Computational Thinking and Algorithms Revision

Computational Thinking

Computational thinking method of studying a problem and formulating a solution that can be provided using a computer.

There are four pillars of computational thinking.

1. Abstraction
The process of of filtering out the characteristics not needed and focusing on those that are

2. Decomposition
Breaking down a complex task into smaller parts that are easier to understand and solve

3. Pattern Recognition
Finding similarities or pattern among decomposed problems to help solved more complex problems

4. Algorithmic(Logical) Thinking
A way of getting to a solution with clear definitions of the steps needed

A design process that could be used for computational thinking	
1. Investigate • Define the problem • Identify the end user • Research	**2. Plan** • Background research • Surveys • Experiments • Research and development
3. Design • Design a solution • Prototype • Sketches • Flow diagrams	**4. Create** • Combine hardware • Code software

Algorithms

Algorithm - a list of instruction or a number of well defined, computer implementable instructions to solve a problem.

We have studied:
Linear and Binary search algorithms
Selection, Insertion, Bubble sort algorithms

Look on Teams at the Python programs for each algorithm and be sure you can explain how they work and be able to identify them.

Algorithmic complexity

We can express algorithmic complexity using something called the '**Big O notation'**.

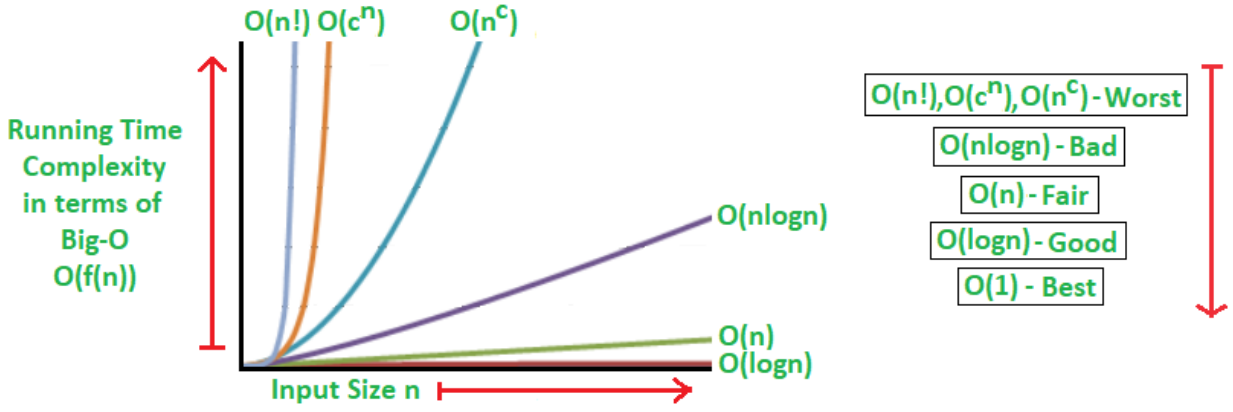
Big(O) notation defines the relationship between a number of inputs and the steps taken by an algorithm to process those inputs.

Big(O) is NOT about measuring speed, it is about **measuring the amount of work** a program has to do based on the size of an input of size N.

Big O is written in the form O(N), which reads as 'Order of N' or for short 'O of N'.

O(1) - Constant Complexity
O(n) - Linear Complexity
$O(n^2)$ - Quadratic Complexity
$O(\log_2 n)$ - Logarithmic Complexity

Polynomial time complexities - O(n), $O(n^2)$, $O(n^3)$



Algorithmic Complexities Compared

Search Algorithms (Linear and Binary)

REMEMBER TO LOOK AT THE CODE ON TEAMS FOR THE COMMENTS

Linear Search

Sequentially going through each item in the list to compare the target to each element and returns the index if found. If the element isn't there, it will return -1 or indicates the element is not there.

Pros

- Data set does not have to be ordered

Cons

- Takes time. Time complexity is O(n), as the list grows the number of comparisons grows.

```
1  myList=['John','Mary','Zoe','Alex','Seamus']
2  name=str(input("Enter name to lookup:"))
3
4  for index in range (len(myList)):
5
6      if myList[index]==name:
7          print("Key is at location: ",index)
8      else:
9          print("Not found")
```

Linear Search Code

Binary Search

- Start at the start of the list
- If middle < target search top half
- if middle > target search bottom half
- if middle = target stop search.

Pros

- Quicker, time complexity $O(\log_2 n)$

Cons

- Data must be ordered

```
1  myList=['A','C','E','D','B','F','H','G','I']
2  myList.sort()
3
4  print(myList)
5  key=input("Enter key value to search for: ")
6  FOUND=False
7
8  while FOUND==False:
9      first_index_value=0
10     last_index_value=(len(myList)-1)
11
12 # **** Start of binary search algorithm*****
13
14     while first_index_value<=last_index_value:
15         mid_point=(last_index_value+first_index_value)//2
16         print("Mid_point value is:",mid_point)
17         if myList[mid_point]<key:
18             first_index_value=mid_point+1
19         elif myList[mid_point]>key:
20             last_index_value=mid_point-1
21         elif myList[mid_point]==key:
22             print("Value is at position: ",mid_point)
23             FOUND==True
24             break
25     else:
26         print("Item not found")
27     break
```

Binary search algorithm

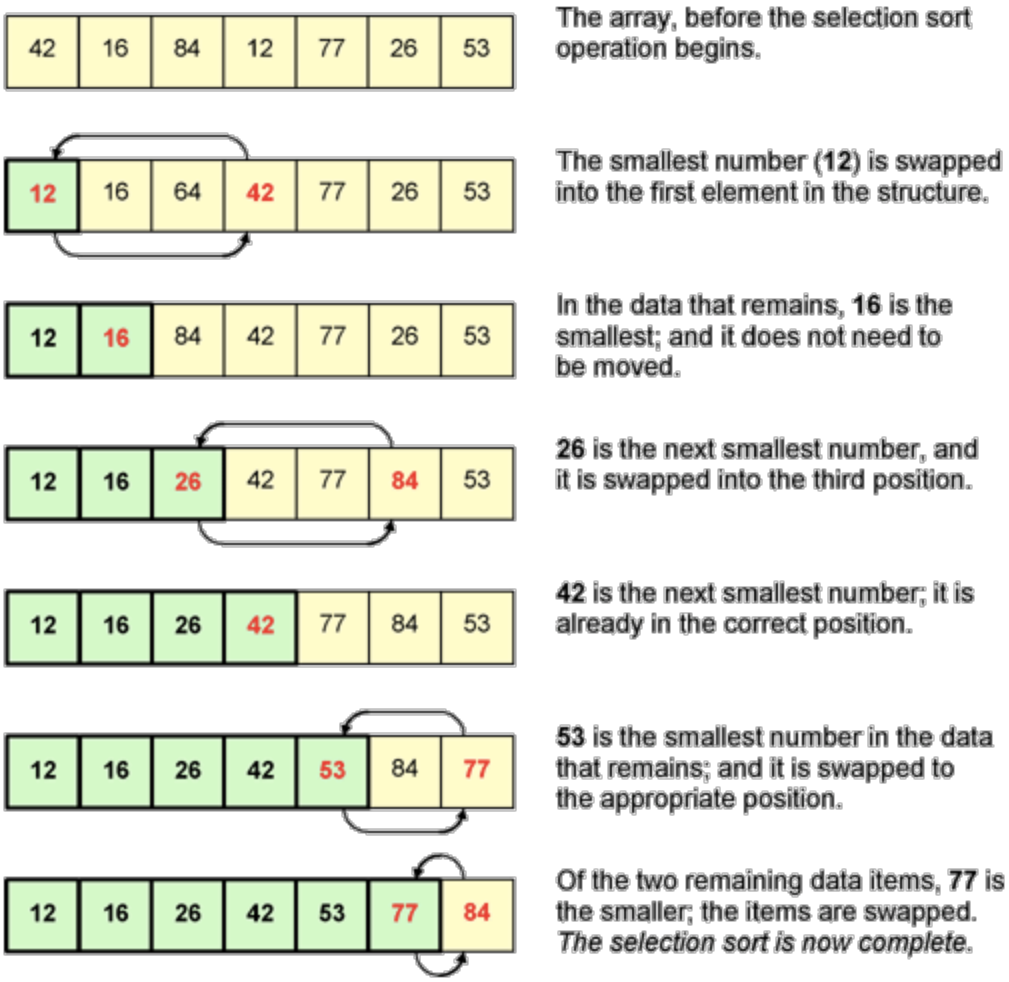
Sort Algorithms (Selection, Insertion, Bubble)

REMEMBER TO LOOK AT THE CODE ON TEAMS FOR THE COMMENTS

Selection Sort.

A selection sort algorithm works by finding the smallest value in a list and then placing it at the start of the list, it then finds the next smallest list and places it in the second position of the list, and then repeats for each value in the list. Each time the a swop occurs, the larger number will take the index position of the smaller number, even if it means the larger number still isn't in the correct location for a fully sorted list. For example, in the image below 26 and 84 swop positions but 84 will swop positions again later on in the sort.

1. The algorithm is looped n times (n = size of array) and each loop requires a maximum of n comparisons (n^2 operations = time complexity $O(n^2)$)



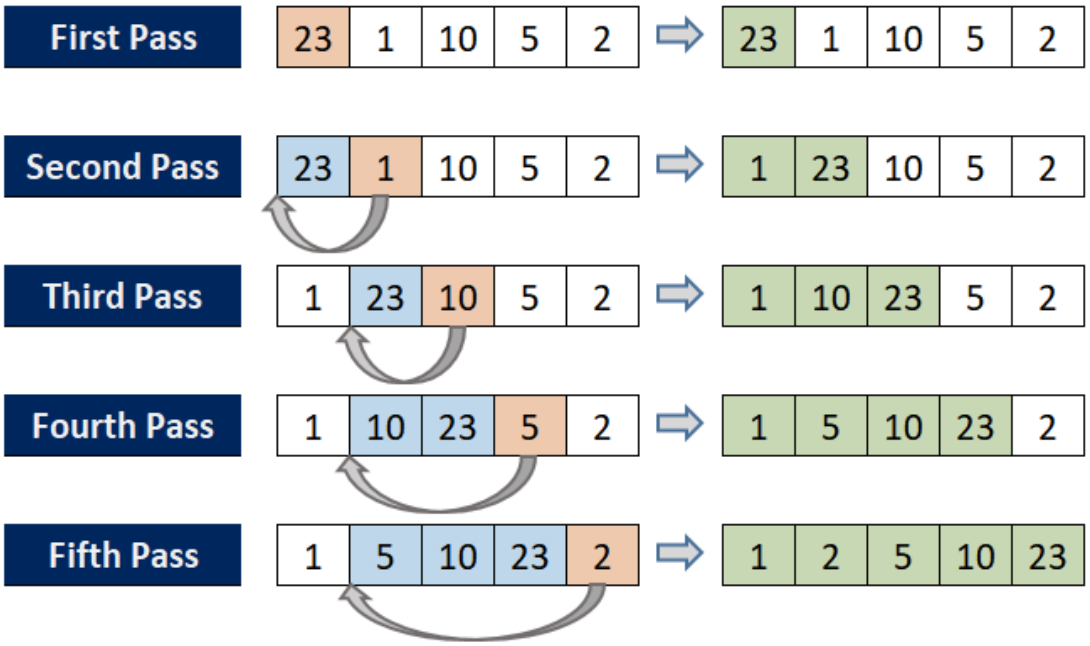
Selection Sort

As the data set size increases the time it takes to sort increases, making it inefficient

Insertion Sort
A insertion sort algorithm is a simple sorting algorithm that **virtually** splits an array or list into two sub arrays, a sorted and unsorted sub array.

The current value is continuously compared to each item before it and then sorted accordingly. For example in the fourth pass below, the value 5 is compared with 23, then 10 and then 1. Since it is smaller than 23 and 10 but larger than 1 it gets placed into index position[1] while 10 goes to index position [2] and 23 goes to index position[3].

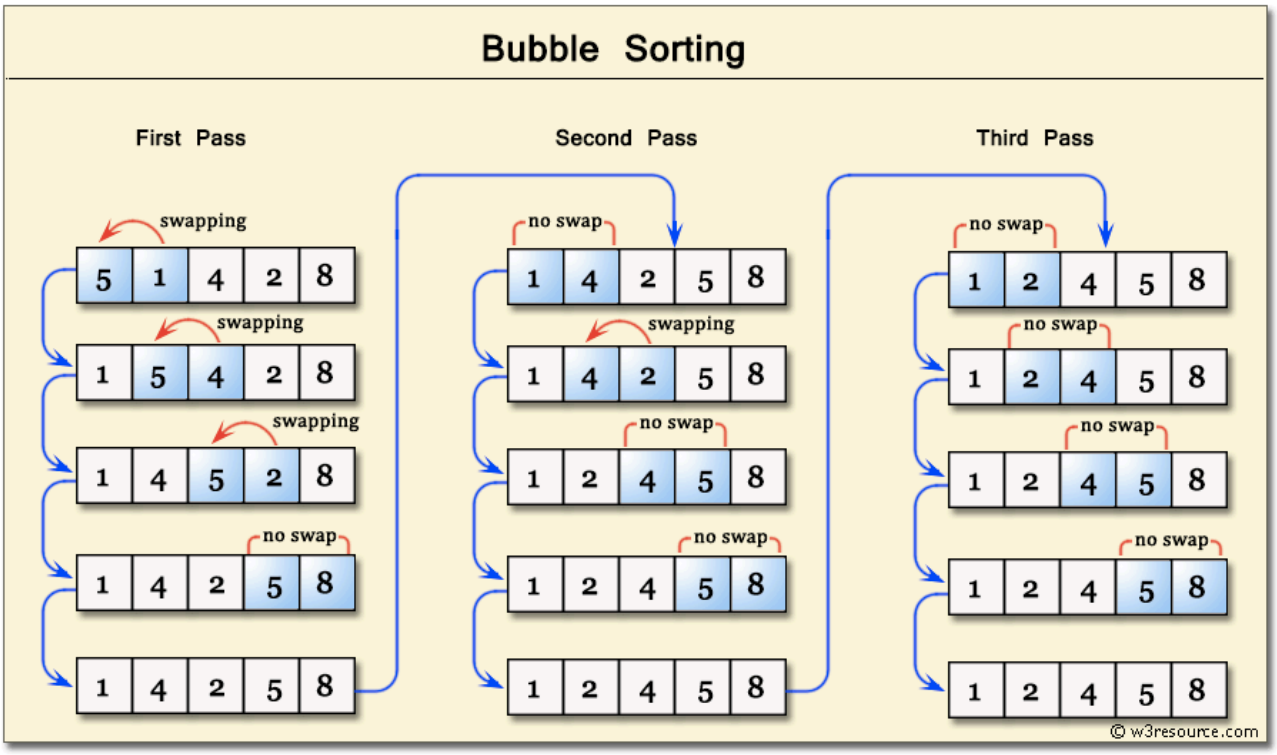
This repeats n times with a maximum of n comparisons (time complexity = $O(n^2)$). As the data set size (n) increases the time it takes to sort increases, making it inefficient.



Insertion sort

Bubble Sort

In a Bubble sort algorithm, each element in an array is compared with the next element and swapped if necessary. After the first iteration (run through) the largest element will always end up in its final position. It is a very slow and inefficient method of sorting data and is almost never used because of this.



Bubble sort

1. Start with the first adjacent pair of elements.
2. Compare them and if they are not in order, swap the elements.
3. Repeat with the next pairs of elements for the length of the array
4. After the first pass of the algorithm the largest value will have bubbled to the top of the array.
5. Repeat steps 1 - 4 until the array is sorted.

Time complexity $O(n^2)$

Bubble sort can be made more efficient by including a count of number of swaps made pass. e.g
if swaps == 0
    stop algorithm