How might we develop algorithms to solve fundamental problems?

1 Describing Algorithms

- Try to develop a more precise language to describe algorithms - pseudo code

2 The first algorithm

- Euclid's algorithm

```
if n==0:
        output m
else:
        set m1=n and n1= m mod n
        set m=m1 and n=n1 and repeat the algorithm
```

2.1 Issues from Euclid's algorithm

- Does it always calculate the GCD?
- Does it halt?
- Is the implementation correct?
- Is it efficient
  - Computation time
  - Memory Usage

3 More on pseudo-code

- It is a list of instructions for menial tasks
- Can change the information in "pigeon holes"
- Ask yes no questions through flow control

4 Recursion

- The idea of "stop what you are doing and repeat from the start on different data; when you are done, pick up where you left off"
- The algorithm is calling itself

```
algorithm: Euclid(m,n)
if n==0:
        output m
else:
        set m=n and n=m mod n
        Euclid(m,n)
```

4.1 Non recursive euclid

Assume that $m \geq n \geq 0$ are non negative integers

```
while n!=0:
        m=m-n
        if n>m:
                swap m and n
output m
```

4.2 Comparison

- One will have more steps than the other, depends on the algorithm
- Compare memory efficiency

5 Yet more on pseudo-code

- Pseudo-code needs to tell us what out inputs, outputs and stored data are
-

Fundamental sorting problem:

- instance: a list of n integers
- output: a list sorted into ascending order

6 Bubble Sort

Assume that our input is always a list A of n integers, held in cells A[0], A[1], . . . , A[n - 1], or A[0..n - 1] for short, and our aim is to output this list of numbers sorted into ascending order.

- The algorithm Bubble-sort repeatedly 'passes' through the input list of numbers, comparing and swapping adjacent numbers in the list.
- In a pass through the input list, consecutive pairs of numbers are compared in turn, and these numbers are swapped (in their locations) if the first number is greater than the second.
- If a swap has been made in a pass through the list then another pass is undertaken, otherwise the algorithm halts

```
change = true
WHILE change == true:
    change = false
    i = 0
    WHILE i < n - 1:
        IF A[i] > A[i + 1]:
            swap A[i] and A[i + 1]
            change = true
        i = i + 1
output A
```

6.1 An execution of Bubble Sort

- During pass 0, bubble sort finds the largest element in the list and puts it in its place
- After that it does a bubble sort on the first n-1 elements of the list
- This suggests proof by induction
- To optimise the number of algorithmic operations, think about this and notice that the previously placed element is still compared, even though the answer is known, so the comparison is unneeded

```
change = true and pass=0
WHILE change == true:
    change = false
    i = 0
    WHILE i ¡ n - 1 - pass:
        IF A[i] > A[i + 1]:
            swap A[i] and A[i + 1]
            change = true
        i = i + 1
    pass = pass +1
output A
```

7 Selection Sort

There are lots of ways to solve our sorting problem, here is how to do selection sort

- Take the number a[0] and store it as x
- pass 0: compare x with the numbers in A[1], A[2], . . . , A[n - 1] in turn, always keeping the smaller number in x and the larger number in the list cell;
- put the resulting number x into A[0];
- take the number in A[1] and store it as x;
- pass 1: compare x with the numbers in A[2], A[3], . . . , A[n - 1] in turn, always keeping the smaller number in x and the larger number in the list cell;
- put the resulting number x into A[1];
- take the number in A[2] and store it as x;
- pass 2: compare x with the numbers in A[3], A[4], . . . , A[n - 1] in turn, always keeping the smaller number in x and the larger number in the list cell;
- put the resulting number x into A[2];

7.1 Pseudo Code

```
pass = 0
WHILE pass < n - 1:
    x = A[pass]
    i = pass + 1
    WHILE i ⩽ n - 1:
        IF A[i] < x:
            swap x and A[i]
        i = i + 1
    A[pass] = x
    pass = pass + 1
output A
```

7.2 How selection sort works

- Selection sort finds the smallest element of the list, puts it in its place and leaves it alone
- To save memory, it is possible to remove the variable x

```
pass = 0
WHILE pass < n - 1:
    i = pass + 1
    WHILE i ⩽ n - 1:
        IF A[i] < A[pass]:
            swap A[i] and A[pass]
        i = i + 1
    pass = pass + 1
output A
```

- It is difficult to say if bubble or selection sort is better
- With an already sorted list:
  - Bubble sort takes 1 pass and n-1 comparisons
  - Selection sort takes $(n-1) + (n-2) + ... + 1 = \frac{n(n-1)}{2}$ comparisons
- However in a reverse sorted list bubble sort makes n(n-1) comparisons

7.3 Implementation Matters

Whilst not as formal as a programming language, pseudocode possesses enough formality for us to express issues relating to data structures and implementation

```
pass = 0
WHILE pass < n - 1:
    x = pass
    i = pass + 1
    WHILE i ⩽ n - 1:
        IF A[i] < A[x]:
            x = i
        i = i + 1
        swap A[pass] and A[x]
    pass = pass + 1
output A
```

- In this method the cell indices are modified, rather than the contents. Any "cell swapping" is left until the end of the pass
- This performs much better if the execution is in an environment where the number of swaps is important

8 Merge Sort

Method

- Chop the input list into roughly two halves so that they have the same length or differ by 1
- Recursively sort each half
- Merge the two sorted lists together

8.1 Pseudocode implementation

```
algorithm: Merge-sort(l, r)      works on portion of list A[l..r]
IF l < r:
  m = ⌈(r - l + 1)/2⌉ - 1        m 'mid-way' between l and r
  Merge-sort(l, m)               recursively sort A[l..m]
  Merge-sort(m + 1, r)           recursively sort A[m + 1..r]
  Merge(l, m, r)                 merge the two sorted portions
main algorithm:
Merge-sort(0, n - 1)
output A
```

8.2 Merging in merge sort

We use an additional workspace to merge our two halved in the form of another list, B

```
algorithm: Merge(l, m, r)
leftptr = l and rightptr = m + 1
counter = l
WHILE leftptr ⩽ m and rightptr ⩽ r:
    IF A[leftptr] ⩽ A[rightptr]:
        B[counter] = A[leftptr]
        leftptr = leftptr + 1
    ELSE:
        B[counter] = A[rightptr]
        rightptr = rightptr + 1
    counter = counter + 1
IF leftptr ⩽ m:
    B[counter..r] = A[leftptr..m]
ELSE:
    B[counter..r] = A[rightptr..r]
A[l..r] := B[l..r]
```

8.3 How merging works

To refine *B*[*counter..r*] = *A*[*rightptr..r*]

```
WHILE rightptr ⩽ r:
    B[counter] = A[rightptr]
    counter = counter + 1
    rightptr = rightptr + 1
```

and to refine *A*[*l..r*] = *B*[*l..r*]

```
i = l
WHILE i ⩽ r:
    A[i] = B[i]
    i = i + 1
```

9 Sorting in Parallel

Are either bubble, selection or merge sort amenable to parallel implementation?

- In bubble sort you could have each processor checking each pair of numbers. However as two processors would be accessing the same number, we would have to take care with mutual exclusion
- There is not an obvious way to make selection sort parallel
- In merge sort each half list could be sorted using distinct processors. But you would need to make sure the lists were sorted before merging

10 Research Glimpses

10.1 Coding Theory - Question from question sheet on this will not be on the exam

- Data Compression
  - The compression of data for more efficient transportation
- Error Correction/Detection
  - Add extra data to messages to cope with errors in transit
- Network Coding

  - Different messages can be combined in transit, but so that the receiver can retrieve the original messages

- Predictive coding
  - The recovery of incomplete electronically stored information for use in legal cases, using mathematical coding theory

10.2 Algorithms for massive data sets

The increase in large data sets ha lead to new models for memory

- **External Memory Model** - The bulk of the data lies in secondary storage
- **Streaming model** - Data can only be accessed sequentially, not randomly

11 Binary Search

We can easily search for an item in a list by amending the inner while loop in selection sort:

```
i = 0 and found = false
WHILE (i ⩽ n − 1) and (found == false):
    IF A[i] 6= x:
        i = i + 1
    ELSE:
        found = true
output i
```

- Note that if x appears in the list then the first location i where it appears is the output; otherwise n is the output
- However this algorithm is more efficient if the list is sorted
- Binary search is designed so that it finds an item in an already sorted list

Method

- Compare the desired item with the median item in the list
- If the median is x then we are done
- If the median is greater than x then x lies to the left of the list, if it appears at all
- Re-perform on the halved list

11.1 Pseudocode

```
algorithm: Binary−search
leftptr = 0 and rightptr = n − 1
WHILE leftptr ⩽ rightptr:
    m = ⌈(leftptr + rightptr)/2⌉
    IF A[m] > x:
        rightptr = m − 1
    ELSE:
        IF A[m] < x:
            leftptr = m + 1
        ELSE:
            return m
return n
```

11.2 Binary Search in Action

- Binary search can be changed into a recursive algorithm

```
algorithm: Rec-Binary-search(l, r)
m = ⌈(l + r)/2⌉
IF A[m] == x:
    return m
ELSE:
    IF l == m:
    return n
ELSE:
    IF A[m] > x:
        index = Rec-Binary-search(l, m - 1)
        return index
    ELSE:
        IF m < r:
            index = Rec-Binary-search(m + 1, r)
            return index
        ELSE:
            return n
main algorithm:
index = Rec-Binary-search(0, n - 1)
output index
```

12 String matching

The string matching problem:

- We are given a test string and a pattern string
- We want to find an occurrence of the pattern of the test (if there is one)

Method:

- The text string is given as an input in the list T[0...n-1] and the pattern string in the list P[0...m-1]
- The algorithm starts with the pattern aligned against the first m symbols of T
- If matches output true
- If not move along one
- Repeat
- Always compare from the right

12.1 Pseudocode

```
algorithm: Naive-String-Match
pos = 0
WHILE pos ⩽ n - m:
    j = m - 1
    WHILE (j ⩾ 0) and (P[j] == T[pos + j]):
        j = j - 1
    IF j < 0:
        return pos
    ELSE:
        pos = pos + 1
return n
```

12.2 Naive string matching in action

The algorithm can be slightly modified to find all occurrences of the pattern in the text

```
algorithm: Naive-String-Match
pos = 0
flag=false
WHILE pos ⩽ n − m:
    j = m − 1
    WHILE (j ⩾ 0) and (P[j] == T[pos + j]):
        j = j − 1
    IF j < 0:
        output pos
        flag=true
    pos = pos + 1
If flag==false:
    output n
```

13 Boyer-Moore string matching

The naive string matching performs more comparisons than needed, because there is unused extra information. Consider the example below:

$$\begin{array}{ll} T & a\ \mathbf{b}\ \underline{b}\ \underline{a}\ b\ a\ c\ a\ b\ a\ a \\ P & c\ a\ \underline{b}\ \underline{a} \end{array}$$

- We know that there are 2 matched symbols (**good suffix**) and a mismatched character, called the **bad character**
- Based on our knowledge of the good suffix and the pattern only, if we have the above mismatch, we can shift the pattern4 cells to the right. This is known as the **good suffix heuristic**
- Based on our knowledge of the bad character and the pattern only, if we have the above mismatch then we can shift the pattern 2 cells to the right before resuming. This is known as the **bad character heuristic**

13.1 Modifying naive string matching

- Two functions, one for the good suffix, one for the bad character
- With good suffix, takes an input number and returns a number greater than zero, depending on the position you are in the string
- With bad character, take a letter from the alphabet, provide an offset
- Shift by maximum of good suffix and bad character

14 Bioinformatics

- Take huge string of genetic characters and look for similarities to look at mutations and evolutions.