Introduction to Algorithms

- Algorithms

  1. What are Algorithms? Design of Algorithms. Types of Algorithms.

- Sorting

  1. Insertion Sort. Merge Sort. QuickSort.

Reading: Weiss, Chapter 5.

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.

```
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).

Value	Time	Value	Time
F_{20}	1.65	F_{24}	9.946
F_{21}	2.51	F_{25}	15.95
F_{22}	3.94	F_{26}	25.68
F_{23}	6.29	F_{27}	41.40

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

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.

```
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_2 = f_1 + f_0;     /* F(i+2) = F(i+1) + F(i) */

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

  return f_0;
}
```

An Iterative Algorithm

An iterative algorithm gives the following times:

Value	Time
F_{20}	0.23
F_{21}	0.23
F_{22}	0.23
F_{23}	0.23

Value	Time
F_{10^3}	0.25
F_{10^4}	0.48
F_{10^5}	2.20
F_{10^6}	20.26

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

$$L = [\text{“dog”}, \text{“cat”}, \text{“aardvark”}, \text{“possum”}].$$

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

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

$$d = \begin{array}{c|cccccc} & A & B & C & D & E & F \\ \hline A & 0 & 2 & 4 & \infty & 1 & 3 \\ B & 2 & 0 & 6 & 2 & 1 & 4 \\ C & 4 & 6 & 0 & 1 & 2 & 1 \\ D & \infty & 2 & 1 & 0 & 6 & 1 \\ E & 1 & 1 & 2 & 6 & 0 & 3 \\ F & 3 & 4 & 1 & 1 & 3 & 0 \end{array}$$

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

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 $\triangleright$ 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).

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 $A[1 \ldots j-1]$
            $i = j - 1$
            **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:

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

Lines 2-7 will be executed n times, lines 4-5 will be executed up to j times for j=1 to n.

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.

Asymptotic Complexity

In computer science it is most common to compare the complexity of two algorithms by comparing how the time taken grows with the size of the input.

This is know as asymptotic complexity, or less formally as *Big O* notation.

If an algorithm runs in $O(n^2)$ (*big O n squared*), then if we double the size of the input, we quadruple the time taken. (Because $(2n)^2 = 4n^2$).

If an algorithm runs in time $O(2^n)$ then if we increase the size of the input by 1, we double the amount of time taken. (Because $2^{n+1} = 2.2^n$).

We will look at this more formally later in the unit.

A better sorting algorithm (in time)

```
**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($A, p, q, r$)
    $n_1 \leftarrow q - p + 1$; $n_2 \leftarrow r - q$
    **for** $i \leftarrow 1$ **to** $n_1$ **do** $L[i] \leftarrow A[p + i - 1]$
    **for** $j \leftarrow 1$ **to** $n_2$ **do** $R[j] \leftarrow A[q + j]$
    $i \leftarrow 1$; $j \leftarrow 1$; $k \leftarrow p$
    **while** $i \leq n_1$ **and** $j \leq n_2$ **do**
        **if** $L[i] \leq R[j]$ **then** $A[k + +] \leftarrow L[i + +]$
        **else** $A[k + +] \leftarrow R[j + +]$
    **while** $i \leq n_1$ **do** $A[k + +] \leftarrow L[i + +]$
    **while** $j \leq n_2$ **do** $A[k + +] \leftarrow R[j + +]$
```

A better sorting algorithm in space

Quicksort has worst case complexity worse than Merge-Sort, but it's average complexity and space usage 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** $i+1$
```

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.

Summary (cont.)

5. *Insertion sort* is a sorting algorithm that runs in time $O(n^2)$.
6. *Merge sort* is a sorting algorithm that runs in time $O(n\lg n)$.
7. *Quicksort* is a sorting algorithm that runs in time $O(n^2)$ but is faster than Merge sort in the average case.
8. Polynomial algorithms (e.g. $O(n)$, $O(n\lg n)$, $O(n^k)$) are regarded as feasible.
9. Exponential algorithms (e.g. $O(2^n)$, $O(n)$) are regarded as infeasible.