Search algorithms

1. Searching

Looking for a specific piece of information within a group of data items (called a *data set*) is known as 'searching'.

Searching is a very common task in computing. So common, in fact, that a lot of effort has been put into creating algorithms to do this task as efficiently as possible.

For this syllabus, the two search algorithms you need to understand are:-

- Linear search
- Binary search

This section describes linear and binary search algorithms along with their advantages and disadvantages.

Search algorithms

2. Data set and search criteria

To begin a search, you need to know two things: what you are looking for and where you are going to look.

We call the first item the *search criteria*, sometimes it is also called the *target*. The second is the *data set*, and can be anything from a list in computer memory, to an open file from secondary storage, to a database.

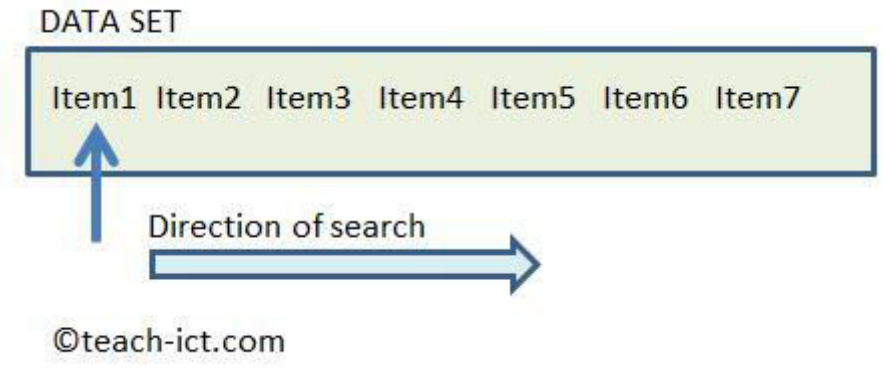
Example
Let the *data set* be a list in memory consisting of (dog, cat, lion, penguin, ant) and let the *search criteria* be 'cat'.

The task of the search algorithm is to determine if 'cat' is in the given data set.

Search algorithms

3. Linear search

Linear search looks for an item within a data set by starting with the first item in the set and comparing it to the search criteria.



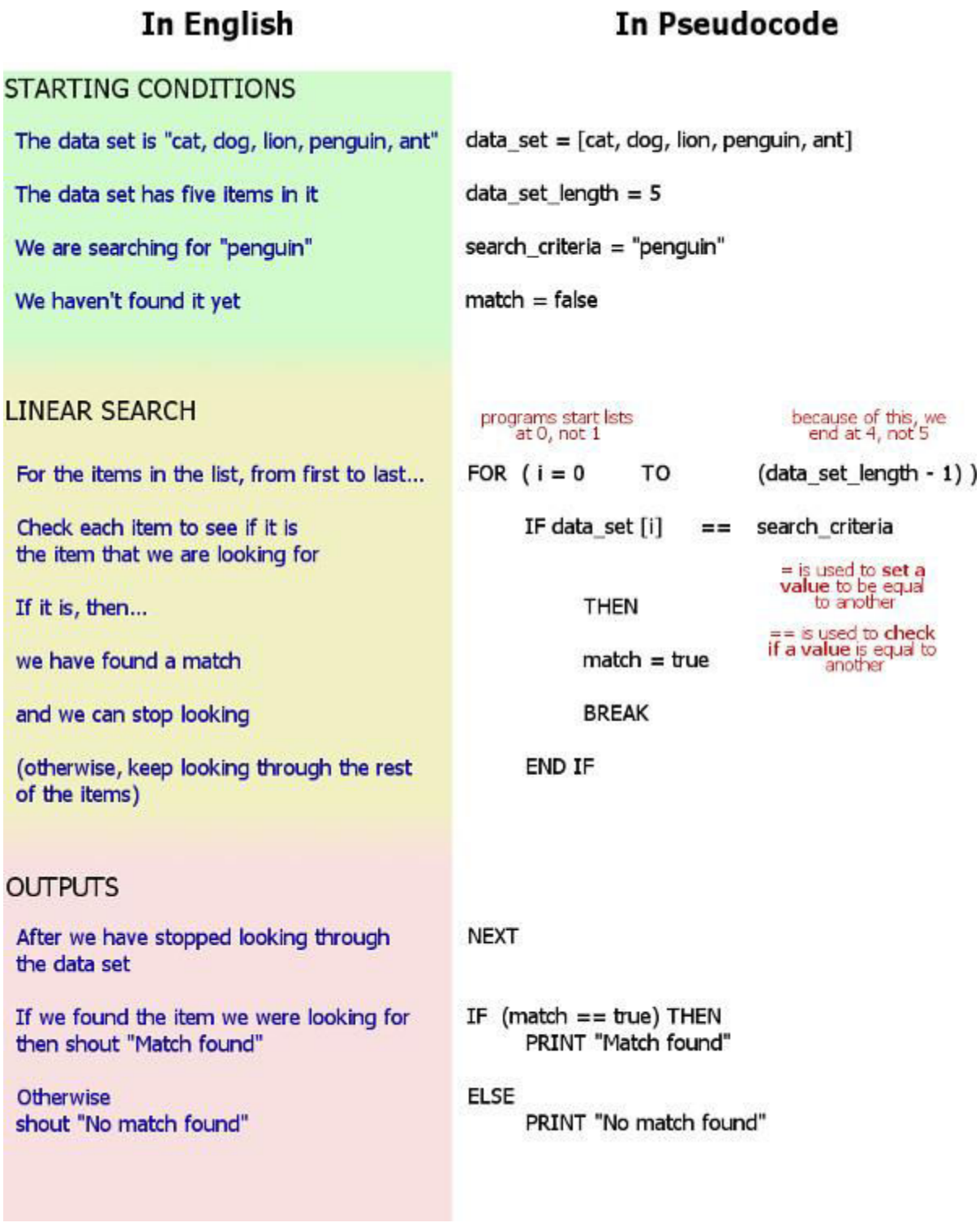
If no match is found, then the next one is compared. This continues until a match is found or the end of the set is reached.

Linear search is also known as the **sequential search algorithm.**

Search algorithms

4. Linear search pseudocode

Much of this pseudocode will look very complicated at this point. You may wish to revisit it after finishing the various sections in the Programming section, where each of the pieces are explained individually. For now, just follow the English-language explanations of each line.



Without annotations, the pseudocode for a linear search algorithm might look like this:

Hover over each line to see comments

```
data_set = [cat, dog, lion, penguin, ant]
```

```
data_set_length = 5

search_criteria = "penguin"

match = false


FOR (i = 0 TO (data_set_length - 1))

  IF data_set[i] == search_criteria THEN

     match = true

     exit loop

   END IF

NEXT i


IF (match == true) THEN

   PRINT "Match found"

ELSE

   PRINT "No match found"

END IF
```

Alternative pseudocode

To emphasise that there is more than one way to write an algorithm, here is an alternative pseudocode that does the same job. It uses a 'while' loop instead of a 'for' loop

Hover over each line to see comments

```
data_set = [cat, dog, lion, penguin, ant]

data_set_length = 5

search_criteria = "penguin"

match = false

position = 0

WHILE position < (data_set_length)

  IF data_set[i] == search_criteria THEN

    match = true

    exit loop

  END IF

  increment position

END WHILE
```

```
IF (match == true) THEN

  PRINT "Match found"

ELSE

  PRINT "No match found"

END IF
```

Despite their differences, they are both ways to code a linear search algorithm. They both start at the beginning of a data set and look at each item in turn to see if it matches the search criteria.

Computer programming allows for more than one way to do a task. This is why there exists more than one word-processor. And from the smartphone point of view, there are thousands of 'apps' doing very similar things.

Search algorithms

5. Linear search pros & cons

Advantages

- Performs well with small and medium-sized lists
- Fairly simple to code
- The data set does not need to be in any particular order (some algorithms need an ordered list)
- It doesn't break if new items are inserted into the list.

Disadvantages

- May be too slow to process large lists or data sets
- If the search criteria only matches the last item in the list, the search has to go through the entire list to find it.

Search algorithms

6. Binary search

Linear search works well for fairly short, unordered lists. For larger data sets you may need a more efficient algorithm i.e. one that can search in fewer steps.

One such algorithm is called the **binary search algorithm**.

First of all, for binary search to work, the list must be arranged in an order.

If a list starts like this:

```
10,7,11,2,1,5
```

It needs to be rearranged like this:

```
1,2,5,7,10,11
```

The basic idea of binary search is to split the ordered list in half and then ask the question "is the target possibly in the upper or lower half list?". This can be answered by checking if the target is greater than the biggest number in the lower list - if it is then discard the lower list, if it isn't then discard the upper list.

This single step means half the list has now been checked in one go.

Split the remaining list again and repeat the question and check. Keep doing this until it is either found or the list cannot be split any further.

Example:

To demonstrate how effective this is compared to a linear search, lets use this example:

Use binary search to find if 19 is in this list:

```
15,7,8,13,6,4,1,20,12,19
```

Rearrange the list to be in order.

0	1	2	3	4	5	6	7	8	9
1	4	6	7	8	12	13	15	19	20

Designate the lowest position in list (0) to be 'Lo' and the highest position in the list (9) to be 'Hi'.

The target is 19.

Step 1: Find the mid point position of the list using this formula:

midpoint = (Lo + Hi) / 2

If (Lo + Hi) is not even, round down (i.e. use floor) to get the nearest whole number . In this case the midpoint position is floor(0+9) / 2 = 4

Step 2: Read the value in position 4, which is 8.

Step 3: There are four possible checks to be made

Check 1: Is target (19) equal to the midpoint value (8)? If it is, then it has been found and search stops.

Check 2: Is target (19) less than midpoint value (8)? If it is, then re-assign 'Hi' position to be one less than the midpoint position (3). This means the upper list is effectively discarded. Search continues.

Check 3: Is target greater than midpoint. If it is, then re-assign 'Lo' to be one more than the midpoint. Search continues

Check 4: This checks to see if every value in the list has been checked, in which case the search stops and returns 'target not found'. This check is done by comparing the midpoint position to the length of the list. If they are equal, then it is time to stop the search.

Here is the algorithm as pseudocode:

```
Let A be the list array.
Let Target be the item to be searched for.

ListLength = A.length    # length of list
Lo = 0        # Lo starts at the lowest position in the list
Hi = ListLength - 1 # Hi starts at the highest position in the
list

WHILE Lo <= Hi  # search complete when this is false
    midpoint = floor (Lo + Hi)/2 # Find the midpoint position
    IF A[midpoint] < Target # It can only be in the upper half
```

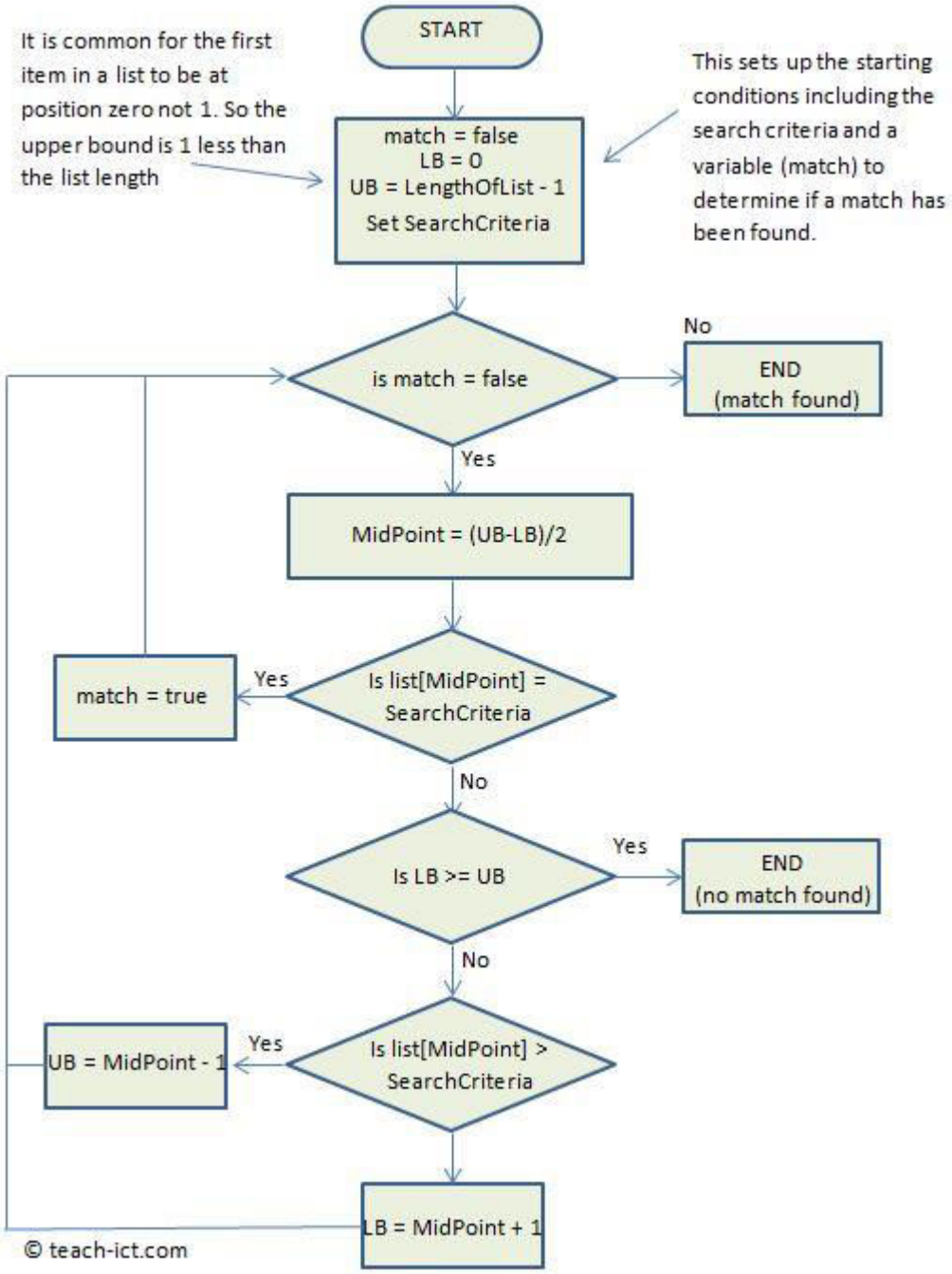
```
                Lo = midpoint + 1 # Stop looking in the lower half
            ELSE IF A[midpoint] > Target # If can only be in lower half
                Hi = midpoint - 1 # Stop looking in the upper half

# this can only be reached if the number is the midpoint value
            ELSE
# target is at the current midpoint position
                return 'target found'
        END WHILE

# All the list was searched and target is not in the list
        return 'target not found'
```

Search algorithms

7. Binary search flow diagram



Search algorithms

8. Binary search pseudocode

The binary search algorithm is set out as pseudocode below.

If the search criteria had to be set by the user, then an input statement would be added to the procedure. To keep things simple, the criteria is pre-set to 50.

NOTE: the symbol '==' is testing for equality and the symbol '!=' means 'not equal to'

```
data_set = [2, 4, 5, 6, 9, 21, 50, 77, 91]
data_set_length = 9
search_criteria = 50
LB = 0
UB = data_set_length - 1
match = false

WHILE match == false AND LB != UB
  MidPoint = roundup((UB - LB)/2) + LB
  IF data_set[MidPoint] == search_criteria THEN
    match = true
  ELSE IF data_set[MidPoint] < search_criteria THEN
    LB = MidPoint + 1
  ELSE
    UB = MidPoint - 1
  END IF
END WHILE

IF match == true THEN
  PRINT 'Match found'
ELSE
  PRINT 'Match not found'
END IF
```

Search algorithms

9. Binary search pros and cons

Advantages

- Very good performance over large ordered lists.
- A binary search takes a maximum of 8 loops to find an item in a list of 1000 items, whilst a linear search worst case would have to search all 1000 items in the list.

Disadvantages

- Binary search can only work on an ordered list. If the list is random, then a linear search is the only way
- It is a bit more complicated than a linear search to code and so for small lists a linear search is just as good.

- If it is a constantly updated list with items added or removed, the list will need re-ordering every time which may slow down the overall performance compared to a linear search.

Sort algorithms

1. Sorting Algorithms

Putting a data set into some kind of logical order is a very common task. For example, the binary search algorithm discussed in the search section requires an ordered data set to work.

Numbered data sets could be ordered in ascending (lowest to highest) or descending (highest to lowest) order. Text data sets could be ordered in alphabetic order.

There are a number of ways to do this, and they can be described using standardised algorithms. For this syllabus, there are three that you need to understand:

- Bubble Sort
- Insertion Sort
- Merge Sort

Each of these will be described in this section along with their strengths and weaknesses.

Sort algorithms

2. Bubble Sort

A bubble sort is a very simple algorithm used to sort a list of data into ascending or descending order.

The algorithm works its way through the list, making comparisons between a pair of adjacent items. Any items found to be in the wrong order are then exchanged. It keeps doing this over and over until all items in the list are eventually sorted into the correct order.

Step by step example

The following list of data (9, 23, 2, 5, 34, 56) needs to be put into ascending order using a bubble sort.

Step 1: Compare the first two items in the list, 9 and 23.

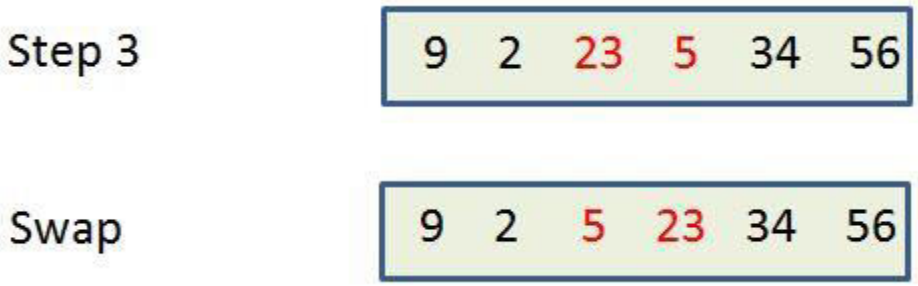
As 9 is smaller than 23 they are in the correct order (ascending) so no action needs to be taken.

Step 2: Move forward by one position and compare the next two numbers in the list - numbers 23 and 2

23 is larger than 2 so the bubble sort will swap the position of those two items.

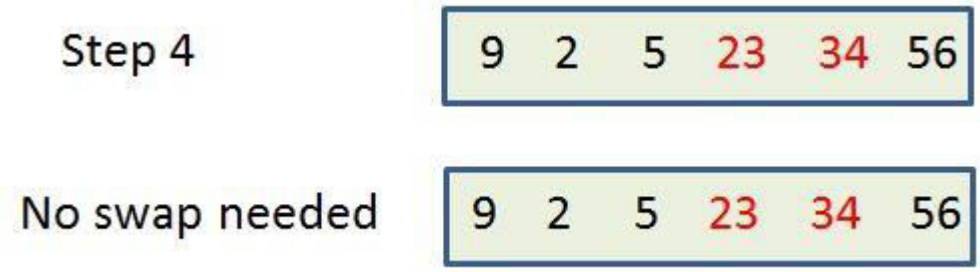
Step 3: Move forward by one position and compare the next two numbers in the list - numbers 23 and 5

23 is greater than 5 so they are swapped.



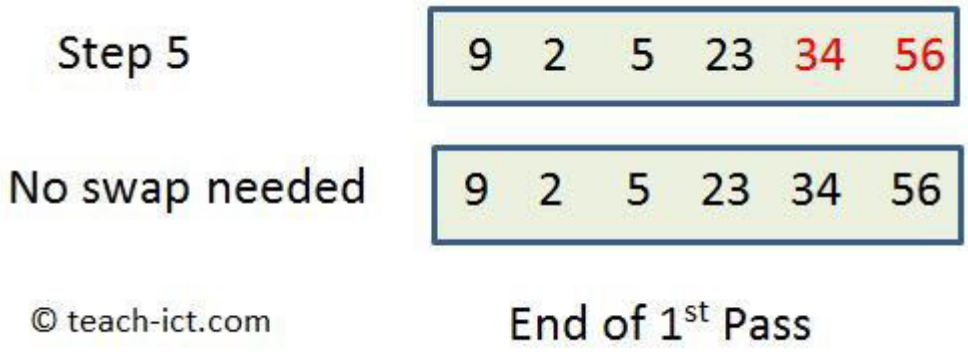
Step 4: Move forward by one position and compare the next two numbers in the list - numbers 23 and 34

23 is not greater than 34 so do NOT swap their position.



Step 5: Move forward by one position and compare the next two numbers in the list - numbers 34 and 56

34 is not greater than 56 so do not swap.



This is the end of the first pass.

You can see from the final image above that the data set is not quite sorted in ascending order (the 9 and the 2 are the wrong way around) so the process is repeated once again, starting at the beginning of the list.

The algorithm is complete when it finishes a pass without having to perform any swaps.

Sort algorithms

3. Bubble Sort pseudocode

The pseudocode below is for the ascending order algorithm. Much like the searching algorithms, you may wish to revisit this page once you have a better grasp on the programming techniques and constructs laid out in our Programming section. For now, just follow the commentaries on each line.

```
data_set = [9,2,5,23,34,56]
last_exam_position = data_set.length - 2
swap = true
WHILE swap == true
  swap = false
  FOR i = 0 to last_exam_position
    IF data_set[i] > data_set[i +1] THEN
      temp = data_set[i+1]
      data_set[i+1] = data_set[i]
      data_set[i] = temp
      swap = true
    END IF
  NEXT i
END WHILE
PRINT "List is now in ascending order."
```

And for completeness sake, a small adjustment to the pseudocode will sort the list in *descending* order, the only line that changes is the one in red shown below, where the 'greater than' becomes 'less than'

```
data_set = [9,2,5,23,34,56]
last_exam_position = data_set.length - 2

swap = true

WHILE swap == true
  swap = false
     FOR i = 0 to last_exam_position
        IF data_set[i] < data_set[i +1] THEN

              temp = data_set[i+1]
              data_set[i+1] = data_set[i]
              data_set[i] = temp   // the swap is complete

           swap = true
        END IF
     NEXT i
END WHILE

PRINT "List is now in descending order."
```

Sort algorithms

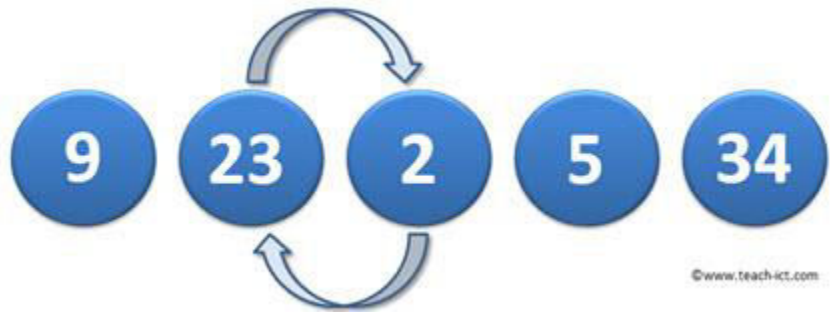
4. Bubble Sort pros and cons

Advantages

- Simple to write the code for.
- Simple to understand.
- The data is sorted in the same memory location that it is held, so you don't need much extra memory to run the algorithm.

Disadvantages

- One of the slowest ways to sort a list. For example, if the list becomes ten times larger than before, it takes almost a hundred times longer to sort. So this method of sorting is very sensitive to the length of the list.



Sort algorithms

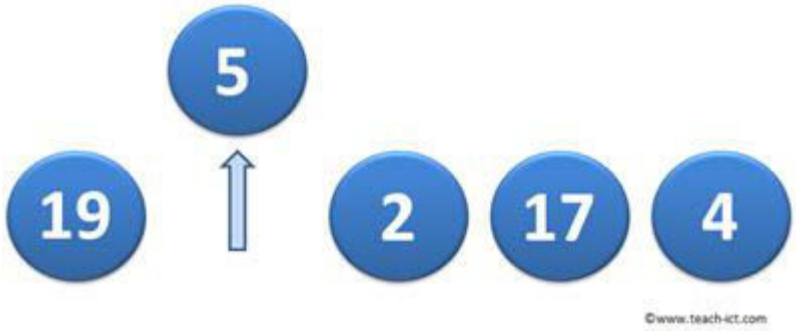
5. Insertion Sort

This is a simple comparison sort, where the sorted list is built up one item at a time. It is more efficient (faster) than bubble sort.

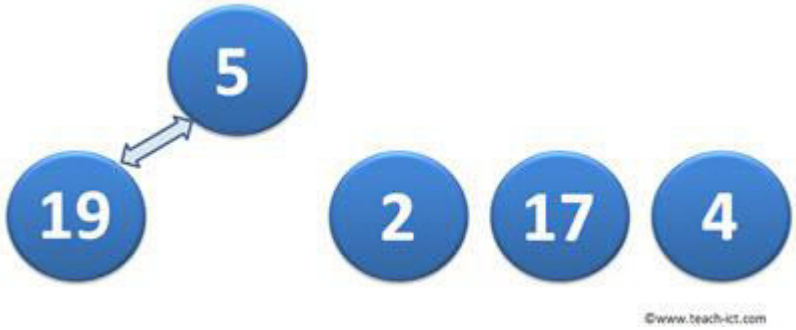
Step by step example

The following list of data (19, 5, 2, 17, 4) needs to be put into ascending order using an insertion sort.

Step 1: Take the second item in the list (5).



Compare the second item (5) with the previous item (19):



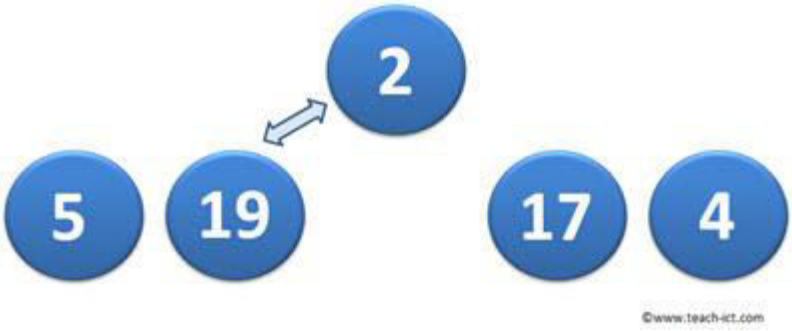
If the second item (5) is greater than the previous item (19), insert it back into the same position

If the second item (5) is smaller than the previous item (19), insert it two positions back (i.e. before 19)

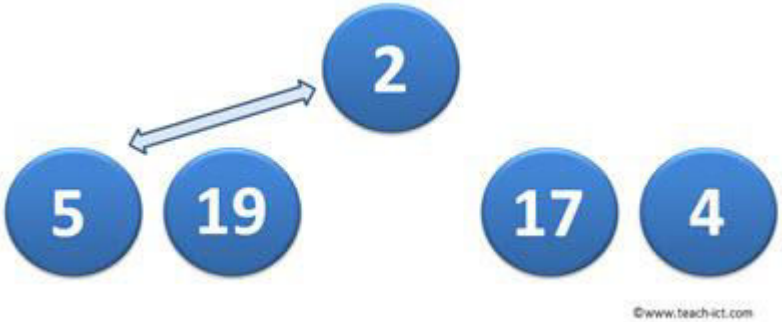
This is what the list now looks like:

Step 2: Move to the next number in the list (2).

Compare the 2 with the previous item (19):



2 is less than 19 so we know that it needs moving to the left. However, we haven't finished yet (if we did move it now then we would simply be doing a bubble sort).



The 2 is now compared to the item in the list prior to 19, i.e. 5. The same checks are run. If it is smaller than 5 then it is moved to the correct position.

These checks will continue, comparing each item with previous data items in the list until everything has been sorted into the correct order:

Sort algorithms

6. Insertion Sort pseudocode

The pseudocode below is one way to code an insertion sort (there are other ways).

In summary, the FOR loop makes sure every item in the list is examined, whilst the WHILE loop does the comparison and insertion.

```
dataSet = [45,23,14,99,15]
FOR i = 1 to dataSet.length - 1
  pos = i
  WHILE pos > 0 AND dataSet[pos-1] > dataSet[pos]
    swap dataSet[pos] and dataSet[pos-1]
    pos = pos - 1
  END WHILE
NEXT i
```

Sort algorithms

7. Insertion Sort pros and cons

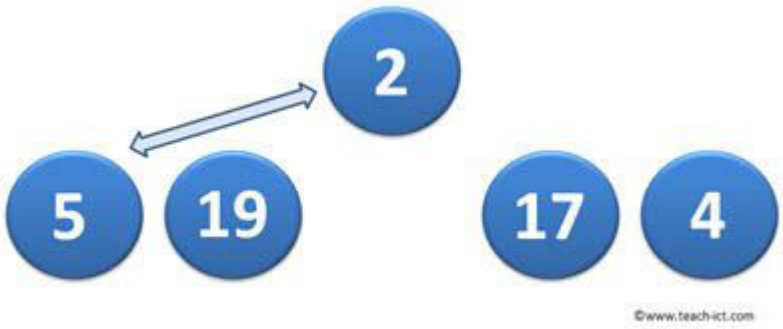
Advantages

- Simpler to code than a merge sort
- Useful for small lists that only take a very short time to sort

- Faster to process a small list than using a bubble sort

Disadvantages

- Not efficient with large lists - merge sort would be a better choice



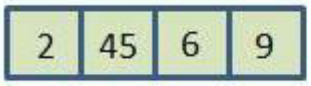
Sort algorithms

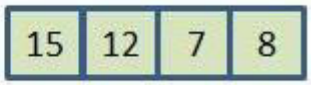
8. Merge Sort

Although bubble and insertion sorts work well on small lists of data, they are inefficient at sorting much larger lists.

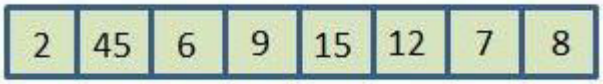
The merge sort was developed to handle the sorting of large lists. It does this by breaking them down into multiple smaller lists, quickly sorting them, and then merging them back together into one larger list

i.e. it is faster to sort these two lists then merge them back together



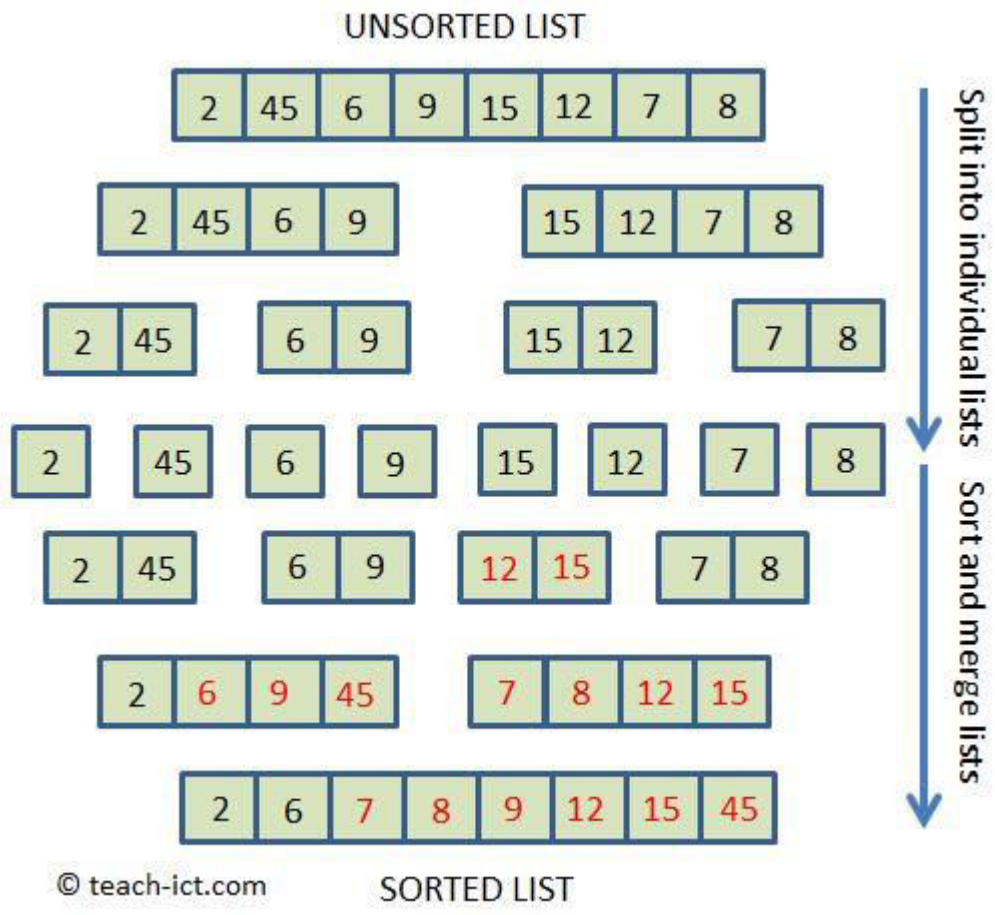


than to sort the list in its entirety.



Merge sort is an example of a **'divide-and-conquer'** algorithm because it splits down a larger problem into a number of smaller ones which are then solved. Each solution is then combined in some way to solve the larger problem.

Let's take a quick look at an example of a merge sort and then on the next page we will break it down into its different stages to help you understand what is happening, and why:



The diagram above showing the first stage of the merge sort is to keep on splitting the lists until they are only 1 item long. Then each list is first sorted and then re-combined to form a fully sorted final list.

Sort algorithms

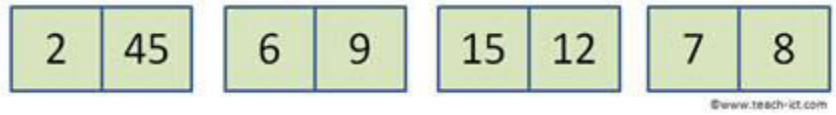
9. Merge sort - step-by-step

We start off with a list of unordered numbers which we want to sort in ascending order:

Step 1: Divide the above list into two smaller lists:

Had there been 9 numbers then we would have a list of 4 items and a list of 5 items.

Step 2: Divide these lists into smaller, equal sized lists (dealing with an odd number if necessary):



As a human, we can look at this pattern and know that there are only two items left, meaning we could theoretically stop here and begin to sort and merge the data.

However, it is difficult for a computer to "look ahead" like this. It works out to be more efficient to simply let the computer run to the end, splitting lists down to 1 or 0 items. So we continue splitting the list until there is only one item per list.

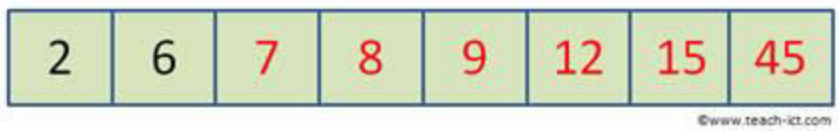
Step 3: Continue to split each list until there is only one item per list (or zero in some cases for uneven lists):

Step 4: Now that the list has been split as far as possible, we can begin to merge and sort the data items:

Two adjacent lists are paired back together and they are sorted (for this example, in ascending order). The ones that changed are the 12,15 pair

Step 5: Two more adjacent lists are merged together and the items within each list are sorted, the red numbers changed position:

Step 6: This process of merging and sorting lists continues until all of the individual lists are merged together and just one list remains. Within this list, all of the data items will be sorted into the correct order:



Sort algorithms

10. Merge sort pros and cons

With a 'divide-and-conquer' algorithm, there is a lot of repeating the same steps in a similar way. In this case the algorithm says 'Keep on dividing the list' and 'keep on merging and sorting'.

The statement 'keep on ...' is a very common and powerful idea in computer programming. There is a word for it: **'recursion'**.
A *recursive procedure* is one that calls itself with slightly different arguments until a stop condition is met.
In order to code the merge sort algorithm efficiently, a recursive procedure is used that keeps on splitting the list and another one is used that keeps on merging and sorting the lists. The pseudocode for this is quite complicated and is unlikely to be asked for in an exam - but here it is.

Advantages

- It is the fastest of three types of sort (bubble, insert, merge)
- It is the best option to use for long lists of data (more than 1000 long)

Disadvantages

- More complicated to code compared to bubble and insert
- It may use twice the memory size of the list - depending on the way it is coded. This becomes important if the list is millions of items long.

Sort algorithms

11. Comparison of different sorting methods

- The main decision as to which sort to use is based on the length of the list to be sorted. In summary

Sort Algorithms		
Sort	**Advantages**	**Disadvantages**
Bubble	Simple to understand and code for. Good for small lists	Slowest performance.
Insertion	Simple to code for and is faster than Bubble.	Not efficient for large lists.
Merge	Fastest performance compared to Insertion and Bubble. Excellent performance for large lists (1000+ items). It is also the fastest for **unordered** lists.	Harder to code for. Overkill / too complicated for short lists