# Algorithms

An **algorithm** is a sequence of ordered instructions that are followed step-by-step to solve a problem. This does *not* need to be on a computer.

**Decomposition** is the breaking down of a complex problem into smaller more manageable problems that are easier to solve.

**Abstraction** allows us to remove unnecessary detail from a problem leaving us with only the relevant parts of a problem thereby making it easier to solve.

**Algorithm Efficiency** More than one algorithm can be used to solve the same problem. Normally we use the algorithm that solves the problem in the quickest time with the fewest operations or makes use of the least amount of memory.

**Dry run testing** is carried out using **trace tables**. The purpose of the trace tables is for the programmer to track the value of the variables and outputs at each step of the program and to track how they change throughout the running of the program.

## Flowchart Symbols

We can represent algorithms using flowcharts

**Start and Stop**

Start    Stop

**Process – An operation that the algorithm performs**

Process

**Connector – Links all the other symbols together**

⟶

**Input and Output of data that is read in and written out**

Input/Output

**Decision is the same as a selection (if then … else)**



```
IF answer is "yes" THEN
        do something
ELSE IF answer is "no"
        do something else
ENDIF
```

# Pseudocode

We can represent algorithms using pseudocode

| | Example | Python equivalent |
|---|---|---|
| **Variable assignment** | a ← 10 | a = 10 |
| **Constant assignment** | constant PI ← 3.142 | PI = 3.142 |
| **Input** | a ← USERINPUT | a = input() |
| **Output** | OUTPUT "Bye" | print("Bye") |
| **Arithmetic Operators** Add Multiply Divide Subtract Integer division Modulus (remainder) | + * / - a ← 7 DIV 2 a ← 7 MOD 2 | + * / - a= 7 // 2 a = 7 % 2 |
| **Relational Operators** Less than Greater than Equal to Not equal to Less than or equal to Greater than or equal to | < > = ≠  or <> ≤ ≥ | < > == != <= >= |
| **Boolean Operators** AND OR NOT | AND OR NOT | AND OR NOT |
| **Selection** if .. | IF i > 2 THEN     j ← 10 ENDIF | if i  > 2:     j=10 |
| if .. else … | IF i > 2  THEN     j ← 10 ELSE     j ← 3 ENDIF | if i > 2:     j=10 else:     j=3 |
| if … else if … else | IF i ==2 THEN     j ← 10 ELSE IF i==3 THEN | if i ==2:     j=10 elif i==3:     j=3 |

```
    j ← 3
ELSE
    j ← 1
ENDIF
```

```
else:
    j=1
```

| | | |
|---|---|---|
| **Iteration** While loops | a ← 1 WHILE a < 4     OUTPUT a     a ← a + 1 ENDWHILE | while a<4:   print(a)   a=a+1 |
| For loops | FOR a ← 0 TO 3     OUTPUT a ENDFOR a ← 1 | for a in range(3):   print(a) |
| Repeat loops | REPEAT     OUTPUT a     a ← a + 1 UNTIL a←4 | |
| **Subroutines** procedure | SUB hello()   OUTPUT "hello" ENDSUB | def hello():   print("hello") |
| Function (with paramerters and return) | SUB add(n)   a ← 0   FOR a ← 0 TO n   a ← a + n   ENDFOR   RETURN a ENDSUB | def add(n):   a=0   for a in range(n+1):   a=a+n   return a |
| **Built-in functions** Length of array | LEN(a) | len(a) |
| Random integer | RANDOM_INT(0, 9) | import random random.randint(0,9) |

## Searching Algorithms

### Linear Search Algorithm

- The purpose of the linear search algorithm is to find a target item within a list.
- Compares each list item one-by-one against the target until the match has been found and returns the position of the item in the list.
- If all items have been checked and the search item is not in the list then the program will run through to the end of the list and return a suitable message indicating that the item is not in the list.
- The algorithm runs in linear time. If $n$ is the length of the list, then at worst the algorithm will make $n$ comparisons. At best it will make 1 comparison and on average it will make $(n+1)/2$ comparisons.
- The performance of the algorithm will be improved if the target item is near the start of the list.

*Example*
Find the position of letter "Z" within the following list. Assume we do not have visibility of the list

| Index position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Value | V | A | S | Z | X | R | T | G |

We compare it with the value in index position 0. We find that the value is "V" so we need to move on to the next index position. At index position 1 and 2 we still have not found Z. However, we get to index position 3 and we compare the target with the value and we find that they match, so the algorithm returns the index position and stops.

*Pseudocode*
```
i ← 0
x ← len(listOfItems)
pos ← -1
found ← False
WHILE i < x AND NOT found
  IF listOfItems[i] == itemSearch THEN
    found ← True
    pos ← i + 1
  ENDIF
  i=i+1
ENDWHILE
OUTPUT pos
```

### Binary Search Algorithm

- The binary search algorithm works on a sorted list by identifying the middle value in the list and comparing it with the search item.
- If the search item is smaller the mid element becomes the new high value for the search area.
- If the search item is larger the mid element becomes the low value for the search area.
- The keeps repeating until the search item is found.
- When the search item is found the index position of the item is returned.
- At each iteration the search are halved in size consequently this is an efficient algorithm.

*Example: Binary search in operation to find 81*



*Pseudocode*
```
low ← 1
high ← LENGTH(arr)
mid ← (low + high) DIV 2
WHILE val ≠ arr[mid]
  IF  arr[mid] < val THEN
    low ← mid
  ELIF arr[mid] > val THEN
    high ← mid
  ENDIF
  mid ← (low + high) DIV 2
  ENDWHILE
OUTPUT mid
```

### Linear search versus binary search

| | Advantages | Disadvantages |
|---|---|---|
| Linear Search | - Very simple algorithm and easy to implement<br>- No sorting required<br>- Good for short lists | - slow because it searchers through the whole list<br>- very inefficient for long lists |
| Binary Search | - much quicker than linear search, because it halves the search zone each step | - The list need to be ordered |

# Sorting Algorithms

## Bubble Sort

- The purpose of sorting algorithms is to order an unordered list. Item can be ordered alphabetically or by number.
- Bubble sort steps through a list and compares pairs of adjacent numbers. The numbers are swapped if they are in the wrong order. For an ascending list if the left number is bigger than the right number the items are swapped otherwise the numbers are not swapped.
- The algorithm repeatedly passes through the list until no more swaps are needed.

*Example*

*Sort the following sequence in ascending order using bubble sort: 5,3,4,1,2.*

| Pass 1 | 5 | 3 | 4 | 1 | 2 | |
|---|---|---|---|---|---|---|
| | 3 | 5 | 4 | 1 | 2 | Compare 5 and 3 – swap |
| | 3 | 4 | 5 | 1 | 2 | Compare 5 and 4 – swap |
| | 3 | 4 | 1 | 5 | 2 | Compare 5 and 1 – swap |
| | 3 | 4 | 1 | 2 | 5 | Compare 5 and 2 – swap; end of pass 1 |
| Pass 2 | 3 | 4 | 1 | 2 | 5 | Compare 3 and 4 – no swap |
| | 3 | 1 | 4 | 2 | 5 | Compare 4 and 1 – swap |
| | 3 | 1 | 2 | 4 | 5 | Compare 4 and 2 – swap |
| | 3 | 1 | 2 | 4 | 5 | Compare 4 and 5 – no swap; end of pass 2 |
| Pass 3 | 1 | 3 | 2 | 4 | 5 | Compare 3 and 1 – swap |
| | 1 | 2 | 3 | 4 | 5 | Compare 3 and 2 – swap |
| | 1 | 2 | 3 | 4 | 5 | Compare 3 and 4 – no swap |
| | 1 | 2 | 3 | 4 | 5 | Compare 4 and 5 – no swap; end of pass 3 |
| | 1 | 2 | 3 | 4 | 5 | |

## Bubble sort Pseudocode

```
A=[5,3,4,1,2]
sorted ← False
WHILE not sorted
  sorted ← True
  FOR I TO LEN(A)-1:
    IF A[i] > A[i+1]:
      temp ← A[i]
      A[i] ← A[i+1]
      A[i+1] ← temp
      sorted ← False
    ENDIF
  ENDFOR
ENDWHILE
OUTPUT A
```

## Merge Sort

- Merge sort is a type of divide and conquer algorithm.
- There are two steps: divide and combine
- Merge sort works by dividing the unsorted list sublists. It keeps on doing this until there is 1 item in each list.
- Pairs of sublists are combined into an ordered list containing all items in the two sublists. The algorithm keeps going until there is only 1 ordered list remaining.
- Merge sort is a recursive function, that calls itself.

*Step 1: Divide*
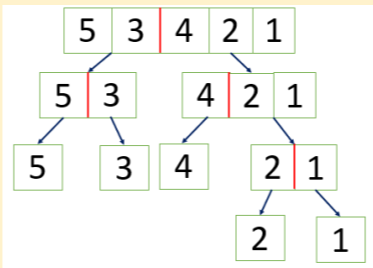


Keep dividing until there is only 1 item in each list

*Step2: Combine*
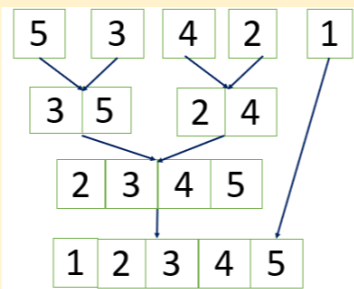


1. The first items in the two sublists are compared, and the smallest value is copied to the parent list.
2. The copied item is then removed from the sublist.
3. When there are no items left in one of the sublists the remaining items in the other sublist are them copied in order to the parent list.

## Merge sort Versus Bubble sort

| | Advantages | Disadvantages |
|---|---|---|
| **Bubble sort** | Very simple and robust algorithm | Can be slow particularly for long lists. As the number of items increases the time taken for the algorithm to run increases dramatically. |
| **Merge sort** | Much faster than bubble sort especially when the number of elements is large | More complex to understand Step 1: Divide Step 2: Combine |