How do we develop algorithms to solve fundamental problems?

1. Detail, using pseudo-code, two different versions of Euclid's algorithm: a recursive one; and an iterative one [6]

Solution:

Recursive:

```
IF n==0:
    output m
ELSE:
    set m=n and n=m mod n
    Euclid(m,n)
```

Iterative:

```
WHILE n ≠ 0:
    m=m-n
    IF n>m
        swap m and n
output m
```

2. What is the fundamental difference between the research areas of formal methods and software testing when it comes to verifying that programs are correct? [4]

Solution:

In software testing, the program is tested against a set of inputs to make sure that the correct answer is returned in the case.
In formal methods the program will be mathematically tested to ensure that in all cases the correct answer will be returned

3. Describe the key difference between recursion and iteration [2]

Solution:

- In recursion an algorithm will call itself
- In iteration the same routine will be performed on slightly different inputs

4. First, give a natural language description of the algorithm Bubble-sort, and, next, give pseudo-code for your algorithm. Describe how your algorithm works on the input list of numbers 4,3,6,6,2,1 [10]

Solution:

- 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
```

5. Here is the pseudo-code for the algorithm Bubble-sort [4]

```
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
```

Can you amend this algorithm so as to improve it? You should explain how you amend the pseudo-code and what improvements you have made

Solution:

```
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
```

Because bubble sort will put the first element in the correct position on the first run and etc for the next runs, this means that it does not need to be checked the next time, by adding the pos it allows you to keep track of the place to which is correct

6. First, give a natural language description of the algorithm Selection-sort, and, next, give pseudo-code for your algorithm. Describe how your algorithm works on the input list of numbers 4,4,6,6,2,1 [12]

Solution:

- Take the number in 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 $A[2], A[3], ..., A[n-1]$ 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=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. Here is the pseudo-code for the algorithm selection-sort

```
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
```

(a) Explain how you might amend the algorithm Selection-sort so that you no longer need the variable x [4]

Solution:

```
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
```

(b) Explain how you might amend the algorithm Selection-sort so that you lessen the number of data swaps made [4]

Solution:

```
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
```

Total for Question 7: 8

8. Amend the algorithm Selection-sort so that it sorts a list of numbers into descending order [3]

Solution:

```
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
```

9. Compare and contrast the efficiency of Bubble-sort and Selection-sort when the input list of numbers is already sorted into descending order. (You should explain how you interpret 'efficiency') [6]

Solution:

Selection sort has a constant time complexity of $O(n^2)$, and in the case where the list is in the reverse order to desired, bubble sort also has its worst case of $O(n^2)$. This makes the two algorithms exactly equal in terms of time complexity in this case.

10. Compare and contrast the efficiency of Bubble-sort and Selection-sort when the input list of numbers is already sorted into ascending order. (You should explain how you interpret 'efficiency') [6]

Solution:

When the list is sorted into ascending order selection sort encounters it constant time complexity $O(n^2)$. Whereas bubble sort encounters it's best case and only has a time complexity of $O(n)$. This makes bubble sort better than selection sort in this case.

11. First, give a natural language description of the algorithm Merge-Sort, and, next,give pseudo-code for your algorithm (you need not provide pseudo-code to describe how two ordered lists are merged into an ordered list). Describe how your algorithm works on the input list of numbers 4,3,6,6,2,1 [14]

Solution:

- Chop the input list into roughly two 'halves' so that both 'halves' either have the same length or their lengths differ by 1
- recursively sort each half' and
- merge the two sorted lists together

```
Merge-sort(l,r)
    if l<r
    m=⌈(r − l + 1)/2⌉ − 1
    Merge-sort(l,m)
    Merge-sort(m+1,r)
    Merge(l,m,r)
    Mergesort(0,n-1)
    output A

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]
```

On the input list given, it will be split into two sublists (4,3,6) and (6,2,1)
These will then be recursively split until the digits are individual.
Then the digits are merged back together, sorting them through the comparisons between the two sublists.
The merging goes all the way up the recursion stack until the input list is sorted

Note that Q12 is omitted as it is non examinable

13. First, describe the algorithm Binary-search using natural language, and, next, give pseudo-code for your algorithm. Describe how your algorithm works on the input list of numbers 1,2,3,5,6,6,8,9 when searching for the number 7 [10]

Solution:

- Compare our item x with the median element of the list
- If this median element is x then we are done
- Otherwise:
  - If this median element is greater than x then x lies to the left in the list, if it appears at all; and
  - If this median element is less than x then x lies to the right of the list, if it appears at all

So we continue to look only in the respective sub-list

```
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
```

14. What is the string matching problem? [2]

Solution:

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

15. The following algorithm solves the string matching problem where the text string is T and the patten string is P: [15]

```
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
    IF pos == n - m + 1:
        return n
```

Explain using natural language how the algorithm works. The BoyerMoore algorithm improves the above algorithm by using two heuristics. Explain what these heuristics are and how they are applied. Assuming that you can pre-compute these two heuristics, what simple change can you make to the above algorithm so as to convert it into the BoyerMoore algorithm

Solution:

- Start with the pattern P aligned against the first m symbols of the text T
- If there is a match halt and output the index within the text where the pattern matches
- Otherwise 'move' the pattern 1 cell to the right so that the pattern is aligned with the text symbols in T[1...m]
- Repeat until a match is found

The Boyer-Moore string matching improves this in the following ways:

- **Good suffix heuristic** - If the end of the string is matching then the string can be shifted by the length of the string, rather than by 1 iff the matching suffix is not repeated in the string
- **Bad character heuristic** - If there is a mismatch then the string can be shifted until the next occurrence of the value

The above algorithm can be turned into the Boyer-Moore algorithm by replacing the line

```
pos=pos+1
```

With

```
pos=pos+max{GS(j),BC(T[pos+j],j)}
```

Where GS(j) is the offset to shift the pattern where there is a mismatch at pattern symbol j according to the good suffix heuristic.
BC(x,j) is the offset to shift the pattern when there is a mismatch with the pattern symbol P[j] according to the bad character heuristic

16. Suppose we have a pattern string abab in the Boyer Moore algorithm where the alphabet of the text string is $\{a, b, c\}$. What are the values of the good suffix and the bad character heuristics at the different mis match positions [7]

Solution:

Good Suffix:

- GS(3)=1 (No good suffix)
- GS(2)=4 (need to shift the full length of the string)
- GS(1)=2 (shift to next occurence)
- GS(0)=2 (shift to next occurence)

Bad Character:

$$BC(C, 3) = 4 \qquad BC(a, 3) = 1$$
$$BC(C, 2) = 3 \qquad BC(b, 2) = 1$$
$$BC(C, 1) = 2 \qquad BC(a, 1) = 1$$
$$BC(C, 0) = 1 \qquad BC(b, 0) = 1$$

17. Give one area of science where string matching algorithms are of direct relevance [1]

Solution:

Bioinformatics - Determining if an amino acid is in a DNA sequence

18. Give an illustration of a principle of Computational Thinking in the context of algorithmic problem solving [2]

Solution:

Algorithmic Graph Theory is the study of how many real life problems can be abstracted as graphs, and the study of the algorithmic properties of graphs and the properties of graphs