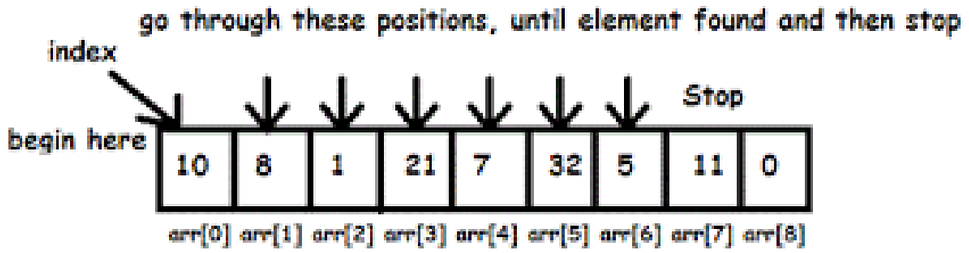
Linear Search Algorithm

Very simple algorithm

Described as a '**brute force**' as it simply looks at every possible item to see if it is the solution

Starts at the beginning of the array and goes through it, item by item, until it finds the data it is looking for or reaches the end of the array without finding it.

The linear search is sequential as it moves through the list item by item.



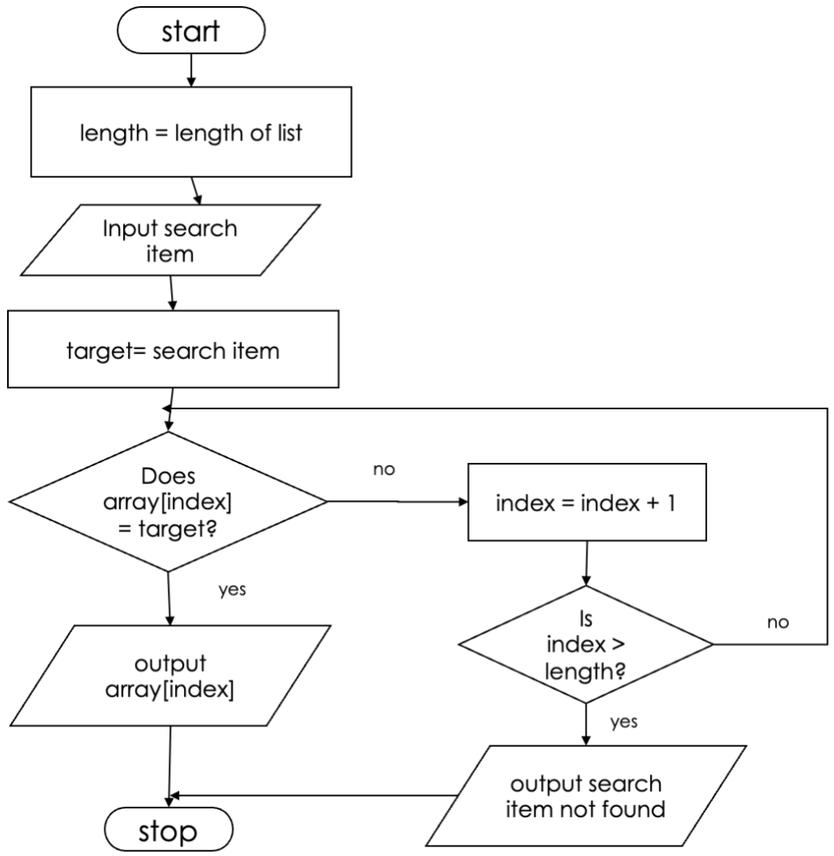
Element to search : 5

Steps for the linear search:

1. Starts at the first item in the list
2. Compare the item with the search item
3. If they are the same, then stop
4. If they are not, then move to the next item.
5. Repeat 2 and 4 until the end of the list is reached

Algorithm

```
alist = ['A', 'F', 'B', 'E', 'D','G','C']
position = 0
found = False
item =input()
WHILE position < len(alist) AND found == False
   IF  item == alist[position] then
          print ("Item found")
          found = True
    ELSE
          position = position + 1
    ENDIF
END WHILE
If found == False then
    print("Item not found")
ENDIF
```



Sorting and Searching Algorithms Knowledge Organiser

Binary Search Algorithm

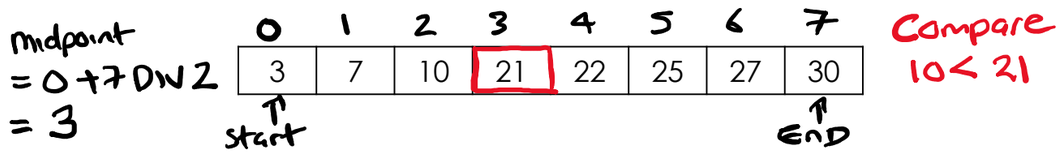
Binary search is a **divide and conquer** algorithm.

Steps for the binary search

1. Sort list
2. Find the midpoint
3. If the search item is found then stop
4. If the search item is less than the midpoint then search the sub-list to the left of the midpoint
5. If the search item is greater than the midpoint then search the sublist to the left of the midpoint
6. Repeat from step 2 until the target has been found.

Finding the midpoint requires adding the start and end pointers of the list and then using DIV to return a whole number (rounded down).

For example: mid = (0+7) // 2 = 3



Algorithm:

```
alist = [1,2,5,7,11,14]
item= input()
found = False
first = 0
last = len(alist) – 1
WHILE found = False AND first <= last
    midPoint = (first + last) DIV 2
    IF item ==alist[midpoint] then
        print (“item found at location”, midpoint)
        found = True
    ELSE
        IF item < alist[midpoint] then
            last = midpoint - 1
        ELSE
            first = midpoint + 1
        END IF
    END IF
END WHILE
if found == False then
    print(“Item not found”)
```

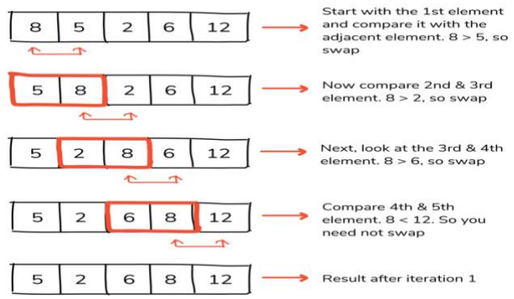
Efficiency of Searching Algorithms

Linear	Binary
Checks against each item until match using brute force, making it a slow algorithm. Works well on small, unsorted lists It can be improved by sorting the list first and can find an item quicker. Program requires less code	Splits lists and checks each half using divide and conquer List must be **sorted** Much quicker than a linear search because the data that needs to be searched halves with each step. Program requires more code

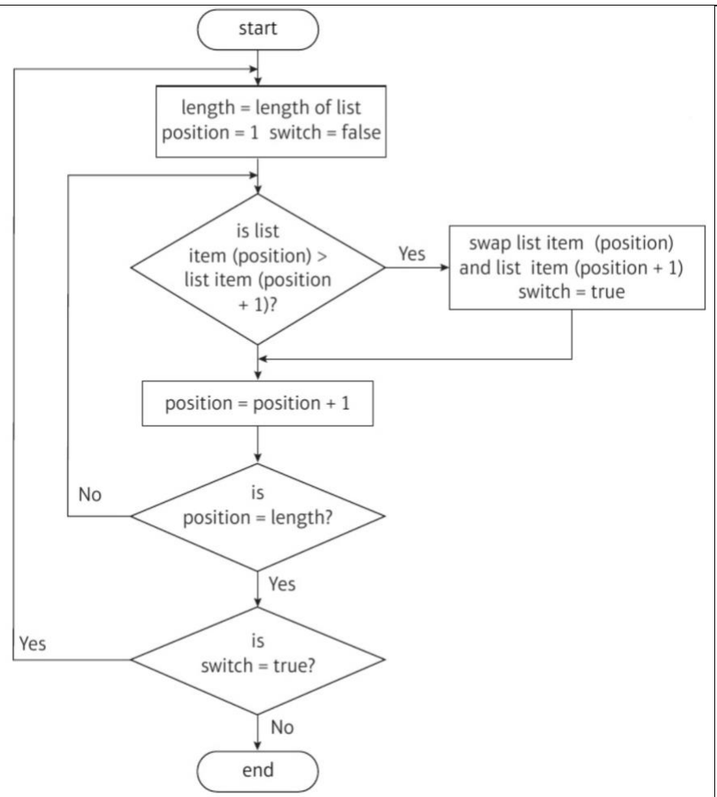
Sorting and Searching Algorithms Knowledge Organiser

Describe the steps of the bubble sort

1. Start at the beginning of the list.
2. Compare the first value in the list with the next one up. If the first value is bigger, swap the positions of the two values.
3. Move to the second value in the list. Again, compare this value with the next and swap if the value is bigger.
4. Keep going until the there are no more items to compare. Note - the last item checked in the list is now sorted, so ignore this next time.
5. Go back to the start of the list.



Each run through the list, from start to finish, is known as a **pass**.
The bubble sort continues until a pass is made where no values have been swapped. At this point, the list is sorted.




```
#Global variables
numbers = [8, 4, 2, 6, 1,3,5,7]
temp = 0
swapped = True

#Main program
length = len(numbers)
while swapped == True:
    swapped = False
    for pos in range (0,length-1):
        if numbers[pos] > numbers[pos+1]:
            temp = numbers[pos]
            numbers[pos] = numbers[pos+1]
            numbers[pos+1] = temp
            swapped = True
print(numbers)
```

Will continue to loop through the list until swapped == False No swaps have occurred during one pass through the list

Make a **traversal pass**

Comparing the next two items

Swap items using a temp variable

Worked Example

Number of passes : 5

0	1	2	3	4	Compares	Swaps	Pass
Lake	Grass	Tree	Rock	Flower	–	–	–
Grass	Lake	Rock	Flower	Tree	4	3	1
Grass	Lake	Flower	Rock	Tree	4	1	2
Grass	Flower	Lake	Rock	Tree	4	1	3
Flower	Grass	Lake	Rock	Tree	4	1	4
Flower	Grass	Lake	Rock	Tree	4	0	5

Sorting and Searching Algorithms Knowledge Organiser

Merge

Merge sorts a list of data by splitting the list into two halves, then splitting again and again until only pairs of data are left. It then sorts the pairs and puts them back together, sorting each subsequent pair again. **It's very fast on large data sets. But uses the most memory than the other sorting algorithms.**

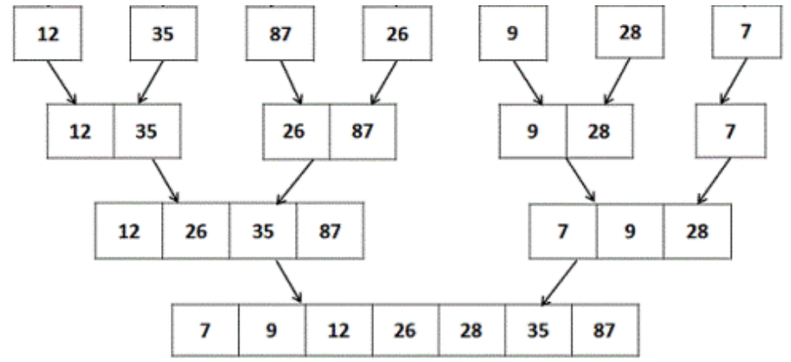
Step 1: Split the arrays into sub-arrays of 1 element.

Step 2: Take each sub-array and merge into a new, sorted array.

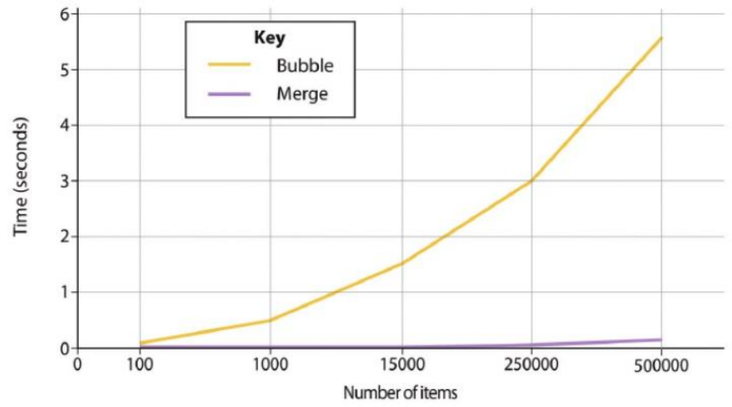
Step 3: Repeat this process until a final, sorted array is produced.

Output: A sorted array

12	35	87	26	9	28	7



Algorithm Efficiency
Comparison of the Bubble and Merge Sort



Bubble Sort

The bubble algorithm is said to be using **brute force** because it starts at the beginning and completes the same task over and over again until it found a solution.

Bubble sort is an **in-place sort** because it does not make copies of the array and does not take up additional memory.

However the number of comparisons is high and not suitable for large lists.

The bubble sort will probably take less time to write because is a simple algorithm.

Merge Sort

The merge sort uses the '**divide and conquer**' method because it repeatedly breaks down the the problem into smaller sub-problems, solves those and then combines the solution.

Merge sort **makes copies** of sub-lists during the divide phase and **additional memory** is required.

The number of comparisons for the merge sort grows in relation to the number of items in the list, however it grows slowly. This means it is efficient for large sets of data.

The merge sort will take more time to write and debug because it is a more complex algorithm.