CITS3210 Algorithms

Introduction

Notes by CSSE, Comics by xkcd.com
Overview

1. Introduction
   (a) What are Algorithms?
   (b) Design of Algorithms.
   (c) Types of Algorithms.

2. Complexity
   (a) Growth rates.
   (b) Asymptotic analysis, $O$ and $\Theta$.
   (c) Average case analysis.
   (d) Recurrence relations.

3. Sorting
   (a) Insertion Sort.
   (b) Merge Sort.
   (c) QuickSort.
What you should already know?

This unit will require the following basic knowledge:

1. Java Programming: *classes, control structures, recursion, testing, etc*

2. Data Structures: *stacks, queues, lists, trees, etc.*

3. Complexity: *definition of “big O”, Θ notation, amortized analysis etc.*

4. Some maths: *proof methods, such as proof by induction, some understanding of continuous functions*
What will we be studying?

We will study a collection of algorithms, examining their design, analysis and sometimes even implementation. The topics we will cover will be taken from the following list:

1. Specifying and implementing algorithms.

2. Basic complexity analysis.


4. Graph algorithms.

5. Network flow algorithms.


7. String algorithms.


What are the outcomes of this unit?

At the end of the unit you will:

1. be able to identify and abstract computational problems.

2. know important algorithmic techniques and a range of useful algorithms.

3. be able to implement algorithms as a solution to any solvable problem.

4. be able to analyse the complexity and correctness of algorithms.

5. be able to design correct and efficient algorithms.

The course will proceed by covering a number of algorithms; as they are covered, the general algorithmic technique involved will be highlighted, and the role of appropriate data structures, and efficient implementation considered.
What are algorithms?

An *algorithm* is a well-defined finite set of rules that specifies a sequential series of elementary operations to be applied to some data called the *input*, producing after a finite amount of time some data called the *output*.

An algorithm solves some computational problem.

Algorithms (along with data structures) are the fundamental “building blocks” from which programs are constructed. Only by fully understanding them is it possible to write very effective programs.
Design and Analysis

An algorithmic solution to a computational problem will usually involve designing an algorithm, and then analysing its performance.

**Design** A good algorithm designer must have a thorough background knowledge of algorithmic techniques, but especially substantial creativity and imagination. Often the most obvious way of doing something is inefficient, and a better solution will require thinking “out of the box”. In this respect, algorithm design is as much an art as a science.

**Analysis** A good algorithm analyst must be able to carefully estimate or calculate the resources (time, space or other) that the algorithm will use when running. This requires logic, care and often some mathematical ability.

The aim of this course is to give you sufficient background to understand and appreciate the issues involved in the design and analysis of algorithms.
Design and Analysis

In designing and analysing an algorithm we should consider the following questions:

1. What is the problem we have to solve?

2. Does a solution exist?

3. Can we find a solution (algorithm), and is there more than one solution?

4. Is the algorithm correct?

5. How efficient is the algorithm?
The importance of design

By far the most important thing in a program is the design of the algorithm. It is far more significant than the language the program is written in, or the clock speed of the computer.

To demonstrate this, we consider the problem of computing the Fibonacci numbers.

The Fibonacci sequence is the sequence of integers starting

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

which is formally defined by

$$F_1 = F_2 = 1 \text{ and } F_n = F_{n-1} + F_{n-2}.$$ 

Let us devise an algorithm to compute $F_n$. 

The naive solution

The naive solution is to simply write a recursive method that directly models the problem.

```java
static int fib(int n) {
    return (n<3 ? 1 : fib(n-1) + fib(n-2));
}
```

Is this a good algorithm/program in terms of resource usage?

Timing it on a (2005) iMac gives the following results (the time is in seconds and is for a loop calculating $F_n$ 10000 times).

<table>
<thead>
<tr>
<th>Value</th>
<th>Time</th>
<th>Value</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_{20}$</td>
<td>1.65</td>
<td>$F_{24}$</td>
<td>9.946</td>
</tr>
<tr>
<td>$F_{21}$</td>
<td>2.51</td>
<td>$F_{25}$</td>
<td>15.95</td>
</tr>
<tr>
<td>$F_{22}$</td>
<td>3.94</td>
<td>$F_{26}$</td>
<td>25.68</td>
</tr>
<tr>
<td>$F_{23}$</td>
<td>6.29</td>
<td>$F_{27}$</td>
<td>41.40</td>
</tr>
</tbody>
</table>

How long will it take to compute $F_{30}$, $F_{40}$ or $F_{50}$?
Experimental results

Make a plot of the times taken.
Theoretical results

Each method call to `fib()` does roughly the same amount of work (just two comparisons and one addition), so we will have a very rough estimate of the time taken if we count how many method calls are made.

Exercise: Show the number of method calls made to `fib()` is \(2F_n - 1\).
Re-design the algorithm

We can easily re-design the algorithm as an iterative algorithm.

```c
static int fib(int n) {

    int f_2; /* F(i+2) */
    int f_1 = 1; /* F(i+1) */
    int f_0 = 1; /* F(i) */

    for (int i = 1; i < n; i++) {
        /* F(i+2) = F(i+1) + F(i) */
        f_2 = f_1 + f_0;

        /* F(i) = F(i+1); F(i+1) = F(i+2) */
        f_0 = f_1;
        f_1 = f_2;
    }

    return f_0;
}
```
An Iterative Algorithm

An iterative algorithm gives the following times:

<table>
<thead>
<tr>
<th>Value ( F^{20} )</th>
<th>Time 0.23</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F^{21} )</td>
<td>0.23</td>
</tr>
<tr>
<td>( F^{22} )</td>
<td>0.23</td>
</tr>
<tr>
<td>( F^{23} )</td>
<td>0.23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value ( F^{10^3} )</th>
<th>Time 0.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F^{10^4} )</td>
<td>0.48</td>
</tr>
<tr>
<td>( F^{10^5} )</td>
<td>2.20</td>
</tr>
<tr>
<td>( F^{10^6} )</td>
<td>20.26</td>
</tr>
</tbody>
</table>
Another solution?

The Fibonacci sequence is specified by the homogeneous recurrence relation:

\[
F(n) = \begin{cases} 
1 & \text{if } n = 1, 2; \\
F(n - 1) + F(n - 2) & \text{otherwise.}
\end{cases}
\]

In general we can define a closed form for these recurrence equations:

\[
F(n) = A\alpha^n + B\beta^n
\]

where \(\alpha, \beta\) are the roots of

\[
x^2 - x - 1 = 0.
\]

- You need to be able to derive a recurrence relation that describes an algorithms complexity.

- You need to be able to recognize that linear recurrence relations specify exponential functions.

See CLRS, Chapter 4.
Recurrence Relations

Recurrence relations can be a useful way to specify the complexity of recursive functions.

For example the *linear homogeneous recurrence relation*:

\[
F(n) = \begin{cases} 
1 & \text{if } n = 1, 2; \\
F(n - 1) + F(n - 2) & \text{otherwise}
\end{cases}
\]

specifies the sequence 1, 1, 2, 3, 5, 8, 13, ....

In general a linear homogeneous recurrence relation is given as:

\[
\begin{align*}
F(1) &= c_1 \\
F(2) &= c_2 \\
& \quad \vdots \\
F(k) &= c_k \\
F(n) &= a_1 F(n - 1) + \ldots + a_k F(n - k)
\end{align*}
\]

For example

\[
F(n) = \begin{cases} 
1 & \text{if } n = 1, 2; \\
2F(n - 1) + F(n - 2) & \text{otherwise}
\end{cases}
\]

specifies the sequence 1, 1, 3, 7, 17, 41, ...
Solving the recurrence

All linear homogeneous recurrence relations specify exponential functions. We can find a closed form for the recurrence relation as follows:

Suppose that $F(n) = r^n$.
Then $r^n = a_1 r^{n-1} + ... + a_k r^{n-k}$. We divide both sides of the equation by $r^{n-k}$.
Then $r^k = a_1 r^{k-1} + ... + a_k$.
To find $r$ we can solve the polynomial equation: $r^k - a_1 r^{k-1} - ... - a_K = 0$.

There are $k$ solutions, $r_1, ..., r_k$, to this equation, and each satisfies the recurrence:

$$F(n) = a_1 F(n-1) + a_2 F(n-1) + ... + a_k F(n-k).$$

We also have to satisfy the rest of the recurrence relation, $F(1) = c_1$ etc. To do this we can use a linear combination of the solutions, $r^n_k$. That is, we must find $\alpha_1, ..., \alpha_k$ such that

$$F(n) = \alpha_1 r^n_1 + ... + \alpha_k r^n_k$$

This can be done by solving linear equations.
Solving the recurrence

The roots of the polynomial are

\[
\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{1 \pm \sqrt{5}}{2}
\]

and so the solution is

\[
U(n) = A \left(\frac{1 + \sqrt{5}}{2}\right)^n + B \left(\frac{1 - \sqrt{5}}{2}\right)^n
\]

If we substitute \( n = 1 \) and \( n = 2 \) into the equation we get

\[
A = \frac{1}{\sqrt{5}} \quad B = \frac{-1}{\sqrt{5}}
\]

Thus

\[
F(n) = \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2}\right)^n
\]
What is an algorithm?

We need to be more precise now what we mean by a *problem, a solution* and how we shall judge whether or not an algorithm is a *good* solution to the problem.

A *computational problem* consists of a general description of a question to be answered, usually involving some free variables or parameters.

An *instance* of a computational problem is a specific question obtained by assigning values to the parameters of the problem.

An algorithm *solves* a computational problem if when presented with *any* instance of the problem as input, it produces the answer to the question as its output.
A computational problem: Sorting

*Instance:* A sequence $L$ of comparable objects.

*Question:* What is the sequence obtained when the elements of $L$ are placed in ascending order?

An instance of **Sorting** is simply a specific list of comparable items, such as

$$L = [25, 15, 11, 30, 101, 16, 21, 2]$$

or

$$L = ["dog", "cat", "aardvark", "possum"].$$
A computational problem: Travelling Salesman

*Instance:* A set of “cities” $X$ together with a “distance” $d(x, y)$ between any pair $x, y \in X$.

*Question:* What is the shortest circular route that starts and ends at a given city and visits all the cities?

An instance of Travelling Salesman is a list of cities, together with the distances between the cities, such as

$$X = \{A, B, C, D, E, F\}$$

$$d = \begin{array}{cccccc}
A & B & C & D & E & F \\
A & 0 & 2 & 4 & \infty & 1 & 3 \\
B & 2 & 0 & 6 & 2 & 1 & 4 \\
C & 4 & 6 & 0 & 1 & 2 & 1 \\
D & \infty & 2 & 1 & 0 & 6 & 1 \\
E & 1 & 1 & 2 & 6 & 0 & 3 \\
F & 3 & 4 & 1 & 1 & 3 & 0 \\
\end{array}$$
An algorithm for Sorting

One simple algorithm for **Sorting** is called *Insertion Sort*. The basic principle is that it takes a series of steps such that after the \( i \)-th step, the first \( i \) objects in the array are sorted. Then the \((i + 1)\)-th step *inserts* the \((i + 1)\)-th element into the correct position, so that now the first \( i + 1 \) elements are sorted.

**procedure** INSERTION-SORT\((A)\)

\[
\begin{align*}
\text{for } j & \leftarrow 2 \text{ to } \text{length}[A] \\
\text{do } & \text{ key } \leftarrow A[j] \\
& \text{Insert } A[j] \text{ into the sorted sequence} \\
& \text{ A}[1 \ldots j - 1] \\
& i = j - 1 \\
\text{while } & i > 0 \text{ and } A[i] > \text{ key} \\
& \text{do } A[i + 1] \leftarrow A[i] \\
& \text{ } i = i - 1 \\
& A[i + 1] \leftarrow \text{ key}
\end{align*}
\]
Pseudo-code

Pseudo-code provides a way of expressing algorithms in a way that is independent of any programming language. It abstracts away other program details such as the type system and declaring variables and arrays. Some points to note are:

- The statement blocks are determined by indentation, rather than `{` and `}` delimiters as in Java.

- Control statements, such as `if...then...else` and `while` have similar interpretations to Java.

- The character ▶ is used to indicate a comment line.
Pseudo-code (contd)

• A statement \( v \leftarrow e \) implies that expression \( e \) should be evaluated and the resulting value assigned to variable \( v \). Or, in the case of \( v_1 \leftarrow v_2 \leftarrow e \), to variables \( v_1 \) and \( v_2 \).

• All variables should be treated as local to their procedures.

• Arrays indexation is denoted by \( A[i] \) and arrays are assumed to be indexed from 1 to \( N \) (rather than 0 to \( N - 1 \), the approach followed by Java).

See CLRS (page 19-20) for more details.

But to return to the insertion sort: What do we actually mean by a good algorithm?
Evaluating Algorithms

There are many considerations involved in this question.

• Correctness
  1. Theoretical correctness
  2. Numerical stability

• Efficiency
  1. Complexity
  2. Speed
Correctness of insertion sort

Insertion sort can be shown to be correct by a proof by induction.

procedure INSERTION-SORT(A)
    for $j \leftarrow 2$ to length[A]
        do key $\leftarrow A[j]$
           $\triangleright$ Insert $A[j]$ into the sorted sequence
           $\triangleright$ $A[1 \ldots j-1]$
           $i = j - 1$
           while $i > 0$ and $A[i] >$ key
               do $A[i+1] \leftarrow A[i]$
                  $i = i - 1$
           $A[i+1] \leftarrow key$

We do the induction over the loop variable $j$. The base case of the induction is:
“the first element is sorted”, and the inductive step is:
“given the first $j$ elements are sorted after the $j^{th}$ iteration, the first $j + 1$ elements will be sorted after the $j + 1^{th}$ iteration."
Proof by Induction

To show insertion sort is correct, let $p(n)$ be the statement “after the $n^{th}$ iteration, the first $n + 1$ elements of the array are sorted”

To show $p(0)$ we simply note that a single element is always sorted.

Given $p(i)$ is true for all $i < n$, we must show that $p(n)$ is true: After the $(n - 1)^{th}$ iteration the first $n$ elements of the array are sorted. The $n^{th}$ iteration takes the $(n + 1)^{th}$ element and inserts it after the last element that a) comes before it, and b) is less than it. Therefore after the $n^{th}$ iteration, the first $n + 1$ elements of the array are sorted.
Aside: Proof by Contradiction

Another proof technique you may need is proof by contradiction.

Here, if you want to show some property \( p \) is true, you assume \( p \) is not true, and show this assumption leads to a contradiction (something we know is not true, like \( i < i \)).

For example, two sorted arrays of integers, \( L \), containing exactly the same elements, must be identical.

**Proof by contradiction:** Suppose \( M \neq N \) are two distinct, sorted arrays containing the same elements. Let \( i \) be the least number such that \( M[i] \neq N[i] \). Suppose \( a = M[i] < N[i] \). Since \( M \) and \( N \) contain the same elements, and \( M[j] = N[j] \) for all \( j < i \), we must have \( a = N[k] \) for some \( k > i \). But then \( N[k] < N[i] \) so \( N \) is not sorted: contradiction.
Complexity of insertion sort

For simple programs, we can directly calculate the number of basic operations that will be performed:

```plaintext
procedure INSERTION-SORT(A)
1  for j ← 2 to length[A]
2    do key ← A[j]
3      i = j - 1
4      while i > 0 and A[i] > key
5        do A[i + 1] ← A[i]
6        i = i - 1
7    A[i + 1] ← key
```

The block containing lines 2-7 will be executed \( \text{length}[A] - 1 \) times, and contains 3 basic operations.

In the worst case the block containing lines 5-7 will be executed \( j - 1 \) times, and contains 2 basic operations.

In the worst case the algorithm will take

\[
(N - 1).3 + 2(2 + 3 + ... + N) = N^2 + 4N - 5
\]
where \( \text{length}[A] = N \).
Correctness

An algorithm is *correct* if, when it terminates, the output is a correct answer to the given question.

Incorrect algorithms or implementations abound, and there are many costly and embarrassing examples:

- Intel’s Pentium division bug—a scientist discovered that the original Pentium chip gave incorrect results on certain divisions. Intel only reluctantly replaced the chips.
- USS Yorktown—after switching their systems to Windows NT, a “division by zero” error crashed every computer on the ship, causing a multi-million dollar warship to drift helplessly for several hours.
- Others...?
Theoretical correctness

It is usually possible to give a mathematical proof that an algorithm in the abstract is correct, but proving that an implementation (that is, actual code) is correct is much more difficult.

This is the province of an area known as software verification, which attempts to provide logical tools that allow specification of programs and reasoning about programs to be done in a rigorous fashion.

The alternative to formal software verification is testing; although thorough testing is vital for any program, one can never be certain that everything possible has been covered.

Even with vigorous testing, there is always the possibility of hardware error—mission critical software must take this into account.
Types of Algorithm

For all solvable problems, you should (already!) be able to produce a correct algorithm. The *brute force* approach simply requires you to

1. enumerate all possible solutions to the problem, and

2. iterate through them until you find one that works.

This is rarely practical. Other strategies to consider are:

- Divide and conquer - Divide the problem into smaller problems to solve.
- Dynamic programming.
- Greedy algorithms.
- Tree traversals/State space search
Numerical Stability

You can be fairly certain of exact results from a computer program provided all arithmetic is done with the integers
\[ \mathbb{Z} = \{ \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \} \] and you guard carefully about any overflow.

However the situation is entirely different when the problem involves real number, because there is necessarily some round-off error when real numbers are stored in a computer. A floating point representation of a number in base \( \beta \) with precision \( p \) is a representation of the form.

\[ d.ddddd \times \beta^e \]

where \( d.ddddd \) has exactly \( p \) digits.
Accumulation of errors

Performing repeated calculations will take the small truncation errors and cause them to accumulate. The resulting error is known as *roundoff error*. If we are careful or lucky, the roundoff error will tend to behave randomly, both positive and negative, and the growth of error will be slow.

Certain calculations however, vastly increase roundoff error and can cause errors to grow catastrophically to the point where they completely swamp the real result.

Two particular operations that can cause numerical instability are

- Subtraction of nearly equal quantities
- Division by numbers that are nearly zero

It is important to be aware of the possibility for roundoff error and to alter your algorithm appropriately.
Efficiency

An algorithm is efficient if it uses as few *resources* as possible. Typically the resources which we are interested in are

- Time, and
- Space (memory)

Other resources are important in practical terms, but are outside the scope of the design and analysis of algorithms.

In many situations there is a trade-off between time and space, in that an algorithm can be made faster if it uses more space or smaller if it takes longer.

Although a thorough analysis of an algorithm should consider both time and space, time is considered more important, and this course will focus on time complexity.
Measuring time

How should we *measure* the time taken by an algorithm?

We can do it experimentally by measuring the number of seconds it takes for a program to run — this is often called *benchmarking* and is often seen in popular magazines. This can be useful, but depends on many factors:

- The machine on which it is running.
- The language in which it is written.
- The skill of the programmer.
- The *instance* on which the program is being run, both in terms of size and which particular instance it is.

So it is not an independent measure of the *algorithm*, but rather a measure of the implementation, the machine and the instance.
Complexity

The complexity of an algorithm is a “device-independent” measure of how much time it consumes. Rather than expressing the time consumed in seconds, we attempt to count how many “elementary operations” the algorithm performs when presented with instances of different sizes.

The result is expressed as a function, giving the number of operations in terms of the size of the instance. This measure is not as precise as a benchmark, but much more useful for answering the kind of questions that commonly arise:

- I want to solve a problem twice as big. How long will that take me?
- We can afford to buy a machine twice as fast? What size of problem can we solve in the same time?

The answers to questions like this depend on the complexity of the algorithm.
Example

Suppose you run a small business and have a program to keep track of your 1024 customers. The list of customers is changing frequently and you often need to sort it. Your two programmers Alice and Bob both come up with algorithms.

Alice presents an algorithm that will sort $n$ names using $256n \log n$ comparisons and Bob presents an algorithm that uses $n^2$ comparisons. (Note: $\log n \equiv \log_2 n$)

Your current computer system takes $10^{-3}$ seconds to make one comparison, and so when your boss benchmarks the algorithms he concludes that clearly Bob’s algorithm is better.

<table>
<thead>
<tr>
<th>Size</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>2621</td>
<td>1049</td>
</tr>
</tbody>
</table>

But is he right?
Expansion

Alice however points out that the business is expanding and that using Bob’s algorithm could be a mistake. As the business expands, her algorithm becomes more competitive, and soon overtakes Bob’s.

<table>
<thead>
<tr>
<th>Size</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>2621</td>
<td>1049</td>
</tr>
<tr>
<td>2048</td>
<td>5767</td>
<td>4194</td>
</tr>
<tr>
<td>4096</td>
<td>12583</td>
<td>16777</td>
</tr>
<tr>
<td>8192</td>
<td>27263</td>
<td>67109</td>
</tr>
</tbody>
</table>

So Alice’s algorithm is much better placed for expansion.

A benchmark only tells you about the situation today, whereas a software developer should be thinking about the situation both today and tomorrow!
Hardware improvement

A time-critical application requires you to sort as many items as possible in an hour. How many can you sort?

An hour has 3600 seconds, so we can make 3600000 comparisons. Thus if Alice’s algorithm can sort $n_A$ items, and Bob’s $n_B$ items, then

\[ 3600000 = 256n_A \log n_A = n_B^2, \]

which has the solution

\[ n_A = 1352 \quad n_B = 1897. \]

But suppose that we replace the machines with ones that are four times as fast. Now each comparison takes $\frac{1}{4} \times 10^{-3}$ seconds so we can make 14400000 comparisons in the same time. Solving

\[ 14400000 = 256n_A \log n_A = n_B^2, \]

yields

\[ n_A = 4620 \quad n_B = 3794. \]

Notice that Alice’s algorithm gains much more from the faster machines than Bob’s.
Different instances of the same size

So far we have assumed that the algorithm takes the same amount of time on every instance of the same size. But this is almost never true, and so we must decide whether to do best case, worst case or average case analysis.

In best case analysis we consider the time taken by the algorithm to be the time it takes on the best input of size $n$.

In worst case analysis we consider the time taken by the algorithm to be the time it takes on the worst input of size $n$.

In average case analysis we consider the time taken by the algorithm to be the average of the times taken on inputs of size $n$.

Best case analysis has only a limited role, so normally the choice is between a worst case analysis or attempting to do an average case analysis.
Worst case analysis

Most often, algorithms are analysed by their worst case running times — the reasons for this are:

- This is the only “safe” analysis that provides a guaranteed upper bound on the time taken by the algorithm.

- Average case analysis requires making some assumptions about the probability distribution of the inputs.

- Average case analysis is much harder to do.
Big-O notation

Our analysis of insertion sort showed that it took about \( n^2 + 4n - 5 \) operations, but this is more precise than necessary. As previously discussed, the most important thing about the time taken by an algorithm is its rate of growth. The fact that it is \( n^2/2 \) rather than \( 2n^2 \) or \( n^2/10 \) is considered irrelevant. This motivates the traditional definition of Big-O notation.

**Definition** A function \( f(n) \) is said to be \( O(g(n)) \) if there are constants \( c \) and \( N \) such that

\[
f(n) \leq cg(n) \quad \forall n \geq N.
\]

Thus by taking \( g(n) = n^2 \), \( c = 1 \) and \( N = 1 \) we conclude that the running time of Insertion Sort is \( O(n^2) \), and moreover this is the best bound that we can find. (In other words Insertion Sort is not \( O(n) \) or \( O(n \log n) \).)
Big-Theta notation

Big-O notation defines an asymptotic *upper* bound for a function $f(n)$. But sometimes we can define a lower bound as well, allowing a tighter constraint to be defined. In this case we use an alternative notation.

**Definition** A function $f(n)$ is said to be $\Theta(g(n))$ if there are constants $c_1$, $c_2$ and $N$ such that

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq N.$$ 

If we say that $f(n) = \Theta(n^2)$ then we are implying that $f(n)$ is *approximately proportional* to $n^2$ for large values of $n$.

See CLRS (section 3) for a more detailed description of the $O$ and $\Theta$ notation.
Why is big-O notation useful?

In one sense, big-O notation hides or loses a lot of useful information. For example, the functions

\[ f(n) = \frac{n^2}{1000}, \quad g(n) = 100 \cdot n^2, \quad h(n) = 10^{10} \cdot n^2 \]

are all \( O(n^2) \) despite being quite different.

However in another sense, the notation contains the essential information, in that it completely describes the asymptotic rate of growth of the function. In particular it contains enough information to give answers to the questions:

- Which algorithm will ultimately be faster as the input size increases?
- If I buy a machine 10 times as fast, what size problems can I solve in the same time?
An asymptotically better sorting algorithm

procedure MERGE-SORT(A, p, r)
    if $p < r$
    then $q \leftarrow \lfloor (p + r)/2 \rfloor$
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)

procedure MERGE(A, p, q, r)
    $n_1 \leftarrow q - p + 1$; $n_2 \leftarrow r - q$
    allocate arrays $L[1 \ldots n_1 + 1]$ and $R[1 \ldots n_2 + 1]$
    for $i \leftarrow 1$ to $n_1$
    do $L[i] \leftarrow A[p + i - 1]$
    for $j \leftarrow 1$ to $n_2$
    do $R[j] \leftarrow A[q + j]$
    $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
    $i \leftarrow 1$; $j \leftarrow 1$
    for $k \leftarrow p$ to $r$
    do if $L[i] \leq R[j]$
    then $A[k] \leftarrow L[i]$
        $i \leftarrow i + 1$
        $i \leftarrow i + 1$
    else $A[k] \leftarrow R[j]$
        $j \leftarrow j + 1$
Merge-sort complexity

The complexity of Merge Sort can be shown to be $\Theta(n \lg n)$. 
The Master Theorem

Merge Sort’s complexity can be described by the recurrence relation:

\[ F(n) = 2F(n/2) + n, \quad \text{where } F(1) = 1. \]

As this variety of recurrence relation appears frequently in divide and conquer algorithms it is useful to have an method to find the asymptotic complexity of these functions.

**The Master Theorem:** Let \( f(n) \) be a function described by the recurrence:

\[ f(n) = af(n/b) + cn^d. \]

where \( a, b \geq 1, \ d \geq 0 \) and \( c > 0 \) are constants. Then

\[
f(n) \text{ is } \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}
\]

See CLRS, 4.3.
The major problem with average case analysis is that we must make an assumption about the probability distribution of the inputs. For a problem like **Sorting** there is at least a *theoretically* reasonable choice—assume that every permutation of length \( n \) has an equal chance of occurring (already we are assuming that the list has no duplicates).

For example, we can consider each of the 24 permutations when sorting four inputs with insertion sort:

<table>
<thead>
<tr>
<th>Comparisons</th>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1234, 2134</td>
</tr>
<tr>
<td>4</td>
<td>1243, 1324, 2143, 2314, 3124, 3214</td>
</tr>
<tr>
<td>5</td>
<td>1342, 1423, 2341, 2413, 3142, 3241, 4123, 4213</td>
</tr>
<tr>
<td>6</td>
<td>1432, 2431, 3412, 3421, 4132, 4231, 4312, 4321</td>
</tr>
</tbody>
</table>

So the weighted average of comparisons is

\[
\frac{(3 \times 2) + (4 \times 6) + (5 \times 8) + (6 \times 8)}{24} = 4.916
\]

(recall that the best case for four inputs is 3, whereas the worst case is 6).
Inversions

**Definition** An *inversion* in a permutation $\sigma$ is an ordered pair $(i, j)$ such that

$$i < j \text{ and } \sigma_i > \sigma_j.$$

For example, the permutation $\sigma = 1342$ has two inversions, while $\sigma = 2431$ has four.

It is straightforward to see that the number of comparisons that a permutation requires to be sorted is equal to the number of inversions in it (check this!) plus a constant, $c$. (For sorting four inputs, $c = 3$)

So the average number of comparisons required is equal to the average number of inversions in all the permutations of length $n$.

**Theorem** The average number of inversions among all the permutations of length $n$ is $n(n - 1)/4$.

Thus *Insertion Sort* takes $O(n^2)$ time on average.
An asymptotically worse algorithm

QuickSort is $\Theta(n^2)$, but it’s average complexity is *better* than Merge-sort! (CLRS Chapter 7)

**procedure** QUICKSORT($A, p, r$)

if $p < r$

then $q \leftarrow $ PARTITION($A, p, r$)

QUICKSORT($A, p, q - 1$)

QUICKSORT($A, q + 1, r$)

**procedure** PARTITION($A, p, r$)

$x \leftarrow A[r]$

$i \leftarrow p - 1$

for $j \leftarrow p$ to $r - 1$

do if $A[j] \leq x$

then $i \leftarrow i + 1$

exchange $A[i] \leftrightarrow A[j]$

exchange $A[i + 1] \leftrightarrow A[r]$

return $i + 1$
The complexity of an algorithm is a measure of how long it takes as a function of the size of the input. For **Sorting** we took the number of items $n$, as a measure of the size of the input.

This is only true provided that the actual size of the items does not grow as their number increases. As long as they are all some constant size $K$, then the input size is $Kn$. The actual value of the constant does not matter, as we are only expressing the complexity in big-O notation, which suppresses all constants.

But what is an appropriate input parameter for **Travelling Salesman**? If the instance has $n$ cities, then the input itself has size $Kn^2$—this is because we need to specify the distance between each pair of cities.

Therefore you must be careful about what parameter most accurately reflects the size of the input.
Travelling Salesman

**Naive solution:** Consider every permutation of the \( n \) cities, and compute the length of the resulting tour, saving the shortest length.

How long will this take? We count two main operations

- Generating the \( n! \) permutations.
- Evaluating each tour at cost of \( O(n) \).

If we assume that by clever programming, we can compute each permutation in constant time, then the total time is \( O(n \cdot n!) \).

Is this a good algorithm?
Good Algorithms

Theoretical computer scientists use a very broad brush to distinguish between good and bad algorithms.

An algorithm is good if it runs in time that is a polynomial function of the size of the input, otherwise it is bad.

<table>
<thead>
<tr>
<th>Good</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>$2^n$</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>$c^n$</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>$n!$</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>exponential</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$O(n^k)$</td>
<td>“polynomial”</td>
</tr>
</tbody>
</table>

A problem for which no polynomial time algorithm can be found is called intractable. As far as we know, Travelling Salesman is an intractable problem, though no-one has proved this.
Summary

1. An algorithm is a well defined set of rules for solving a computational problem.

2. A well designed algorithm should be efficient for problems of all sizes.

3. Algorithms are generally evaluated with respect to correctness, stability, and efficiency (for space and speed).

4. Theoretical correctness can be established using mathematical proof.

5. Numerical stability is required for algorithms to give accurate answers.
6. Different kinds of algorithms have been defined, including brute-force algorithms, divide and conquer algorithms, greedy algorithms, dynamic algorithms, and tree traversals.

7. The efficiency of an algorithm is a measure of complexity that indicates how long an algorithm will take.

8. Big “$O$” is a measure of complexity that is the asymptotic worst case upper bound.

9. $\Theta$ (big $\theta$) is a measure of complexity that is the asymptotic worst case tight bound.

10. Average case analysis attempts to measure how fast an algorithm is for an average (typical) input.
11. *Insertion sort* is a sorting algorithm that runs in time $O(n^2)$.

12. *Merge sort* is a sorting algorithm that runs in time $O(n\log n)$.

13. *Quicksort* is a sorting algorithm that runs in time $O(n^2)$ but is faster than Merge sort in the average case.

14. Polynomial algorithms (e.g. $O(n)$, $O(n\log n)$, $O(n^k)$) are regarded as feasible.

15. Exponential algorithms (e.g. $O(2^n)$, $O(n)$) are regarded as infeasible.