula above are not relevant - we are interested in the performance of the fragment for large values of ***n***- so we simply ignore the constant***2***(See page 6 Section 3-2 of Tutorial Letter 102).
So →***6n*** + **2** (ignoring 2)
→***6n*** (ignoring the constant factor)
→***n***

Hence the Big-O Notation of Fragment#1 is ***O(n).***

This tells us, that the running time of the program is proportional to ***n***. That is, if we double the value of ***n*** we can expect the running time to be approximately double as well. This gives us an indication of the scalability of the program – how well it will perform when the value of ***n*** is large.

This kind of analysis is actually quite exhausting. We can often analyze the running time of a program by determining the number of times selected statements are executed. We can usually get a good estimate of the running time by considering one type of statement such as some statement within a looping structure. Provided the loop is iterating sequentially in a linear fashion. Thus if we just consider `sum++` we note that it runs ***2n*** times. Ignoring the constant - the Big O Notation of Fragment#1 is ***O(n)***. This works because Big O is just an estimate of the running time.

```
//Code Fragment #2
for(inti = 0; i< n; i++)
     for(int j = 0; j <i; j++)
          sum++; //Line 3
```

We begin by analyzing the number of times Line 3 is executed. When `i = 1`, Line 3 is executed once. When `i = 2`, Line 3 is executed twice. In general, Line 3 is executed exactly `i` times. Therefore, the total number of executions of Line 3 is: ***1 + 2 + 3 + . . . + n - 1*** times.
By mathematical proof, we know that:

$$1 + 2 + 3 + \ldots + n = n(n + 1)/2$$

For mathematical buffs, this formula will be familiar. But if you have not seen this before, *do not worry* about the proof, we will not expect you to know this and there are easier ways of finding the Big O, which will be discussed shortly.

Applying the formula: $1 + 2 + 3 + \ldots + n - 1 = (n - 1)((n -1) + 1)/2 = (n - 1)n/2$
$= n^2/2 - n/2$
Omitting constants and insignificant terms (See page 6, Section 3-2 of this Tutorial Letter) the Big O Notation of Fragment#3 is expressed by $O(n^2)$.

As you can see trying to count the exact number of times Line 3 is executed as a function of ***n*** gets tricky, especially for this fragment. The following rule may be applied:

The Multiplication Rule: The total running time of a statement inside a group of nested loops is the running time of the statements multiplied by the product of the sizes of all the for loops. We need only look at the maximum potential number of repetitions of each nested loop.

Applying the Multiplication rule to Fragment #2 we get:

The inner `for` loop *potentially* executes ***n*** times (the maximum value of `i` will be ***n-1***) and the outer `for` loop executes ***n*** times. The statement `sum++;` therefore executes not many times less than $n * n$, so the Big O Notation of Fragment#3 is expressed by $O(n^2)$.

```
//Fragment #3
for(int i = 0; i < n*n; i++)
          sum++;
for(int i = 0; i < n; i++)
     for(int j = 0; j < n*n; j++)
          sum++;
```

We consider only the number of times the `sum++` expression within the loop is executed. The first sum expression is executed n^2 times. While the second `sum++` expression is executed $n * n^2$ times. Thus the running time is approximately $n^2 + n^3$. As we only consider the dominating term - the Big O Notation of this fragment is expressed by $O(n^3)$.

Note that there were two sequential statements therefore we added n^2 to n^3. This problem illustrates another rule that one may apply when determining the Big O. You can combine sequences of statements by using the **Addition rule**, which states that **the running time of a sequence of statements is just the maximum of the running times of each individual statement.**

```
/Fragment #4

for(int i =0; i < n; i++)
     for(int j = 0; j < n*n; j++)
          for(int k = 0; k < j; k++)
               sum++;
```

We note that we have three nested for loops: the innermost, the inner and the outer for loops. The innermost for loop potentially executes ***n*n*** times (the maximum value of `j` will be ***n*n-1***). The inner loop executes ***n*n*** times and the outer for loop executes ***n*** times. The statement `sum++;` therefore executes not many times less than ***n*n * n*n * n***, so the Big O Notation of Fragment #4 is expressed by ***O(n^5)***.

```
//Fragment #5
for(inti = 0; i<2^10; i++)
     for(int j = 0; j < n; j++)
          sum++;
```

Note that the outer loop runs ***2^{10}***times while the inner loop runs ***n*** times. Hence the running time is: ***2^{10} * n***. The Big O Notation of this Fragment is expressed by ***O(n)***(ignoring constants).

```
//Fragment #6
for(int i = 1; i < n; i++)
     for(int i = n; i > 1; i = i/2)
          sum++;
```

The first loop runs ***n*** times while the second loops runs ***log n*** times. (Why? Suppose ***n*** is 32, then `sum++;` is executed 5 times. Note $2^5 = 32$ therefore $\log_2 32 = 5$. Thus the running time is actually ***$\log_2 n$***, but we can ignore the base.) Hence the Big O Notation of Fragment #6 is expressed by ***O(n logn)*** (Multiplication Rule.)

CONTACTED SOURCES

Baase, S. & Van Gelder, A. 2000. *Computer Algorithms: Introduction to design and analysis.* 3rd Edition. Addison-Wesley: Massachusetts.

Malik, DS. 2003. *Data structures using C++.* Thompson: Canada.

Scaffer, CA. 1998. *A practical introduction to data structures and algorithm analysis (Java edition).* Prentice-Hall: New Jersey.