# Week 4 - Sorting and Searching Algorithms

This lab sheet will introduce two algorithms from computer science. After this session you will be able to sort and search lists using the two following algorithms:

* Insertion and Bubble sort algorithm;
* Binary search.

A YouTube playlist with all the videos for this lab sheet can be found [here](http://www.youtube.com/playlist?list=PLnC5h3PY-znyEYYOnDbQHq1PUuNgbhdD3).

## Sorting Algorithms

1. **TICKABLE** The following code creates a list of digits from 1 to 31.

* l = range(1, 31)  
  print l
* If we import the random library we can pick a random sample of the list and shuffle this it (do not worry too much about this):
* import random  
  jumbledlist = random.sample(range(1, 31), 20)  
  print jumbledlist
* Using pen and paper, sort the above list, attempting to understand a general approach to doing this. Write a function jumbledlist that takes as arguments: maximumnumber and sizeoflist which returns a jumbled list of integersas above.
* [Video hint](http://www.youtube.com/watch?v=18LLHQ7JR1s)

1. **TICKABLE** Python has a built in method on lists to sort them: sort():

* l = jumbledlist(30, 20) # Use the function you created above.  
  print l  
  l.sort()  
  print l
* In this question we will take a look at one type of algorithm that can be used to sort a list: "Selection sort".
* The main idea behind this algorithm is to create a new (empty at first) list and go through the old list and slowly pick out the 'next' element to go in the new list.
* 
* Here is some **pseudo code** that describes this:
* INITIATE NEWLIST  
  WHILE MORE ELEMENTS IN NEWLIST THAN IN OLDLIST:  
   FIND SMALLEST ELEMENT IN OLDLIST  
   MOVE THAT ELEMENT TO NEWLIST
* It should be straightforward to see that at every step of this algorithm the sum of the sizes of NEWLIST and OLDLIST stay the same. As such we can simply put the NEWLIST at the beginning of the OLDLIST so that at each step of our algorithm we are basically moving elements from the unsorted part of the list to the sorted part of the list.
* 
* Here is some **pseudo code** that describes the 'insertion sort' algorithm:
* SET FIRSTUNSORTED TO 0  
  WHILE NOT SORTED:  
   FIND SMALLEST UNSORTED ITEM  
   SWAP FIRST UNSORTED ITEM WITH EARLIEST UNSORTED ITEM  
   SET FIRSTUNSORTED TO FIRSTUNSORTED + 1
* Here is some python code that carries out the above algorithm, experiment with it and include comments:
* def insertionsort(data):  
   firstUnsorted = 0  
   while firstUnsorted < len(data) - 1:  
   indexOfSmallest = firstUnsorted  
   index = firstUnsorted + 1  
   while index <= len(data) - 1:  
   if data[index] < data[indexOfSmallest]:  
   indexOfSmallest = index  
   index += 1  
   data[firstUnsorted], data[indexOfSmallest] = data[indexOfSmallest], data[firstUnsorted]  
   firstUnsorted += 1
* [Video hint](http://www.youtube.com/watch?v=O1HBxikerUQ)

1. There are various other algorithms that can be used to sort lists. The following pseudo code is for an algorithm called 'bubble sort'. Attempt to write out the corresponding python code:

* SET FIRSTUNSORTED TO 0  
  SET SWAP TO TRUE  
  WHILE FIRSTUNSORTED < LENGTH - 1 AND SWAP:  
   SET SWAP TO FALSE  
   "BUBBLE UP" THE SMALLEST ITEM IN AN UNSORTED LIST  
   SET FIRSTUNSORTED TO FIRSTUNSORTED + 1
* Here's the pseudo code for the "BUBBLE UP" part of the above code:
* SET INDEX TO LENGTH - 1  
  WHILE INDEX > FIRSTUNSORTED:  
   IF DATA[INDEX] < DATA[INDEX - 1]  
   SWAP DATA[INDEX] AND DATA[INDEX - 1]  
   SET SWAP TO TRUE  
   SET INDEX TO INDEX - 1
* [Video hint](http://www.youtube.com/watch?v=SXDWTZpj--M)

1. The 'time' module allows you to get the current system time on your machine:

* import time  
  print time.time()
* Using this we can write a function that will evaluate how long it takes to run a particular function:
* def timing(string):  
   starttime = time.time()  
   eval(string)  
   return time.time() - starttime
* This uses the eval function which runs any string of code. We can define the following test function:
* def testfunction():  
   k = 0  
   while k < 10 \*\* 6:  
   k += 1
* We see how long a single run of this testfunction takes using our timing function:
* print timing("testfunction")
* Modify the timing function so that it returns the average time taken over 10 evaluations of the passed code. Furthermore use this function to evaluate the performance of the bubble sort and insertion sort algorithms.
* [Video hint](http://www.youtube.com/watch?v=4MC3wPCKpyg)
* (Note that python also has a timeit library which offers timing functionality.)

## Searching algorithms

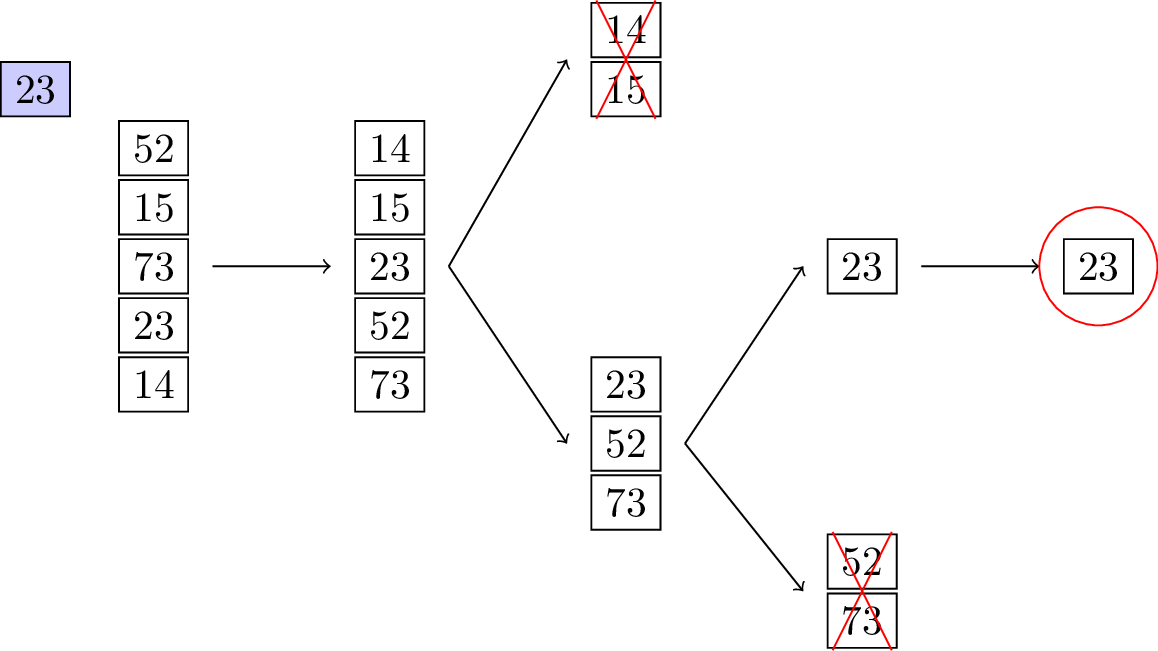
1. **TICKABLE** Consider the data in [W04\_D01.txt](./Data/W04_D01.txt). Search this file for the index of 4558. Do this by hand and then check your answer using the index method. How could you do this if it was to be done in pairs (2 students searching the list)?

* [Video hin](http://www.youtube.com/watch?v=D7-3Oh_oZA4)

1. One searching algorithm we will look at is called "sequential search". This algorithm starts by sorting a list, and then going through it until it either reaches the element in question or gets to a point in the list that the item in question **should** be at.

* SORT THE LIST  
  SET INDEX TO 0  
  SET FOUND TO FALSE  
  WHILE INDEX < LENGTH and NOT FOUND:  
   IF DATA[INDEX] = ITEM:  
   FOUND = TRUE  
   ELSE IF DATA[INDEX] > ITEM:  
   INDEX = LENGTH  
   ELSE:  
   INDEX = INDEX + 1  
  IF FOUND:  
   RETURN INDEX  
  ELSE:  
   RETURN "ITEM NOT IN LIST"
* Write some python code for this and use it to find the index of following numbers:
* targets = [19,  
   592,  
   9507,  
   4221]
* in the file [W04\_D01.txt](./Data/W04_D01.txt).
* [Video hint](http://www.youtube.com/watch?v=CsBlbFaZD_I)

1. **TICKABLE** Another searching algorithm is called 'binary search'. In this algorithm, a **sorted** list is split in two recursively and by considering the first and last element of each list we immediately know which sublist to search.

* 
* Here is some pseudo code that describes this:
* SORT THE LIST  
  SET INDEX TO 0  
  SET LAST TO LENGTH - 1  
  SET FOUND TO FALSE  
  WHILE FIRST <= LAST AND NOT FOUND:  
   SET MIDDLE TO (FIRST + LAST) / 2  
   IF DATA[MIDDLE] = ITEM:  
   SET FOUND TO TRUE  
   ELSE:  
   IF DATA[MIDDLE] > ITEM:  
   SET LAST TO MIDDLE - 1  
   ELSE:  
   SET FIRST TO MIDDLE + 1  
  RETURN MIDDLE
* Here is some python code that carries out the above algorithm, experiment with it and include comments:
* def binarysearch(data, item):  
   data.sort()  
   first = 0  
   last = len(data) - 1  
   found = False  
   while first <= last and not found:  
   middle = int((first + last) / 2)  
   if item == data[middle]:  
   found = True  
   elif item < data[middle]:  
   last = middle - 1  
   else:  
   first = middle + 1  
   if data[middle] == item:  
   return middle  
   return False
* [Video hint](http://www.youtube.com/watch?v=UyIhxhURX-M)

1. Use the timing function of question 4 to compare the performance of the binary search and sequential search algorithms.
2. The binary search algorithm is a very nice algorithm to write in a recursive way, attempt to do this.