Tutorial Letter 101/3/2020

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

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!

We hope that you will enjoy this module. All the best.

1 INTRODUCTION TO THE MODULE

Please note that this module is offered in a blended format, which means that although all the material will be available online, some material will be printed and some will be available only online.

All study material for this module will be available on *my*Unisa. It is thus very important that you register on *my*Unisa and access the module site on a regular basis. You must be registered on *my*Unisa to be able to access your learning material, submit your assignments, and gain access to various learning resources and to participate in online discussion forums.

Because this is an online module, you need to go online to see your study material 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 COS2611-20-S1/S2 in the row of modules in the orange blocks across the top of the webpage. Remember to check in the 'More Sites' tab if you cannot find it in the orange blocks. Click on the module you want to open.

Printed materials to support the online module

We will also provide you with some of the study material in printed format. You will receive a copy of this tutorial letter.

Note, however, that other tutorial matter will not be printed (such as tutorial letters 102 and 103, and assignment solutions that are issued as 200 series of tutorial letters). You will be able to download them as PDFs from *my*Unisa.

You can access all the study material on *my*Unisa**. You should NOT wait for the printed document to arrive to start studying.**

Please consult the *Study @ Unisa* publication for more information on the activation of your *my*Life email address as well as obtaining access to the *my*Unisa module site.

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.

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:

- Demonstrate an understanding of algorithm analysis and the Big-Oh notation used in algorithm analysis.
- Demonstrate an understanding of the basic properties of pointers and linked lists.
- Demonstrate an understanding of how to use recursion to solve problems and how to think in terms of recursion.
- Demonstrate an understanding of Abstract Data Types (ADTs) and how they are stored on computers.
- Demonstrate an understanding of search techniques used to retrieve data held in data structures.
- Demonstrate an understanding of sort techniques used to sort data held in data structures.

3 LECTURER(S) AND CONTACT DETAILS

3.1 Lecturer(s)

The details of the lecturers will be provided on the home page of the COS2611 site on *my*Unisa.

The details of the lecturers will also be communicated in a COSALL tutorial letter.

When you contact the lecturers, please do not forget to always include your student number and module code. This will help the lecturers to assist you.

3.2 Department

You can contact the School of Computing as follows:

Telephone number: **+27 (0) 11 670 9200**
E-mail: **computing@unisa.ac.za**

3.3 University

To contact the University, follow the instructions in the brochure ***Study @ Unisa***. Remember to have your student number available whenever you contact the University.

4 RESOURCES

4.1 Prescribed book(s)

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 book(s)

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

Recommended books can be requested online, via the Library catalogue.

4.3 Electronic reserves (e-reserves)

E-reserves can be downloaded from the library catalogue. More information is available at:

http://libguides.unisa.ac.za/request/request

4.4 Library services and resources information

The Unisa Library offers a range of information services and resources:

- for brief information go to: https://www.unisa.ac.za/library/libatglance
- for more detailed Library information, go to http://www.unisa.ac.za/sites/corporate/default/Library

- for research support and services (e.g. Personal Librarians and literature search services), go to http://www.unisa.ac.za/sites/corporate/default/Library/Library-services/Research-support

The Library has created numerous Library guides: http://libguides.unisa.ac.za

Recommended guides:

- request and find library material/download recommended material: http://libguides.unisa.ac.za/request/request
- postgraduate information services: http://libguides.unisa.ac.za/request/postgrad
- finding and using library resources and tools: http://libguides.unisa.ac.za/Research_skills
- Frequently asked questions about the Library: http://libguides.unisa.ac.za/ask
- Services to students living with disabilities: http://libguides.unisa.ac.za/disability

Important contact information:

- https://libguides.unisa.ac.za/ask - ask a Librarian
- Lib-help@unisa.ac.za - technical problems accessing library online services
- Library-enquiries@unisa.ac.za - general library related queries
- Library-fines@unisa.ac.za - for queries related to library fines and payments

5 STUDENT SUPPORT SERVICES

The *Study@Unisa* brochure is available on myUnisa: www.unisa.ac.za/brochures/studies

This brochure has all the tips and information you need to succeed at distance learning and, specifically, at Unisa.

5.1 E-Tutors

Unisa offers online tutorials (e-tutoring) to students registered for modules at NQF level 5 to 7, this means qualifying first year to third 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 *my*Unisa in order to receive further information on the e-tutoring process. If you login into *my*Unisa 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 COS2611-20-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 *my*Unisa 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 Downloading study material and software

One of the requirements for studying at the School of Computing is to have regular access to *my*Unisa. 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 *my*Unisa. The study material is updated regularly, thus you need to check the COS2611 website at least once a week on *my*Unisa.

The ***software*** should also be downloaded from *my*Unisa under *Additional Resources* for COS2611. Please note that you only need to download Code::Blocks only for COS2611.

6 STUDY PLAN

Use your *Study@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	**Activity**
1	Install software Study Chapter 1 Study Appendix A
2	Study Chapter 3
3	Study Chapter 5
4	Study Chapter 6 **Do Assignment 1 and submit it.** **You must submit this assignment to attain EXAM ADMISSION.**
5	Study Chapter 7
6	Study Chapter 8
7	Study Chapter 9

8	Study Chapter 10
9	Study Chapter 11 Do Assignment 2
10	**Submit Assignment 2**
11	Study Chapter 12
12	Do Self-Assessment Assignment Do NOT submit.
13-15	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

7.1 Prescribed compiler

The prescribed C++ compiler for COS2611 is the GNU C++ compiler.

Note that this prescribed compiler is the one used for COS1511 and COS1512 and can be downloaded from *my*Unisa 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.

8 ASSESSMENT

8.1 Assessment criteria

Assessment for COS2611 is in the form of 2 assignments and a final exam.

The final mark is made up of the semester mark and the exam mark based on the following assessment plan guidelines.

8.2 Assessment plan

The Syllabus

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 analyse 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.

The concepts and sections which have not been included during the discussion of the foregoing syllabus are NOT part of the syllabus of COS2611. Consequently you will NOT be examined on these concepts. *Hashing* which is part of Chapter 10 should for example NOT be studied.

Specific outcomes and assessment criteria

Below we give 6 specific outcomes of this module as well as the criteria used to assess these outcomes.

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.

Range: Queues, Stacks, Graphs and Trees

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).

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.

Range: Sequential Search and Binary Search

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.

Specific outcome 6:

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

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

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].

The combination of formative and summative assessments consists of the following structure:

Component	**Weight**	**Total weight**
Assignment 1	25%	
Assignment 2	75%	
Semester mark	**100%**	**20%**
Exam mark		**80%**
Final mark		**100%**

8.3 Assignment numbers

8.3.1 General assignment numbers

Below is a breakdown of the assignments as they occur in the respective semesters. Take note of the following information for each assignment: semester, assignment number, unique assignment number, weight (for semester mark) and due date.

Semester 1

Semester	Assignment number	Unique assignment number	Weight	Due date
1	01	770462	25%	24 February 2020
1	02	669070	75%	14 April 2020

Semester 2

Semester	Assignment number	Unique assignment number	Weight	Due date
2	01	549716	25%	17 August 2020
2	02	731876	75%	21 September 2020

8.3.2 Unique assignment numbers

See table above.

8.4 Assignment due dates

Assignment due dates are provided above and at the beginning of each assignment and NO EXTENSIONS will be granted for late submission of any assignment.

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 *my*Unisa and assignment 2 (as a .pdf file) via *my*Unisa. 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.

8.6 The assignments

Assignment 01 [For First Semester Students Only]

Due date:	**24 February 2020**
Tutorial matter:	**Algorithm Analysis (Chapter 1 + Appendix A)**
Maximum marks:	**25**
Year mark contribution:	**25%**
Unique assignment number:	**770462**

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 – 15 please indicate the run time complexity of the respective code fragments:

Question 1

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

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

Question 2

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

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

Question 3

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

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

Question 4

```
for (int i = 0; i < 6n; i+=2)
        sum++;
```

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

Question 5

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

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

Question 6

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

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

Question 7

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

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

Question 8

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

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

Question 9

```
for (int i = 1; i < n; i++)
{
        sum++;
        total += i;
}
```

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

Question 10

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

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

Question 11

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

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

Question 12

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

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

Question 13

```
for (int i = 0; i < 22; i++)
        for (int j = 0; j < n; j++)
          sum++;
```

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

Question 14

```
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 15

```
for (int i = 1; i < 1000; i++)
        sum++;
```

1. O(1)
2. O(1000)
3. O(n)
4. O(log n)

Question 16

An algorithm takes 10 seconds for an input size of 1000. How long will it take for an input size of 10 000 if the running time is log n?

1. 1 second
2. 10 seconds
3. 13 seconds
4. 100 seconds

Question 17

An algorithm takes 1 second for an input size of 10. How long will it take for an input size of 50 if the running time is in O(1)?

1. 1 second
2. 10 seconds
3. 50 seconds
4. 500 seconds

Question 18

An algorithm takes 2 seconds for an input size of 5. How long will it take for an input size of 50 if the running time is exponential (2^n)?

1. 2^{11} seconds
2. 2^{46} seconds
3. 20 seconds
4. 2^{20} seconds

Question 19

An algorithm takes 10 seconds for an input size of 1000. What will the largest size of input be that can be executed in 80 seconds if the running time is cubic?

1. 500
2. 2000
3. 4000
4. 8000

Question 20

An algorithm takes 5 seconds for an input size of 80. What will the largest size of input be that can be executed in 15 seconds if the running time is linear (O(n))?

1. 512
2. 480
3. 240
4. 48

Question 21

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

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

Question 22

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

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

Question 23

Order the following functions by growth rate from smallest to largest (in terms of their Big-O values): 10 log n; $2n^2$; n log n

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 24 and 25. The table shows the running time against input size (n) of 2 algorithms.

Algorithm	**n=10**	**n=20**	**n=30**
A	17 minutes	728 days	2 043 years
B	1½ minutes	7 minutes	15 minutes

Question 24

What is the running complexity of algorithm A?

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

Question 25

What is the running complexity of algorithm B?

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

Assignment 02 [For First Semester Students Only]

Due date:	**14 April 2020**
Tutorial matter:	**Linked Lists (Chapter 5)** **Recursion (Chapter 6)** **Stacks (Chapter 7)** **Queues (Chapter 8)** **Searching (Chapter 9)** **Sorting (Chapter 10)** **Trees (Chapter 11)**
Maximum mark:	**80**
Year mark contribution:	**75%**
Unique assignment number:	**669070**

Question 1: Linked Lists [5]

Consider a linked list called `list` depicted below. Assume the node is in the usual info-link form. (`list` and `Ptr` are pointers of type `nodeType`.)

Suppose calling `exchange(2,4)` on the list will rearrange the nodes such that the second node is D and the last node is B.
Without changing the order of the nodes in the above diagram, adjust the links of this list below to show the results after calling `exchange(2,4)`:

(2)

Note that the `exchange()` function does not swap the elements of the nodes.

Write C++ code to implement (a). Do not provide the complete function.

(3)

Question 2: Linked Lists [15]

2.1 Add the following operation to the class `orderedLinkedList`:

```
void mergeLists(orderedLinkedList<Type> &list1, orderedLinkedList<Type> &
list2)
         //This function creates a new list by merging the elements
         //of list1 and list2.
           //Postcondition: first points to the merged list; list1 and list2
         //are empty
```

Example: Consider the following statements:

```
orderedLinkedList<int> newList;
orderedLinkedList<int> list1;
orderedLinkedList<int> list2;
```

suppose `list1` points to the list containing the elements `2 6 7` and `list2` points to the list containing the elements `3 5 8`. The statement:

```
newList.mergeLists(list1, list2);
```

creates a new linked list with the elements in the order `2 3 4 5 7 8` and the object `newLists` points to this list. After the preceding statement have executed, `list1` and `list2` are empty.

2.2 Write the definition of the function template to implement the operation `mergeLists`. Write a program to test your function thereafter.

Question 3: Recursion [6]

Write a **recursive** function template, `identicalS` that checks whether two stacks provided as parameters are identical. If that is the case, the function should return the Boolean value `true` otherwise the Boolean value `false` should be returned.

Use the following header:

```
template <class Type>
bool identicalS (stackType<Type> S1, stackType<Type> S2);
```

Question 4: Stacks [6]

$L=\{a^n b^n\}$ 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.
- Examples of words that belong to L are

  ab, where n=1;

  aabb, where n=2;

  aaabbb, where n=3;

  aaaabbbb, 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);
```

Question 5: Queues [16]

a) 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. (6)

b) Implement question 4 using a queue instead of a string as in the following header:

```
bool isInLanguageLQ (queueType<char> &w)
```

(10)

Question 6: Searching [6]

Describe how the binary search algorithm will search for 27 in the following list:
5, 6, 8, 12, 15, 21, 25, 31.

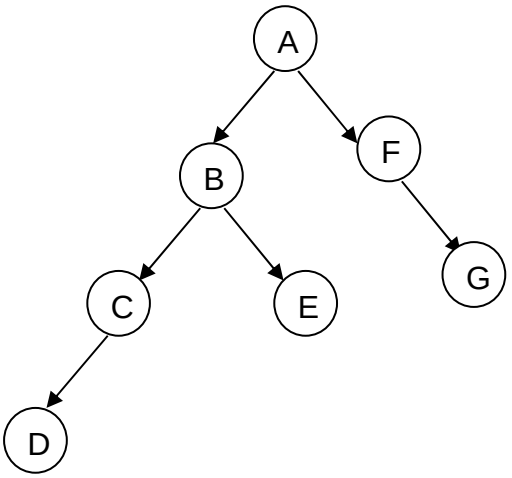
Question 7: Sorting [20]

Consider the following sequence of numbers 11, 8, 9, 4, 2, 5, 3, 12, 6, 10, 7

a) Sort the list using selection sort. Show the state of the list after each call to the swap procedure.
b) Sort the list using insertion sort. For each iteration of the outer loop of the algorithm show the state of the list.
c) Sort the list using quick sort with the middle element as pivot. For each iteration of the outer loop of the algorithm show the state of the list.

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

Question 8: Trees [6]



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

Assignment 01 [For Second Semester Students Only]

Due date:	**17 August 2020**
Tutorial matter:	**Algorithm Analysis (Chapter 1 + Appendix A)**
Maximum marks:	**25**
Year mark contribution:	**25%**
Unique assignment number:	**549716**

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

For Questions 1 – 15 please indicate the run time complexity of the respective code fragments:

Question 1

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

1. O(1)
2. O(Log N)
3. O(N)
4. O(N log N)

Question 2

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

1. O(1)
2. O(Log N)
3. O(N)
4. O(N log N)

Question 3

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

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

Question 4

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

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

Question 5

```
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 6

```
for(int i = 1; i < n; i = 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 7

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

1. O(N^2)
2. O(N^5)
3. O(N^3)
4. O(N log N)

Question 8

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

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

1. $O(N^4)$
2. $O(N^2)$
3. $O(N^3)$
4. $O(N^5)$

Question 9

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

1. $O(N^4)$
2. $O(N^2)$
3. $O(N \log N^2)$
4. $O(N^5)$

Question 10

```
for(int i = 0; i < 100000; i++)
     for(int j = 0; j < 20000; j++)
          sum++;
```

1. $O(1)$
2. $O(N^2)$
3. $O(N^3)$
4. $O(N \log N)$

Question 11

```
for(int i = 1; i < n*n; i++)
{
     int j =n;
          while (j > n)
          {
               Sum++;
               j--;
          }
}
```

1. $O(N^4)$
2. $O(N^2)$
3. $O(N^3)$
4. $O(N \log N)$

Question 12

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

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

Question 13

```
for(int i = 1; i < 2^10; i++)
    for(int j = 0; j < n; j++)
        sum++;
```

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

Question 14

```
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 15

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

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

Question 16

An algorithm takes 0.5 seconds to execute for an input size of 100. How long will it take for an input size of 500 if the running time is linear?

1. 5 sec
2. 2.5sec
3. 0.5sec
4. 1sec

Question 17

An algorithm takes 10 seconds for an input size of 100. How long will it take for an input size of 10 000 if the running time is log N?

1. 10 sec
2. 20.5sec
3. 20 sec
4. 40sec

Question 18

An algorithm takes 1 second for an input size of 10. How long will it take for an input size of 50 if the running time is exponential (2^n)?

1. 2^{40} sec
2. 20 sec
3. 20^{40} sec
4. 2^{10} sec

Question 19

An algorithm takes 1 second to execute for an input size of 10. What will the largest size of input be that can be executed in 25 seconds if the running time is quadratic (N^2)?

1. input size of 100
2. input size of 5
3. input size of 50
4. input size of 625

Question 20

An algorithm takes 10 seconds for an input size of 1000. What will the largest size of input be that can be executed in 80 seconds if the running time is cubic?

1. input size of 800
2. input size of 1250
3. input size of 2000
4. input size of 3000

Question 21

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

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

Question 22

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

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

Question 23

Which one of the following sets of functions is ordered according to growth rate from the smallest to the largest?

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

Questions 24 and 25 are based on the table below. The table displays the running time of two algorithms for an input size of n..

Algorithm	**n = 4**	**N = 8**	**N = 16**
A	16	256	65536
B	6	11	20

Question 24

What is the runtime complexity of algorithm A?

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

Question 25

What is the running complexity of algorithm B?

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

Assignment 02 [For Second Semester Students Only]

Due date:	**21 September 2020**
Tutorial matter:	**Linked Lists (Chapter 5)** **Recursion (Chapter 6)** **Stacks (Chapter 7)** **Queues (Chapter 8)** **Sorting (Chapter 10)** **Trees (Chapter 11)**
Maximum mark:	**80**
Year mark contribution:	**75**
Unique assignment number:	**731876**

Question 1: Linked Lists [5]

Draw what is produced by the following C++ code. Assume the node is in the usual info-link form with info of type `int`. (`list` and `ptr` are pointers of type `nodeType`.)

```
list = new nodeType;
list->info = 20;
ptr = new nodeType;
ptr->info = 28;
ptr->link = NULL;
list->link = ptr;
ptr = new nodeType;
ptr->info = 30;
ptr->link = list;
list = ptr;
ptr = new nodeType;
ptr->info = 42;
ptr->link = list->link;
list->link = ptr;
ptr = List;
while (ptr != NULL)
```

Question 2: Linked Lists [15]

Given two ordered linked lists `L1` and `L2`, write a function that deletes every element of `L2` contained in `L1`. Your function should be a *friend* function of the class `orderedLinkedListType`.

The following header should be used:

```
Template<class Type>
Void deleteOc (orderedLinkedList<Type> & L1, const
ordereLinkedList<Type> & L2)
```

Question 3: Recursion [6]

Operating systems use virtual memory to expand the amount of memory available for running programs by swapping pages between RAM and virtual memory. One of the algorithms that can be used to determine which page gets swapped out is the least recently used (LRU) algorithm. One way of implementing this algorithm is to use a stack of page numbers, and when a page is referenced it is removed from the stack and put on top. Thus the least recently used page reference is at the bottom of the list. This is best implemented by using a linked list because page references are removed from the middle of the list. However, for this exercise, assume the LRU algorithm is implemented by using a stack.

Write a member function of class `StackType` that updates the stack when a page is referenced. Assuming a stack that can hold 5 values, and the next page referenced is 7, then

- the function searches the stack for page reference 7;
- if it finds it, it removes it from the stack and places it at the top;
- if there is no 7 in the list, the last page reference in the stack is removed and the 7 is placed at the top of the stack.

Below is an example illustrating the contents of the stack before and after the function has run.

4	To the left is the stack	7
8	before the 7 is	4
2	referenced; to the right	8
7	is the stack after the 7	2
1	is referenced	1

Use a driver function

```
template <class Type>
void stackType<Type>::updateRecursive(stackType<Type> &s, Type
t);
```

that calls the recursive function.

```
template <class Type>
bool stackType<Type>::updateRecursiveDo(stackType<Type> &s,
Type t);
```

Question 4: Stacks [6]

$L=\{a^n b^{2n}\}$ 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 **twice** the number of `a`'s.
- Examples of words that belong to `L` are

`abb`, where `n=1;`

`aabbbb`, where `n=2;`

`aaabbbbbb`, where `n=3;`

`aaaabbbbbbbb`, 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 i`sInLanguageL2` should return `true` otherwise `isInLanguageL2` should return `false`.

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

Question 5: Queues [16]

a) Write a function `removeFirst` that removes only the first occurrence of an item in a circular queue without changing the order of the other elements in the queue. Use the following header:

```
template <class Type>
void queueType<Type>::removeFirst(queueType<Type>& Q,
                                  const Type& x)
```

(6)

b) Implement Question 4 using a queue instead of a string as in the following header:

```
bool isInLanguageL2Q (queueType<char> &w)
```

(10)

Question 6: Sorting [20]

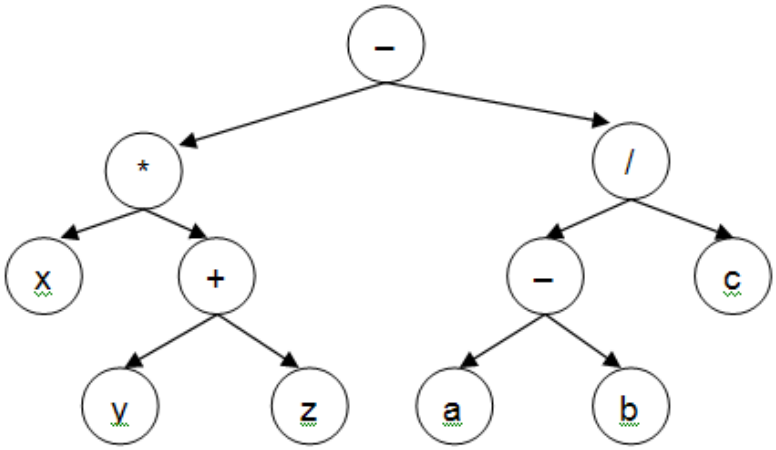
Consider the following sequence of numbers 5, 12, 3, 2, 4, 10, 1, 6.

a) Sort the list using a selection sort. Show the state of the list after each call to the swap procedure.
b) Sort the list using an insertion sort. Show the state of the list each time after the outer for loop of the algorithm has iterated.
c) Sort the list using a 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 [6]

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.



8.7 Other assessment methods

None

8.8 The examination

A 2-hour examination for this course will be scheduled for the end of the semester. The Examination Section will inform you of the time, date and venue. You can also obtain the examination date from *my*Unisa.

Please note that you will be admitted to the examination if-and-only-if you submit at least one assignment before the relevant due date.

The exam will assess your mastery of the outcomes of the course as covered in the assignments. After the due date of the final assignment, additional information about the exam will be included in TL103 (which will be made available under Additional Resources).

9 FREQUENTLY ASKED QUESTIONS

None

10 SOURCES CONSULTED

See Appendix A.

11 IN CLOSING

Do not hesitate to contact your e-tutors or lecturers by email if you are experiencing problems with the content of this tutorial letter or any aspect of the module.

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

Enjoy the journey!

12 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 analyse 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 analyse algorithms, we should employ mathematical techniques that analyse algorithms independently of *specific implementations, computers, or data.*

To analyse 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

```
if (n < 0)
absval = -n
else
absval = n;
```

Operation	Cost
compare	1
Assignment	1
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

```
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 analyse?

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 →**6*n* + 2** (ignoring 2)
→**6*n*** (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 analyse 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 analysing 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.

13 APPENDIX B: Errata

13.1 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.

13.2 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.