Comparing Algorithms: Complexity Theory

An **algorithm** may be defined in simple terms as a finite sequence of instructions that solves a problem.

Example: Shampoo Algorithm (Dirty Hair Problem!)

1. Wet hair
2. Apply shampoo
3. Rinse hair
4. Repeat steps 1,2,3 twice
5. Stop

Here we have a finite number of instructions. They are in a specific sequence i.e. the order is important. You cannot apply the shampoo until the hair is wet.

The term algorithm is believed to derive from the name of a 9th century Arabian mathematician Al-Khowarizmi.

A computer program is simply an implementation of an algorithm on a computer.

Frequently, there may be a number of algorithms to solve the same problem. The question arises as to which one should be used. Is one algorithm better than another?

This is a non-trivial question and leads to the **analysis of algorithms** and the means by which they can be compared.

This area of study is sometimes called **Complexity Theory** in Computer Science.

One way to compare algorithms is to compare their performance in terms of how quickly they solve the problem.

Some algorithms arrive at a solution faster then others. We often choose the fastest algorithm when solving a problem.

Think of how you use a telephone book to look up a number. What other way could you do it ?

Which way is faster ? Which way is simpler ?

Another way of comparing algorithms is to look at the amount of space (memory) they require.

Some algorithms require large amounts of space but arrive at a solution quickly. Others require small amounts of space but arrive at a solution less quickly.

You are often presented with a **space/time trade-off**.

You will find time and time again in comparing algorithms and computing systems (and in many aspects of life!) that you have a trade-off situation. Improvement in one aspect leads to degradation in another.

Examples

Fast algorithm may require a lot of space
Slow algorithm may require small amount of space

Expensive car may consume less fuel
Cheap car may consume much fuel

Each trade-off situation must be evaluated separately. You cannot tell in advance which solution is appropriate until you know the details of the situation. Thus, in advance you cannot specify the ‘best’ solution.

However, you can specify in advance, the relative merits of each solution, so that in any particular situation you can choose the solution that best suits the situation.

If we have a number of algorithms, we can specify their relative performance and use this when choosing one for a particular problem in a particular environment.

We will only look at one measure of performance here, the relative amount of time an algorithm takes to solve a problem.

This is called the **time complexity of an algorithm**.

When we speak of the time complexity we are **not** interested in **absolute** times, i.e. how many seconds it takes to solve a particular problem.

The actual absolute time taken to solve a problem depends on a number of factors:

- how fast the computer is
- RAM capacity of the computer
- OS the computer uses
- quality of code generated by the compiler
- etc

If you change any of these, then the absolute time changes.

Thus **absolute time is not useful as a measure of an algorithm's performance**.

One reason for computing complexity is to compare algorithms. We need a measure which will allow us compare two algorithms.

Each algorithm consists of a finite sequence of instructions. The more instructions in an algorithm the longer it will take to execute.
One way to compare algorithms would be to **count** the **instructions** that the algorithm requires to solve a problem.

The **number of instructions will vary depending on the input**. A payroll program for 100 people will repeat more instructions than one for 10 people.

Thus we compute the **number of instructions as a function of the input size**.

But do we need to count all instructions?

When we look at many algorithms, we see that they consist of a **basic loop** which is **executed for each item** of the input.

As the **input size grows**, the number of instructions carried out in the loop far **exceeds** those before and after the loop.

To simplify our calculations we do **not** count instructions **outside** the basic loop.

Do we need to count all instructions inside the loop?

Again when we study algorithms, we find that the number of instructions inside the loop for any two algorithms to solve a specific problem does not vary widely !

But **what does differ** is the **number of times the loop is executed**.

So, essentially what **we need to count is the number of times the loop is executed.**

In a searching algorithm, the loop compares the current element of a list with the item to be found. In this case it is the **number of comparisons** that we count in order to compare one searching algorithm with another.

But obviously the number of comparisons depends on the size of the list.

Thus we express the number of comparisons in terms the list length.

Given a list of **length n** for a **sequential** (linear) **search algorithm** we find that the **number of comparisons** is **n**, in the case *where the item to be found is not in the list*.

We use the notation **O(n)** to describe this time complexity. It reads "**Order of n**". This is called the **Big-O notation.**

For a list of length 1000, we would make 1000 comparisons.

If we double the list length we double the number of comparisons.

But sometimes (**on average**) we are likely to find the item in the list. Sometimes near the beginning, sometimes near the end.

On average, we will have to search **half of the list**. So, on average we would make **O(n / 2)** comparisons.

This is called the **average case complexity**.

We have seen, that if the item is not in the list we make n comparisons, this is called the **worst case complexity**.

Note that we still regard the complexity of a linear search algorithm as O (n) despite the fact that on average we find the item in n/2 comparisons.

This is because as n grows very large, the difference between n and n/2 may will become less significant i.e. they are of the same order (e.g. 2 billion versus 1 billion).

When comparing algorithms it is important to know both the average and worst case complexity.

Faster Search Algorithm: Binary Search

But there is another algorithm for searching a list called the **binary search algorithm** where the number of comparisons is substantially less. It is **O(log n**), using base-2 logs.

Thus if the list length is 1000, using binary search we will find the item in at worst 10 comparisons (as opposed to 1000 for linear search).

But consider a list length of **1 billion**. With a binary search algorithm we will find an item with **at worst 30 comparisons**.

It is obvious that the binary search algorithm is far superior from a time complexity viewpoint to a linear search algorithm.

We can see for any two algorithms that an algorithm of O(log n) is far superior to one of O(n) as the following table illustrates:

Iterations for O(n) Algorithm	**Iterations for O(log n) Algorithm**
100	7
1000	10
1 million	20
1 billion	30
1 billion billions	60

Binary Search Algorithm

There is a major restriction with the binary search algorithm. It **can only be used** when the list is in a particular order i.e. **when the list is sorted**.

It is the method we use to search for a number in phone directory. We open the directory in the middle, compare the entry there with what we are looking for, and depending on the outcome we know if the number is in the lower half of the directory, the upper half or on the page we have opened. We thus eliminate half of the directory with 1 comparison!!

We repeat this procedure until we find the number or know it is not present. With each comparison we reduce the list length (number of pages of directory) by a factor of 2.

Thus with 10,000 pages after 1 comparison we have 5,000 left to search, then 2500, then 1250, then 625 etc.

The maximum number of comparison is log(10,000) (to base 2) i.e. the number of times we can divide 2 into 10,000.

We can only use this method, because the list is sorted.

Consider a list **A** of **n** integers, sorted in increasing order. The binary search algorithm to search **A** for an element **e** takes the form:

```
while not finished and found == false do
begin
    compute middle of list
    if e == middle item then found = true
    else if e < middle item search lower half
    else if e > middle item search upper half
end
```

The **binary search (chop) algorithm** is a standard algorithm for searching sorted lists. It is described in most textbooks dealing with algorithms.

There are numerous standard algorithms in Computer Science for performing a variety of tasks. Before designing an algorithm from scratch, it is worth checking whether the problem (sub-problem) is a general one and if so are there standard algorithms available. This obviously saves you a lot of time and effort as there is little point in reinventing the wheel!

One problem for which there exists a host of standard algorithms is that of sorting. The sorting problem is: given a list of items in any order, design an algorithm to arrange the list in a particular order (ascending or descending) i.e. sort the list.

Before we can use the binary search algorithm, the list must be sorted. Occasionally, the list of interest in a problem may already be sorted, but usually you will have to sort the list yourself.

Searching and sorting algorithms are important because they arise so frequently in practical programming applications. A significant percentage of **all** computing time is spent searching and sorting lists.

Bubble Sort

There are a surprising number of ways to sort a list. We will only consider one method called the **bubble sort algorithm**. We will sort the list in ascending order i.e. the largest values are at the end.

Consider the list L:

10, 34, 2, 16, 23, 8, 12

We wish to sort L so that it becomes:

2, 8, 10, 12, 16, 23, 34

In order to sort the list, we have to swap (exchange) values, so that the larger values are swapped with the smaller ones, the larger values being moved to the end of the list and the smaller values being moved to the start of the list.

We want to move the larger values to the end of the list. Remember we can only compare two values at any point.

In bubble sorting, we begin at the start of the list. We compare the first two elements: L[0] and L[1]. If the lower index element (L[0]) is bigger we swap it, thus moving the larger element towards the end of the list. We then compare L[1] with L[2] and we swap them if L[1] > L [2]. Next we compare L[2] with L[3] and swap if necessary.

We continue this process of comparing pairs of elements and swapping if necessary until we come to the end of the list.

We can now be sure of one thing. The largest element of the list is at the end. The rest of the list is still unsorted. We have passed through the list once.

Initially list L had the form

10, 34, 2, 16, 23, 8, 12

After first pass of bubble sort:

10, 2, 16, 23, 8, 12, 34

We now begin all over again. We compare L[0] with L[1] swapping if L[0] > L[1] and so on as before, but this time we have only to proceed to the second last item in the list, since the last item is the largest. When we have finished, we now have the second largest item in the second last position. We have passed through the list twice.

List L now has the form:

2, 10, 16, 8, 12, 23, 34

We start all over again, this time stopping at the second last item. On this pass through the list, we get the third largest item in position.

List L after third pass

2, 10, 8, 12, 16, 23, 34

The algorithm continues, passing through the list, once for every item in the list, but the distance passed getting smaller by one each time.

The algorithm finishes when we have passed through the list for the **nth** time for a list with n items.

After the final pass, L has the form:

2, 8, 10, 12, 16, 23, 34

L is now sorted. Bubble sort is named because if you trace its action, the larger elements gradually move towards the end of the list, just like bubbles rising in a glass of water !

Exercise: Apply bubble sort to the list 2,4,15,6,7,9,10

Do you notice anything inefficient about it ?

Complexity of Bubble Sort

For a list of n items we pass through the list n times, thus giving a complexity of $O(n^2)$.

We have now seen the complexity of three well-known algorithms:

- linear search $O(n)$
- binary search $O(\log n)$
- bubble sort $O(n^2)$

The complexity of list processing algorithms will **typically** be at least $O(n)$, since nearly always we have to make at least 1 pass through the list.

Binary searching is somewhat of an exception. It takes advantage of the fact that a list is sorted and thus does not have to make a pass through the list.

But what if the list is unsorted, as is usually the case. Then for searching, we have two choices. We can use the linear search algorithm which works with unsorted and sorted lists. Alternatively, we can first sort the list using a sorting

algorithm and then use the binary search algorithm. However, we now have the overhead of sorting and unless we intend searching the list a large number of times the linear search algorithm would be superior than the combination of sorting followed by a binary search.

If we intend to search the list many times, then the overhead of sorting the list once, would be repaid by the greater efficiency of the binary search algorithm.

It is important to note that algorithms of O(n) are in fact very efficient algorithms especially compared to say a bubble sort with O(n^2) complexity.

Infeasible Algorithms

There are many algorithms whose complexity is such that for even small values of n, they cannot be feasibly used.

Algorithms with complexity O(**n!**) (factorial n) or O($\mathbf{2^n}$) (exponential complexity) are **infeasible**.

One such algorithm is one for the **travelling salesman problem**.

A salesman has to travel to say, 50 cities. He wishes to take the cheapest (perhaps shortest) route whereby he visits each city only once.

One simple algorithm is to compute all possible routes and pick the cheapest.

But how many routes are there?

There are 50! possible routes. This is a 65 digit number, and would take billions of years to compute on a computer. If there were 300 cities then we would have a 600 digit number.(So what ! Well the number of protons in the

universe is a 126 digit number).

Thus we can see that the algorithm is infeasible for even small values of n.

It is important to note that this problem occurs frequently in different guises: e.g. laying telephone cables (roads, rail tracks) between a number of towns. A small saving in distance can result in huge financial savings. It is an example of a routing problem.

Another example of an infeasible problem might be a process control situation in a power plant. There are 50 sensors around the plant connected to a computer. For simplicity, we assume each sensor sends a binary signal to the computer, indicating normal/abnormal conditions.

We are interested in testing that the computer program will behave correctly with every possible combination of inputs from the sensors.

Well how many combinations are there ? There are 50 sensors each providing yes/no responses thus there are 2^{50}

possible combinations. If we carried out 1 instruction per microsecond it would take over 35 years to compute all solutions.

If we had 60 sensors, it would take 366 centuries to compute.

This is another infeasible problem.

This last example illustrates another important issue. We cannot test complex computer programs to check if they will work with all possible inputs. We can only partially test programs on a very small subset of the possible inputs.

It is also very important to remember that testing in itself does not prove anything. Just because a program works for some test data, in no way proves the correctness of the program.

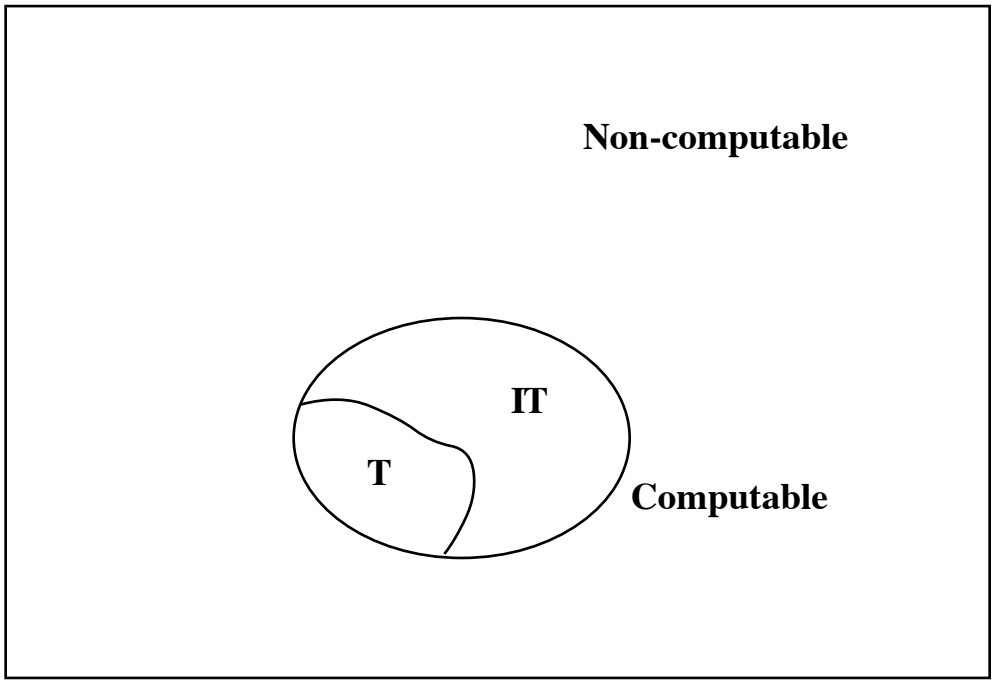
The question arises then as to how we deal with infeasible problems. One approach is to develop algorithms that are not guaranteed to give the best answer, but will on average give a good answer.

Computability

In the field of computability, we are interested in studying what can and cannot be computed. We have seen above some problems that were computable but they were infeasible i.e. it would take an impossible amount of time to compute the results.

But there are also problems, in fact the majority of problems, which cannot be computed. The computable problems form a very small subset of the universe of problems. The set of feasibly computable problems forms a small subset of the computable problems as shown in the figure below.

Obviously it is important to be able to distinguish infeasible and non-computable problems from feasible problems, so that we do not waste time seeking solutions that either do not exist, or if they do exist, they are useless.



T: Tractable or Feasibly Computable
IT: Intractable or Infeasible

A Non-Computable Problem: Halting Problem

A common error that programmers make is to write programs that contain endless loops i.e. they do not terminate (halt). When such a program is executed, it must be interrupted by the user to halt it, when the user realises after a while that the program is not going to terminate.

It would be very useful, if we could write a program or modify a compiler so that it could test if a given source program terminates. If we had such a compiler, then we would never execute a program with an endless loop.

The sad fact is that this problem is non-computable. It is impossible to write a program, which given another program P as input, will determine whether P terminates. This problem is known as the halting problem.

Informally, we can show that the halting problem is non-computable as follows.

Assume there exists a function Halt() which takes an arbitrary program P as a parameter. Function Halt() is

defined to return True if P halts and False if P does not halt e.g.

```
if Halt( P ) then printout('P halts')
else printout('P does not halt');
```

Consider the following program F which takes an arbitrary program P as input and uses the function Halt() as follows:

```
Label1:   if Halt( P ) then goto Label1
          else Stop ;
```

This program loops forever if program P does halt i.e. Halt(P) returns True and it stops only if P does not halt.

What happens with program F if we use program F itself as input i.e. as the parameter to Halt(). F now takes the form:

```
Label1:   if Halt( F ) then goto Label1
          else Stop ;
```

which says that if F halts then it loops forever and if F does not halt it stops !

We thus have a contradiction which is due to our assumption that a function Halt(P) could computed. We thus conclude that the halting problem is non-computable.

Computability is explored in the area of Computer Science called **Theory of Computation.**

Correctness

Most programs in current use contain errors (bugs). Errors in computer programs have serious consequences such as the loss of a spaceship sent to Venus. This is just one of many publicised errors. However, there are very many more unpublicised ones. Every (honest) programmer will recount tales of silly, funny and serious errors they have made in their programs.

It is estimated that as much as 70% of the effort and cost in constructing complex software systems is devoted to error correcting. This may be due to poor program specifications (imprecise, vague, ambiguous problem definitions), extensive debugging to find errors and worst of all rewriting large parts to correct bugs. The later an error is discovered, the more costly it is to correct.

It is important to be aware that compilers and operating systems also have bugs. It is quite common that new versions of compilers and especially operating systems are regularly released, which correct bugs in previous versions. Sadly, the new versions frequently introduce new bugs which in turn are corrected in later versions and the cycle continues.

Thus the importance of writing correct programs cannot be overstated. Beginners, often believe that their algorithms do precisely what they intend them to do. This has no justification. A disciplined or methodological approach to programming must be adopted if errors are to be eliminated. Computer Science courses in Programming Methodology, Formal Specifications and Programming Design and Verification deal with the notion of developing correct programs.

Methods of producing correct programs can be grouped in two categories: testing and proving. Testing a program consists of executing a program on test data in order to discover the output of the program for that data. The important feature is that the outcome is only tested for that

particular set of test data.

Proving a program correct means that the program is correct for all permissible input data. This is obviously a much more desirable property. (It is important to make sure that there are no errors in the proof!).

Program testing has been more widely used because it is a much easier technique to apply, requiring less thought than proving. Informal proofs are often used to reason about a program's behaviour. They can increase our confidence in the correctness of a program. Informal proofs can be made more formal and detailed, but this is more difficult and tedious. However, in some situations where the application is a life critical one, a formal proof of correctness may be required.

The method of proof by induction is primarily used in proving the correctness of a program.

What does correctness mean?

It is meaningless to speak of program correctness in isolation, it must be related to the purpose of the program. “The program produces the right answer” is meaningless unless we can specify what is meant by “right” in any particular situation. The purpose of a program is provided in the form of the problem specification, which defines (precisely, unambiguously and clearly) the problem to be solved and the output to be produced.

A program is correct with respect to its specifications if it produces the results specified for those input values defined by the specification.

There are a number of facets to correctness. We speak of partial correctness, when we know that if a program terminates it will always produce the correct result, but we do not know if it will always terminate.

We speak of total correctness if in addition to always producing the correct result, the program always terminates. Different techniques can be used to prove partial and total correctness.

Another facet of correctness is feasibility, which was discussed earlier. Will the program use a reasonable amount of resources e.g. finish is a feasible time span.

In conclusion, the ability to reason about program correctness and produce correctness proofs is likely to become increasingly important in the future. Already, the major cost in computing systems is the development of software.

The cost of software is in terms of programming effort. Techniques which can significantly improve programmer productivity will become more important.

In addition, the requirement that software in important applications (process control in industry, life critical applications) be formally correct is likely to become a standard requirement.

Further Reading:

The Complete Plain Words, Gowers, Pelican.

Computer Science, A Modern Introduction second edition; Goldschlager and Lister; Prentice Hall

Algorithmics, The Spirit of Computing; Harel; Addison-Wesley

The Science of Programming; Gries, Springer-Verlag

The Practice of Programming, Kernighan & Pike, Prentice Hall