Practical Sheet 2

Searching and Sorting

1 Searching

1.1 Linear Search

Exercise 1.1: Implementing linear search in Python

Implement the linear search algorithm in Python. Here is the pseudo-code:

```
index = 0
while index < length of array
   if A[index] equals target
      return index
   index = index + 1
return not found
```

You could complete the following outline, which is followed by two test cases:

```
def findLin(A, target):
    # find the target in array A
    # return index or -1
    ...
    ...

print(findLin([2,3,4], 3))
print(findLin([2,3,4], 7))
```

1.2 Computational Complexity of Algorithms

Exercise 1.2: Complexity of Linear Search

Linear search has complexity of O(N). Discuss with another member of the course whether each of the following statements is true:

1. A list of 1000 items always takes 1000 nanoseconds to search, using the linear search algorithm.
2. A test of a linear search program runs once and takes 20 microseconds to search a list of 10,000 items. If we run the program once on a list of 20,000 items it will take 40 microseconds.
3. If it takes an average of 30 milliseconds to search a particular list on a particular computer, then another computer cannot possibly be faster.
4. The average time to search a list of 15,000 items is 30 microseconds on a particular computer; we should expect that a list of 150,000 could be searched in an average of 300 microseconds on the same computer.

1.3 Binary Search

Exercise 1.3: Binary Search

Using playing cards and a pointer (or pointers), show how the following search algorithms work

- Linear search
- Binary search

A pen or pencil can be used as a pointer, showing which cell is currently being processed. Do we need a pointer to explain the algorithms, or can a less detailed explanation be given without?

Exercise 1.4 : Complexity of Binary Search

The following table shows the average times taken on a particular computer to search a (array) list using binary search.

Recall that the complexity is O(log N). So, for example, the time for the array of 1000 items is: 10 x (log 1000 / log 100). That is, the time increases in proportion to the log of the length of the list.

Length of Array	Time Taken (Average of 25 runs)
100	10
1000	
5000	
10,000	
50,000	

Note: if you do not have a calculator handy, use Python. Import the module 'math' and use the function `math.log()`. Does it matter that the time units are not shown?

Exercise 1.5: Questions About Binary Search

Somebody says: "*Binary search obviously does not work. Think about it: to see if an item is in a list, you have to look at every item. It is simply not valid to skip items without checking*". Explain why this reasoning is wrong.

Exercise 1.6: Binary Search in Python

Copy the code shown in the lecture for binary search. Try it out. Check you understand how it works.

2 Sorting

2.1 Bubble Sort

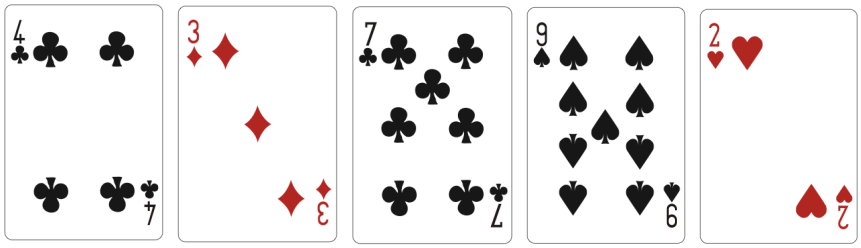
Exercise 2.1: Sort Sequence

The following array of numbers is sorted using bubble sort. Show the array after each pass, starting from the left hand side:

56	21	43	61	12	37	13

Exercise 2.2: Demonstrating Bubble Sort

Using playing cards (e.g. 5 or 6) to show the bubble sort algorithm. Lay the cards face up:



- Start by pointing (use a finger or a pencil) at the left hand card.
- Ask "do I swap with the one to right?"; if so, swap it.
- Then move the pointer one step to the right; the last pointer position is the second last card (why?).
- Repeat again from the start, until no swaps occur.

2.2 Insertion Sort

Exercise 2.3: Sort Sequence

Repeat Exercise 2.1 for insertion sort. Each row should show the order after each insertion.

Note that it is also possible to show the order after each swap operation (in the example given, this requires more rows).

Exercise 2.4: Demonstration Insertion Sort

Repeat Exercise 2.2 for insertion sort. Again, this can be done in more or less detail:

- In the less detailed version, use one pointer to point at the card that is being inserted. Start with the second last card. Treat the insertion as one operation.
- In the more detailed version, you need two pointers:
  - The first pointer points to the card that is to be inserted (as above) but does not move during the insertion.
  - The second pointer is used to keep track of the insertion: it moves as the card is inserted. At each step, compare the value of the card being inserted with the card to its right.

Discuss which version is better.

Exercise 2.5: Sorting Demo

Download the Python program sortingDemo.py.

- Explore and understand the swap order of the algorithms.
- Draw a graph of the number of swaps used for different sized lists (you can use a spread sheet).

2.3 Quicksort

Exercise 2.6: Demonstrate Quicksort

Demonstrate Quicksort in the style of Exercises 2.4 and 2.2. You are recommended not to show the partition step in too much detail.