Introduction to Algorithms

Paul Taylor

www.dcs.qmul.ac.uk/~pt/algorithms

Teaching materials for a first year BSc G500 Computer Science module at Queen Mary, University of London, taught in the spring of 1999, 2000 and 2001. The course is also taken by second year GG15 and GG51 Mathematics and Computer Science students. It is about understanding the behaviour, implementation, correctness and complexity of some well known array algorithms, especially for sorting and searching.

Please note that these are not *lecture* notes. They are an amalgamation of

- instructions for experiments and programming that students do during supervised labs;
- model answers for exams and tests on the theoretical material that had been presented on the blackboard in lectures; and
- administrative documents that were originally published as Web pages.

Full treatment of some theoretical topics is therefore missing from the collection.

© 2001 Paul Taylor.

Paul Taylor hereby asserts his moral right of paternity (the right always to be identified as the author of this work) in accordance with the UK Copyright, Designs and Patents Act (1988).

Contents

Chapter 1

Course Information Sheet

Course code: DCS/127, **Semester:** 2.
Acknowledgements: Richard Bornat, Samin Ishtiaq and Jules Bean
Resources: my TESTER, some textbooks and Web links and the mailing list.

1.1 Aims

- To understand why formal logic is essential for getting programs to work correctly, and
- to learn some simple methods using logical notation to specify properties of programs.
- To give an appreciation of the range and depth of knowledge of programming solutions to apparently simple problems;
- to increase perception of the beauty and intricacy of known algorithms;
- to increase understanding of the role of analysis in the construction of practical programs;
- to become more reflective when designing programs.

1.2 Objectives

- To understand and be able to employ predicate calculus notation in the specification of several algorithms, especially in describing the invariants of loops;
- to understand and be able to apply big-O notation for worst-case complexity analysis of programs including the algorithms treated in the course;
- to understand and be able to reproduce some algorithms for sorting and searching;
- to be able to construct arguments about the relative worst-case space and time complexity of some algorithms for sorting and searching.

1.3 Prerequisites

Introduction to Logic (DCS/122) and Programming I (DCS/101) are pre-requisites.

An understanding of mathematics up to A-level will be useful for the complexity analysis of the algorithms presented.

The *essential* skill is precise thinking, of a kind similar to that used in mathematics, though very little use is made of particular facts from the A-level (or degree-level) mathematics syllabus.

1.4 Description

The study of algorithms — carefully-crafted solutions to particularly important programming problems — is one of the oldest areas in Computer Science. In order truly to *understand* an algorithm you have first to understand why it works at all, then to understand how it performs in general — how long it takes and how much space it takes up while it is working — and last, understand how to tweak the algorithm to make its performance fit the particular circumstances in which you want to use it. In the case of some of the famous algorithms we shall meet, like quicksort, you also have to understand why it isn't easy to do better.

So the course will focus on two issues: *correctness* — how you say what an algorithm is supposed to do, and how you can argue that it actually does it — and so-called *complexity* — how to estimate the amount of work an algorithm must do to perform its task in general, or on average, or in the worst case. There will be some experimental work, trying to verify in practice the theoretical measures that can be developed by using mathematical arguments, but much of the course will be about analysing algorithms with pencil and paper.

Some of the algorithms we shall analyse are intensely beautiful. To appreciate their beauty you have to realise how much better they are than the first naive attempts one of us might make to solve the same problem, and how concisely they solve the problem which they are designed to solve. All of the algorithms we shall analyse are useful. Some of them are not entirely understood — it is a recurring feature of Computer Science that we continually invent things which we only partly understand.

1.5 Course Plan

From each of the algorithms that we study, we learn some general principle about the correctness of programs in general. These general principles, and not programming recipes for these particular examples, are what you are meant to take away from this course.

- **Linear search**: writing a method that satisfies a specification, fitting it in to a test program, and proving its correctness.
- **Binary search**: there are at least three published algorithms, differing by ± 1 in various *important* places.
- **Array shifting**: assignment messes up the logic.
- **Insertion sort**: search and shift are used as subroutines in this quadratic algorithm.
- **Selection sort**: another quadratic sorting algorithm.
- **Merge sort**: this $O(n \log n)$ algorithm is your first recursive program.
- **Quick sort**: another recursive program, that goes fast when you get a design decision right, it's easy to get this wrong.
- **Heap sort and priority queues**: your first object-oriented program.

Some harder topics will be treated in the lectures at the end of the course but will be optional for assessment:

- **Hash tables**: how to insert, look up and delete data by name very quickly.
- **Finding a path through a network**

1.6 Assessment

There are **three** items of assessment, each worth 15% of the course marks, together with the exam (55%). Two of these items are tests (mini-exams) conducted during the periods timetabled for lectures, and the third is work to be done during lab sessions and handed in by a deadline.

The relatively high percentage of test marks is intended to deter the "listen; forget; revise" style of course attendance. All of the assessment is intended to make you **think**, and will contain traps to deter copying and rote learning.

1.7 Reading

- *Data Structures and Problem Solving Using Java* by Mark Allen **Weiss**, (Addison-Wesley);
- *Reasoned Programming* by Krysia **Broda**, Susan **Eisenbach**, Hessam **Khoshnevisan** and Steve **Vickers**, Prentice Hall, International Series in Computer Science, 1994.

1.8 Calendar (2000–1)

- Monday 11am-noon, **lecture**, Skeel Lecture Theatre, People's Palace
- Monday 2-6pm, **supervised labs**, ITL ground floor
- Tuesday 3-5pm, **lecture**, Physics Lecture Theatre

The middle item in each triplet is the lab work; the other two are lectures.

- 15/16 January (Week 1)
  - Themes, timetable, assessment, resources, this week's work.
  - Linear search (from left and right)
  - Using semi-formal logic to describe programs
- 22/23 January (Week 2)
  - Linear search and shift loop invariant diagrams.
  - Search for insertion; shifting a segment of an array
  - Four quadratic sorting algorithms
- 29/30 January (Week 3)
  -
  - Experiments with quadratic sorting algorithms
  - Loop invariant diagrams for shift, insertion sort and selection sort.
- 5/6 February (Week 4)
  -
  - Observations and timings of quadratic sorting algorithms
  - Variations on Binary search
- 12/13 February (Week 5)
  -
  - Experiments with binary search

  - Mergesort; picture of recursion.
- 19/20 February (Week 6)
  - Recursion; the idea of mergesort+quicksort; merge loop invt.
  - Implementing merge and mergesort
  - Binary trees, logarithms, exponentials, complexity. Hybrid mergesort.
- 26/27 February (Week 7)
  - **Test** in the Great Hall on the results of lab work.
  - Implementing mergesort and quicksort.
  - Specifications. Split algorithms and choice of pivot for quicksort.
- 5/6 March (Week 8, Eid-al-Adha)
  - Lecture cancelled for Eid.
  - Implementing split and quicksort.
  - Heaps, priority queues and heap sort
- 12/13 March (Week 9)
  - **Test** in the Great Hall: correctness proof for one algorithm.
  - Implement sift up and down for heaps.
  - Algorithms on networks.
- 19/20 March (Week 10)
  - Lecture cancelled because I was at a conference
  - Implement heapification, heap sort and priority queues.
  - Hash tables for fast access to dictionaries.
- 26/27 March (Week 11)
  -
  - Additional coursework options or finish regular coursework.
  -
- 2/3 April (Week 12)
  - Recursion- and data- (class-) invariants (non-examinable)
  - Additional coursework options or finish regular coursework.
  - Revision lecture.
  - Friday 6 April (End of term): **Coursework deadline**
- Thursday 9 May
  - exam at 10am in Stratford Old Town Hall.

1.9 Monitoring student work

If you are a student registered for this course, you should find in your personal filespace a directory (folder) called **DCS127**.

All of your work for this course must be done in this directory.

(Think of it as the electronic equivalent of a school exercise book for a particular subject.)

As the various pieces of lab work for the course are set, template files will be copied into this directory for you to work on. They contain the structure of the methods and classes that you have to implement, instructions for writing and testing your code, and questions for you to answer about the behaviour of the code.

Your answers to these questions, as well as your Java code, should be written in these files. Then all of the things that you are supposed to learn about each algorithm will be collected together in one place for you to learn and revise.

The directory also contains a short-cut (**pt**) to the teaching materials for the course. It is also the *only* place in which you can do the `run`, `trace` and `check` commands that invoke the automatic tester.

Everything that you do in this directory will be copied overnight to a place where your lecturer, teaching assistants, adviser and tutor can read it.

Only the contents of *this* directory will be copied and made available to these people. Other parts of your personal filespace (such as your email records) will *not* be copied. Other students will not be able to see the copy; in fact the access permissions on this directory itself are automatically reset so that only you, and not other students, can read its contents.

Only plain text files (Java *source* files and notes that you make using xemacs in plain ASCII) will be copied. Compiled Java *classes* and files that appear to be mail or Microsoft Word format are not copied. Nor are directory trees called `private`, `not_mine` or `RCS`, and some other types of file[1] are also excluded.

When files are copied, a "receipt" for them is recorded in the file `COPY_LOG`. You cannot delete or modify this file yourself.

Modification histories of the files are also maintained (in the place where the copies are made) using the Unix RCS tool. It may be possible to recover old versions of the file from this.

The purpose of this is to ensure that you do the lab work that is set each week.

To submit your end of term coursework, you simply leave it in this directory.

The Tester will be run on the copied files, but they will be printed out and looked at individually before being marked.

Files that you put in this directory must be entirely your own work.

Since some of the things that you put here will be marked as coursework, contributing to your final grade for this course and ultimately to your degree, they must be treated in the same way as exam scripts. The College examination regulations therefore apply.

If you wish to keep materials in this directory that are relevant to your study of this course, but are not your own work (for example, code that you have downloaded from the Web), you must put it in the subdirectory **not_mine**. It will then not be treated as coursework.

1.10 Assessment of coursework

The assessment for this course consists of 55% for the May exam and 15% for each of three items of term-time work, two of these being tests.

All lab work is collected automatically from the directory **DCS127** in your filespace, every night. *All* work for this course should be done in this directory, as a matter of routine. In particular, you "**submit**" coursework *simply* by leaving it there.

There are **two alternative ways** of gaining the third 15% of marks:

[1] Emacs backup files (`file~` and `#file#`); "dot" files (`.login`); your "receipt" `COPY_LOG`; files whose names contain spaces, quotes or other strange characters; symbolic links; PostScript; PDF; executables; images (`picture.gif`); archives (`tester.jar`, `folder.zip`, `folder.tar.gz`).

- On the basis of your weekly coursework, or
- By doing one of the options listed below in the last three weeks of term.

You will be awarded the better of the two marks. You do not need to tell me which you want to count.

Please note:

- To gain marks for each week's work (approximately 1% of the final credit per week), it must be done **on time**, not at the end of term.
- Normally, **the deadline is the end of the day of the lab**, but it will be set and changed each week depending on how difficult the class as a whole has found the exercise. **No extensions of deadlines will be given for colds or single absences** (and you will only cause annoyance by asking for them), but of course allowance will be made for *medical* problems that last *several weeks*. The intention behind this method of assessment is to persuade you to **do the work that is set in each week's lab**.
- Formally, the assessment will be for the term's work *collectively*, not for individual items. Assessment will include a elements for programming style, *etc.*, in which the items that have been submitted will be examined qualitatively, not quantitatively.
- **Do not think that you can miss labs and make up the marks by submitting coursework at the end of term instead.** Such work will be examined *critically and suspiciously* for correctness and evidence of copying. But lab work is there to help you learn, so it is rather unlikely that you will be able to do the (more difficult) end of term coursework well unless you have done the earlier stuff properly.
- The optional topics are intended for the best students. **Mediocre work will get far fewer marks than it did last year.**

1.11 The optional topics

- **Hash tables** for creating, accessing and updating dictionaries.
- **Finding a path between two nodes in a network** was covered in 2000–1 for the first time.
- **External sorting** of a list of names (such as the US population) that fits on disk but not in RAM. You have to do your own reading on this option — there will be no lectures.
- **Heap sort and priority queues** were an option in 1998–9 and 1999–2000, but became a core part of the course (with structured lab work and exam question) in 2000–1.
- **Minimax and alpha-beta pruning** for playing games such as chess or draughts. This was an option in 1998–9 and 1999–2000, prompted by the Software Engineering projects about implementing games, but not very many students took it, so it was droppped in 2000-1.
- **Black box testing of an alleged sorting method** was an option in 1999–2000, but only one student (Daniel Dor-Chay) took it. He discovered the "sawtooth" array that makes quicksort quadratic, but otherwise this option was not a very good idea.

You can get credit for the options listed above **twice**:

- 15% for **assessed lab-work**. You have to **implement** the algorithm, and hand in your code on 6 April, along with some of the results of testing that the code works.
- Approximately 14% more for an optional **essay** in the exam.

Do not write an essay for the coursework. Postpone further reading and experimentation (beyond what is necessary to get your program working) until after you have handed in your program.

1.12 The essay question in the exam

You do not have to choose the same topic for the coursework and the exam, but it would obviously be more effective in gaining marks to do so. However, the coursework and exam test different aspects of the topic.

For the exam you are not expected to remember and reproduce 200 lines of code, but you will be expected to give the code for the **core algorithms**. Your essay should also summarise what you have **learned** about the algorithm from running and modifying your own program or my compiled model answer, my lectures, Richard Bornat's lecture notes, Weiss's book, and any other sources you can find.

The essay question was compulsory in May 1999, but in 2000 and 2001 students had to choose four questions out of eight, one of them being the essay; the 14% figure above is based on this.

Writing an "essay" in an exam is **not** a piece of English composition (although I would like you to know how to spell "pigeonhole", and the difference between "its" and "it's"). At school you were probably told to write "a beginning, a middle and an end", but any "introduction" or "conclusion" you write for this essay is unlikely to (contain material which will) gain you any marks. An exam essay is like one of those TV game shows where you have to remember as many items as possible that have gone past on a conveyor belt ("TV, teddy bear, toaster, ..."). How many facts can you remember about this topic?

The **exam rubric** for the essay question [in May 2000] read as follows:

> Write an essay about *one* of the [above] topics.
>
> Your answer should include a selection of the most important parts of the code: do not include JAVA declarations unless they illustrate some aspect of the algorithm. Explain the main points in proving correctness. State the main facts about complexity.
>
> You are unlikely to gain any marks at all for this question unless you have implemented the algorithm in the lab sessions and thought carefully about its behaviour. Remember that this is an exam about algorithms, not about English composition, and that your answer will be marked according to the information that it contains about algorithms: wordy answers to questions like this gain very few marks.

The marking of both the coursework and essay question will be open-ended: the mark-scheme will give full marks for the basic issues, but generous bonus marks will be given for anything else you discover by experimentation with your program. Unfortunately, it is not within my power to give total marks above 100% for the course, or to give any prizes for supererogation.

1.13 Warning about plagiarism

It is the College's policy to punish plagiarism **severely**.

- This is part of **your degree assessment**, and is intended to be **individual** work.
- Copying from other students' work (with or without their consent) is plagiarism.
- If you work in **pairs** or larger **groups**, each member of the group must submit **identical** work, which must state the **names of all of the members of the group** and the agreed proportions of the work that they contributed. If you have had the help of others to do the work you submit then you will of course get fewer marks, to compensate.
- To copy code out from Richard Bornat's notes, Weiss's book or web pages, or from any other sources without identifying the source *precisely* is also plagiarism,
- If you use a book, you must give the **author, title, publisher and page numbers**.
- If you use the web you must give the **full URL of the relevant page**, not just the name of the site.

- **Changing** what you copy from books, web pages or other students counts as **deliberate attempted deception**. It is also insulting to expect someone (a lecturer or a colleague) to give individual attention to something that does not in fact have any original content.

Giving precise references is a basic professional habit, and is **useful**. For example if you've filched someone else's code like this, and adapted it to your own needs, and then it doesn't work, you (or your colleagues) can go directly back to the original source.

You don't have to cite your sources in the exam (in my course).

Chapter 2

How I teach this course

This course is about understanding the behaviour, implementation, correctness and complexity of some well known array algorithms, especially for sorting and searching.

It is taken by first year BSc (G500) and MSci (G501) Computer Science and second year BSc Computer Science and Mathematics (GG15) and Mathematics and Computing (GG51) students.

It was introduced by Richard Bornat in January 1997, and I took it over in January 1999, so I have taught it three times.

The **message** that course is meant to send to students is

- *technically* about **components** and their **specifications**, and
- *philosophically* that **it is not obvious what programs do** — you have to think about them and do experiments.

The purpose of this note is to explain (mainly to my colleagues at Queen Mary) my own approach to this course, and in particular to address certain misunderstandings that are in circulation: that

- the course is a *collection of sorting algorithms*,
- it's *about big-O notation*,
- it's *too mathematical*,
- it *ought* to be about *data structures* and
- it has a *high failure rate*.

It also describes the things that I have contributed to the course, in particular my TESTER, and my treatment of components, specifications, correctness proofs and the subtleties of some of the algorithms.

2.1 Data structures

Similar courses to mine are often called *Algorithms and Data Structures*, and the material that I inherited from Richard Bornat concluded with treatments of topics such as balanced binary trees.

I am not clear whether those of my colleagues who advocate teaching data structures in this course mean

- brief *descriptions* of how to program linked lists, trees, *etc.* in JAVA, or
- consideration of *algorithms* that make use of such structures, such as balancing of binary trees.

If they mean real consideration of algorithms, then this complaint is inconsistent with the other common complaint that the course is *too difficult*. Partly on Richard Bornat's advice, I took out the balanced binary trees, because I felt that there were plenty of things to learn about algorithms that could be done just with arrays.

If they just mean brief descriptions, it seems to me that such a course would just be a **zoo**, which would warrant exactly the criticism of the *collection of sorting algorithms* that is made.

I was persuaded to teach Dijsktra's algorithm for finding the shortest path between nodes of a network, as one of the end-of-term options. I tried very hard to present this using arrays, but it turned out to be essential to use linked lists of objects. I feel that this went beyond the proper scope of the course. (On another occasion, I might teach Walshall's algorithm for the transitive closure, which contains similar concepts but can be implemented with arrays alone.)

In point of fact, **Java is a singularly inappropriate language for abstract data types**. For example, to implement an arithmetical expression tree, we need an abstract class for expressions, together with subclasses for each of the **several cases** of the arithmetical operators. The JAVA compiler requires each of these subclasses to be in a **separate file**. On the tree, we intend to implement **several algorithms** (in this case it might be *evaluation*, the *encoding* in some machine or programming language, or maybe *symbolic differentiation*). There is therefore a **matrix** of pieces of code to write.

- Natural, modular programming would put all of the cases for **each algorithm** in a file, but
- Object-oriented programming requires **each case** for all of the algorithms in a file, so when we add an algorithm, we have to add methods to every one of the existing files.

(It was Appel's book, *Modern Compiler Implementation in Java*, pp99-100, that first drew my attention to this "transposed matrix" problem, but I have since experienced it myself in several ways.)

Maybe that makes sense for the GUI applications for which JAVA was designed, but **this style of programming would be an additional obstacle for first year students**.

In fact I consider that **there ought to be a course on data structures** — in the *second* year.

Despite my complaints about JAVA, I am happy to use it to teach such a course. JAVA has its virtues, and I would teach its use for such problems seriously. But I would at the same time invite students to form their own judgements, and take a level-headed attitude to the dogmas of Computer Science such as Object-Oriented programming.

2.2 Sorting algorithms

The goal of the group projects in the second year Software Engineering course in 2000–1 was to write a web server to help people choose holiday destinations. In the previous three years, the projects were about playing Othello, Draughts and Backgammon.

Does this mean that Norman Fenton teaches a course about travel agencies, or that Graem Ringwood and Victoria Stavridou taught about games?

Of course not. These applications are **vehicles to carry ideas** of Software Engineering, or programming in the large.

The ideas of my course are about **specification of components**, or programming in the small.

Sorting serves well as my top-level specification, because there is no need to spend time motivating or explaining it. As it is a common theme throughout the course, the students and I can put this top-level specification to the back of our minds and concentrate on the things that we can learn from the implementation of different algorithms and the specifications of their components.

Having inherited teaching materials about sorting algorithms, it was quite difficult for me (as well as the students) to see what we were supposed to do in the labs, when JAVA code for these algorithms is readily available in textbooks and on the Web.

I believe that I have now identified those things, and that is what I concentrate on teaching.

I am, of course, open to suggestions for **other** algorithms that I might consider in place of the well know sorting algorithms. In fact, such suggestions would be extremely valuable, as I am running out of ideas for exam questions.

In many ways, the textbooks (such as Weiss) **seriously got in my way**, which is why I stopped recommending any of them.

2.3 Components and specifications

For teaching the algorithms, I adopt the slogan, **one problem, one specification, one program, one loop, one invariant, one proof**.

The quadratic sorting algorithms (in particular insertion sort and selection sort) have two loops, where the inner one solves one of the following problems:

- search a sorted segment of an array for where to insert a new value,
- shift a segment of an array
- search a segment of an array for the position of the least value.

I get the students to study and implement these algorithms before putting them together with the insertion or selection sort proper. (In fact I start with simple linear search, in both directions).

These preliminary exercises are extremely valuable.

- Left-to-right linear search is the *Hello World* problem for my TESTER and this course. It shows up those students that have failed to learn **basic Java syntax** for loops and arrays, or who do not understand the difference between **pointers and values** in an array.
- Right-to-left linear search is a symmetrical problem, but they write much more complicated code, so this is an opportunity to teach **style** and **simplicity** of programming.
- I forget from one year to the next how traumatic they find the shift problem (at first they have emotional difficulties with trampling over data!), so this exercise is a **rite of passage** from writing toy programs and starting to be a programmer. Again style and simplicity of coding are the key, especially when it comes to getting both the left-to-right and right-to-level versions to work.
- I make them check the validity of the arguments of the shift method and throw an **exception**. This problem is therefore a hook on which to hang discussion of **overflowing arrays** and its consequences for **security** of programs, as well as **when to catch exceptions**. This can be illustrated by the Ariane V disaster.
- When they come to fit the components together, they learn how to **match actual and formal arguments** of subroutines.

2.4 Complexity theory

Complexity theory, like Number theory, is a peculiarly discontinuous mathematical discipline. There are some easy things, after which there is a *huge gap*: the next things to consider are *much* more difficult mathematically.

The (mainly American) textbooks on Algorithms use a lot of **Calculus**. I consider this largely **irrelevant** to the analysis of algorithms at the level that is feasible for undergraduates in Computer Science or even Mathematics. I suspect that it appears in the textbooks *simply* because of the heavy emphasis on Calculus that is traditional in the American university curriculum as a whole.

The formal definition of the *O(n)* notation itself was inherited from Real Analysis and is a handy way of describing convergence, and in particular the *asymptotic* behaviour of functions.

But *asymptotic* behaviour is **wholly inappropriate** for the understanding of how real programs behave on real computers:

- **Real hardware works differently** for kilobytes (in CPU cache), megabytes (in RAM), gigabytes (in disk) and terabytes (on some other media), so the mathematical models that apply to one do not apply to the others. In fact some of my exercises actually demonstrate the discontinuities when caches overflow.
- **Real programs** written by real students consist of nested loops, so certain **standard functions** are more or less certain to **fit statistically** throughout the range (rather than asymptotically), if anything will.

For these reasons I decided to teach the big-O notation simply as **jargon**, with terms such as *logarithmic* and *quadratic* as equivalents. (Clearly this is not an approach that a *mathematician* would choose lightly!)

What is the reason for teaching complexity at all at this level? I believe that it is to make students aware of *one* of the reasons why programs still take such a long time to execute, namely **indiscriminate use of nested loops** that traverse datasets.

Unfortunately, there is a difficulty at a much lower level: students themselves only understand mathematical formulae at a jargon or symbolic level. I am still struggling to find a way to get them to understand the practical implications of fast-growing functions (see the exam question).

2.5 Automatic testing

There is no point in writing these programs if students only **test them by poking**. My impression during the first year in which I taught the course was that many students *thought* that they were doing and had completed the exercises when in fact they had not.

The TESTER was written to be an **automatic teaching assistant** that would never let them get away with incorrect programs. As they are supposed to be writing **components**, it also provides the **main()** method: an interface whereby they can supply specific or random data to their code. The TESTER does comprehensive correctness testing, and complexity analysis. The TESTER checks the students' implementations of the algorithms for them, but by an extension of the same theme, *they* could be (black box) testing alleged implementations of specifications. The original version of the TESTER included stubs of code of code with this in mind, but I have not as yet had time to develop either this feature of my program or teaching material that would exploit it.

Having totally abandoned a mathematical approach to complexity theory, I invested a lot of effort in the **empirical** approach, *i.e.* **timing programs in labs**. In his lecture notes, Richard Bornat talks about *running races* to compare the various sorting algorithms, but the fact is that he did not have the technology to do this, and nor did I until I wrote my TESTER.

The TESTER obviously embodies a rather large **investment of time**, both in the correctness testing and the timings. Such an investment ought to be developed and utilised by the Department, not discarded.

Its primary job in the latter case is to decide *which function* fits the data, and secondarily to report to coefficients. This decision begins from the assumption that *one* of the standard functions (constant, logarithmic, linear, $n\log(n)$, quadratic or some power) is appropriate, based on the crude observation of the general nature of the code that students are likely to have written. The best fit for each function is calculated, using a standard least squares method with weighting. Then

- binary search *only* fits the logarithm (so long as the array sizes range over several orders of magnitude, say from 4 to 250,000) whilst the other algorithms do not fit it, *i.e.* the correlation coefficients are 99% and 40%;
- quadratic and $n\log(n)$ are similarly orthogonal;

- the quadratic coefficient for linear algorithms is less than 10 picoseconds, which is far less than one machine operation takes;
- the $n\log(n)$ coefficient for linear algorithms does not have such a wide margin of error, so can often be mis-identified.

2.6 Mathematical requirements

As I have explained, I have **removed** mathematical material that is traditionally contained in a course such as this.

What I **added** mathematically was the Floyd–Hoare–Dijkstra style **logical proofs of correctness**. I had learned this from small group teaching on the corresponding course at Imperial College, out of which came the book by Broda *et al.*.

I am very skeptical of the value of formal approaches of any kind (whether to correctness proofs or to Software Engineering), so my treatment is **considerably less formal** than the one I had learned, and is **largely pictorial**.

In fact I had learned **loop invariants** for myself at school, simply by **documenting** the meanings of program variables at the top of each loop. (I wrote a suite of programs to calculate mathematical functions from power series, for a programmable calculator with a small number of numbered memories, so I had to document the usage of these memories (variables), and did so "at the point of the loop test".)

In the second **exam** that I set (May 2000), students **at pass level** could reproduce reasonably accurately the loop invariant **diagrams** that I had drawn repeatedly for insertion and selection sort. So far, only the A grade students could reproduce the arguments that used the diagrams to demonstrate correctness.

Although this logical thread dominated my treatment in the *first* year that I taught the course (1998–9), I played it down in the second year. Instead, I spent more time on **empirical** observations of the behaviour of the algorithms, expressed without formal logical or programming notation.

2.7 Exam results

I put a lot of care into marking — and am criticised for doing so.

I get a spread of marks all the way from 0 to 100%, whereas a bookwork course such as *Discrete Structures* gets a bell curve between 30% and 75%.

I give a lot of well deserved A grades. These students come away having thrown themselves wholeheartedly into the course and having really learned some Computer Science.

I also give a lot of C grades. These students have done most of the work in the labs and have learned from the course.

The results trail down to and below the pass mark. I consider very carefully whether students near the boundary have learned *something* from the course, and my resit paper is worded and marked to do this too.

The percentage that pass my course is similar to that for *Introduction to Programming* and other first year courses in the Department.

At the very bottom are maybe 20 **students that have clearly opted out** of the course at an early stage (the first three weeks). The **modular course system** forces all students to register for exactly eight courses before Week 2 of the Spring term. Many students, wisely or otherwise, study actively for fewer than eight courses in each year and there is **no incentive for them to deregister**.

Despite this, the actions and opinions of these students impose a **dumbing-down pressure** on courses such as mine, both through their influence on the exam statistics and by giving them the opportunity to make **anonymous personal insults** about lecturers through multiple-choice course questionnaires. I am happy to receive considered criticism from students who *participate* in the course (the mediocre ones as well as the best), but it is **offensive** that comments by non-participants should be given **equal weight** in reviewing courses.

Therefore, **percentages are misleading** because the performance of these students is **beyond my control**. Without imputing negligence to our departmental student administrator, who of course only has the information about attendance that lecturers give her, these additional numbers are essentially a **bureaucratic oversight**.

The way that I advocate for dealing with the failure rate is to put **greater pressure on students to do the work**, and to **deregister** those that are not participating, under the College's General Regulation 5.8 regarding Attendance at lectures, *etc.*.

The TESTER provides one way of monitoring students' work, as it sends a logging message to me (by UDP) every time it is run.

The list of students currently logged in to the machines in the ITL during timetabled labs is also automatically saved.

Students have been told to do their work in a particular directory, which is copied every night to a place where I can read and analyse it.

Chapter 3

The Tester

This course is about **components**, not whole programs: you will be writing and testing **single methods** to do sorting and searching. The **main()** method that provides the "whole program" and calls your method is in the TESTER.

3.1 Aims of the Tester

- To give students immediate **feedback** on whether they are achieving the objectives of the exercises.
- To monitor the rate of **progress of the class** of students as a whole, *i.e.* that I may pass on to the next topic when 60% have achieved the objectives of this one.
- To monitor whether **individual students are participating** as required in the exercises and lab sessions.
- To teach **complexity** in an empirical way.
- To teach **trip testing** as a verification technique: a *program* (JAVA class file) is to consist not only of the code, but also the informal specification in English, the formal logical specification and correctness proof, and test data. (The term *trip test* was introduced by Donald Knuth for debugging TEX and verifying subsequent versions and implementations.) *(This has not been implemented.)*

3.2 Disclaimers

- This is not an automated marking tool and will not be used as such.
- The monitoring of students is not secure.
- Only the principal versions of the core exercises in the course are covered.

3.3 Objectives

The TESTER

- provides a `main()` so that you *only* write the *relevant* code;
- provides arguments (either explicitly or randomly) to your method and prints out the results;
- provides a `Trace.print()` method for you to use (instead of `System.out.print()`) for debugging and observing loop invariants; it is enabled or disabled from the command line;

- does thorough trip-testing of your method and prints (and logs) a report;
- does time-complexity testing, reporting the coefficients of the best-fitting curve, and drawing it for you (using `gnuplot`).

Later versions of the Tester will

- provide animation, profiling and a GUI;
- run your own trip tester both on your code and on a supplied collection of correct and incorrect algorithms.

3.4 Using the Tester

The Tester can

- `run` your method once with random or specified arguments,
- `trace` its behaviour, or
- `check` that it is correct and time how long it takes.

These three commands (`run`, `trace` and `check`) are provided in the directory **DCS127** in which you are to do your work for this course. They are not standard Linux commands and will not work elsewhere.

The `run` and `trace` commands compile your class before running it, as there are some complicated extra arguments that have to be given to `javac` to make it connect your class with the Tester.

The `check` command does not compile your class first. This is to make you do some individual runs before the comprehensive test.

Do not use the `javac` and `java` commands directly.

You invoke the Tester by typing commands such as

- `run Linear.java`
- `trace Linear.java size=15 strictly reverse sorted`
- `trace Linear.java array=2,4,6,8,0,9,7,5,3,1 seek=5`
- `check Linear.java`
- `check Insertion.java sorted`
- `check Insertion.java displacedpc=1`

where the first argument after the command name is the name of your source file.

You are recommended to keep the Java filenames of the exercises as they are given, but you may, if you wish, copy the templates to files with different names, for example if you want to try a different way of doing things. In this case you also have to change the name of the **class** that the file defines. Then you can simply run the Tester using the new filename.

3.5 Running your method once using the Tester

To run the Tester on the first exercise (linear search), type

```
run Linear.java
```

to which the response will be something like this:


```
Recompiling your class file ...
Using /homes/pt/DCS127/tester.jar
QMW/DCS/127 "Intro to Algorithms" Tester, (v2: 2001.03.29)
<pt@turing> Java v1.2.2 11:13 Fri  2 Nov 2001.

Problem: Searching unsorted data; Experiment: run with multiple threads;
Source: Linear.java; Last modified: 13:13 Sat  3 Feb 2001.

Your method was asked to search for 11 in the following array:
# 18 13 12 12 12 10 18  8  0  0  7  8 10 16  5 16  5  4 17  5  9#
The position returned was -1, which is outside the array,
the required value being absent.
```

The hash character (`#`) is placed in the printout of the array to the left of the position that was returned by your method. If the value is not present, or if your method alters the array, returns an incorrect result, fails to terminate or throws an exception, then this is reported.

If you do this again you will get another random array of length 20.

You can specify the `size` (length) of the array, the `range` (magnitude) of the numbers that it is to contain, or that they are to be `sorted` (in ascending order), as follows:

```
run Linear.java size=100 range=50 sorted
```

However, if the array is too big for the width of your screen then it will not be printed out in the report.

You can instead specify the actual numbers to be used in the array, and the value to be found:

```
run Linear.java array=3,5,7,9,0,8,6,4,2,1 seek=4
```

Beware that **no spaces** are allowed in the `array` specification.

3.6 Debugging and tracing

If you are having difficulty with getting your method to work, you can make it print out messages by including lines like

```
 Trace.println("position=" + position + ", contains " + array[position]);
```

in the code. **Do not use `System.out.print` to do this**. `Trace.println()` does the same thing, but it is silent unless the `trace` keyword is used:

```
trace Linear.java
```

This means that, when you come to do time-complexity testing (which runs your method *thousands* or even *millions* of times), the resulting spew of output will be suppressed.

(In general, you should never delete debug lines like this from your source code. When you have finished with them, either **comment them out** (*i.e.* put `//` in front of them) or — better — make them switchable by writing things like

```
if (verbose > 3) { Trace.println (''position='' + position); }
```

so that you can use the debugging lines again later by setting `verbose` to an appropriate value.)

You can watch the progress of the algorithm on the array by calling the method

```
Trace.printarray (array, i, j, k);
```

whose arguments are the array and **up to five** pointers. This prints output similar to that shown in the simple report above.

3.7 Thorough correctness testing

When you are satisfied with the way that your method works, you can subject it to much more thorough testing by typing

```
check Linear.java
```

the TESTER will then make a more general report on the behaviour of your method, such as

```
For values that are actually PRESENT in the array, your method finds the
LEFTMOST occurrence.  For ABSENT values, it returns -1.
```

This is **not an alternative to your own testing**. The report is deliberately not specific enough to tell you how to modify your code to make it work: in real life program bugs manifest themselves in far more obscure ways than this. You must do single runs, as in the previous section, to debug your code.

3.8 Timing and time-complexity testing

If the TESTER considers that your algorithm is reasonably close to being correct, it will run it several thousand times on arrays of varying sizes, and tell you how long this took:

```
    size      time (seconds)    repetitions
      30      0.000 112 5          10814
      40      0.000 110 6          10000
      50      0.000 112 6           9445
     ...      ...                    ...
   70000      0.003 778             1094
   80000      0.004 295             1074
   90000      0.004 718             1056
  100000      0.005 19               180
Your method has LINEAR time complexity, within 5%:
for size n it takes 50 + 0.0535 * n microseconds.
To see the graph, please type
     gnuplot LinearInsertion.plot
This is a PLAIN TEXT file: LOOK AT IT to see
how to send the graph to the printer.
```

It takes considerably longer than the time shown to do this test, because (in order to achieve some experimental accuracy and reliability) your method is called thousands of times. The timing shown is for *each call*.

When several timings have been made, the TESTER attempts to fit a straight line or some other function to the graph of timings. The percentage at the end shows how successful that was: 2% is good, 9% is bad and any more than that is reported as failure.

The TESTER then draws a graph to show both the individual timings and the best-fitting curve through them. To see this do

```
gnuplot Linear.plot
```

(or whatever filename the TESTER tells you). This file is *plain text* and contains commands in the `gnuplot` plotting language. *Read it* for instructions on how to send the plot to the printer.

You can run the test for another sequence of sizes like this:

```
check Linear.java size=10000 by=10000 to=100000
```

This will give the arithmetic progression 10000, 20000, 30000, ..., except in the case of `Binary` search, where `by` *multiplied* instead, so


```
check Binary.java size=1 by=2 to=1000000
```

gives the geometric progression 1, 2, 4, 8, 16, ..., 1048576.

As with the single runs, you can use `sorted` arrays and state the `range` of numbers that they contain, but specifying the actual contents of the `array` or the value to `seek` has no effect.

The TESTER may find that the times reported are too erratic to fit the appropriate function to them. Unfortunately, linear search is the most unreliable algorithm in the course from this point of view! This is because *the entire program and array* is loaded into the CPU cache and executed very fast, without accessing RAM at all — usually. Occasionally, however, this process is interrupted by something else, and ends up taking *much much* longer.

You should **shut down Netscape** before trying to do these timing experiments, because (even when you're not looking at any Web pages) from time to time it takes up memory and CPU time with its activities. The Gnome window manager is another culprit: it is Similarly, if you are competing with the other students for CPU time on the communal machines (bronwyn or linus) then your timing will be less reliable than on a machine that you alone are using.

The printout of timings in the TESTER was designed to emphasise that you have at best three digits of precision, because the JAVA system clock only measures milliseconds, and may not do that very well. This does not, however, seem to be an important factor in how well curves can be fitted to the experimental data.

3.9 Summary of Tester command-line arguments

For single runs, the TESTER provides a random unsorted array that can be printed across your screen, containing values from 0 to the size. So *on average* each value in this range occurs once.

The various problems that the TESTER encodes may set different default values, and the comprehensive correctness tests largely set their own parameters.

The timings are done with arrays of various sizes, so some of the options below set parameters in ways that depend on the size.

Many of the options are applicable only to certain problems, or to certain algorithms.

In fact, the arguments are passed using the normal Unix/JAVA mechanism, being separated by spaces. It does not matter in what order you write them. An argument is either a single keyword, or a `keyword=value` pair. If the value contains special characters such as space or `*()[]<>?|$!&` then you must put (single) quote characters around the argument.

The following command line parameters are used, with examples:

- **array=1,3,5,7,9,0,8,6,4,2**: use exactly this array (be careful *not* to include space characters in the string).
- **choice=3**: the TESTER calls `choice(3)` in your class (this allows you to put several attempts or versions in the same class file, as the cases of a `switch`, as for example in the exercise on binary search).
- **convex=true**: make a sawtooth like 0,1,2,3,2,1,0 instead of 3,2,1,0,1,2,3 (for *quicksort*).
- **cutoff=25**: call `cutoff(25)` in your class (for hybrid *mergesort* and *quicksort*).
- **debug**: verbose output for me to debug the TESTER.
- **density=2.0**: use range=size/2, so each value occurs on average twice.
- **dest=5**: call `shift (array, source, 5, len)` (for *shift*).
- **displaced=5**: scatter 5 random values in a sorted array (for *insertion* and *bubble sort*).
- **displacedpc=1**: scatter 1% of random values in a sorted array.
- **easy=true**: give the `shift` method easy arguments (for *shift*).

- **fit=linear**: accept linear complexity and generate the graph, however bad the experimental results are (similarly **logarithmic**, **quadratic**, **nlogn** and **power**).
- **help**: print a help text.
- **largesizes** use a sequence of array sizes from 8192 to 262144.
- **len=5**: call `shift (array, source, dest, 5)` (for *shift*).
- **'markers=[]/'**: use these characters instead of # in `Trace.printarray` (be careful to enclose this in quotes to protect it from interpretation by the operating system).
- **maxtime=60**: how long the Tester should wait before concluding that your method is not going to terminate.
- **plot=file.plot**: the name of the file in which to write gnuplot commands to draw a graph.
- **quiet**: reduce the output, in particular don't print out all the timings.
- **range=100**: how big the random numbers should be.
- **range_lower=10**: random numbers to be at least 10.
- **range_upper=100**: random numbers to be less than 100.
- **restmore=true**: the numbers in the unsorted segment after *upto* are to be greater than those in the sorted segment before this point (for `LocatingMinimum` for *selection sort*).
- **reverse=true**: sorted in descending order.
- **sawtooth=2**: sawtooth pattern like 6,3,1,2,3,4,5 that *starts* with a descending sequence of 2 numbers (for *quicksort*).
- **sawtooth=**−2: sawtooth pattern like 5,4,3,2,1,3,6 that *ends* with an ascending sequence of 2 numbers.
- **sawtoothpc=50**: sawtooth pattern with its vertex at 50% of size.
- **screenwidth=80**: this controls line-breaking of text messages and the default array size, and is set in the `run` perl script.
- **seek=5**: call `search (array, 5)` or `where_to_insert(array, upto, 5)` (for *linear* and *binary search*).
- **showfit**: print information about curve fitting for me to debug the Tester.
- **size=15**: use an array of size 15 for a single run.
- **sizes=30,40,50,60,70,80,100,120,140,160,180,200,220,250,300,400,500,600,700,800, 1000,1200,1500,2000,2500,3000,4000,5000,6000,7000,8000,10000,12000,15000,20000, 25000,30000,35000,40000,50000,60000,70000,80000,90000,100000**: use these sizes for timing (this list is the default, in case you want to cut and paste part of it).
- **smallsizes** use a sequence of array sizes from 256 to 8192 (for *binary search*).
- **sorted=true**: generate sorted arrays.
- **source=5**: call `shift (array, 5, dest, len)` (for *shift*).
- **strictly**: generate sorted arrays without repetitions.
- **thread=true**: use threads to catch non-terminating processes and for timing.
- **upto=5**: call `where_to_insert(array, 5, seek)` or `where_is_min (array, 5)` (for *search to insert*).

3.10 Running the Tester without the perl script

If you are working with Linux in the ITL, you will find that Tester and the `run`, `trace` and `check` commands for using it have been set up for you already in your work directory (DCS127). You do not need to do anything to install them.

The Tester itself will be updated as the course progresses, as each exercise that you do needs me to write additional testing code. So long as you leave the commands in your work directory as they are, you will get the new versions automatically.

The Tester **may** be able to work with Windows NT, but I haven't tried it. The timing experiments have been calibrated to work under Linux on the student workstations on the ground floor of the ITL: the timings may work on other machines, but this is not guaranteed.

In order to take the Tester home you will need both the Java Archive file `tester.jar` containing the Tester itself and the perl script for the `run`, `trace` and `check` commands. This *single* file, called `run`, can do "`run`", "`trace`" OR "`check`" — which of the three it does depends on the name by which you call it. So install it as `run` and then do

```
chmod +x run
ln -s run trace
ln -s run check
```

to make it executable and create symbolic links (aliases) for it.

You can use the Tester without using this perl script at all (for example if you only have Microsoft Windows at home), but this is more complicated.

Assuming that "tester.jar" is in the current directory, to compile your class (say, `Insertion.java`) you have to do

```
javac -classpath tester.jar Insertion.java
```

and then to perform a single run you do

```
java -cp .:tester.jar Tester Insertion.java
```

or with tracing,

```
java -cp .:tester.jar Tester Insertion.java trace
```

and to do the comprehensive correctness and timing test, do

```
java -cp .:tester.jar Tester Insertion.java test
java -cp .:tester.jar Tester Insertion.java time
```

You can add other options to the end of the command line in the same way as you would using this script under Linux in the ITL.

If your code throws an Exception, you can get a more informative indication of where it happened than "Compiled code" by doing

```
java -Djava.compiler=NONE -cp .:tester.jar Tester Insertion.java
```

Since this runs the Java run-time *interpreter* (instead of compiling the Java class file into machine code immediately before running it), your code will run more slowly. *This doesn't matter*, since we're learning about the *functions* (linear, logarithmic, quadratic, $n\log(n)$, *etc.*). It's just that you can't *compare* timings made using the interpreter with those made using the compiler.

The `classpath` argument connects your code to the Tester, whilst `NONE` makes the Java run-time system use the interpreter, as a result of which you get line numbers (instead of "Compiled code") in the stack trace when an exception occurs. By default, Java 1.2 uses a "just-in-time" compiler, *i.e.* it translates the (machine-independent) class file into machine code immediately before running it.

Chapter 4

Linear search

The first exercise is to find (an occurrence of) a value in an array on integers by examining each cell from one end of the array to the other.

You need to write fewer than six lines of Java.

4.1 Setting up your work for this course

There are two *very important* differences between the exercises for *Programming I* and those for this one. **This is not "just another programming course".**

- This course is about *components*: you *only* write the method that does the specific job in hand, in this case searching an array. It is *not* about *whole* programs, so you will *never* be writing

  ```
  public static void main (String[] args)
  ```

  as the `main` method is provided by the TESTER program that I have written for this course. Nor will you be allowed to write

  ```
  System.out.print (message);
  ```

  because the TESTER will call your method *millions* of times (to check that it's correct in different cases and to time it). Instead you can write

  ```
  Trace.print (message);
  ```

  which will only print the message if you invoke the TESTER with the `trace` command, and not if you invoke it with the `run` or `check` commands.

- This course is about *specification* of problems and the *behaviour* of algorithms to solve the problems. Much of your time will be spent on *experiments* to find out how the different algorithms work and how fast they are. This course is as much a continuation of *Introduction to Logic* as it is of *Programming I*: it is the *application* of logical techniques to programming. You will be using *logic* to describe *precisely* what the algorithms do.

As in *Programming I*, you are to use the **Linux** operating system for this course, and run your programs from a "command line" and not from a GUI (graphical user interface). You will need just two windows on your screen:

- an editor, for which I strongly recommend **xemacs**, and
- a terminal window (shell, **xterm**).

Begin by making these as large as possible, to make good use of the space on your screen. The TESTER will generate quite a lot of lines of output, and you will want to see as much of this and of your program as you can fit on the screen.

You do not need Netscape. Later you will be doing experiments to find out how long your code takes to run. Netscape is an enormous and bug-ridden program that is very wasteful of system resources; if it is active on your workstation, it will interfere with these experiments, and the results of these experiments will be inferior.

4.2 Writing the Java code

In your personal filespace you will find that a directory called `DCS127` has recently been created for you. *All of your work for this course must be done in this directory.* Do

```
cd ~/DCS127
```

In the directory is a file called `Linear.java`. This is a template in which you will fill in the code for the search method.

It already contains the following:

```
// DO NOT CHANGE the "class" line
public class Linear implements Searching {

   // Here is the one method that you have to write.
   // Do not change the method declaration
   // (yes, it must be "public" and NOT "static").

   public int search (int[] array, int seek) {
     int size = array.length;

     // A logical formula
     // (the "precondition" of the method)
     // belongs here.
     // This will be discussed in the next lecture:
     // please just leave the comments about logic
     // here as they are,   and add the logical
     // formulae during or after the lecture.

     // You ONLY need to declare this one local variable.
     int position;

     // Another logical formula
     // (the "precondition" of the loop)
     // belongs here.

     // Please use "while", NOT "for", in this course.
     // (The thing that you write in brackets
     // will be called the "loop test" in this course.)
     while (     ) {

         // Yet another logical formula
         // (the "loop invariant")
         // belongs here.

         // Write your code (the "body" of the loop) here.
```

```
    }

    // The fourth and final logical formula
    // (the "postcondition" of the loop and of the method)
    // belongs here.

    // Here we say where "seek" occurs in the array
    // and go home.
    return position;
  }

  // PLEASE IGNORE THESE LINES - DO NOT CHANGE THEM
  public static int choice = 0;
  public void choice (int c) { choice=c; }
  public boolean sorted_data () { return false; }
}
```

Note: "seek" is an irregular verb: I *seek*, I *sought* (past tense), I have *sought* (past participle), the value is *sought* (passive participle).

Be careful to distinguish between *pointers* or *indices* into the array and the *values* that are contained in it.

4.3 Running your program

When you have written your code, **in the terminal window** do

```
run Linear.java
```

This will compile your code for you (do not use `javac`) and `run` it on a random array of length 20.

Do this several times.

You can make the TESTER call your method with a longer or shorter array, or one that is sorted (ascending), reverse sorted (descending) or strictly sorted by addition combinations of words like

```
run Linear.java size=10 strictly reverse sorted
```

to the command.

Alternatively, you can tell it *exactly* what array to use:

```
run Linear.java array=2,4,6,8,0,9,7,5,3,1 seek=5
```

(beware that there must be *no spaces* in the list of numbers).

4.4 The experiments

Now answer the following questions *in your* JAVA *source file* (`Linear.java`). The source files for each algorithm will then contain everything that you learn about that algorithm, and will form the basis of your exam revision notes.

In the situation where `seek` occurs *twice or more often* in the array, which `position` of the two or more occurrences does your method indicate?

What `position` does your method return if `seek` is *not* in the array? In what way does this number depend on the size of the array?

Why is this an *appropriate* result for the method to return in this case?

What other result value might you choose to return in this case instead? Give a reason.

How would you change the code to make it return this alternative result? (Insert your changed code following this question in `Linear.java` — do not *change* what you have written towards the beginning of the file.)

Now do this:

```
check Linear.java
```

"Cut and paste" what the TESTER reports at the place indicated in `Linear.java`.

Does the TESTER confirm what you observed about the cases where `seek` is either absent or occurs more than once in the array?

Assuming that you have written your code in the very simple way that was intended, you should find that its behaviour can be described *more precisely* than what you were originally asked to do, or expected yourself than it would do. In what way is it more precise?

When your code is correct and you have finished your answers to these questions, print out this file (including your answers) by doing

```
a2ps -Pitlevel1 Linear.java
```

or

```
enscript -Pitlevel1 Linear.java
```

Show your printout to the teaching assistant, and bring it to the next lecture, so that you can write in the logical formulae mentioned above.

Then look at the file `RLinear.java`. It is essentially identical to `Linear.java`, but you are asked to do write code to do something slightly different, and then answer the same questions about it.

4.5 Searching to insert

The first few exercises lead up to the implementation of **insertion sort**. The idea of this algorithm is to insert successive values into the segment of the array that has been sorted so far, by **searching** for the appropriate position and **shifting** the intervening stuff.

The specification for **search to insert** is more complicated than that for the linear search. It has to meet the **requirements** of the application that will use it (insertion sort). These are different in two ways:

- you have to say where to **insert** the value (`seek`), rather than where it **already occurs** — which it probably won't;
- you are only looking in *part* of the array, namely **upto** the end of the so-far sorted segment.

What you have to *provide* is therefore slightly more difficult, but what you are *given* is also more generous: this segment of the array is **already sorted**.

The file `ILinear.java` is in your coursework directory.

The template is

```
class ILinear implements SearchingToInsert {
   public int where_to_insert (int[] array, int upto, int seek) {

   }
   public void choice(int c) {}
}
```

which is similar to linear search, apart from the extra argument `upto`.

When linear search failed to find `seek` in random data, it could do nothing more informative than return `-1` or `size`.

However, when the array is `sorted`, it can say `where_to_insert` the new value **to keep the array sorted**.

If `position` is the place *before* which to make the insertion, what inequalities hold amongst `seek` and the values in the array?

Write the code and use the TESTER to try it out as you did for the striaghtforward linear search.

4.6 Correctness and complexity

This was part of the model answer for the first test in 2000; the rest of the test was about binary search.

Linear search takes typically

size $* 50$ nanoseconds

to execute. The time depends *linearly* on the length of the array, but you should also be aware that it takes a fraction of a microsecond to go round the loop once.

Also, if there are not very many *different* values occurring in the array (for example, if they are integers that are significantly smaller than the length of the array), then the first occurrence of a particular value will be found sooner.

This code returns the position of the *first* or *leftmost* occurrence of `seek`. The array does not have to satisfy any special requirement for the method to conform to this specification.

More generally, if we regard the array as a *function*, $f : i \mapsto \texttt{array}[i]$, the search for a particular value x is trying to calculate the *inverse* of the function, $f^{-1}(x)$. This inverse value is a *set*, which may be empty, a singleton or have more than one element. However, `search` is specified to return, not a set, but an `int`, which, moreover, must be an *element* of the set $f^{-1}(x)$ if this set is non-empty. It is not, therefore, able to tell us everything there is to say about the set $f^{-1}(x)$.

Treating it as an inverse function is therefore not the appropriate way of making the specification of `search` more precise. Instead, we consider the way in which it may be *used*, for example as a subroutine in insertion sort.

In the case where $f^{-1}(x) = \emptyset$, *i.e.* the value sought does not occur anywhere in the array, a user such as insertion sort would be interested in the position before (or after) which to insert the value. This requirement only makes sense if the array is sorted.

By changing the code to

```
int position = 3; while (position < 14)
```

it will search the segment `array[3]` to `array[13]` inclusive, instead of the whole of the array.

Note that `while (pos >= 3 && pos <= 13)` *does not work*: following the initialisation `pos=0`, the loop condition fails immediately, so `array[3]` is never considered!

4.7 Loop invariant diagram

(To be written.)

4.8 Searching for the minimum

(To be written; this is for selection sort. There could be more exercises on other similar linear algorithms.)

Chapter 5

Shifting a segment of an array

Every time you move a window around on your screen (or even move the mouse), a *very* low-level subroutine is being called to copy a chunk of stuff from one place to another, overwriting whatever was there before.

The code for **shift** is not as easy as it may seem. When you have *completed* this exercise, you will have undergone a "rite of passage" towards being a programmer, as opposed to someone who merely uses a computer.

You are not allowed to allocate a new array!

Like linear search, this is a very low-level operation, and one that not quite so low-level applications will call *very* frequently and expect to be done *very* quickly.

We shall learn logical techniques for proving *correctness* of these algorithms.

5.1 Writing the code

There is a file called `Shift.java` in your coursework directory. As for linear search, you should do all your work for this algorithm in this file, including what you learn about it from the TESTER and in lectures.

The class declaration is

```
public class Shift implements Shifting
```

and your shift method has the signature

```
public void shift (
   int[] array,   //  work on this array, in place
   int source,    //  the index of the first cell from which to copy
   int dest,      //  the index of the cell into which to copy this value
   int len)       //  the number of cells to copy
```

At the top of the file you will find a **diagram** that shows which parts of the array are to be copied where, and *which parts stay the same.*

The loop invariant diagram should be included here!

Make a habit of drawing diagrams like this before you start to write Java code. As you see, it can be done adequately with ASCII characters. There is no point in trying to be elaborate or artistic.

What is the intended effect of the shift method? State, not only what has been changed, but also what has stayed the same. This is the **post-condition**. Get into the habit of **stating it as a comment** in your code.

As usual in this course, the part of the code that does the useful work of this method is a single loop, and you should use `while`, not `for`, because we want to write some logic or draw some diagrams in certain places in the code.

There are three things that the loop **counter** might count:

- the position *from* which you are copying,
- the position *to* which you are copying,
- from 0 to `len-1`, or
- there may be two counters, doing both of the first two.

Write all four versions of the code in the file.

Which is the clearest of the three, and why? Make a note of your reason as a comment in the code, and leave the other three versions there as comments.

Which version would be most efficient in machine code? It is one of the jobs of a compiler to generate machine-efficient machine code from your human-clear source code. *Write your code for humans to read, and make the compiler do the work to make it efficient for the machine.*

5.2 Running your program

When you have persuaded the JAVA compiler to accept your code, you can use the TESTER to perform individual runs as follows:

```
run Shift.java size=20 source=15 dest=3 len=4
```

As before you can specify the actual values in the array, rather than getting random ones.

The TESTER provides random arguments (with which the first version of your code should work) when you use the command

```
run Shift.java easy
```

You can watch your method at work by adding lines like

```
Trace.printarray (array, a, b, c, ...);
```

to your code, where `a`, `b`, `c`, ... are pointers into the array. (You can give up to five pointers as arguments to `Trace.printarray`.) Do this immediately before, immediately inside and immediately after the loop. Use the `trace` command instead of `run` to invoke the TESTER in order to make these `Trace.printarray` statements effective.

5.3 The pre-condition

What numerical inequalities must be satisfied by `source`, `dest`, `len` and `array.length` for this to be meaningful? This is the **pre-condition**. Get into the habit of **stating it as a comment** in your code.

On this occasion, it is appropriate to **test the pre-condition** at the beginning of your method and

```
throw new IllegalArgumentException ("bad arguments for shift");
```

if it is not satisfied. (Do not use `System.out.print()` or `Trace.print()`.) You can make the message more informative if you wish.

This is the *only* Exception that you will `throw` when implementing the algorithms in this course.

Do not catch this exception. It indicates that **your caller has a bug**. Do not conceal bugs by indiscriminately catching exceptions. It is better to have your program (or your caller's program) **crash** during development than when it is in service and could do some damage.

5.4 Shifting towards the right

Does your code work with the following arguments?

```
trace Shift.java size=20 source=10 dest=13 len=6
```

In which direction is the shift being made (leftwards or rightwards) in the two cases?

What has gone wrong?

On what condition, involving `source`, `dest` and `len` does the shift work or not work?

Write a version of the code that would work in the case where the original did not. Would this work in the situation where the original one did, making it redundant, or do you need both versions to cover all cases?

As with the right-to-left search, you should take advantage of the *symmetry* of the problem to make your code as clear and simple as possible.

When you have code that you think will work in all situations, use the `check` command to test it thoroughly. No timings will be done for this algorithm.

5.5 Another way to do the two cases of shift

You probably implemented the two cases as *separate* loops contained in the two branches of a conditional.

Now, instead, define an extra variable `step` that takes the value `+1` or `-1` and re-write the code as a single loop, in which the loop variable `j` is incremented by

```
j += step;
```

instead of by `j++;` or `j--;`. You will need to define `step` and the initial and final values of the loop variable within a conditional *before* (not containing) the loop.

Discuss these two programs — one with two versions of the loop in the branches of a conditional, the other with just one more complicated loop — in your small-group tutorial.

- What advantage does the first have, in terms of *speed*?
- What advantage does the second have, in terms of *future maintainability* of the program?

5.6 Loop invariant for shift

This is the model answer for the test that was set in Week 5 in 1998–9.

Given an array `array[ ]` of `int`s of length `n`, together with two pointers `src` and `dest` into it, and a number `len`, we are required to **shift the segment of length `len` that starts** at `src`, so that **the copy starts** at `dest`.

Be extremely careful before you read English meanings into the names of program variables — this does not say that `src+len == dest`! [When this was origianlly written, the variables were called `from` and `to`.]

For most of the discussion (as mathematicians say, "without loss of generality"), we assume that `src` > `dest`.

It is possible that the source and destination segments

- *overlap* (`src < dest+len`), in which case part of the source gets overwritten, or
- they may *abut* (`src == dest+len`), or
- there may be a *gap* between them (`src > dest+len`), the entries in which are likely to stay the same.

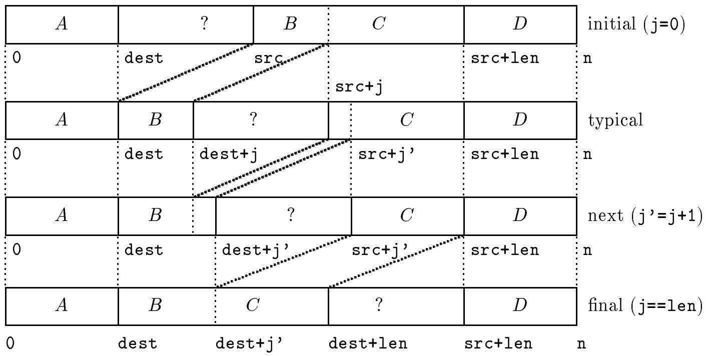
(a) The *loop invariant* is described by the following diagram. The parts B and C get copied, whilst we guarantee that A and D will remain the same. There is *no guarantee* that any part of the source segment will remain the same. In the "typical" situation, B has already been copied, and C remains to be done. In the third case, there would be a third segment (the *gap* between the destination and source segments) that we could also *guarantee* would stay the same.

The diagram is based on the first case, because this draws attention to the *possibility* that the source segment may get overwritten. We are not *assuming* an overlap: the point is that we *must not assume* that the source stays intact (compare drawing diagrams "in general position" in geometry).

In practice, the program will leave the right part of the source intact — in the case of the left shift — whilst the right shift would leave the left part intact. *Regard this as an accident.* You do not *guarantee* anything. Some of the copied values might accidentally be equal to the values that they overwrite, but we wouldn't think of guaranteeing that!

Notice also that (to mis-quote Abraham Lincoln) the program shifts *some* of the values *all* of the way — not all of them some of the way, like a tube train going through a tunnel!

Always label the *right* of the boundaries. It is *wrong* to put the label *on* the boundary: an array consists of *discrete cells* indexed by *integers* (like days of the week), it is not *continuous* with pointers to infinitesimal instants of time! It is *bad practice* to label (the right of the lower boundary and) the left of the upper boundary (`len-1`): the length of the segment is given most simply by *subtraction*. (It is for the same reason that arrays in modern programming languages are indexed from `0` and not `1`.) Do not introduce unnecessary difficulties into your program in the form of `-1`s that have nothing to do with the problem! In practice, programs are complicated enough as it is, without making them gratuitously worse.



For this diagram (and therefore the program) to be meaningful,

(b) the input values `n`, `src`, `dest` and `len` must all be ≥ 0, and `src` + `len` $\leq$ `n`.

In practice, a method in a library that implements this algorithm should *check* these conditions *before* doing anything to the array, and *raise an exception* if any of them is not true. As a point of programming style, always make *all* of these checks *at the beginning.* Think of it this way: if a surgeon is about to perform an operation on you, you want to be completely sure that s/he's fully qualified, *before* opening you up!

(c) The loop is *initialised* by `int j=0;`

(d) Since then `dest == dest+j` and `src == src+j`, the B part of the diagram is empty, so the *typical* situation is in fact the *initial* one: nothing (B) has been copied, and everything (C) remains to be copied. To achieve this instance of the loop invariant, we have to do *nothing* apart from setting `j=0;`

(e) The loop *condition* is `while (j < len)`

(f) The *body* of the loop is `array [dest+j] = array [src+j]; j++;`

(g) We need to assume that the loop *condition* (e) was **true** before executing the code inside the loop in order to know that the code is copying the appropriate values. Otherwise it would violate our guarantees about leaving A and D alone, and might even `throw new ArrayBoundsException`.

Being enclosed in a `while` (or `if`) *does not prevent* the code from being executed by the machine. Think of the condition as a ticket collector on a gate: there may be ways to dodge the check! There are tools (*debuggers*) that can interfere with how programs run in arbitrary ways. Also, if there is a *label* inside this block of code, and a `goto` that jumps to it (some programming languages and many obscure programmers can do much worse than this) then the machine may find itself inside the block without the condition being true. The ticket collector *only* stands at the gate: there are no "spot checks" inside the block!

So the force of the loop condition is not on the *execution* of the program, but on what we *know* (can *prove*) about it.

(h) We show that the loop *invariant* (a) is restored *after* this code, and is therefore valid at the beginning of the next iteration of the loop, as follows:

- The loop invariant says that A and D (and the gap, if any, between the source and destination segments) are unchanged from the initial situation to the top of the present loop. But the body of the loop doesn't touch them, so they are still unchanged at the bottom of the loop body.
- Similarly, the part B that has already been copied stays there.
- Also, the part C that remained to be copied was guaranteed by the loop invariant to be the same as it was initially, and is not overwritten by the body of the loop.
- One element is transferred and the pointers are incremented, but by the loop condition they remain within the appropriate bounds.

In this way the *next* version of the diagram is the same as the *typical* one.

We prove that the loop terminates by observing that `len-j` is a non-negative integer, but is decreased by 1 on each iteration.

(i) When the loop condition (e) is **false** (so the loop exits), we have `j==len`, so the *typical* version of the loop invariant diagram coincides with the *final* one. Hence *the whole segment has been copied as required* and *the guarantees* about A and D *have been honoured.*

(j) If instead `src` < `dest`, the program attempts to do a right shift, but it may overwrite part of the source segment before it is copied to the destination, giving repetitions of part of the source instead of copying it correctly.

(k) The reasoning above with the loop invariant is no longer *applicable*, because parts B and C *overlap*, so we no longer know that C has remained intact.

(l) If, however, `src` + `len` $\leq$ `dest`, so the destination is on the right of the source with no overlap, the program works, and more or less the same reasoning applies, because C does remain intact.

Notice that, in (k), the reasoning *is still valid*: it is a case of $P \Rightarrow Q$ where both P (the assumptions or pre-conditions) and Q (the conclusions or post-conditions) are false.

There are two *symmetries* in this problem: between the left shift and the right shift, and between the source and destination.

The assignment that lies at the heart of this program is designed to exploit the second symmetry: the loop counter makes the correspondence between the source and destination cells *obvious* by not being biased towards one view or the other.

Any modern compiler should be capable of translating this clear idiom into

```
int end_src = src + len;
while (src < end_src)
   { array [dest++] = array [src++]; }
```

which is a well established idiom of *machine code* programming. *Make the machine work for you, not vice versa!*

Chapter 6

Quadratic sorting algorithms

You have probably heard of a sorting algorithm called "bubble sort" — it is included in this course simply to show how stupid it is.

We shall concentrate on algorithms based on these two ideas:

- **insertion sort**: you have some sorted stuff already, and insert the next value in the appropriate place;

- **selection sort**: you have some of the lowest values already sorted, and search the rest of the data for the least remaining value.

A fourth algorithm, **permutation sort**, is only considered very briefly. The idea of this one is to find out the final position of each value before we move anything at all.

It is very common to be given data that are *almost* sorted — just a few (or a small percentage of) items are out of place. Insertion sort performs very well in this situation: it basically verified that the values are in order, and does additional work proportional to the extent of the divergence from sortedness. Bubble sort apparently does something similar — sometimes, but it can go very wrong; there is an exam question about this.

We study these algorithms from three points of view:

- The "model answers" built into the TESTER allow you to watch the algorithms in action, and find out how long they take to run, without having to program them correctly yourself first.

- Then you will program them yourself.

- We shall also prove correctness using loop invariant diagrams.

There is also a handwritten handout that describes the four quadratic sorting algorithms. This needs to be incorporated.

6.1 Animations

Use Netscape to watch the animations of insertion sort, selection sort and bubble sort that Jon Rowson and Daniel Dor-Chay wrote. (Daniel was a first year *Algorithms* student when he did this.)

These run correctly with Netscape (with, of course, JAVA enabled) under **Linux**, but for some reason they don't work properly with Netscape on NT.

There are other animations elsewhere on the web.

However, you should close Netscape down altogether before you start doing timing experiments, as it will muck things up for you!

6.2 The model answers inside the Tester

The TESTER can also animate the algorithms, albeit in a less colourful way.

It has model answers for most of the exercises in the course built in to it. These can be tested in much the same way as you would test your own code, except that the first argument to `run`, `trace` or `check` is now the name of the algorithm *without* `.java` on the end.

(a) Try the following, adding suitable arguments to consider almost sorted data:

```
trace insertion-sort
trace selection-sort
trace bubble-sort
```

As we shall see, bubble sort is the slowest, but the tracing output is a bit misleading, as it doesn't show the search operations that insertion sort and selection sort have to perform first.

(b) Now do the following to find out how long the various sorting algorithms take:

```
check insertion-sort
check insertion-sort sorted
check insertion-sort reverse sorted
check insertion-sort displaced=5
check insertion-sort displacedpc=1
check selection-sort
check bubble-sort
check permutation-sort
check merge-sort
check quick-sort
check heap-sort
check shell-sort
```

together with the obvious variations.

(c) Rank the algorithms in order of speed.

(d) At what size do the $n\log(n)$ algorithms overtake insertion sort on unsorted data?

(e) How does the coefficient of n^2 depend on the amount of unsorted data?

(f) Work out (from the formulae) how long these algorithms would take to sort a million numbers.

(g) **Conversely, what is the *biggest* array that they could sort in an hour?**

Textbooks indulge themselves in tabulating the values of functions such as these (*i.e.* the time taken by certain algorithms) for various given values of n. These values sometimes need to be written using *towers* of exponents.[1] The *practical* question is, however, the *inverse* of this: visitors to your web site are willing to wait for a certain amount of time (one second, maybe), so *these functions place a bound on the maximum size of your dataset.*

6.3 Almost sorted data

The TESTER can supply various kinds of random data to your method, both in the single runs and for the timings. Try the following arguments:

- sorted
- reverse sorted
- strictly sorted
- displaced=10

[1]How long do you think the Universe will last? The result is qualitatively the same whether you believe in modern physics or divine creation in historical time.

- displacedpc=.5

where the last two provide sorted data and then scatter either 10 or 0.5% respectively of random values in random positions.

The meanings of the follow should be obvious:

- size
- range
- range_upper
- range_lower

For quicksort we shall use

- sawtooth
- sawtoothpc

to provide a “triangular” (decreasing, then increasing) distribution.

When you use `check` to do timings, the TESTER now creates a gnuplot command file to draw a graph. The you can look at the graph on the screen by doing

```
gnuplot insertion-sort.plot
```

This file is **plain text**: you can **look at it** with emacs. Inside you will find out **how to print** the graph (on a printer).

6.4 Programming insertion and selection sort

There are templates `Insertion.java`, `Selection.java` and `Bubble.java` in your coursework directories.

These do not assume that you have already written `Shift.java` *etc.* and got them working — they use methods out of the TESTER. (If you compare the “object-oriented hocus-pocus” at the end of `Insertion.java` you can see how it chooses whether to use your code or mine.)

Selection sort requires a method called `where_is_min`. You can write your own using this template.

`Bubble.java` shows all that you need to connect your code for a sorting algorithm to the TESTER: the class line has to be

```
public class MySorter implements Sorting
```

where Sorting.java is the following **interface**, *i.e.* a list of the methods that you have to supply:

```
public interface Sorting {
   int[] sort (int[] array);
   void choice (int c);
   void cutoff (int n); // used for hybrid mergesort and quicksort
}
```

(**cutoff** will be used for mergesort and quicksort; the **choice** feature of the TESTER will be used in the binary search exercise.)

6.5 Other things to think about

(a) Given a reverse-sorted array (*i.e.* in descending order), for *each* element that insertion sort considers, how many times round the loop in the `shift` (and `linear_search`) methods do you go? How many times do you go round this loop altogether, as insertion sort passes over the entire array, as a *function* of $n \equiv$ `array.length`? (This is the *worst* case.)

(b) Suppose instead you have a random array. Where, on average, would you expect each new element to be inserted in the already-sorted part? Repeat the previous calculation in this, the *average*, case.

(c) How much work does insertion sort do in the *best* case, of already-sorted data?

Warning: it is an *accident* that the average case in this situation is mid-way between the best and worst!

(d) When the data are mostly already sorted in ascending order, but with some *number* of misplaced values, how does the execution time depend on this number?

(e) When there is instead some *percentage* of misplaced values, how does the execution time depend on the percentage?

(f) In implementing the `search`, why should we look for the *last* existing occurrence (if any) of the new value?

(g) For the search part of the algorithm, you could have used *binary* search (instead of linear search). Which was quicker, and by how much? Does it change the overall time-complexity of insertion sort from being quadratic? Even though binary search is inherently quicker than linear search, what still costs more time now?

(h) *For the best programmers to think about but not try to program*: How could this cost be eliminated? If you did eliminate this cost, would it still be possible to use binary search?

6.6 Further thoughts on bubble sort

(a) What happens to the *greatest* value on the first pass of bubble sort?

(b) What about the second greatest value? For definiteness, consider an array of length 18 in which the greatest value occurs at position 12 and the second greatest at position 6.

(c) Why does it take at most n passes to sort the array?

Try to formulate a proof, using a loop invariant diagram.

(d) Describe an array that does take all n passes.

(e) Compare the way in which bubble sort works with selection sort (reversing the order). Make precise the statement that selection sort does the same thing, but with fewer assignments.

(f) You have seen that bubble sort has a directional bias to it, which could be removed by passing alternately up and down the array.[2] Implement this and find out how it compares with insertion sort on almost sorted data.

(g) Can you devise almost sorted arrays that force the two-way bubble sort to take $O(n^2)$ steps?

6.7 Using a nested loop instead

We implemented insertion sort using *separate* search and shift methods in order to understand how these *components* fits together. If you look up insertion sort in any of the algorithms textbooks (*do not do this until you have done the exercises in the way that I have told you*), you will find instead that the code is written with nested loops.

Copy all three methods (`where_to_insert()`, `shift()` and `sort()`) into `Insertion2.java` and delete the `searcher.` and `shifter.` Object-Oriented Hocus-Pocus. Do the timings again. As you see, it's a lot faster without the OO stuff.

[2] There are pre-classical Greek texts that are written in this fashion — alternately from left to right and back again, as you would lead an ox ploughing a field (hence the term *boustrophedon*) — which demonstrate the transition in the use of the alphabet from its origins in the Semitic languages.

Copy `Insertion2.java` to a third file, `Insertion3.java`. Strip the `where_to_insert()` and `shift()` methods down to the code that is actually used (discarding the comments). Do the timings again.

Non *in-line* them — *i.e.* replace the method-calls with the code itself, giving something like this:

```
while (  ) {
   while (   ) {
   }
   while (   ) {
   }
}
```

Do the timings again.

What do you notice about the pointer-values through which the two inner loops run? Combine the two inner loops into one. Try to make the text and the execution of the code as short as possible. Do the timings again.

Now (and *only* now) look up the code for Insertion Sort in a book.

Do you understand how the textbook works?

Would you have understood the textbook code if you had been given it straight away, or would you have just treated it as a piece of hocus-pocus like the object-oriented stuff above?

Did any of the steps in trimming this code down make any difference to the *form* of the dependency of the time on the size of the array and on how far it was from being sorted? What changed?

Of these various steps, which one made the *most* difference to the time?

6.8 Permutation sort

The idea is to find out, for the value originally in `array[i]`, *exactly* what its final position is in the sorted array, *without moving anything*.

In *Discrete Structures* you will learn the formal input–output definition of a *function*. Sorting must induce a permutation, which is a *bijective* function. Permutation sort determines this function explicitly.

(a) Assume at first that there are no repetitions in the array. If the value `array[i]` ends up at `array[j]`, the number `j` *counts* something — what? Write a subroutine to do this.

(b) Now for the case with repetitions. (Consider first the situation where all of the values are equal.) Suppose that the various occurrences of each particular value have to remain in the same order in the sorted array ("if you can't decide on a new order, use the old one"). What does `j` count now? In order for the function to be a permutation, different values of `i` have to give different values of `j`.

6.9 Shell sort

(Write about it.)

6.10 The time-complexity graphs

(Include them!)

Chapter 7

Loop invariant for insertion sort

This is the proof of correctness for insertion sort, using a loop invariant diagram. It is the model answer for a test that was set in week 9 in 1998–9.

You should write accounts similar to this for the other algorithms in the course that process arrays from left to right:

- array shift (already done in the model answer to the first test),
- linear search,
- binary search,
- selection sort,
- insertion sort (already done on this sheet),
- merge (the loop, not recursive merge-sort),
- split (as used in recursive quick-sort).

The pattern for merge-*sort* and quick-sort is different, as they involve recursion instead of loops. See the end of this sheet for a brief discussion. You will not be expected to prove the correctness of the recursive algorithms in the exam.

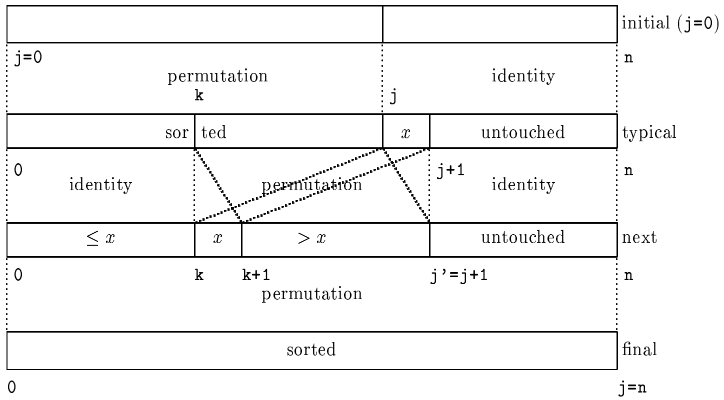
You are given an array

```
int[] A = new int[n];
```

that has somehow been filled with `int`s, and are required to use the **insertion sort** algorithm to sort it.

(a) Draw a *loop invariant* diagram to show all of the pointers (indices) into the array that you would use to code the algorithm. It should indicate typical positions of these pointers during the execution of the loop, and what is known about the values in the segments into which the pointers divide the array.

There should be *four* parts to the diagram: initial, typical, next and final, and the relationships between these should be clearly indicated.



(b) Write the JAVA code to initialise the pointers.

`int j=0;` (or 1)

(c) How do you know that the loop invariant (a) is justified by the initialisation (b)?

The "typical" version of the loop invariant in the case `j=0` says that the "sorted" part is empty (or has exactly one element), so is trivially sorted, whilst the "untouched" part is the whole (or rest) of the array, *i.e.* that the array is in its original form.

(d) What is the condition that goes inside the `while()` statement to decide whether to execute the body of the loop (again) or to exit from it?

```
while (j < n)
```

(e) When the insertion sort algorithm was presented to you in lectures, two subroutines (methods) were used inside the main loop of the algorithm. Write the JAVA code that goes inside the body of the loop, *making use of these two subroutines.*

```
int x = A[j]; // the next "unsorted" entry
int k = search (A, j, x);
shift (A, k, k+1, j-k);
A[k] = x; // re-insert the new entry
j++;
```

The pieces of code `search` and `shift` that are called within the body of the loop above are properly called *subroutines.* The term *method* refers to the way in which encapsulated *objects* are accessed from outside, in the object-oriented programming paradigm, which is not the subject of this course.

(f) In order to prove that insertion sort is *correct*, what is the *specification* of the first of these subroutines? That is, what condition, expressed in logical notation, must the inputs, output and possible side-effects satisfy? Be careful to allow for the boundary cases. You are not asked to say anything about the implementation of the subroutine (or even name the algorithm that it might use), or how long it takes.

`k = search (A, j, x)` finds `x` in `A[]` before `j`:
$(k = 0 \vee A[k-1] \le x) \wedge (0 \le k \le j) \wedge (x < A[k] \vee k = j)$

(g) What is the specification of the second subroutine? Describe it in English, not logical notation.

`shift (A, k, k+1, j-k)` shifts the segment of length `j-k` starting at `k` to start at `k+1`, leaving the rest of the array ($< k$ and $\ge j$) as it was.

(h) How do you know that the loop invariant (a) is valid *after* the code in part (e) has been executed? Explain the relevance to this of the *assumption* that the loop invariant was valid beforehand, the specifications of the two subroutines in (f) and (g) and the actual code in part (e).

The shift and re-insertion effect a permutation on the array; in fact they perform a $(j-k+1)$-cycle on the affected entries and the identity on the rest. The loop invariant, as assumed at the top of the loop, says that the segment to the left of j was sorted. When j has been incremented, it delimits another such segment, containing an extra entry with value x inserted before position k; this is sorted because the specification of the search subroutine says that the appropriate inequalities hold.

If you chose instead to use the in-line version of the code in part (p), you would have had to give a second loop invariant and proof for the inner loop. *This would involve reproducing the whole of the answer to the first test!*

(i) How do you know that the loop terminates?

The positive integer $n - j$ is reduced in each iteration.

(j) What can you deduce from the loop condition (d) and the loop invariant (a) when the loop has terminated?

When the loop *condition* fails, $j = n$. The loop *invariant* in this case says that the "sorted" part of the array is the whole of it, and the "untouched" part none. It also says that the array as it currently stands is a permutation of the original state. In other words, the array has been sorted as required.

(k) What is the total number of operations that are performed by the second subroutine throughout the execution of the insertion sort algorithm, in the *worst* case? What "operations" are meant here? Express your answer as a function of n, the size of the array.

In the worst case, where $k = 0$, `shift` performs j comparisons and assignments on the jth call, totalling $\frac{1}{2}n(n-1)$.

(l) What kind of data in the array give rise to the worst-case behaviour?

This happens when $\forall j. \forall i < j. A[j] < A[i]$, *i.e.* when the array is reverse-sorted (without repetitions).

(m) How many operations are performed in the *average* case? Make clear the relationship between this answer and that to part (k).

In the average case, $A[j]$ is approximately the median of the sorted segment, so `shift` has to do about $j/2$ assignments on the jth call, totalling approximately $n^2/4$.

(n) How many operations are performed in the *best* case, and how must the subroutines be implemented in order to achieve this? You need only describe the essential feature in words: there is no need to write the JAVA code. What kind of data in the array give rise to the best-case behaviour?

In the best case, $k = j$, so the loop in `shift` is not executed. This happens when $\forall j. \forall i \langle j. A[j] \ge A[i]$, *i.e.* the array was sorted (possibly with repetitions). To take advantage of this situation, `search` must return the *rightmost* occurrence of x, otherwise any repetitions of x will get shifted unnecessarily. Linear search from the right will do this, but binary search will take $n \log n$ steps altogether.

(o) In "$O()$" notation, what is the complexity of insertion sort in this best case?

$O(n)$, because there are other steps in the main loop (and in the initialisation of `search` and `shift`) that are performed once for each iteration of the main loop, *i.e.* n times.

(p) It is, of course, more efficient to do the job of the two subroutines "in-line" *i.e.* by writing a nested loop instead of calling them as separate methods. Write the whole of the insertion sort program in this way.

```
int j=2;
while (j<n) {
   int x = A[j];
   int k = j;
   while (k > 0 && A[k-1] > x) {
      A[k] = A[k-1];
      k--;
   }
   A[k] = x;
   j++;
}
```

Chapter 8

Test: loop invariant for selection sort

You have 50 minutes.
The discussion is entirely about the *outer* loop of the algorithm, *i.e.* the `sort` method itself..

```
public int[] sort (int[] A) {
   int n=array.length;
   int j=0;
   while (j<n) {
      int k = where_is_min (A, j);
      int x = A[j];
      A[j]  = A[k];
      A[k]  = x;
      j++;
   }
   return array;
}
```

In order to eliminate any discussion of the inner loop, this makes use of the following private method.

```
private int where_is_min (int A[], int after) {
   int size = array.length;
   int position = after;
   int result = after;
   int value = Integer.MAX_VALUE;

   while (position < size) {
      if (A[position] < value) {
         result = position;
         value = A[position];
      }
      position++;
   }
   return result;
}
```

(a) State, in one English sentence, the **idea** of selection sort. [1 mark]

(b) Draw a *loop invariant* diagram to show all of the pointers (indices) into the array that you would use to code (the outer loop of) the selection sort algorithm. It should indicate typical positions of these pointers during the execution of the loop, and what is known about the values in the segments into which the pointers divide the array.

There should be *four* parts to the diagram: initial, typical, next and final, and the relationships between these should be clearly indicated.

Hint: there are *three* logical conditions in the loop invariant. [**5 marks**]

(c) How do you know that *each of the three conditions* in the loop invariant (b) is justified when the loop is entered for the first time, *i.e.* just after the initialisation of the pointers? [1 mark]

(d) What logical property, relating `A[j]` to the rest of the array `A`, is true after the following code has been executed, assuming that `(j < A.length)`?

```
int k = where_is_min (A, j);
int x = A[j];
A[j]  = A[k];
A[k]  = x;
```

Write your answer in logical notation, paying close attention to whether strict ($<$) or non-strict ($\leq$) inequalities are required. [1 mark]

(e) Assuming that the loop invariant in (b) was valid at the beginning of the execution of a particular iteration of the body of the loop, prove that (*each of the three conditions* in) the loop invariant is valid again after the body of the loop has been executed once.

You must state clearly where the following things are used in the proof:

- each of the three conditions that make up the loop invariant that was assumed to be true at the top of the loop,
- your answers to part (d), and
- the actual code overleaf.

[**5 marks**]

(f) How do you know *that* the loop terminates, rather than continuing forever? [1 mark]

(g) What logical property is true when the loop has terminated, that you know simply because the loop has indeed terminated? [1 mark]

(h) Putting (g) together with the loop invariant, explain what can you deduce when the loop has terminated, and why. [1 mark]

Chapter 9

Binary search

This chapter is a rather crude amalgamation of several kinds of teaching materials, and needs to be re-written before being given to students again. I introduce binary search in the lectures by handing out the eight versions of the code. I divide the class into groups, and ask each group to do a "dry run" of one of the algorithms on certain input data. In the labs, they are asked to verify correctness, in particular finding out whether the leftmost or rightmost occurrence is found, and making various alterations to the code. These things are then covered in a mini-exam. Formal proof of correctness is covered in the lectures. This has also been an exam question, for which the model answer is incorporated here.

When an array of data is sorted, there is a *much* faster way of finding a value than by *linear* search. Think of the way in which you use a dictionary. In reality, we employ our knowledge of the frequency of initial letters — the *London Residential Telephone Directory* used to be published in volumes for A–D, E–K, L–Q and S–Z — but we shall not use such knowledge in this algorithm.

The idea is to look half way through, then, depending on whether that was too high or too low, *either* $\frac{1}{4}$ or $\frac{3}{4}$ of the way through. At each iteration we have `top` and `bot` pointers, and look at the value at the position `mid`-way between them. You should try programming this for yourself before looking at the code below, which gives the best possible form of this algorithm.

In fact we shall study this algorithm from a completely different point of view. Many textbooks give you complete JAVA code, and students regard this as a *recipe*. The purpose of our (novel) approach to is gain an appreciation that *every symbol in the program means something*. We shall see this by *changing them*, and first by looking at four equally correct versions of the same algorithm. You will learn that ± 1 in a program is "a matter of life and death" — that the behaviour is different in crucial ways that you cannot see from a superficial look the code.

When you get a new job as a programmer, you won't be writing virgin code, as in most official and unofficial student exercises, but modifying someone else's code. Usually, you will find that other people's code is unspeakably awful, but you have to live with it. It is difficult to simulate this real-life situation in a teaching environment, but this exercise is a tiny example.

9.1 Four versions of binary search

"There are nine and sixty ways of constructing tribal lays,
And every single one of them is right!"
In the Neolithic Age, Rudyard Kipling

In the algorithms textbooks, there are *three* different ways of writing the code for binary search. Although the version that we call CX below is the one that a naïve programmer would write, version AL is the best, and BL is a less tidy version of the same thing.

The `+1`s and `-1`s in the assignments to `top` and `bot` are what distinguish the versions, and you will be asked to change everything else about these versions *except* for these things. The fourth version completes the symmetry.

Each version of the algorithm solves the simple problem of locating *some* occurrence of any value that is actually present. It also reports that a value is absent by returning a pointer to a position that the caller can check doesn't contain the required value.

However, version AL is more precise than this: it behaves in a specific, predictable and useful way when there are *multiple* or *no* occurrences of the value, as we shall see.

`Binary.java` in your own coursework directory contains a copy of the following JAVA code.

```
int AL (int[] array, int seek) {
    int bot = -1;
    int top = size;
    while (top - bot > 1) {
        int mid = (top + bot)/2;
        if     (array[mid] <  seek)   bot = mid;
        else /* array[mid] >= seek */ top = mid;
    }
    return top;
}

int BL (int[] array, int seek) {
    int bot = 0;
    int top = size;
    while (top > bot) {
        int mid = (top + bot - 1)/2;
        if     (array[mid] <  seek)   bot = mid + 1;
        else /* array[mid] >= seek */ top = mid;
    }
    return top;
}

int CL (int[] array, int seek) {
    int bot = 0;
    int top = size - 1;
    while (top >= bot) {
        int mid = (top + bot)/2;
        if      (array[mid] <  seek)    bot = mid + 1;
        else /* (array[mid] >= seek) */ top = mid - 1;
    }
    return bot;
}

int DL (int[] array, int seek) {
    int bot = -1;
    int top = size - 1;
    while (top > bot) {
        int mid = (top + bot + 1)/2;
        if     (array[mid] <  seek)   bot = mid;
        else /* array[mid] >= seek */ top = mid - 1;
    }
    return top+1;
}
```

Note: `/*...*/` is another way of delimiting a comment. Whereas `//` begins a comment that continues to the next newline, a comment that begins with `/*` continues until the next (not necessarily matching) `*/`. As a matter of style, when using this way of delimiting comments that span many lines of the source file, it is usual to start each intervening line with another `*` and align the `*`s vertically.

9.2 Dry runs

The following "dry runs" show what the search executes and returns when it is asked to search for 6 or 13, respectively in the array

1, 3, 4, 6, 6, 6, 8, 8, 8, 10, 12, 14, 14, 14, 15

seek=6	bot	mid	top	seek=13	bot	mid	top
bot=-1	−1			bot=-1	−1		
top=15			15	top=15			15
while (15 - -1 > 1)				while (15 - -1 > 1)			
mid=(15+-1)/2		7		mid=(15+-1)/2		7	
array[7]=8>=6				array[7]=8<13			
top=7			7	bot=7	7		
while (7 - -1 > 1)				while (15 - 7 > 1)			
mid=(7+-1)/2		3		mid=(15 + 7)/2		11	
array[3]=6>=6				array[11]=14>=13			
top=3			3	top=11			11
while (3 - -1 > 1)				while (11 - 7 > 1)			
mid=(3+-1)/2		1		mid=(11 + 7)/2		9	
array[1]=3<6				array[9]=10<13			
bot=1	1			bot=9	9		
while (3 - 1 > 1)				while (11 - 9 > 1)			
mid=(3 + 1)/2		2		mid=(11 + 9)/2		10	
array[2]=4<6				array[10]=12<13			
bot=2	2			bot=10	10		
! (2 - 1 > 1)				! (11 - 10 > 1)			
return top=3				return top=11			

9.3 Experiments

Use `check` to verify that AL–DL are correct by doing things like

```
check binary-al
check Binary.java choice=3
```

of which the first version uses the "model answer" built into the TESTER and the second uses the template file `Binary.java` in your filespace, in which AL is selected by setting `choice=1`, BL=2, CL=3 and DL=4. Paste the correctness reports of the TESTER into your copy of `Binary.java`.

Verify that each version is able to find *some* occurrence of any value that is actually present, using tests like

```
run Binary.java choice=1 array=1,3,4,6,6,6,8,8,8,10,12,14,14,14,15 seek=6
```

Be careful to check the extreme values, 1 and 15.

When a search is done for a value that occurs more than once, can you characterise *which* occurrence each of the eight versions finds?

Now consider a search for a value that does not occur in the array. What position does each method return? Don't forget the extreme cases, *i.e.* values that are smaller (or larger) than all of the values that do occur in the array.

What happens if the array is of zero length?


```
run Binary.java choice=1 size=0
```

Do you get an `ArrayIndexOutOfBoundsException`? What position is returned? Is this consistent with the specification of the behaviour that you have just formulated, where an absent value was sought in a non-empty array?

Change the initialisation `bot=-1;` in AL to `0` or `-2`, and check the behaviour for small values. What goes wrong?

Similarly, change the initialisation `top=size` to `size-1`.

Explain how you would change the code to search the segment `array[3]` to `array[13]` inclusive, so that it *never* accesses `array[2]` or `array[14]`.

Change the return value from `bot` to `top` or *vice versa* in all four versions.

There is a slight difference in the amount of time that they take. Find out the coefficient of $n\log(n)$ and paste it in as a comment in the source file. Which is faster?

Does the timing graph depend on the `range` of random increments in the array values? (You will need to make `range` very small or `density` large to see any difference.)

9.4 Time complexity

On paper, find out how many times you go round the loop if the array is of size 3, 7, 15, 31, 63, ..., or more generally of of size $2^k - 1$.

How many bits long is the number n, written in binary, if $2^{k-1} \leq n \leq 2^k - 1$?

You may have learned about **logarithms** to base 10 or base $e \approx 2.718$ in Calculus at school. In Computer Science we *always* use logarithms to base 2, and usually round them up to the nearest whole number.

Now use the TESTER's `check` command to find out how long binary search takes on arrays of size 3, 7, 15, 31, 63, ... and so on up to the biggest that the JAVA run-time system will allow.

Shut down all unnecessary programs, especially Netscape.

It will be possible to run the program for arrays that are larger than the available RAM, because your workstation will start using *virtual memory* — actually on disk. This will completely ruin your timings. You can tell that this is happening by listening to the disk and watching the lights on the box.

The time taken to execute the code depends *logarithmically* on the length, and is therefore *substantially* faster than linear search.

Look inside the file Binary.plot to find out how to **plot a graph of time against the logarithm of the size**, so 3, 7, 15, 31, 63, ... are equally spaced.

If you are *very* careful with the experiments, you can find out the size of the **primary memory cache within your CPU**, and how long it takes to fetch data from the secondary cache within the CPU.

Include the graphs here.

9.5 The pre-condition

These algorithms **only** work when the array satisfies a certain **precondition**.

What is this condition? Write it in logical notation.

What does the program do if it is not satisfied?

Write the JAVA code to test whether an array satisfies this property.

How long would it take to execute the check, as a function of the size of the array?

When you use a telephone directory, do *you* (as a human) make such a check? Describe carefully would be the effect on the usefulness of the directory if a few of the words failed to satisfy this condition.

This verification takes time that depends *linearly* on the length of the array (and *sorting* it, *i.e.* *enforcing* rather than *verifying* it, would take time at least $O(n\log n)$), which is *much* greater than the time that the search itself takes. To verify or enforce the precondition would therefore entirely defeat the point of using binary search.

In the `shift` exercise, the TESTER insisted that you should check the precondition before proceeding with the shift, and throw an Exception if it fails.

Discuss the consequences of trying to proceed with shift, or with binary search, in a situation whether the precondition is not satisfied. How, on the other hand, do the costs of checking the precondition compare?

9.6 Mirror-image algorithms

Within your copy of the file `Binary.java`, make a *copy* of the `case`s 1–4 (AL–DL), calling them `case 11` to `case 14`, and change `<` to `<=`. So, for example, `case 2` (BL) becomes

```
case 12: {
    int bot = 0;
    int top = size;
    while (top > bot) {
        int mid = (top + bot - 1)/2;
        if     (array[mid] <= seek)   bot = mid + 1;
        else /* array[mid] >  seek */ top = mid;
    }
    return top;
}
```

Does this still return the position of the *leftmost* occurrence when `seek` occurs several times?

How does the position returned by the algorithm now relate to the position where the value should be inserted if it is absent?

You may need to change the `return` values in cases A, C and D to get this right.

What are appropriate names (instead of AL–DL) for these four versions of the algorithm?

What behaviour (in particular for missing values) would be appropriate if this algorithm were to be used as the search subroutine for insertion sort?

9.7 Early exit

If you tried to program binary search for yourself without looking at (the four versions of) the code at the beginning of this chapter, you probably wrote something like

```
int CX (int[] array, int seek) {
    int bot = 0;
    int top = size - 1;
    while (top >= bot) {
        int mid = (top + bot)/2;
        if      (array[mid] == seek)    return mid;
        if      (array[mid] <  seek)    bot = mid + 1;
        else /* (array[mid] >= seek) */ top = mid - 1;
    }
    return bot;
}
```

which certainly appears in many of the textbooks. We shall say that this version of the algorithm **makes an early exit**.

Include CX in your copy of `Binary.java`, calling it `case 23`. The other three versions can be modified in a similar way, and we call them AX, BX and DX or `case`s 21, 22 and 24.

Now, when you use CX to search for a value that occurs more than once, does it return the leftmost or rightmost occurrence?

On the other hand, if the value is not present, does CX still return the position *before* which to insert it?

Which line do you have to change to obtain the position *after* which to insert a missing value?

As you see, this "early exit" test means that the algorithm satisfies a weaker specification. But the point of adding this test was to make it run faster. On the other hand, the three-way test in CX takes longer than the corresponding two-way test in AL.

Does CX actually perform the search any quicker?

Let's do some mathematical analysis of this.

Assuming $n = 2^r$, that no value is repeated in the array, that we search for each value with equal likelihood, and never search for absent values, in how many cases is the search successful after exactly k iterations? What is the average number of comparisons made in the search? Hint:

$$\sum_{k=1}^{\infty} \frac{k}{2^k} = \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \frac{5}{2^5} + \frac{6}{2^6} + \cdots = 2$$

What saving does this represent, on average, compared to the r comparisons that are made in every case with no early escape?

In our code (comparing integers), a three-way test is actually twice as expensive as a two-way test, so is there any advantage?

Now consider that, in practice, unsuccessful searches are performed, as well as successful ones. How does this affect the conclusion?

What difference does this make to the time?

If, however, the cost of the test were very high compared to that of just going round the loop, or the means of making the test provided a three-way result at the same price as a two-way one, this analysis would be different.

Suppose that we were instead searching a `String[]` array, using the JAVA library method

```
int comparison = string1.compareTo (string2);
```

which returns -1, 0 or $+1$ according as `string1` comes before, at the same place as or after `string2` in the dictionary. This method, which we would only call once for either CX or AL, takes *considerably* longer than the rest of the code, so avoiding one or two calls to it could quite easily outweigh the cost of the three-way test for whether `comparison==0`, `>0` or `<0`.

The `COMP` instruction in SF4 machine code (from the CS1 course) also makes a 3-way test, in the sense that it sets `N` and `Z` flags inside the CPU in such a way as to distinguish the three cases. As for `compareTo`, we would only use it once in either CX or AL. Donald Knuth[1] claims that CX is faster unless the array is very large, though I don't believe this.

Note: this is still a comparison between the two algorithms CX and AL, not between JAVA and machine code.

9.8 Changing the loop test and the calculation of `mid`

99% of `while` loops that you will ever meet can be proved to terminate by observing that the loop variable is a whole number that increases or decreases by 1 or more on each iteration, between pre-determined bounds. As we shall now see, termination of binary search is not quite this obvious, and, on the other hand, some curious things happen if you exit the loop too early.

Change the **loop condition** (`while (top > bot)` *etc.*).

What happens if you make the condition stronger? For example, change AL to

[1] *The Art of Computer Programming*, volume 3, *Sorting and Searching* (Addison–Wesley, 1973, second edition, 1997), closing paragraphs of section 6.2.1 (page 422), and exercise 23 (page 425).

```
while (top - bot > 2)
```

What happens if you make it weaker?

```
while (top > bot)
```

In this case it more readily continues going round the loop.

Beware that the effect may depend on the input data: do several experiments before you assert that something *always* happens.

The one remaining thing to change is the calculation

```
mid = (top + bot)/2;
```

A easy mistake to make (you will do this often) is to miss out the brackets. Try this, and find out whether this makes the algorithm give *wrong results*.

On the other hand does it necessarily produce any result at all?

A more reasonable version of this "mid-point" calculation is `(top+bot+1)/2` or `(top+bot-1)/2`. Do the same experiments.

Keeping your alternative calculation for `mid` for which the algorithm fails to terminate, change the loop condition to fix the problem. What is wrong now? Can you fix it with extra code after the end of the loop? Make sure that this still works with arrays of length zero.

9.9 Proving termination

The basic idea behind a proof of termination is that the value of `top-bot` is strictly reduced on each iteration.

The modifications that you made to the programs, and the analysis of attractor sets both show that this argument needs to be considered more carefully.

In version AL, the loop condition says that

```
top >= bot + 2
```

from which it is easy to deduce that the calculation of `mid` makes

```
bot < mid < top
```

but these inequalities are different for the other versions.

Using these little bits of arithmetic, we can check that `top-bot` really is reduced on each iteration.

9.10 How the versions differ

The difference amongst the four L versions of the algorithm is simply a matter of whether the pointers `bot` and `top` are on the left or the right of the boundary.

		$<$ `seek`		seek		$\geq$ `seek`	
AL:	0	bot		mid		top	n
BL:	0		bot	mid		top	n
CL:	0		bot	mid	top		n
DL:	0	bot		mid	top		n

The R versions are the same, except that the left-hand segment is $\leq$ `seek` and the right-hand one $>$ `seek`.

9.11 Logical analysis of version AL

Note: be careful to distinguish between the old and new values of variables to which assignments are made (`top` and `bot`).

a. The assignment `bot=mid` is executed after the test `(top - bot > 1)` has succeeded and the initialisation `int mid = (top + bot)/2` has been made. For any integers b and t, we claim that
$$t - b > 1 \Rightarrow (t+b)/2 > b.$$
But, as these are integers, $t \geq b+2$, so $(t+b)/2 \geq (b+2+b)/2 = b+1$ as required. Hence the variable `bot` is increased by at least 1.

   Beware that, since we are dealing with *integer* division, it is wrong to "multiply through-out by 2" to eliminate the division. Mathematical notation — which, after all, you learned at school — is *essential* for such reasoning, as English words are imprecise. Maths is also much quicker to write!

   Note that `mid` may often be closer to `bot` than to `top`, for example if `bot=8`, `top=11`, so `mid==9`.

b. This calculation would no longer be valid if the loop condition were changed to `while (top - bot > 0)`. Eventually we (may) reach the state `top==bot+1` *and continue executing the loop*, in which case the value of `mid` is $(t+b)/2 = (b+1+b)/2 = b$, leaving `bot` unaltered.

c. Suppose, for example, `seek=7`, `bot==3`, `top==4` and `array[3]=6` in Section 9.8(c). Then we re-assign the same value, `bot=3`, and go round the loop again in exactly the same state as before, and again, and again, ... On the other hand, with `seek=5`, we would instead assign `top=3` and leave the loop. So this version *sometimes* fails to terminate.

d. Similarly, the assignment `bot=mid` makes the new value of `bot` at most `top-1`, because, as in (a), $(t+b)/2 \leq (t+t-2)/2 = t-1$.

e. After the assignment `bot=mid` has been made, `array[bot] < seek`.

f. Similarly, after the assignment `top=mid` has been done, `seek <= array[top]`.

g. The two logical statements in (e) and (f) are not, as they stand, valid just after the initialisations `bot=-1` and `top=array.length`, since these are "out of bounds". We may, however, restore meaning to the statement either by adopting the convention that $\texttt{array}[-1] = -\infty$ and $\texttt{array}[\texttt{array.length}] = +\infty$, or by using the more complicated statement
$$(\texttt{bot} = -1 \vee \texttt{array}[\texttt{bot}] < \texttt{seek}) \wedge (\texttt{top} = \texttt{array.length} \vee \texttt{seek} \leq \texttt{array}[\texttt{top}])$$
(though even makes illegitimate use of `array[-1]` and so depends on a convention about $\vee$). Of course, this convention is *only* appropriate to the discussion of an array that's sorted in ascending order!

h. Suppose that the logical statement in (e) happened to be valid just inside the beginning of the body of the loop, but that the `else` branch (including assignment `top=mid`) is executed. Then (e) is still valid, because `bot`, `seek` and the contents of the `array` have not been altered. Such statements about what has *not* happened are just as important as saying what has happened, *cf.* that the rest of the array has stayed the same when part of it has been shifted.

i. Therefore, on exit from the loop we may conclude either that `array[bot] < seek <= array[top]`, or the more complicated statement in part (g). This is because this statement

   - has been verified on entry to the loop, and

- is preserved by each iteration,
- so is therefore still true on exit —
- assuming that the loop does eventually exit!

j. We exit the loop exactly when the loop condition fails, *i.e.* `!(top - bot > 1)`.

k. Part (d) said that `(top - bot >= 1)` after the loop body has been executed, irrespectively of whether it was true at the top of the loop, though in fact it is also true after the initialisation, so this statement is part of the loop invariant. Putting this together with part (j), we have `(top == bot+1)` on exit from the loop.

l. Substituting (k) into the loop invariant, we have `array[top-1] < seek <= array[top]` on exit from the loop.

m. We have already said that the array must be sorted as a precondition for binary search to work. Then,

$$\forall i.\ 0 \leq i < \mathtt{top} - 1 \Rightarrow \mathtt{array}[i] \leq \mathtt{array}[\mathtt{top} - 1] < \mathtt{seek}.$$

so it is not possible for `array[i]==seek` where `i<=top-2`.

n. Hence, if the value `seek` occurs in the array, `top` is the position of its leftmost occurrence.

o. Suppose that `array[top] != seek`. Then, by the same reasoning as in (m), $\forall i.\ \mathtt{top} < i \Rightarrow \mathtt{seek} < \mathtt{array}[i]$. Hence the number `seek` does not occur anywhere in the array.

p. More precisely, the value `seek` lies strictly between those in the array in positions `top-1` and `top`, so `top` is the position before which to insert.

q. If we replace the loop condition by `while (top - bot > 2)`, the best that we can say is that `top==bot+1` or `top=bot+2`, so `array[top-2] < seek <= array[top]`, but we know nothing about `array[top-1]`.

9.12 The intermediate value theorem

Binary search provides a constructive proof of a standard theorem in Analysis. ("Analysis" is what mathematicians call Calculus done rigorously.) Clearly this would not be part of a test in a computer science course.

"If you're on one side of a river in the morning and on the other side in the afternoon, you must have eaten your lunch on a boat!"

Let $f : [0, 1] \to \mathbb{R}$ be a real-valued continuous function on the real unit interval. Suppose that $f(0) \leq s \leq f(1)$. Then $\exists x. 0 \leq x \leq 1 \wedge f(x) = s$.

The proof usually found in the textbooks considers the point

$$x = \sup\{y \in [0, 1] \mid f(y) \leq s\}$$

and uses continuity to prove that $f(x) = s$, without actually offering any way to *calculate* x.

Here is a constructive way to solve the problem (so long as the function f is such that you can *decide*, for each $y \in [0, 1]$, whether $f(y) < 0$ or $f(y) \geq 0$). You can apply this method to solving equations like $n \log_2(n) = 10^6$ if you wish.

We define three sequences b_n, t_n and m_n of real numbers by a recursive procedure.

First, put $b_0 = 0$ and $t_0 = 1$, as in the first two lines of the binary search algorithm.

Then, for each $n \in \mathbb{R}$, let $m_n = (b_n + t_n)/2$, and

- if $f(m_n) < s$, put $b_{n+1} = m_n$ and $t_{n+1} = t_n$;
- if $f(m_n) \geq s$, put $b_{n+1} = b_n$ and $t_{n+1} = m_n$.

These assignments correspond to those in the loop in binary search, except that this iteration goes on forever.

Hence b_n is an increasing sequence, and t_n a decreasing sequence, and moreover $t_n - b_n \leq 2^{-n}$, so both sequences converge to a common limit. Call it x. This satisfies

$$\forall n.\ x - 2^{-n} \leq b_n \leq x \leq t_n \leq x + 2^{-n}.$$

We haven't used continuity of f yet, just as in binary search we didn't rely on the array being sorted until the very end (Section 9.11(m) and (o)).

By definition, we say that f is continuous at x if

$$\forall \epsilon > 0.\ \exists \delta > 0.\ \forall y.\ |y - x| < \delta \Rightarrow |f(y) - f(x)| < \epsilon.$$

Suppose $f(x) > s$, so $\epsilon = f(x) - s > 0$, and let $\delta > 2^{-n} > 0$ be as in the definition of continuity. But then $y = b_n$ satisfies $|y - x| < \delta$, so $|f(y) - f(x)| < \epsilon$ by continuity. However, by construction $f(y) = f(b_n) < s < f(x)$, so $f(x) - f(y) > \epsilon$, which is a contradiction.

The case $f(x) < s$ also leads to contradiction by considering t_n, so we are left with $f(x) = s$ as required.

Chapter 10

Test: sorting algorithms and binary search

You have 90 minutes.

Calculators are NOT allowed.

There are rather a lot of questions on this test paper, and you are not expected to be able to do all of them. Just do as many as you can.

The later parts ((f), (g), (h), ...) of each section (1, 2, 3, ...) are more difficult than the earlier ones. If you get stuck, move on to the next section.

Full credit (15/15) will be awarded for considerably less than the entire paper, depending on the distribution of marks from the whole class.

Warning: The "dry run" questions (where you are asked to simulate the algorithm on paper) will take you quite a long time, but not gain you marks in proportion to this time. You should read quickly through the other questions before starting these, and skip them if you feel reasonably confident of scoring marks elsewhere.

10.1 Sorting algorithms in general

(a) In order to justify calling a method a "sorting" algorithm, its result or output must, of course, be in (non-strictly) ascending order. What *other* property must the method satisfy? Write your answer in English, not formal logical, mathematical or programming language; there is one significant word.

(b) Briefly describe in English a method whose output is in ascending order, but which fails this condition and is therefore not a sorting algorithm. Make it as simple as possible.

(c) List the sorting algorithms that have been discussed during the course, in order of the time that they take to sort large arrays, **fastest first**.

(d) Based on your observations during labs of the algorithms built in to the TESTER, what is the **largest** unsorted array of integers that the **fastest** of the algorithms could sort in **one minute**. Your answer should be a **power of 10**: 10, 100, 1000, ...

(e) What is the largest unsorted array that the **slowest** algorithm could sort in **one minute**?

(f) What is the largest unsorted array that the **fastest** algorithm could sort in **100 minutes**? Your answer will be judged by whether it is consistent with your answer to (d).

(g) What is the largest unsorted array that the **slowest** algorithm could sort in **100 minutes**? Again, it is consistency with (e) that matters.

(h) Most of the algorithms work **in place**, *i.e.* by rearranging the given array without using additional scratch space. Which one **does** need additional scratch space, and why?

(i) You have found on the Internet a JAVA method that begins

```
public int[] hp_srt (int [] A) {
```



```
    ...
}
```

so you try it out on some arrays of length 1000, 2000, 3000, 4000 filled with random numbers. Repeating the test many times with new random numbers, you find that it consistently returns in 1, 2, 3 and 4 milliseconds respectively. Without looking at either the code or its output, on what basis can you deduce that `hp_srt` is almost certainly *not* a sorting algorithm?

10.2 Insertion sort and bubble sort on almost sorted data

Be careful to emphasise the *distinctions* between your answers to parts (a) and (f), and also amongst (e), (f), (h) and (i).

(a) State, in one English sentence, the **idea** of **insertion sort**.

(b) Insertion sort takes about 150 microseconds to sort a random array of length **100**. How long would you expect it to take to sort into *ascending* order an array of length **100** that was reverse sorted, *i.e.* given in *descending order*?

(c) How many **seconds** would you expect it to take to sort a random array of length **10000**?

(d) Suppose that insertion sort is given the following array

1, 2, 4, 5, 6, 7, 8, 3, 9

which has one *small* value out of place amongst otherwise sorted data.

Describe the steps that insertion sort executes on this array, using the notation:

- write out the data shown above so that it takes the full width of the page, with plenty of space between the digits;
- allow one line per iteration of the main loop;
- **underline** (____) each position with which a **comparison** is made;
- write each **number** that is **assigned** to cells in the array, carefully aligning it with the numbers above to make its position in the array clear; **if no assignment is made, leave the column blank**;
- do not indicate temporary storage.

State how many comparisons and assignments are made.

(e) Using the same notation, describe what insertion sort does with

1, 2, 8, 3, 4, 5, 6, 7, 9

where there is just one *large* value that is mis-placed among the *small* ones.

(f) State, in one English sentence, the **idea** of **bubble sort**.

(g) What does bubble sort do with the following array?

1, 2, 4, 5, 6, 7, 8, 3, 9

Use the following notation:

- **underline** each pair of numbers that are are **compared**;
- rewrite the **numbers** only when they are **swapped**.

State the total number of comparisons and assignments.

(h) Using the same notation as in (g), describe what bubble sort does with the following array?

1, 2, 8, 3, 4, 5, 6, 7, 9

(i) Insertion sort is very fast when it is given an array that is already sorted in ascending order *but contains repeated values.* What design decision must be made in its implementation to reduce to the *absolute minimum* the work that it does in this case?

(j) In what way does the time taken by insertion sort to sort an almost sorted array depend on the *number* or the *percentage* of displaced values? Explain these result in the light of the answers to the questions above.

10.3 Binary search

The rest of the questions on this test paper are based on the following (correct) JAVA code for binary search. Notice that one of the lines is "commented out"; it will be discussed in Question 10.5.

```
int search (int[] array, int seek) {
   int bot = -1;
   int top = size;
   while (top - bot > 1) {
      int mid = (top + bot)/2;
      // if (array[mid] == seek) return mid;
      if (array[mid] < seek) bot = mid;
      else                   top = mid;
   }
   return top;
}
```

In this question, `array` is of length 15 and contains the values

1, 3, 4, 6, 6, 6, 8, 8, 8, 10, 12, 14, 14, 14, 15

(a) Do a "dry run" to find out what result the method returns when it is asked to search for **6**. Indicate clearly *when* assignments to the program variables take place, and what numbers are assigned to them.

(b) Does the result returned indicate the leftmost, rightmost or just some occurrence of the value `seek`, assuming that it is present?

(c) Without doing a "dry run", what number would be returned given `seek=8`?

(d) Do another "dry run" to find out what the method returns when it is asked to search for **13**.

(e) When the value `seek` is absent, should it be inserted before or after the position that the method returns?

(f) Without doing a "dry run", what number would this method return given `seek=0` or `seek=16`?

(g) List clearly and explicitly the lines of code that are executed if this method is instead given an array of length 0.

(h) What number does the method return in this case?

(i) Explain why it does *not* `throw ArrayIndexOutOfBoundsException`.

10.4 Complexity and pre-conditions

(a) How much time does it take this method take to execute, as a function of the size of the array?

(b) What special requirement (*pre-condition*) must the array satisfy for the method to be able to work?

(c) Write this in logical notation (using $\forall$, $\Rightarrow$ *etc.*.).

(d) Write a JAVA program

```
boolean ok (int[ ] array) { ...  }
```


to find out whether or not the `array` satisfies this requirement.

(e) What does binary search do if the pre-condition is not satisfied?

(f) Should the code for binary search be altered to check the pre-condition before proceeding with the search? Give your reasons.

10.5 Early exit

Now consider the version of binary search where the line

```
if (array[mid] == seek) return mid;
```

(which was "commented out" before) is now included.

(a) If the value `seek` occurs several times in the `array`, does this altered code find the leftmost, rightmost or just some occurrence?

(b) If the value `seek` does not occur at all in the `array`, should it be inserted before or after the position that the altered code returns?

(c) *Why* might this alteration to the code make it *faster*?

(d) *Why*, on the other hand, might the alteration make it *slower*?

(e) Is it, in fact, faster or slower than the original code?

(f) Suppose that the value `seek` does occur exactly once in the `array`, which is of length $n = 2^k - 1$. How many times does the **unaltered** code go round its loop?

(g) On average, how many **fewer** times does the **altered** code execute the loop? Explain your answers by drawing a binary tree.

(h) Suppose that you were instead searching a `String[]` array, using the JAVA library method

```
int comparison = string1.compareTo (string2);
```

which returns -1, 0 or $+1$ according as `string1` comes before, at the same place as or after `string2` in the dictionary. Would you expect the analogous code with or without the alteration to be faster. Why?

(i) Again, suppose that you were using the `COMP` instruction in SF4 machine code (from the CS1 course) to compare integers. Which would you expect to be faster now, and why?

10.6 Other alterations

The purpose of this question is to get you to appreciate that each symbol in a program has a meaning, and affects the program as a whole.

The alterations to the code are to be taken individually: you have to "change it back" when you move on to the next part.

If something goes wrong, state clearly whether it is *always* or *sometimes* incorrect in the way that you describe.

If it only sometimes goes wrong, try to give an example of a value of `seek` (in the array used in Question 3, or in some other array that you describe) where this incorrect behaviour would occur. Say what value the method returns, or, if it throws an `ArrayIndexOutOfBoundsException`, what position it illegally tries to access.

(a) What happens if you initialise `bot=0` instead?

(b) What happens if you initialise `bot=-2` instead?

(c) How could you alter the code to search the segment `array[3]` to `array[13]` inclusive of the array (so it is allowed access `array[3]` and `array[13]` but not `array[2]` or `array[14]`)?

(d) What happens if you change the loop condition to this?

```
while (top > bot) { ...  }
```

(e) What happens if you change the loop condition to this?

```
while (top - bot > 2) { ...  }
```

You will only be able to answer the final two parts if you have done the experiments in the lab (or can do the mathematical analysis).

(f) What happens if you change the calculation of `mid` to this?

```
int mid = (top + bot - 1) / 2;
```

(g) What happens if you make **both** of the changes in (e) and (f)?

Chapter 11

Mergesort and quicksort

11.1 Mergesort and quicksort

These are your first two **recursive** programs in JAVA(you have already written lots of recursive programs in MIRANDA). There are two approaches to teaching recursion. One is to pull your hair out with worry about how circular definitions could possibly be meaningful. The other is just to get on with it, and that's what we shall do.

Mergesort takes an array,

$$\{91\ 46\ 16\ 46\ 61\ 46\ 89\ 78\ 27\ 81\ 83\ 37\ 95\ 34\ 75\ 56\}$$

and splits it in two **without any attempt to sort it**:

$$\{91\ 46\ 16\ 46\ 61\ 46\ 89\ 78\} \qquad \{27\ 81\ 83\ 37\ 95\ 34\ 75\ 56\}$$

then sorts two parts **recursively** (*i.e.* by calling itself):

$$\{16\ 46\ 46\ 46\ 61\ 78\ 89\ 91\} \qquad \{27\ 34\ 37\ 56\ 75\ 81\ 83\ 95\}$$

and finally **merges** them **in order**:

$$\{16\ 27\ 34\ 37\ 46\ 46\ 46\ 56\ 61\ 75\ 78\ 81\ 83\ 89\ 91\ 95\}$$

So **mergesort** does the work **on the way out**.

Quicksort (maybe we should call it **splitsort**) takes the array,

$$\{91\ 46\ 16\ 46\ 61\ 46\ 89\ 78\ 27\ 81\ 83\ 37\ 95\ 34\ 75\ 56\}$$

chooses a **pivot** value, say 46, and splits the array into two *non-empty* parts, one consisting of elements that are less than *or equal* to the pivot value, the other consisting of values greater than *or equal* to the pivot value:

$$\{34\ 37\ 16\ 27\ 46\} \qquad \{61\ 89\ 78\ 46\ 81\ 83\ 46\ 95\ 91\ 75\ 56\}$$

sorts the two parts recursively:

$$\{16\ 27\ 34\ 37\ 46\} \qquad \{46\ 46\ 56\ 61\ 75\ 78\ 81\ 83\ 89\ 91\ 95\}$$

and finally **concatenates** them as they are *already in order*:

$$\{16\ 27\ 34\ 37\ 46\ 46\ 46\ 56\ 61\ 75\ 78\ 81\ 83\ 89\ 91\ 95\}$$

So **quicksort** does the work **on the way in**.

Notice that the "≤" and "≥" parts are neither sorted nor in the same order that these elements appeared in the given array, and that "=" values may be put in either part. It is not important what order they're in (of course, because we're about to sort them), but this is the order that you get from the splitting process according to the simplest algorithm for doing it.

(It could alternatively split the array into three parts, including one for values *equal* to the pivot, but this involves a more complicated splitting algorithm, known as the "Dutch National Flag".)

The recursive parts of the programming are easy. What is tricky is to get the **merge** and **split** operations to work. We concentrate on those first.

Before you start work on the programming, do `trace merge-sort` and `quick-sort` a few times to get a feel for how the splitting and merging work.

11.2 Write the code for merge

Do not confuse merge with mergesort!

There are no more templates any more: you create your own file called `Merge.java` containing

```
class Merge implements Merging {
   public int[] merge (int[] B, int[] C) {

       your code goes here

   }
   public void choice (int c) {}
}
```

Assume that the arrays `B` and `C` are already sorted (when you have the whole of mergesort in place they will have just been sorted).

Allocate a new array `A` that's just big enough to contain the merge of `B` and `C`.

Merge `B` and `C` into `A`, so that `A` ends up sorted and containing the same stuff as `B` and `C`.

Draw the diagram showing how **merge** is going to work.

Don't forget to deal with the situation where one of the arrays has been exhausted before the other, for which a simple copy (with no comparisons) is needed.

Run the TESTER on `run Merge.java` in the usual way. The `size` command line argument controls the size of `B`; use `size2` for `C`.

You can use `Trace.printarray (B, j)` *etc.*. to see your code in action.

Give it the example above explicitly:

```
trace Merge.java array=16,46,46,46,61,78,89,91 array2=27,34,37,56,75,81,83,95
```

These arrays are already sorted, so the precondition for merge is satisfied, and you should get a sorted result.

Test the situations where *all* of `B` comes before *all* of `C`, or *vice versa*.

There is no need to time this code: it is only a subroutine, and is pretty obviously linear in the (total) size of the arrays.

11.3 Mergesort

Create a file `Mergesort.java` containing

```
class Mergesort implements Sorting {
   public int[ ] sort (int[ ] A) {
      int size = A.length;

      your code goes here

   }

   // access to merge(B,C)
   private Merging merger;

   // this is for the hybrid algorithm (Section 6)
   private int cutoff=0;
   public void cutoff (int n) { cutoff=n; }
   private Sorting insertion_sorter;

   public void choice (int c) {}

   public Mergesort() {
      merger = new TestMerge();  // or Merge() to use your own
      insertion_sorter = new Insertion();
   }
}
```


Begin by finding the midpoint:

```
int mid = size/2;
```

Assign this *explicitly* to a variable — don't just use this expression verbatim in the code. Remember that `size` may be odd; it is more important to make the calculation of the left and right "halves" consistent than to make `mid` exactly `A.length/2`.

Now declare two new arrays (`B[ ]` and `C[ ]`) of appropriate sizes and copy half of the given array into each of them.

Next is where the the recursive calls to `sort(B)` and `sort(C)` belong. Write the code, but comment it out (*i.e.* put `//` before it) for the time being.

Now use `merger.merge (B,C)` to merge them, and `return` the result of the merge as the result of `sort`.

The code as given above (with `TestMerge`) uses the `merge` method that's built into the TESTER, so you can be confident that this part is correct. Substitute your own code (with `Merge`) later, when you have mergesort working correctly.

Try this out using an array consisting of two sorted halves:

```
run Mergesort.java array=16,46,46,46,61,78,89,91,27,34,37,56,75,81,83,95
```

There is a **pitfall** here. If you fall into it, you will get back the same (unsorted) array that you put in. This is to do with the distinction between JAVA *variables* and the *objects* (or arrays) to which they refer. You have recently done some exercises in *Introduction to Programming* concerning this point.

If you didn't actually make this mistake, you have missed out on a learning opportunity — alter your code to make this happen.

If we had been using `C` or `C++`, you would have fallen down this hole much earlier.

11.4 Recursive Merge Sort

Now enable the recursive calls to `sort(B)` and `sort(C)` and run the TESTER again.

What happens?

What if `A` is empty or only contains one element?

When your mergesort is correct, the TESTER will, as usual, time it for you. This time you will see **$n \log n$ time complexity** (instead of the n^2 complexity for insertion sort *etc.*.).

You can time it for larger sizes of array using commands like these:

```
run Mergesort.java size=1000000 thread=no
check Mergesort.java sizes=200000,500000,1000000 maxtime=60
```

Because of the allocation of scratch space within the recursive calls, you will not be able to run this for very large arrays: 1,000,000 may be the largest that the JAVA run time system allows.

Does it make any difference to the performance of merge sort if you give it already-sorted data?

11.5 Seeing recursive mergesort in action

It's rather more difficult to get to see what recursive programs are doing than for `while` programs. Here's something that you can try.

First, take out the calls to `Trace.printarray()` that you put in the loop(s) for merge.

There is a special option to demonstrate mergesort in action:

```
Trace.printmerge (int[] A, int indent, int level, boolean direction);
```

The `direction` argument is there to show when you enter and leave `sort`. Call it with `true` on entry and `false` on exit; then an appropriate bracket will be printed.

The other two arguments keep track of how deeply nested the recursion is, and what part of the array you're working on. You will need to alter your code like this to use them:

```
public int[] sort (int[] A)
   { return mergesort (A, 0, 0); }

private int[] mergesort (int[] A, int indent, int level)
   { /* your original code for sort(A) */ }
```

and replace `sort(B)` with

```
mergesort (B, new_indent, level+1);
```

and similarly `C`.

The `level` argument causes that many dots to be printed, to show the depth of the recursion.

The `indent` argument shifts the printout of the array across the screen by some number of cells.

Work out what value of `indent` has to be passed to each of the two recursive calls to align them with the two halves of the larger array from which they came. *How many elements of the original array are to the left of the part we're looking at?*

11.6 Hybrid of mergesort and insertion sort

Is there some size below which insertion sort is faster than merge sort? Work out, by experiment, what it is.

Why does merge sort perform so badly on small arrays?

Why is this important for its use on large arrays too?

Make a new sort method that uses insertion sort for small arrays and merge sort for big ones.

Use the `cutoff` variable to decide whether to use mergesort or insertion sort on the present array, depending on its size. This variable can be set by adding `cutoff=25` or similar to the TESTER command line.

Time this one.

Is it better than the the original `merge_sort` for big arrays? By how much? What *function* of the size is this?

Try to find the optimum value of `cutoff`.

Carefully compare the coefficients of both $n \log n$ and n in the formulae for the dependence of execution time on size.

Mergesort is very profligate with *space*. Sections 11.10ff consider ways of reducing the space used and replacing the recursive calls with `while` loops.

11.7 Quicksort

Mergesort does the work (merging) on the way *out* of the recursion.

Quicksort does it on the way *in*, and also does it *in place.*

Just as mergesort needs a `merge` subroutine, quicksort needs `split`, but this time we shall treat the main algorithm as the priority and leave `split` for the better programmers to do later.

Create a file `Quicksort.java` containing

```
class Quicksort implements Sorting {
   public int[ ] sort (int[ ] A) {
      return quicksort (A, 0, A.length);
   }

   private int[ ] quicksort (int[ ] A, int left, int right) {

       your code goes here

   }

   // access to split (A, left, right)
   private Splitting splitter;

   // this is for the hybrid algorithm
   private int cutoff=0;
   public void cutoff (int n) { cutoff=n; }

   public void choice (int c) {}

   public Quicksort() {
      splitter = new TestSplit();  // or Split() to use your own
   }
```

Do not allocate any new arrays.

As with `shift()`, `where_is_min()` and `where_to_insert()`, we shall work with **segments** of the array, delimited by `left` and `right`. As usual, `left` is *inside* and `right` is *outside* the segment. At the top level, `left=0` and `right=A.length`.

The interesting thing about quicksort is how to choose the pivot value. Start by using `pivot=A[left]`.

Call `split` with appropriate arguments to tell it what segment to consider and what the pivot value is. It re-arranges the segment and returns the *position* at (before) which to make another split.

Call `quicksort` recursively on the two halves.

Where is the sorted result now?

Try it out using the TESTER.

There is a special tracing routine to let you see how quicksort works:

```
Trace.printquick (A, left, right, level, direction)
```

As you did with `mergesort`, you will need to add an argument to `quicksort` to count the level.

11.8 Choosing the pivot for quicksort

You can do this experiment with the version of quicksort that's built in to the TESTER.

```
trace quick-sort pivot=first
```

Other ways of choosing the pivot are `last`, `middle`, `mean`, `random` and `median3`.

What is meant by the **average**, **mean**, **median** and **mode** of a collection of numbers?

Which of them can be calculated easily? Which of them would you like to use, ideally, to choose the pivot value for quicksort?

The `first` value in the segment is the obvious one to try, but what happens when you run quicksort on already sorted data? Use `trace` to watch it.

Is the `last` value any better than this? What about the `middle` one, or one in a `random` position?

The textbooks tell you to use the **median** of the first, middle and last values.

Write the code to do this. It is more complicated than you might like for such a simple job.

As is unfortunately rather common, people write textbooks and copy ideas like median-of-three from previous generations of textbooks (but adding the word "JAVA" in fluorescent letters to the cover of the book, to make you but it) without considering the ideas carefully themselves.

Do

```
run quick-sort sawtoothpc=50
```

to see what I mean by a **sawtooth** array. (I got the word from sawtooth *waves*, which are used to move the electron beam across a television screen, as my father was an electronics engineer designing televisions.)

You can control the position of the **vertex** of the sawtooth by using `sawtooth=2` to put it 2 places from the left, `sawtooth=-2` to put it 2 places from the right or `sawtoothpc=25` to put it 25% of the way through the array.

Do

```
trace quick-sort sawtooth=1 pivot=median3
```

and guess what the result of `check` might be *before* you try it.

The rest of this exercise sheet is for the serious programmers.

11.9 Splitting

(To be written. See the exam question.)

11.10 Merge Sort with `while` loops: easier way

Merge sort does not have to be recursive. We can do it with loops instead. It's just that the programming is more difficult.

The print-out from part 11.5 shows that recursive merge sort breaks the array down into single elements and re-combines these into 2s, 4s, 8s, *etc.*, doing as much work as it can at the left before moving towards the right.

A different way to proceed is to combine *all* the 2s first, throughout the array, then all the 4s, then the 8s, up to the whole array.

Modify your merge code to copy segments of the array between certain pointers, instead of the whole array.

You will need nested loops like this:

```
 for (int frag_len = 1; frag_len < size; frag_len *= 2) {

      for (int from = 0;  from < size;  from += 2 * frag_len) {

            // merge B[from...from+frag_len-1] and B[to...to+frag_len-1]
            // to corresponding segment in A
      }
 }
```


When you've done each merge, you can either copy it back, or alternately merge from `A[]` to `B[]` and *vice versa*.

Don't forget to make allowance for the size not being exactly a power of 2.

11.11 Merge Sort with `while` loops: disk-efficient way

Part 11.10 wasn't the way in which the recursion in part 11.3 actually worked. When sorting amounts of data that are too large to fit in memory and must be stored on disk, the order of work that the recursive method exhibits is better because we can put whole blocks of sorted data on disk, and only have to read them back when we get to merging very long segments. The method in part 11.10 involves reading and writing to disk for *every* power of 2, not just the bigger ones.

How can you implement this method using while loops? How does this compare with the way that recursion is implemented on the machine?

This is not easy. It is only for the most confident programmers.

11.12 Merging runs instead

Here is yet another way to do merge sort, which will take linear time on already-sorted data.

Instead of dividing the array into 2s, 4s, 8s, 16s, and so on without regard to any order that may already exist, look for **runs** and merge those.

A **run** is a sequence of consecutive elements that are *already* in order.

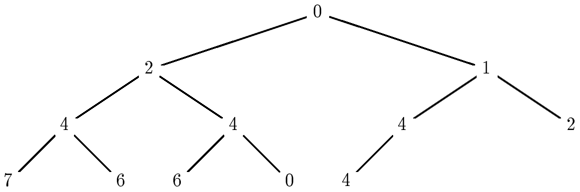
Plainly it is inefficient to copy a run of length 1000 in order to merge a few elements into it, so we need a scheme of runs of diminishing lengths, so that you only merge runs that are of approximately equal lengths.

Try to mimic what the ordinary binary mergesort does.

Chapter 12

Heaps and priority queues

A **heap** is a **binary tree, represented as an array**. For example,



is strored in the array

$$\{0, 2, 1, 4, 4, 4, 2, 7, 6, 6, 0, 4\}$$

Each node carries a value that is $\geq$ the value of its **parent**. This is what we mean by **heap order**. Any sorted array is in heap order, but not conversely. For example, the sequence 1,3,2 is in heap order, but it is not sorted.

This means that the **root** carries the **least** value. Insertion and deletion can be performed in $O(\log n)$ time.

A heap could be used to implement a **priority queue**, *i.e.* in which the entries need not be processed in precisely the order in which they arrived, but according to some assigned priority measurement.

- In the Accident and Emergency department of a hospital, this assessment is called *triage*.
- A printer or other shared computer resource ought to process small jobs quickly, and leave big ones until a quiet period.
- The basic idea of **selection sort** — to find the next smallest value — can be implemented in $O(n \log n)$ time instead of $O(n^2)$ since heaps provide the next smallest value in $O(\log n)$ time instead of $O(n)$.
- We shall use priority queues later as a subroutine in searching for the shortest path between points in a network.

12.1 The place of this topic in the course

The following instructions assume that you are familiar with Richard Bornat's notes about the heap data structure and the sifting up, sifting down, heapifying and sorting algorithms. See also Chapter 20 of Weiss's book.

Heap sort does seem to be an excellent teaching example, because

- the shortest version of the code is utterly meaningless, so you *have* to use theory to get it to work, and
- it contains several traps that nicely test students' understanding of the algorithm.

Some of the common errors obviously make the program *logically* incorrect, but other misunderstandings lead to larger time complexities, and in particular to a quadratic sorting algorithm. This was one of the things that inspired me to write the TESTER.

The operations of moving things up and down the tree are called by different names by the various authors. I call it *sifting*;[1] this follows Peter Burton, who has taught the same algorithm in his third year course, *Algorithms and Complexity.* Mark Allen Weiss calls it *percolation.* Richard Bornat calls it *bubbling*, but this has got students confusing it with bubble *sort.*

12.2 The tree as an array

Draw a 15-node binary tree on paper. Number the nodes, starting with 1 at the root, and going from left to right across each level of the tree.

If a ("parent") node is numbered p, what is the number of its *left* child?

If a ("child") node is numbered c, what is the number of its parent?

These are paper exercises — move away from your computer to do them!

However, the first element of an array in JAVA is numbered 0. Draw another tree, starting the numbering from 0, and work out the formulae for the parent and left child.

In each of the JAVA classes that you implement (for various X), you will need a copy of the following code, with the formulae that you have just found inserted:

```
class HeapX implements HeapXing {
   final static int ROOT = 0;
   public int root () { return ROOT; }
   public int parent      (int child)  { return ???; }
   public int left_child  (int parent) { return ???; }
   public int right_child (int parent) { return ???; }
   public int sibling     (int child)  { return ???; }
   public void choice (int c) {}

   public void heap_X (...) {
   }
}
```

A **sibling** is either a sister or a brother. This method should convert between the left- and right-children of the same parent.

We shall also need to refer to the **contents** of the heap and **capacity** of the array used to implement it, as numbers.

Think of the array as a jug: its *capacity* may be 1 litre, but it may currently *contain* just 0.6 litres of water. The capacity is the greatest amount of water that it can contain. We may pour some of the water out, or add some more water to the jug, but if we want to store more than 1 litre of water, we shall need a *bigger* jug.

At first we shall pass around the **array** and its **contents** as program variables (of course, **capacity=array.length**), but at the end these will become instance variables in an objected-oriented implementation of heaps.

[1] *Sifting* (not "shifting") is what you might do with a sieve to flour or soil to separate the finer grains from the lumps.

12.3 Sifting up and insertion

To *insert* a new value in a heap, you first put it at the end (increasing the `contents` by 1) and then *sift it up*.

So if the value is $<$ that of its parent (violating heap order), you swap parent and child. The new value is now in the parent position, but may still violate heap order (by being $<$ the value of the grandparent of the original position), so you carry on swapping.

This makes the algorithm sound like bubble sort, but

- you pass from `position` to `parent(position)`, not to `position+1`, and
- you only make one pass up the tree, since the rest of it is assumed already to be in heap order, so
- really it's more like *insertion sort*, as the ancestors (parent, grandparent, ...) are successively demoted, and then the new value is inserted once.

See Richard Bornat's lecture notes for a pictorial explanation.

You can also watch my code in action by doing

```
trace sift-up
```

in the usual way. The array will be shown as a tree, with the branches drawn in different ways depending on whether heap order is satisfied or not.

Sifting up very often terminates after one generation, so you may have to run it several times to see it working.

The `sift_up` method has the following signature, and belongs in a class that `implements HeapSiftingUp`. It also needs the `parent` and `child` formulae above.

```
class SiftUp implements HeapSiftingUp {
    public void sift_up (int[] array, int position, int contents) {
    }
}
```

The `sift_up` method does not itself do an insertion, or increment `contents` — that will be done in the object-oriented version in Section 12.9.

What the TESTER actually does here is to create a random heap, choose some (leaf) `position` in it, reduce the value there so that heap order is violated, and give the `array` and the faulty `position` to your method to fix.

If the heap has n elements, what is the (worst case) time complexity for inserting and sifting up a new value?

The TESTER will not do the timings for you here, because the time taken to restore the array between runs is greater than that taken to do the run of your code — it is far too delicate a measurement! You saw how difficult it was to get good results for binary search, but that doesn't alter the array, so it is not necessary for the TESTER to restore it between runs.

The tracing output from my code uses the method

```
Trace.printheap (int root, boolean root_is_max,
                 int[] array, int contents);
```

for which the first two arguments should be `0` and `false` if you have followed the instructions above.

12.4 Sifting down and deletion

In the applications of heaps to priority queues and sorting, it is only the *root* element (which is at `array[ROOT]`) that we need to delete, so we shall only consider that case. This element is at the *head* of the queue, so it is deleted from the queue to be "dealt with".

However, we cannot simply discard `array[ROOT]` — *something* has to go in its place! Although it may perhaps seem the least appropriate value to choose, we move the *last* value in the heap, `array[contents-1]`, to the root, and then *sift it down* to restore heap order.

Sifting *up* involved comparing the position that potentially violates heap order with its *parent*. Sifting *down* is similar, except that now there are (potentially) *two children*.

As before, you can `trace sift-down` to watch my code in action.

The code for sifting down is the most difficult part of the programming in this topic. Usually, students write complex, deeply nested `if` statements to handle all of the possibilities:

- there may be 0, 1, or 2 children — don't forget the case of 1 child — and
- the values of the parent, left and right children need to be compared.

It is rather common in programming — especially when doing filthy things like form-processing for Web sites — to have proliferations of cases like this. So it is an important programming skill to be able to **prune** tangled trees of `if` statements.

In fact, there is a trick in this case that makes `sift_down` only a couple of lines longer than `sift_down`. The idea is to decide between the two children (if there are indeed two) *before* looking at the parent at all.

You will, of course, find this trick implemented in the many versions of the code that can be found on the Web. Needless to say, these versions *do not explain how they work*. So, if you manage to work out this trick, *use comments to explain it*. I hardly need to say that it is good programming practice to do so, but it will also tell me whether you have found out how to do it for yourself or just copied the code from the Web.

Even though we shall only be *deleting* the root value, we shall later need to sift down from other positions. As usual, this generalisation is only a matter of using the argument as the initial value of the loop variable.

The template is as follows, together with the `parent` and `child` formulae above.

```
class SiftDown implements HeapSiftingDown {
   void sift_down (int[] array, int position, int contents) {
   }
}
```

Once again, the `sift_down` method does not itself do a deletion. The TESTER creates a random heap, with one of the values near the root increased, so that heap order is violated. The `array` and the faulty `position` are given to your method to fix.

Again, what is the (worst case) time complexity?

The TESTER cannot, unfortunately, tell you, for the same reason as before.

12.5 Making an array into a heap — standard way

The way that you *normally* construct a data structure is — to use its constructor methods.

In this case, you turn an array into a heap (*i.e.* impose heap order on it) by *inserting* the values one by one into the heap. Do

```
trace heapify-by-insertion
```

to watch the model answer in the TESTER doing this.

The template is as follows, together with the `parent` and `child` formulae above.

```
class Heapify implements Heapifying {
   public int[] heapify (int[] array, int contents) {
   }

   private HeapSiftingUp sifter_up;
   public Heapify () {
      sifter_up = new TestHeapUp(); // or SiftUp() to use your own code
   }
}
```

The **contents** argument for the **sift_up** method is *not* that of the **heapify** method: in fact it is **heapify**'s loop counter.

You may allocate a new array if you wish, but the input and output can share the same array, with the output (the heap) first, essentially as you did with the sorted and unsorted segments in insertion sort and selection sort.

Considering the time complexity of each insertion operation, what would you expect the time complexity of **heapify** to be?

Use the TESTER to check your answer.

12.6 Making an array into a heap — fast way

Although what you have just done is the *standard* way to build a data structure, there is in fact a faster way to turn a random array into a heap.

Think of it as a tree. Each node also defines a tree, consisting of its descendants.

In particular, the *leaves* of the big tree (which, remember, constitute *half* of its nodes) define one-element trees.

Are these one-element trees in heap order?

Now consider a node whose tree of descendants is not, perhaps, in heap order, but assume that its two subtrees are. (This is the loop invariant.) What operation (that you have already programmed) do you need to perform to turn the bigger subtree (the node under consideration plus its two subtrees) into a heap?

So, working up (backwards) from (the parent of) the last element in the tree, you can impose heap order on the whole tree (array). Do

```
trace heapify-by-sifting-down
```

to watch the model answer in the TESTER doing this.

In **Heapify2.java**, implement this new way of heapifying an array.

Considering the time complexity of the operation that you performed on each node, what would you *expect* the time complexity of **heapify** to be?

Use the TESTER to check your answer.

12.7 Some more difficult questions to think about

(a) Explain why the complexity of the faster **heapify** is what the TESTER says, not what you originally thought. The analysis is the same as that for the "early exit" from binary search and the hybrid merge sort algorithm.

(b) Prove that this way to do **heapify** works, *i.e.* that the result is in heap order and is a permutation of the original array. The appropriate diagram is a tree, not a rectangle!

(c) If you look carefully at the graph, you will see a bend in it, just as you did in the logarithmic plot for binary search. The explanation is the same: fetching stuff from the secondary memory cache within the CPU or from RAM. Why, considering the way in which modern microprocessors are designed, is heap sort not such a good idea?

(d) If make an arbitrary change to the value at a particular node, or if you want to delete some node other than the root, what combination of **sift_up** and **sift_down** does the job?

12.8 Sorting

Use **heapify** to turn a given array into a heap, and then the other heap operations to extract a sorted array from the heap. As before, you can use the same array for input an output.

Your **class HeapSort implements Sorting** as before.

What do you expect the complexity to be? What does the TESTER say?

12.9 Heaps as Objects

Use the heap methods that you have written *and already tested* to fill in the code in the following template file called **Heap.java**.

When you **insert** a new element, check whether **contents** would exceed **capacity**. If so, create a new bigger **array** and copy the old one into it (*cf.* the jug). What is the best new **capacity** to choose?

```
class Heap {
   private int capacity;
   private int contents;
   private int [ ] array;

   final static int ROOT = 0;
   public int root () { return ROOT; }
   private int parent      (int child)  { return ???; }
   private int left_child  (int parent) { return ???; }
   private int right_child (int parent) { return ???; }
   private int sibling     (int child)  { return ???; }

   private int choice = 0;
   public void choice (int c) { choice=c; }

   // make a new heap with contents=0 and capacity=size
   public Heap (int size) {
      ???
   }

   // make a new heap with default size
   public Heap () { this (10); }

   // make a new heap by heafifying a given array
   public Heap (int[ ] data) {
      ???
   }

   // transfer the heap to a bigger array (cf. jugs)
   private void rebuild (int new_size) {
      ???
   }

   private sift_up (int position) {
      ???
```

```
}

private sift_down (int position) {
   ???
}

public void insert (int new_value) {
   ???
}

public int delete_root () {
   ???
}

private void heapify () {
   ???
}

public int[ ] sort_me () {
   ???
}

// use a "new Heap" so that your class "implements Sorting"
public int[ ] sort (int[ ] array) {
   ???
}

// maybe write your own (command line) user interface
public static void main (String[ ] args) {
   ???
}
}
```

You can add other methods to make this class implement the various TESTER interfaces and thereby re-test your code. However, the TESTER will only allow you to do these one at a time.

Chapter 13

Hash tables

Hash tables were invented by the authors of compilers for programming languages in the 1960s, as a trick for maintaining the list of user-defined names (such as program variables) and associated properties. An interpreter for the formerly popular teaching language `BASIC`, for example, consists of little more than a calculator together with a hash table in which variable-names and their current values are recorded.

The function of a hash table is to enter and retrieve data as quickly as possible. It does not sort the data, which would take $O(n \log n)$ operations. Entry, deletion and retrieval of single items in a hash table can be done in *constant* time, irrespectively of the number of items in the table, so building the table takes just $O(n)$ operations.

The basic idea is to put each item in a numbered "pigeonhole", the number being obtained by some quick and repeatable (but otherwise meaningless) piece of arithmetic, called a *hash function* (Section 13.2.2). As there are vastly more possible names than one can provide pigeonholes, there must also be a strategy for resolving *conflicts*, *i.e.* when the pigeonhole that we want to use for one name is already occupied by another.

The main factor on which the performance of the hash table depends is how crowded it is. Experiment shows that there should be approximately twice as many available pigeonholes as items to go in them. When the table gets too full, it needs to be rebuilt.

If you are planning to write an essay on hash tables in the exam, you should use my implementation to do further investigations of the performance when the load factor and other features are changed. You should not do more programming work for the exam.

[Explain why prime numbers are used.] [Section on performance.] [References.]

These notes were written after I had marked the April 1999 course-work, and are based on the points that arose from marking. They are *complementary* to the treatments in Richard Bornat's lecture notes and textbooks such as that by Weiss.

Other sources:

- Richard Bornat's notes,
- Chapter 19 of Weiss's book,
- The documentation of the JAVA library.

13.1 Using hash tables

In the subdirectory `Hash` of the course directory, you will find

- a test program called `Main.java`
- the compiled version of this program, `Main.class`

- a file `dcs-login` that contains the login and human names of everyone in the Department and is read by the test program
- the compiled version `Hash_Table.class` of my implementation of a hash table
- a template `template.java` that is a simplified version of my implementation, with its three most important methods removed.

Start by running `Main.class`. It will print out some statistics as it builds two hash tables (for the login-to-human *and* human-to-login translations) from the data, and prompt for queries. Try your own login name first: it will look this up in both tables, responding with your human name in one case and "not found" in the other. For the reverse translation you have to type your (human) name exactly as it appears in the password file.

Apart from answering queries, as an application would be expected to do, the program also prints some debugging output. This shows each of the entries in the tables that were considered during the search. In a good implementation, there should only be one, two or maybe three of these, but in order to show you how the algorithm works, these hash tables have been built with a very high load factor, requiring multiple attempts to locate the entries.

One table uses linear rehash, the other quadratic (Section 13.2.3), so you can see what these do since the printout gives the table index, the long hash code, the key and the data.

13.1.1 Object-oriented aspects

If you now look at the JAVA source code provided, you will see that `Main.java` has **method calls** like

```
login_to_human.put (login_name, human_name);
```

whilst the corresponding **method definition** in `template.java` is

```
public Object put (Object key, Object data) { ... }
```

with no dot (or anything to go before it).

JAVA is an **object-oriented programming language**, and this program is written in the object-oriented style. The *algorithms* for hash tables are enclosed in a **class** declaration, whilst a *particular* hash table (such as `login_to_human`) is an **instance** of the class (this is object-oriented programming jargon). Each instance is declared (in `Main.java`) using the **constructor**

```
Hash_Table login_to_human = new Hash_Table ();
```

Objects, like arrays, may be passed around and assigned **by reference**: the constructor returns a **reference** to a hash table. Its details are meaningless to the application (`Main.java`): the reference simply says *which* hash table to use.

Imagine that you are in a foreign country and are given a visa or some other official document, written in a foreign language, that's meaningless to you, but which the local bureaucrats think is very important. All you have to do with it is take it off one bureaucrat to give to another.

You access different objects (such as the two hash tables `login_to_human` and `human_to_login` in `Main.java`) by prepending their names to the method name, *e.g.*

```
login_to_human.remove("pt");
```

From the point of view of the implementation (`Hash_Table.java`), what this means is that you can write the code as if there were only one single hash table in the world. You (almost) never have to refer to any *particular* instance (hash table): there is, it seems, only one `contents` variable.

If ever it is necessary to say whose `contents` variable something is, it's called

```
login_to_human.contents
```


as with the method calls. However, it is advisable to **hide** or **encapsulate** variables such as this, only allowing the external user to read or change them via **methods**. One reason for this is that, in the case of the hash table, this variable is part of a **data invariant** (Section 13.3.1), and must only be changed in harmony with the rest of the data.

There is another course about object-oriented programming: everything that you need to know for this exercise, or for this course, you can find out by looking at how the dot is used in `Main.java` and `template.java`.

The dot syntax is rather clumsy when it comes to using binary operations such as `lessThan` or

```
if ( skey.equals (key[i]) ) { ... }
```

because the natural symmetry between the arguments is lost. However, there is a reason for this: the object `skey` is an instance of a particular class (say `String`), and the actual code that is executed is that which is the definition of the `equals()` method in that class. This means that

```
if ( key[i].equals (skey) ) { ... }
```

may possibly do something quite different, should `key[i]` happen to be of a different class, containing a different `equals()` method (Section 13.1.3).

Usually, the data (*e.g.* the variable `contents`) belong to the *object*, *i.e.* the *instance* of the class, not the class itself. This is how JAVA makes it possible to use many instances of a class, even though only one version of the code, with an apparently global `contents` variable, has been written.

However, if you declare a variable as `static` it belongs to the class instead. For example, you might want to accumulate usage statistics for the search algorithms without distinguishing between instances.

13.1.2 Public methods for accessing data

We have already seen how to use the **constructor**,

```
Hash_Table login_to_human = new Hash_Table ();
```

and one of the **public access methods**,

```
String pt_name = login_to_human.get ("pt");
```

of which the others are

```
boolean present = login_to_human.containsKey ("pt");
boolean present = login_to_human.contains ("Paul Taylor");
String pt_old_name = login_to_human.put ("pt", "Paul Taylor");
String pt_old_name = login_to_human.remove ("pt");
```

Notice that, although the last two methods would have `void` return types in natural usage, we use the return value to tell the application whether this entry had previously been defined or not, and if so what its old value was. The application can then report an error if it so chooses. It may also want to maintain usage statistics about the old data value before it is discarded.

The `contains()` method is only included because it appears in the hash table implementation to be found in the JAVA library; it does a *linear search through the entire table*, which is plainly stupid.

13.1.3 What objects can you put in a hash table?

Almost all uses of hash tables are indexed by strings; the data are often strings too, but may be numbers.

However, hash tables may be used to look up other things, with very little change to the code.

One application is playing games such as Chess. The tree of possible moves needs to be searched to some depth using a minimax algorithm (with alpha–beta pruning). However, there are often pairs of "independent" moves, that can be done in either order, resulting in the same position. This means that the minimax algorithm does a lot of unnecessary work, by considering the same position twice.

The solution to this problem is to hash the positions (as the keys) and minimax's calculations (as the data). For this there must be a `GamePosition` class.

The hash table does not need to convert game positions into strings before storing or manipulating them. All it needs is to be able to call the following two methods in the `GamePosition` class:

```
int long_hash_code = key.hashCode ();
boolean same = key1.equals (key2);
```

Any class that has these methods (can be declared to say that it) `implements` the `Hashable interface`.

If you do not provide an `equals` method for the class, the default one from the `Object` class is used. This just compares the references, so, for example, `String`s that contain the same letters in the same order but which were declared separately are considered *unequal*.

13.1.4 Methods to access the control parameters

With no arguments, the constructor has to make its own guess as to the size of the table required. As the table can be rebuilt automatically with a larger size, this is OK. However, the constructor may be given arguments to specify the original size of the table, what the acceptable load factor is, and how fast it should grow when this is exceeded.

The application may perhaps wish to change these parameters after the table has been created, or to find out how full it is or how efficiently it's working.

My implementation (`Hash_Table.class`) has numerous bells and whistles that you can ring/blow by making minor adjustments to Main.java. You are **not** expected to implement these yourself, or be put off by how many there are! They are there for you to do experiments and design your own implementation. In each case, if the method is called with no arguments, the current value is returned.

```
int capacity (int new_size)
int used () { return used; }
int size () { return contents; }
int contents () { return contents; }
boolean isEmpty () { return (contents==0); }
double inflation (double new_inflation)
int increment (int new_increment)
boolean report_rebuild (boolean new_report_rebuild)
boolean use_quadratic_rehash (boolean new_use_quadratic_rehash)
double conflict () { return ((double)nconflicts)/((double)nsearches);}
double loading () { return ((double)used)/((double)capacity); }
double real_loading () {return ((double)contents)/((double)capacity);}
double load_factor (double new_load_factor)
```

The meaning of `increment` and `inflation` is apparent from

```
private int rebuild_size ()
     { return next_prime ((capacity + increment) * inflation); }
```


Changing `capacity` or, if necessary, certain of the other parameters, triggers a rebuild of the table. The JAVA library's `ensureCapacity (new_size)` is the same as `capacity (new_size)`.

Find out what happens to the behaviour when you change the load factor or switch to quadratic rehash. With load factor 0.9, quadratic rehash crashes; there is a little number-theoretic problem for the maths students to see why; how could *cubic* rehash solve this problem?

13.1.5 Other methods

The debugging output that `Main.class` produces is obtained from

```
String output = login_to_human.show ("pt");
```

and the whole table is rendered in this way using `toString()` (or just the name of the table as part of a string concatenation) in the usual way.

I'll leave you to guess what these methods do:

```
login_to_human.clear ();
Hash_Table copy = login_to_human.clone ();
```

You can get all of the keys, or all of the data, using

```
Object[] data =  login_to_human.element_array ();
Object[] key  = login_to_human.key_array ();
```

There is another way, using the `Enumeration` interface (see the JAVA library documentation):

```
Enumeration list = login_to_human.elements (boolean quick);
```

or

```
Enumeration list = login_to_human.keys (boolean quick);
while ( list.hasMoreElements () ) {
      Object key  = list.nextElement ();
      Object data = login_to_human.get (key);
      ...
}
```

The enumeration methods maintain a cursor, which has to be part of *another* object in order to allow you to do double enumerations (*e.g.* running through all *pairs* of vertices in a graph). The `quick` switch, if true, just has this pointing into the hash table itself, so it is invalidated whenever a change is made to the table (the corresponding methods in the `Vector` class in the JAVA library, for example, have this bug). With `quick=false`, the enumeration uses a private copy, which is made using one of the to-array methods.

13.2 Implementation

`template.java` is my implementation with the "guts" taken out. The bits that (I have taken out and) you have to write are, in this order:

- the private **locate (search_key)** [or find] method, which is the core of the algorithm; it must return the index into the table where either the key has been found or there is an `EMPTY` or `DELETED` cell that can be used to insert it;
- the public **put (key, data)** [or insert] method for inserting items in the table, and
- the private **rebuild (new_size)** [or rehash] method that rebuilds the whole table when it has got too full.

Please note that the test program (Main.java/class) uses a more elaborate version of (my implementation of) Hash_Table.class than is given in the template. You are **not** expected to implement all these fiddly bits yourself!

13.2.1 The data

The data in the table consist of several arrays, the most important of which are

- `Object[] key  = new Object [capacity]` and
- `Object[] data = new Object [capacity]`

The **key** is what you put in (*e.g.* a name) to unlock the **data** (*e.g.* a telephone number). In my implementation, the keys and data are **Object**s (the most general JAVA class: see Section 13.1.3), but you may safely substitute String for Object throughout.

Along with the data itself, various control parameters must be maintained, so that the program knows *easily* when to rebuild the table, without repeated counting:

- `capacity`: the number of cells available altogether;
- `contents`: the number of currently valid entries;
- `used`: the number of cells currently occupied by valid or `DELETED` entries.

It is part of the **data invariant** (Section 13.3.1) that the numerical value of the `contents` variable should really be the number of valid entries in the table.

In my implementation, the long hash code is saved in a third array,

- `int[] code = new int [capacity]`

for use when rebuilding the table; this avoids calling `hashCode ()` again.

I use "special" values in `code[]` to indicate `EMPTY` and `DELETED` positions. This means that the return value of each call to the `hashCode()` method must be tested, and replaced with an `OTHER` value if it happens to be equal to one of these.

Another approach is to use two `boolean[]` arrays, or better two `BitSet`s (see the JAVA library), to record whether an entry is `EMPTY`, `EMPTY|DELETED` or valid. (Examination of the tests that actually need to be done in the code shows that this is better than storing `DELETED` instead.)

13.2.2 The hash coding function

Despite extensive attempts to design better ones, experience has shown that it is not important *how* the hash function is calculated. In any case, in any single application (for example compiling a particular program written by a particular person) there are regularities in the distribution of names that would compromise any "rationally" designed hash function.

The rule is simply that *all of the bits in the key must be used.*

From the design of spelling checkers, it is known that the erroneous form of a word almost always differs from the intended one by a single change of one of four kinds: insertion, deletion or alteration of a single letter, or transposition of two letters.

Applying a similar idea to the design of a hash function, we must ensure that each letter contributes to the value. For example, `item1` and `item2` must get different values, This might be done, for example, by adding up their ASCII or Unicode values.

However, the sum of the values (or any other associative commutative operation, such as bitwise OR) fails the transposition requirement. For example, in `Main.java` there are variables called `login_to_human` and `human_to_login`, which contain the same letters in different orders. A common solution to this problem is to *multiply* each ASCII code by its *position*; this trick is used, for example, to calculate check digits in ISBNs, bank account numbers, *etc.*.

In fact the JAVA `String` class (along with many others in the JAVA library, in particular `Integer`, `Float`, *etc.*) already has a `hashCode()` method. You should use this instead of writing your own, although *when you have your hash table working* you may like to experiment with alternative hash functions.

The `hashCode()` method in the library returns a 32-bit `int` value, which is of course too big to use as the index into an array. I call this the **long hash code**; it is printed (in hexadecimal) in the debugging output of `Main.java`.

The index into the table, which I call the **short hash code**, is obtained by finding the remainder modulo the size of the table. However, the implementation of the `%` (modulo or remainder) operator in JAVA and other programming language standardly has the behaviour that `(-5) % 3` is not `1`, as it ought to be for our purposes, but `-2`. I cannot think of any application in which this would be the desired behaviour, but that's how it is. Consequently, you have to check for negative results and either add `capacity` or change the sign.

In my implementation, the long hash code of each entry is stored in a third array, called `code[ ]`, but special values of `code[i]` are used to indicate that position `i` is `EMPTY` or `DELETED`. If you do things this way, you must check each value returned by `hashCode()` to see if it accidentally happens to be equal to one of these special codes, and if so replace this value with an `OTHER` one.

Otherwise, one day (come the millennium) you will find that some particular key cannot be entered into the table: whenever you do so, it mysteriously disappears, being treated as an `EMPTY` or `DELETED` position. (Worse still, the inappropriate insertion of a spurious `EMPTY` position might make other entries disappear too!)

13.2.3 The private `locate` method

See Section 13.2.2 for the long and short hash codes. In fact, in Section 13.2.4 we shall see that it's slightly more convenient for the caller (`put()`, `get()`, `remove()`) to calculate the long hash code and pass it to `locate()` as a second argument.

There is no point in doing

```
if (code[i] == EMPTY) { return i; }
else
while (code[i] != EMPTY) { ... }
```

because you're merely evaluating the same test for `EMPTY` twice.

In the loop, you have to test

- `if (code[i] == EMPTY)` as numerical values and
- `if (skey.equals (key[i]))` as equality of Strings (or Objects),

although as the second test may be expensive, it is a good idea to find out first whether the saved long hash code is equal to the one we're looking for.

The table is effectively circular: if we reach `capacity` we have to return to `0`.

Some versions of this code used `while (i<capacity)` or the same thing with `for`, then reset to `0` and repeated the code with a second loop. This is no more efficient (the same test `(i<capacity)` is made), and is less safe because when you write code twice you have two opportunities to get it wrong.

There is some benefit in testing that we don't go round the loop again and again, but proper consideration of correctness would deal with that. Looping forever is as good a way of reporting a run-time error as is raising an exception; better, in fact, because infinite loops can't be silenced by code like

```
try { code; }
catch (Exception e) { }
```

It is the whole point of marking cells as `DELETED` that they must *not* be treated like `EMPTY` ones (Section 13.3.3).

Very little of the code needs to be changed to implement **quadratic rehash**, *i.e.* to visit positions i, $i+1$, $i+3$, $i+6$, $i+10$, $i+15$ instead. It is not necessary to calculate j^2 or $j(j+1)/2$: notice that the *jumps* are 1, 2, 3, 4, ... Beware, however, that quadratic rehash considers `(capacity-1)/2` positions, and then repeats the same path (backwards). Problem for mathematics students: why? Could cubic rehash avoid this problem? (For cubic rehash, we don't increment `jump` by 1 each time, but by an amount that is itself incremented.)

13.2.4 The public `insert` method

This method is an application of the private `locate()` method above, which returns either the position of the key if it already exists in the table, or the position where it can be inserted. These cases are distinguished by the `if` statements that follow.

You could instead repeat the code for `put()`, `get()` and `remove()`, but this is **unsafe** (Section 13.3.3).

There is no loop in this code!

As (in my implementation) the long hash code is saved in `code[i]`, it is more convenient for `put()` to calculate it and pass it as the second argument to `locate()`.

It is important to maintain the `contents` and `used` counts correctly (Section 13.3.1). There are different things to do in the three cases.

The key may already be present in the table. In this case we replace the old data with the new, returning the old data as our return-value.

Using the `used` count, we test whether the table has got too full, and if so call `rebuild()`.

13.2.5 The private `rebuild` method

Notice the way in which this is divided into two methods, one of which is then available for the constructor to make a new table when no valid table is already available.

The caller should be able to choose the (minimum) new size, and the actual size should be the next prime above this.

Beware that this is **not** a simple copy of one array to another: each old entry must be inserted in the new table in the same way as it would be using the `put()` method; this is why it's useful to save the 32-bit long hash code.

This method could be implemented by mimicking the code in `locate()` and `put()`, but it is much **safer** to call `put()` instead (Section 13.3.3).

The old `EMPTY` and `DELETED` entries must not be re-inserted.

The control parameters, in particular `contents`, `overflow`, `contents` and `used` must be set correctly (Section 13.3.1).

When using the `put()` method, the hash table must first be initialised correctly, as `newhash()` does. As this assigns to the `key`, `data` and `code` variables, these must be saved for later use in the loop.

It is *not* necessary to save the *contents* of these arrays, element-by-element. This is because the `key` and other variables are *references* to arrays, and the assignment inside `newhash()` changes these *references* to point to new arrays, leaving the contents of the old ones alone.

Nor is there any need to assign `old_key = null` at the end. As this is a *local* variable, its value is lost when `rebuild()` exits. As it is *private* to the class, this local variable is the only reference to the array. JAVA counts the references and, when it has exhausted the available memory, de-allocates the arrays and class instances that have zero reference counts. This process (**garbage collection**) is not straightforward: when an array or object is de-allocated, any references to other arrays and objects that it contains also disappear, so there are long chains of de-allocations that have to be made. Maintaining this is not so difficult, but there may also be *loops* of references, which can only be detected by scanning *all* of the objects that the program has allocated. JAVA worries about this for you: usually you don't have to do so, but sometimes this can go wrong!

13.3 Correctness

In the earlier parts of the course you have learnt how to prove correctness of algorithms such as linear search and array shifting by using **loop invariants**.

However, the only **loops** in `Hash_Table.java` are

- `locate()`: a linear search along a "path" in the table (Section 13.2.3);
- `newhash()` and `clear()`: make all the cells `EMPTY`;
- `next_prime()`
- `rebuild()`: a linear copy of one array to another, though not into the same positions;

plus a few inessential things to make the "bells and whistles" for my implementation.

Therefore, it is not in the use of loops where the correctness issues lie.

When using a class in object-oriented programming, there is another, bigger, loop that you can't see in the code:

```
Hash_Table login_to_human = new Hash_Table ();
while ( ! bored ) {
     login_to_human.call_some_method ( ... );
}
```

So the invariant — which we now call a **data invariant** or **object invariant** or **class invariant** — is a statement of

- what we *may assume* about the state of the data when we *enter* a method, and
- what we *must ensure* about the state of the data when we *leave* it.

So we have to prove something about the **body** of the method, in exactly the same way as we did about the body of a loop.

13.3.1 Counting

One part — the simplest part — of the data invariant concerns the variables `contents` and `used`. These count the number of cells in each of the three states: `EMPTY`, `DELETED` and `VALID`.

So they must be updated every time one of the cells changes state, and also when the table is rebuilt.

Otherwise, we will not know when the table has got too full and needs to be rebuilt, with the result that `locate()` will search the entire table instead of two or three entries, or maybe loop forever. Alternatively, we might get into a loop rebuilding the table bigger and bigger until we run out of memory.

13.3.2 Partial functional relations

Neither of the previous two subsections actually addresses the issue of what the hash table is *for*, and therefore what it means for it to be correct or otherwise.

Remember that the entries in the table are not sorted. All we do is put them in, look them up and take them out.

There are three ways that we can look at this:

- the sequence of `put (key, data)` and `remove (key)` method calls that were used to create the table,
- the finite set of `(key, data)` pairs that the table contains, and
- the `key`$\to$`data` response of the `get (key)` method.

The last two directly define **partial functional** or **single-valued relations**: *if you consider the same key twice, you get the same data back.*

Considering the `remove (key)` method calls, and the `put (key, data)` calls that overwrite the `data` already associated with the `key`, the first way of looking at the hash table is like a sequence of **assignment** statements. The overall effect of this (when we consider just the current state, rather than the assignments that got us there) is that this view of the table is also a partial functional relation.

To show that the implementation of the hash table is correct, we must therefore show that these three views actually define the same partial functional relation.

Part of this is easy to check: every `put` or `remove` method call must actually enter or delete an entry in the table, unless it can see that this has already been done.

However, this is not the whole story. We must also ensure that, when one of the methods accesses a key that is contained in a particular cell of the table, this is the *only* cell where this key is to be found.

This concrete property is in fact just the same as the abstract mathematical property of being a partial functional relation, that you should have learned in the *Discrete Structures* course.

In fact this is not quite right. Mathematically, we say that R is a partial functional relation if

$$\forall x.\forall y_1, y_2.xRy_1 \wedge xRy_2 \Rightarrow y_1 = y_2.$$

If we were actually doing this in the JAVA implementation, there would somewhere be code like

```
if ( data1.equals (data2) ) { ... }
```

whereas in fact we test equality of *keys*. The logical property that our table satisfies is actually

$$\forall ij.0 \le i \langle j \langle \texttt{capacity} \Rightarrow \texttt{key}[i] \ne \texttt{key}[j],$$

but I shall leave it to you to prove that this implies that the three binary relations defined above are in fact functional in the mathematical sense.

13.3.3 Repeatability of `locate`

The considerations about partial functional relations in the previous subsection come down to proving that the `locate()` method correctly solves the **search problem**:

> to determine whether *or not* the search key occurs in the table, and if so where.

The linear search algorithm solved this problem in a very crude way, by possibly considering *every* cell in the table. Binary search was able to do it much more quickly, only considering $\log n$ cells, by relying on the assumption that the table was sorted.

A hash table is not sorted, and we certainly don't want to consider every single cell. Correctness must therefore depend on guaranteeing the "efficient filing" property that,

> *if* the key occurs anywhere, *then* it must be *here*,

so if it's not here, it's not present at all.

We achieve this for `get()` by knowing where `put()` would have left the entry. So these methods must consider *the same sequence of positions*, which is most **safely** achieved by making them call the same code, a private `locate()` method. This means that

> `locate()` must be repeatable.

Recall that we call a `hashCode()` method to turn the key (usually a `String`) into a long hash code, and then the `%` operator to reduce this to the short hash code, which is the entry point to the table.

Really, it is better to consider `hashCode()` and `locate()` together as defining, not a single cell-index in the table, but a **path** (sequence of cells) in the table. In principle, this path consists of a large number of positions, to be considered in a particular order. The single loop within this method runs through this list of positions.

This path may be defined in various ways:

- **linear rehash** visits positions i, $i+1$, $i+2$, $i+3$, ..., modulo `capacity`, where i is the short hash code;
- **quadratic rehash** visits i, $i+1$, $i+3$, $i+6$, $i+10$, ...;
- maybe these intervals, like the starting-point i, are also obtained from the long hash code, for example by taking its remainder modulo `capacity-2`. Richard Bornat says (9 April 1999):

> I have heard of it. It's the sort of thing that was tried early on in history; my recollection is that it doesn't work better than (say) quadratic rehash, and that Knuth's analysis of linear rehash means that we don't have to look for better rehash schemes. But Sedgewick's book may have references.

In practice, of course, we only want to consider up to about three positions. However, the sequence of positions that are *actually* visited by a latter call to `get()` may be longer than the sequence visited by the call to `put()` that originally inserted the entry, because an intervening insertion may have changed the terminating `EMPTY` cell into a valid one. This is why, in proving correctness, we must consider the entire (potential) path.

The loop in `locate()` breaks when it encounters either

- the required key, or
- an `EMPTY` cell.

The second case implies that the key is absent, so long as every attempt to look up this key follows exactly the same path, and the insertion is made in the first available `EMPTY` cell. When the later `get()` finds an `EMPTY` cell without first finding the key, it knows that there has been no earlier `put()` — because that would have used this `EMPTY` cell, or an earlier one in the path that has since been used for another key.

What happens when we delete one of the other keys that is considered along this path? If this cell were marked `EMPTY`, the loop searching for our key would break prematurely. Therefore, this other key must be marked in some other way (`DELETED`) that does not cause the loop to break at that point.

We have not considered the `rebuild()` method in this discussion. However, this works by

- taking all of the valid `(key, data)` pairs in the old table, and
- re-inserting them in the new one.

Therefore the old `(key, data)`-relation (defined by the old table) coincides with the new one, as defined by its `put()` method, and hence (by the above argument) with that defined by its table and its `get()` method.

Chapter 14

Minimax for games

Minimax is an excellent example of the difference between

- **long, complicated but naive coding**, in this case especially for the method that finds the list of valid moves for a particular player from a particular position, and
- **short, subtle algorithms**, in this case alpha-beta pruning.

For this option you should do the following:

- write the minimax algorithm
- adapt your tic tac toe exercise, adding the methods that minimax needs,
  - list of valid moves for this player from this position,
  - effect (new position) of this move for this player from this position,

  and changing the main loop to call minimax for one of the players instead of prompting the user
- adapt a more complicated game (such as Connect Four) in the same way; for this you will need to cut off the game tree at a certain depth, and calling a "crude evaluation", which is another game-specific method;
- speed up the game with alpha-beta pruning.

See also

- Samin Ishtiaq's model answer to the *Introduction to Programming* exercise on Tictactoe.
- Sections 7.7 and 10.2 of Weiss's book,
- A. N. Walker's game theory notes, including alpha-beta pruning at Nottingham University.
- The bible of mathematical games is *Winning Ways* by E. R. Berlekamp, J. H. Conway and R. K. Guy (Academic Press, 1982); two large volumes, and rather expensive, but absolutely fascinating. Only a tiny fraction of the material is relevant to this module, but you will want to read it anyway. [This book was by 1997 out of stock at the publishers, though not out of print, so is becoming harder to find.] (That's what Walker says about it; minimax does not, I think, occur anywhere in this book, so it's not really relevant to this coursework option at all.)

Received wisdom has it that the "crude evaluation" of a position should be a very simple function. "How many different legal moves do I have?" is thought to be good enough. Complicated functions, with elaborate ways of scoring each square on a board, often turn out to be based on a mis-conception of the game. Any such function that depends on enumerating the possible moves is likely to be *worse* at winning the game than simply going one level further down the game tree with a simpler evaluation function.

Alpha-Beta Pruning

Alpha-beta pruning does not play any better moves: it just plays the same moves more quickly. Apparently, if used properly, it considers the square root of the number of positions than minimax alone would consider, so, since minimax is exponential in the depth, you can go twice as far down the game tree in the same time — and *thereby* play better moves.

The addition of alpa-beta pruning to an exiting program that uses minimax does not involve very much extra code. You just need

- two extra arguments (traditionally "alpha" and "beta", but "floor" and "ceiling" are more informative names) to the minimax method,
- an adjustment to the initialisation of the maximum or minimum so far before the beginning of the loop,
- appropriate values of the two extra arguments in the recursive calls, and
- a test that breaks the loop early.

The descriptions of alpha-beta pruning by Weiss and Walker are game-theoretic and not, in my view, very clear from the point of view of the logical analysis of algorithms.

When the higher level calls

```
score = minimax (position, depth, floor, ceiling);
```

it's saying that, "whatever value you give me for the score, if it's bigger than ceiling I shall treat it like plus infinity, and if it's smaller than floor I'll treat it like minus infinity".

This means that (at our level) the "maximum so far" of the list is initialised to floor instead of minus infinity, and the loop breaks as soon as one of the recursive calls returns a value above the ceiling.

The maximum so far and the ceiling provide the floor and ceiling for the recursive call, except that, as they are for the other player, their roles are interchanged.

Think about this: the explanation that I have just given follows from the properties of the $\max(a, b)$ function: if $b \le a$, so max chooses a instead of b, then it doesn't matter how small b is — it might as well be minus infinity.

To take advantage of alpha-beta pruning, the method that lists the available legal moves from any given position must put the best moves (on the basis of naive principles) first. For example, in chess or draughts, you would consider taking an opponent's piece before any other move, and would prefer to queen an advanced pawn instead of moving one on the back rank. Of course, such a move may be a trap, but it's minimax's job to find that out!

Chapter 15

May 2001 Exam

The **rubric** on the front of the exam paper reads:

- **DCS/127 INTRODUCTION TO ALGORITHMS**
- Time allowed: **TWO AND A HALF HOURS**.
- Answer **FOUR** questions. There are **eight** questions altogether; they carry equal weight (30 marks). Question 8 is an **essay** that itself has three options, but you may only choose *one* of these.
- Calculators are **NOT** allowed.

15.1 Complexity

This option is made up of two groups of questions; you must answer both.

First group

The Hewlett–Packard HP9810A programmable desk calculator cost $2960 in 1971 and could perform about 100 instructions per second.

Below is a table showing the sizes of the largest problems that it could solve in **one minute**.

For less than this price, you can now buy a computer that is $\mathbf{10^6 \approx 2^{20}}$ **times as fast**.

State the worst-case **time complexity** of each algorithm in the form "$O(\cdots)$" and then estimate the **size of the largest problem** that the **modern** computer could solve in the same amount of time (one minute).

All numbers must be expressed in standard decimal ("Arabic") notation: no marks will be given for formulae or exponential notation.

	algorithm	largest problem size in 1971	marks
(a)	Multiplying two $n \times n$ matrices by the naïve algorithm.	$n = 6$	[3]
(b)	Heap sort on an array of n integers.	$n = 32$	[5]
(c)	Selection sort on an array of n integers.	$n = 32$	[3]
(d)	Looking for a "true" line in the truth table for a propositional formula involving ($\wedge$, $\vee$, $\neg$, $\rightarrow$, $\top$, $\bot$ and) n variables.	$n = 6$	[5]
(e)	Linear search on an array of n integers.	$n = 200$	[2]

You are advised to write down the mathematical equation that needs to be solved before you do any arithmetic, and set out your rough calculations as clearly as possible. Alternatively, you should be able to guess the answers on the basis of your own experience of running test programs in the lab sessions, without doing any arithmetic. It is enough to give **order-of-magnitude** answers, so long as they correctly indicate the **relative** values.

Second group

For each of the three fragments of code, what is its worst-case time complexity, in the form "$O(\cdots)$".

(f) Given an array `A` of appropriate size, what is the complexity in terms of `n`?

```
int i=n, x=0;
while (i>0) {
   i--;
   x += A[i][n-i];
}
```

[2 marks]

(g) Given three 2-dimensional arrays `A`, `B`, `C` of appropriate sizes, what is the complexity in terms of `n`, `m` and `p`?

```
for (int i=0; i<n; i++) {
   for (int k=0; k<p; k++) {
       int x=0;
       for (int j=0; j<m; j++) {
           x += A[i][j] * B[j][k];
       }
       C[i][k] = x;
   }
}
```

[2 marks]

(h) Given `int x` and an array of appropriate size, what is the complexity in terms of `n`?

```
int p = -1;
int q = n;
while (p+1 < q) {
   int m = (p+q)/2;
   if     (A[m] <  x)   { p=m; }
   else /* A[m] >= x */ { q=m; }
}
```

[2 marks]

(i) Given an array of appropriate size, what is the complexity in terms of `n`?

```
int m = A[0];
for (int i=1; i<n; i++) { if (A[i] > m) m = A[i]; }
int k = 0;
for (int i=0; i<n; i++) { if (A[i] == m) k++; }
```

[2 marks]

(j) Given an array of appropriate size, what is the complexity in terms of `n`?

```
for (int i = n-1; i > 0; i--) {
  int x = a[i];
  a[i]  = a[0];
  int p = 0;
  while (true) {
     int c  = 2*p+1;
     if (c >= i) break;
     if (c != i-1 && a[c] < a[c+1]) c++;
     if (x >= a[c]) break;
     a[p]   = a[c];
     p = c;
  }
  a[p] = x;
}
```

[4 marks]

15.2 Specifications

(a) What are meant by the terms *pre-condition* and *post-condition*? [4 marks]

(b) Explain how these two conditions provide a *right* (or *guarantee*) for the user of a method and impose a *duty* on its programmer, or *vice versa*. [3 marks]

(c) *Very briefly*, state the *post-condition* of *one* method that you have implemented during this course. [1 mark]

(d) Name one method that you have implemented for which you did not need to state any *pre-condition*. [1 mark]

(e) Does a *specification* determine the way in which the method is to be *implemented*? Give an example to explain your answer: no marks will be given for “yes” or “no” alone. [3 marks]

(f) What is the precondition for the following method?

```
void shift (int array[ ], int src, int dest, int len) {
  if (src > dest) {
    int j=0;
    while (j < len) {
        array [dest + j] = array [src + j];
        j++;
  }   }
   else {
```



```
        int j=len;
        while (j > 0) {
            j--;
            array [dest + j] = array [src + j];
    }   }   }
```

[3 marks]

(g) Can this condition be verified in an acceptable number of operations? [1 mark]

(h) What would happen in JAVA if this method were to proceed with its execution when the pre-condition was **not** satisfied? What would happen in C? Does this matter? [3 marks]

(i) What is the precondition for **binary search**? [1 mark]

(j) Can this condition be verified in an acceptable number of operations? [2 marks]

(k) What would happen if this method were to proceed with its execution when the pre-condition was not satisfied? Does this matter? [2 marks]

(l) You were told to implement *insertion sort* using a separate *search* method. Your insertion sort method is therefore a *user* of the search method, in the sense of part (b) above. What is the specification of the **search** method, if it is to be used for this purpose? [2 marks]

(m) How does insertion sort make use of this property to conform to its *own* specification? [4 marks]

15.3 Merge sort

(a) What is the **idea** of merge sort? Make sure that your description applies unambiguously to merge sort and not to quick sort. [4 marks]

(b) Suppose that recursive merge sort is called with an array of length 16. Draw a tree to show how many recursive calls are made, labelling the root “$M(16)$” and each of the other nodes “$M(size)$” in the same way to show the sizes of the arrays that they are each required to sort. [3 marks]

(c) In general, how many method calls does recursive merge sort make when sorting an array of length n, assuming that $n = 2^k$? [2 marks]

(d) Each of the recursive calls to mergesort has to merge two arrays into one, which involves a certain number of assignment operations. How many such assignments are made altogether? Annotate your tree with the total number that are made at each level. [2 marks]

(e) In general, how many such assignments does recursive merge sort make when sorting an array of length n, assuming that $n = 2^k$? [2 marks]

(f) It is possible to make a *hybrid* of insertion sort and merge sort that sorts large arrays faster than pure merge sort itself. You are *not* required to write out the code. Instead, draw a tree similar to that in (b) and explain in one English sentence how the hybrid differs from pure merge sort. Annotate the nodes of your tree with “$I(s)$” where you want to call insertion sort on an array of size s. Choose the size of the original array appropriately (say, 128 or 256) to illustrate the design decision that you are making. [6 marks]

(g) How many method calls does your hybrid algorithm make for an array of size $n = 2^k$? [2 marks]

(h) How many assignments does insertion sort make, on average, when sorting an array of length s? [1 mark]

(i) How many assignments does your hybrid algorithm make for an array of size $n = 2^k$? [4 marks]

(j) How long do the pure and hybrid mergesort methods take to sort an array of length n, *as a function of* n? It is the *form* of the function that is required: you are not required to give numerical time values for the coefficients. Which part of this mathematical expression is different for the hybrid? [4 marks]

15.4 Quick sort

(a) Given an array of numbers, what is meant by their **mean**, **mode**, **average** and **median** values? [4 marks]

(b) One of these values (mean, mode, average or median) is **very easy** to calculate. Which one? Write (the significant part of) the JAVA code to do it. [2 marks]

(c) In a single English sentence, what is the **idea** of **quick sort**? Make sure that your description applies unambiguously to quick sort and not to merge sort. [3 marks]

(d) What is the **best**-case time complexity of quick sort? [1 mark]

(e) What is meant by the **pivot** in quick sort? [2 marks]

(f) Which of the values in (a) (mean, mode, average or median) would you *ideally* use for the pivot? [1 mark]

(g) What would the depth of the recursion be if this ideal choice were used to sort an array of length $n = 2^k$? Explain why this is. [4 marks]

(h) Explain how this ideal choice would result in the time complexity that you have stated in (d). [2 marks]

(i) What is the **worst**-case time complexity of quick sort? [1 mark]

(j) Describe briefly what quick sort does with *sorted* data if the *first* value is used for the pivot. [1 mark]

(k) Why does this result in worst-case complexity? [3 marks]

(l) What is meant by the **median of three** method of choosing the pivot? [2 marks]

(m) On what kind of almost sorted data does this method of choosing the pivot still result in worst-case time complexity? Describe what happens. [4 marks]

15.5 Programming Split

This question guides you through the design of a method

```
int parity_split (int [ ] array)
```

to rearrange the values in `array` so that

- the odd numbers come first (in any order, not necessarily sorted),
- followed by the even numbers (also any order),
- returning the position of the first even number.

So, for example,

`{1,0,8,7,5,3,9,6,2,7,4}` might become `{1,7,9,7,5,3,8,6,2,0,4}`

and the value 7 (the new position of 8) is returned.

This is similar to the `split` method used in quicksort.

You must not allocate a new array inside your method.

In the typical situation during the execution of the loop inside your method, there are some odd numbers on the left, some even numbers on the right and a mixture of odd and even numbers that have not yet been considered in the middle.

(a) Draw a diagram to illustrate the typical situation as just described, and the way in which program variables will be used as pointers into the array during the execution of the loop. [4 marks]

(b) What are the initial values of the pointers? [2 marks]

(c) Under what condition can you simply increment the left-hand pointer, without altering the values in the array? [2 marks]

(d) Write the JAVA code to do this as much as possible. [2 marks]

(e) Similarly, write the JAVA code to move the right-hand pointer as far left as possible. [2 marks]

(f) After you have moved the pointers together like this, what can you deduce about the values near the pointers in the diagram? Describe the situation in English or JAVA, and also indicate it on your diagram in (a). [3 marks]

(g) What operation can you now perform on the values in the array, after which you may advance both pointers? [3 marks]

(h) When does the algorithm terminate, and what value does it return? Write it in JAVA as `if (...)  return ...;` [2 marks]

(h) Now write the complete method. [10 marks]

15.6 Heap sort

(a) Considering it as a **tree**, explain what is meant by a **heap**. Draw an example. [4 marks]

(b) How can a heap be represented as an array? In particular, what array represents your example? [2 marks]

(c) How do you calculate the positions of the **parent** and **children** of a particular node in this representation? [3 marks]

(d) Describe how to **insert** a new value in the heap. Draw an example. [4 marks]

(e) What significance does position 0 have in the tree representation, and what property does the value there have? [2 marks]

(f) Describe how to **delete** the value at position 0 from the heap. Draw an example. [4 marks]

(g) If the heap has n elements, what is the (worst case) time complexity for insertion and for deletion? [1 mark]

(h) Suppose you have a random array of n elements that you want to rearrange into a heap. One way to do this is by *inserting* the elements (using (e)) one by one. What is the (worst case) time complexity of doing this? [2 marks]

(i) Describe a faster way to **heapify** an array, and state its time complexity. [5 marks]

(j) Explain how to use a heap to sort an array, and state its time complexity. [3 marks]

15.7 Correctness of binary search

This question guides you through the proof of correctness of the following binary search method:

```
int search (int[] array, int seek) {
   int bot = 0;
   int top = size;
   while (top > bot) {
      int mid = (top + bot - 1)/2;
      if     (array[mid] <= seek)   bot = mid + 1;
      else /* array[mid] >  seek */ top = mid;
   }
   return ??;
}
```

Notice that the return value is missing.

(a) After the assignment `bot=mid+1` has been made, what can you say about `array[bot-1]`? [2 marks]

(b) Suppose instead that the assignment `top=mid` has just been executed. What can you say about `array[top]`? [2 marks]

(c) What do (a) and (b) tell you about *other* values in the array, and why? [2 marks]

(d) Draw a diagram to illustrate these statements, indicating the positions of the `bot` and `top` pointers, and the properties of the segments that they delimit. [5 marks]

(e) Are the two logical statements in (a) and (b) valid just before the loop is entered? If so, why? If not, what similar statements are in fact valid, and why? [2 marks]

(f) Suppose that the logical statement in (a) happened to be valid just inside the beginning of the body of the loop, but that the `else` branch (including assignment `top=mid`) is executed. Is (a) still valid at the end of the body of the loop? Why? [2 marks]

(g) What may you therefore conclude is true on exit from the loop, assuming that it does terminate? [1 mark]

(h) Suppose that the assignment `bot=mid+1` has just been executed. Show that the variable `bot` is increased by at least 1. Be careful to distinguish between the original and new values of `bot`. [3 marks]

(i) What logical statement is true on exit from the loop, simply by virtue of having just come out of the loop? [2 marks]

(j) What can you deduce about the relationship among `seek`, `array[top-1]` and `array[top]` on exit from the loop? [3 marks]

(k) Deduce the value that the method must return, if it is required to point to some occurrence of `seek` if there is one. [2 marks]

(l) Is this the left- or right-most occurrence, or just some occurrence? [1 mark]

(m) What does the result returned by the method mean in the situation where the value `seek` does not occur in the array? [1 mark]

(n) What would happen if you changed the loop condition to

```
while (top >= bot)
```

and ran the program? Explain this in terms of the calculation in (h). [2 marks]

15.8 Essay

Write an essay about *one* of the following topics. [30 marks]

Your answer should include a selection of the most important parts of the code: do not include JAVA declarations unless they illustrate some aspect of the algorithm. Explain the main points in proving correctness and state the main facts about complexity.

- Sorting a list of names that fits on disk but not in RAM.
- Hash tables for creating, accessing and updating dictionaries.
- Finding a path through a network.

Chapter 16

Further reading

None of these books is specifically *recommended.*

Knuth's book is the Great Grand-daddy of books on algorithms, but much of its emphasis is on intricate calculations of precisely how many instructions are executed.

Sedgewick's and Weiss's books are more modern and more student-friendly, but owe much to Knuth.

All of these books make heavy use of Calculus. The reasons for this are more a matter of the culture of American universities than what is actually relevant to the performance of these algorithms when you program them.

On the other hand, what is *missing* from the American tradition is *proof of correctness* of the algorithms.

The book by Broda *et al.*, which was based on the courses at Imperial College that correspond to our Logic and Algorithms courses, heavily influenced my way of teaching this course.

The book by Baase is actually the most similar to my course, but it is seriously lacking in precision, both in correctness and complexity analysis.

I am not telling you to **buy** any of these books, but if you do choose to buy any of them, you should consider the balance between your short-term need to learn the material in this course, and the long-term usefulness of the book in your programming career. Broda is better as a teaching book, Weiss as a programmer's book. You will find yourself looking at Weiss's book long into the future, but you will probably never use Broda's book again.

Richard Bornat recommended Weiss's book when he taught this course, but I found that his "look — no hands!" attitude to programming was a serious obstacle to my teaching students what I wanted them to learn.

The **QA76** numbers tell you where to find the books in QMW Library.

Just going to the appropriate section of the library is often a quicker way of finding things out than using bibliographies and catalogues.

If the Web interface for the library catalogue had been designed according to the straightforward and well thought out original principles of the Web, it would have been possible for me to do the search for you and bookmark the pages for the books themselves. Then you would have been able to find out directly whether there are any copies currently available in the library. Unfortunately, the designer of this Web interface was "clever" and used JavaScript, so this simple and obvious application of the library catalogue is not possible.

The links behind the authors' names take you via Hypatia to other information about these people, such as their other publications and how to contact them.

My own book is not relevant to this course: it is listed for reasons of vanity.

Baase, Sara, van Gelder, Allen, **Computer Algorithms: Introduction to Design and Analysis**, Addison-Wesley, 1999, QA 76.9.

Broda, Krysia, Eisenbach, Susan, Khoshnevisan, Hessam, Vickers, Steven, **Reasoned Programming**, Prentice Hall, International Series in Computer Science, 1994, 0–13–098831–6.

Knuth, Donald E., **The Art of Computer Programming: Volume III, Sorting and Searching**, Addison-Wesley, 1973, 0–201–89685–0, QA76.6.K64.

Knuth, Donald E., **Literate Programming**, In *Literate Programming*, Stanford University Center for the Study of Language and Information, Lecture Notes, **27**, 1991, 0–937073–80–6 (paperback), 0–937073–81–4 (hardcover), QA76.6 .K644 1991.

Rowe, Glenn, **An Introduction to Data Structures and Algorithms with Java**, Prentice-Hall, 1997, 0–13–857749–8, QA76.73.

Sedgewick, Robert, **Algorithms**, Addison-Wesley Series in Computer Sience, 1988, 0–201–06673–4, QA76.6.

Sedgewick, Robert, **Algorithms in Java, Parts 1–4**, Addison-Wesley, 1998, 0–201–36120–5, QA76.73.

Taylor, Paul, **Practical Foundations of Mathematics**, Cambridge University Press, Cambridge Studies in Advanced Mathematics, 1999, 0–521–63107–6.

Weiss, Mark Allen, **Data Structures and Problem Solving Using Java**, Addison-Wesley, 1998, 0–201–54991–3, QA76.73.