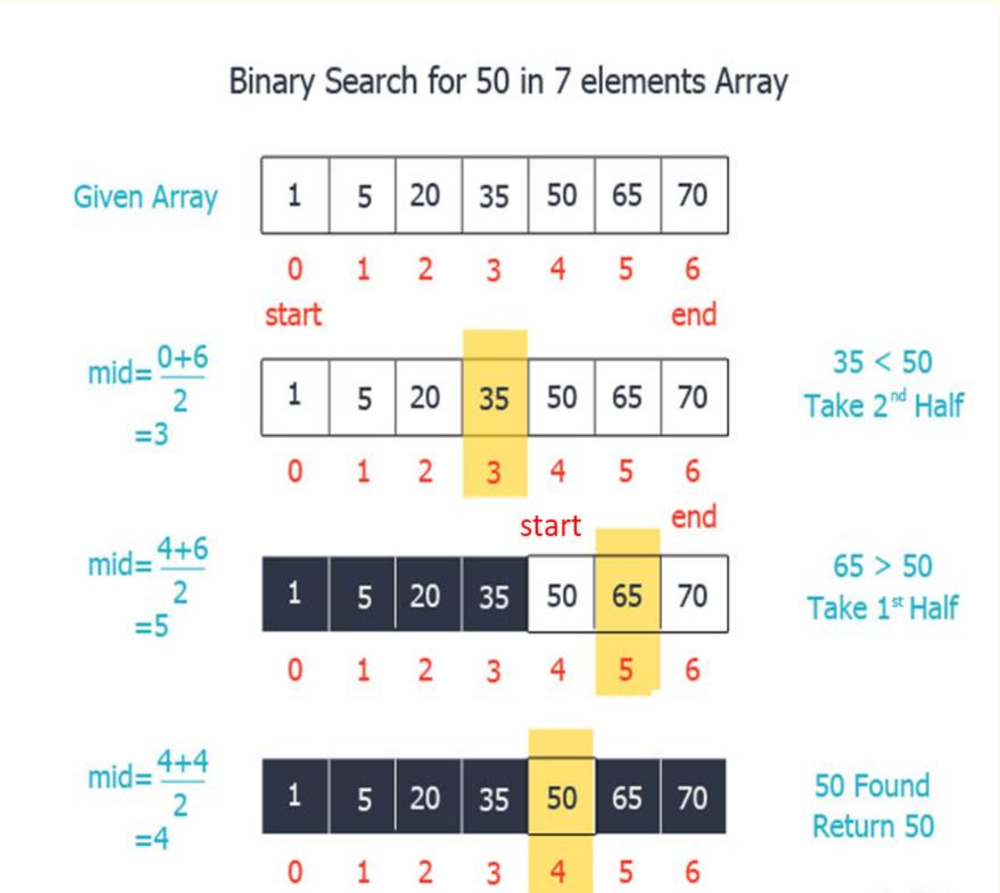
Searching algorithms are specific algorithms a computer can use to **search** for **data** in an **array** (list).

Binary Search

Used to efficiently search large sorted lists of data for a specific item.

Example: Binary Search for 51



Advantages
Binary searches are more suitable for large lists.
In general takes less steps than a linear search.

Disadvantages
Not suitable for small lists.
They can only be used on ordered lists

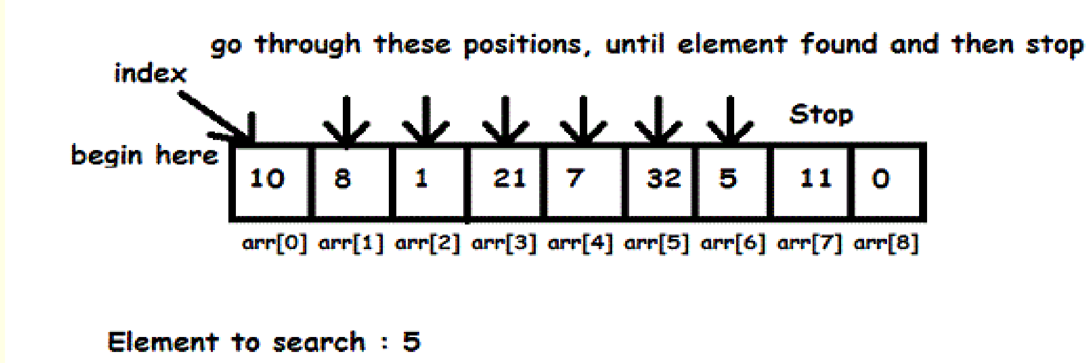
Algorithm:

```
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”)
```

Complexity: O(log N) logarithmic

Linear Search

Linear search is an algorithm that can used on any list to find a specific item. To complete a linear search you simply start with the first item and compare it to the search item.
This process is repeated till the search item has been found.



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
```

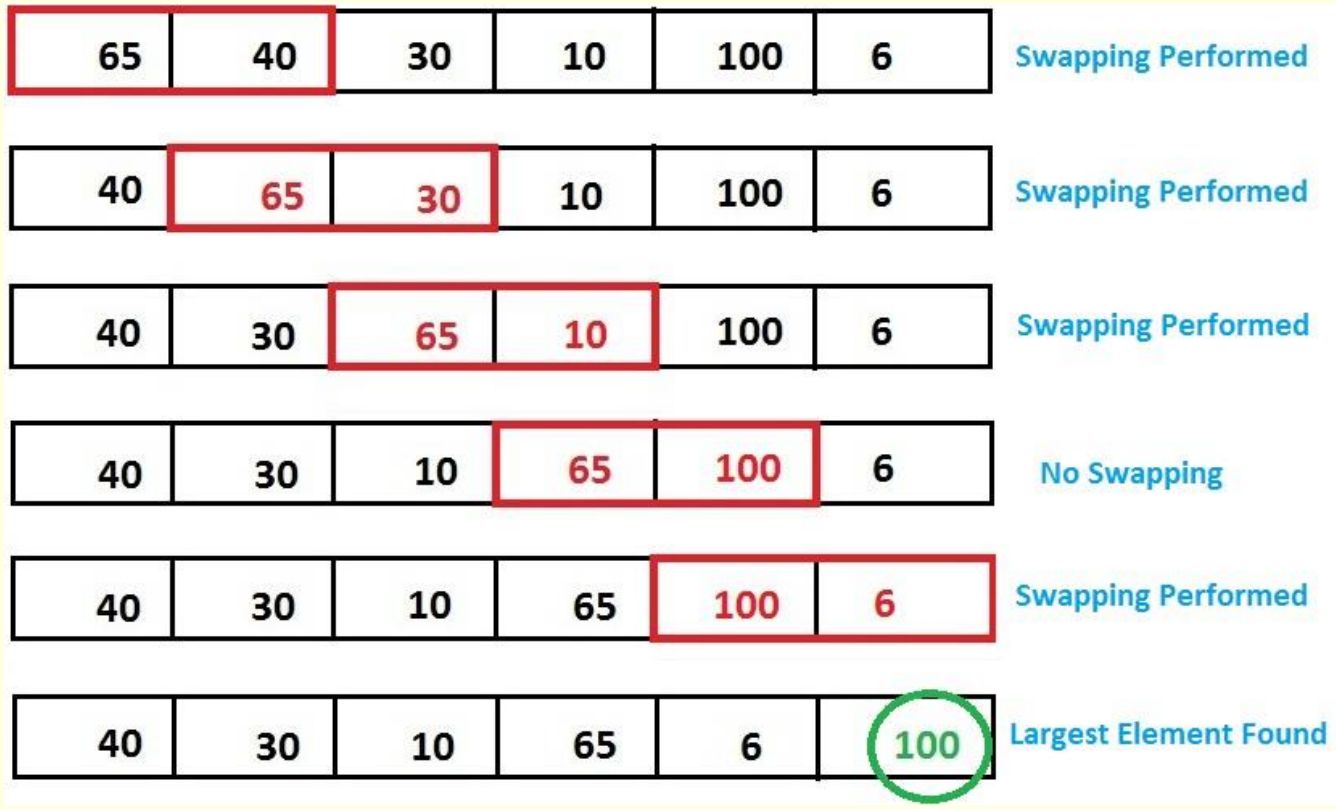
Advantages
Linear search can be used on any list.
Very efficient on small lists.

Disadvantages:
This algorithm is not very efficient on large lists.
Usually takes more steps than a binary search.

Complexity: O(N) – Linear

Bubble sort

To complete this sort you must compare the first and second item of the list, if the right item is the smallest swap the items around.



Advantages

This algorithm works very well for small lists.

Disadvantages:

Even when the list is sorted each number still needs comparing to check. This is very slow to run as the algorithm has to do multiple passes of the data.

This algorithm is very inefficient for large sets of data.

Algorithm

```
swapMade = True
WHILE swapMade
      swapMade = False
      position = 0
      FOR position=0 to listLength-2
            IF list[position]>list[position+1]then
                  temp = list[position]
                  list[position] = list[position+1]
                  list[position+1] = temp
                  swapMade = True
            ENDIF
      END FOR
END WHILE
```

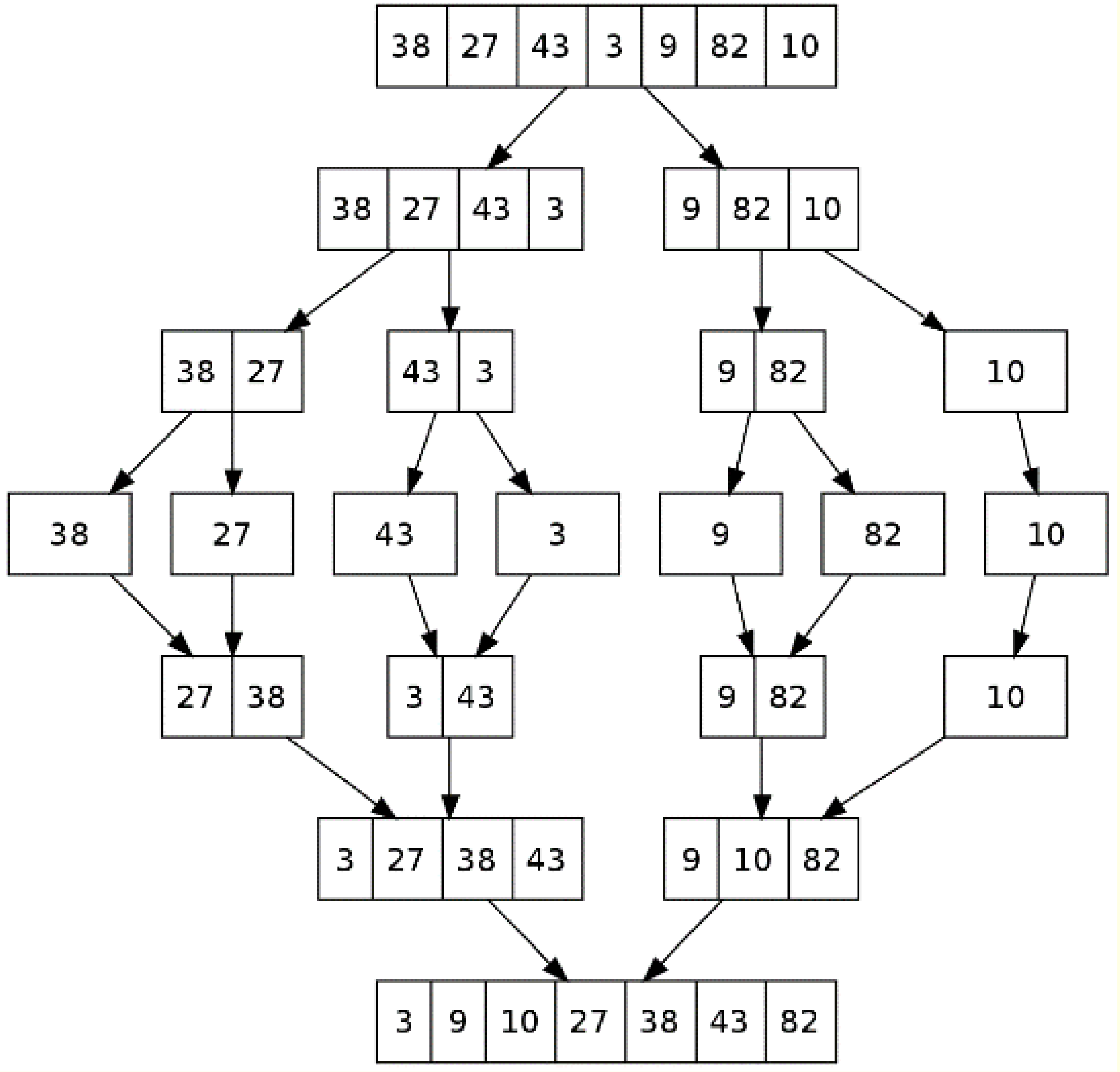
It will continue to pass through the data until it has no swaps

Loops through the elements in the list

Compare the first and second item and makes a swap if first item is greater than second item.

Complexity: O(n^2) polynomial

Merge Sort Can be used efficiently on very large lists of data.



Algorithm

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

Advantages

Very high performance on any list.

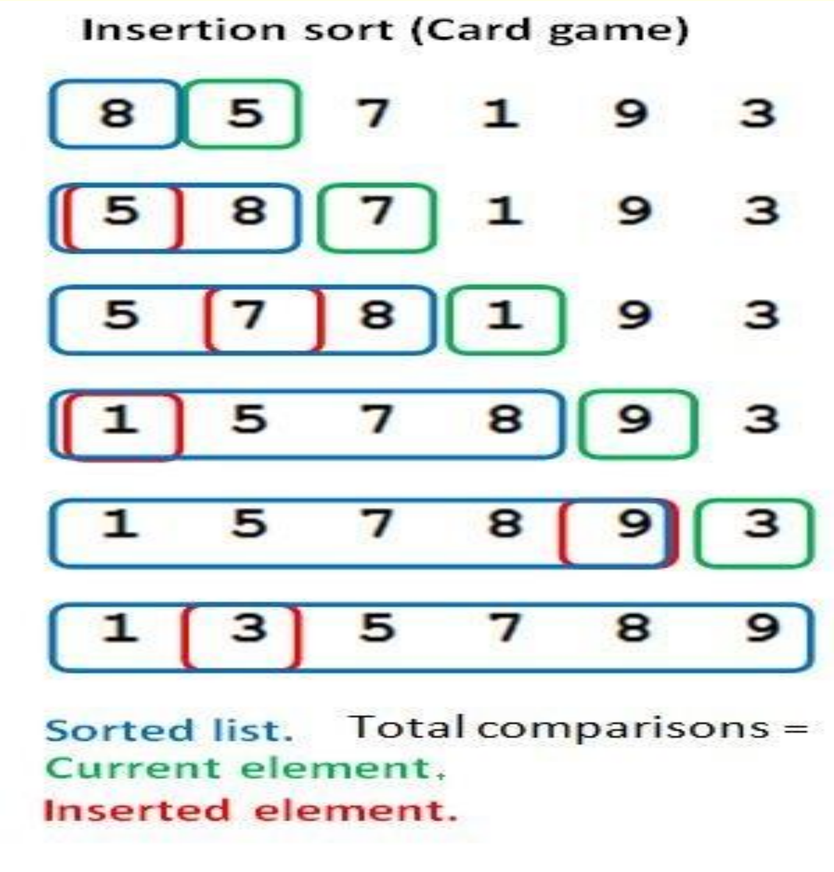
Disadvantages:

Uses a lot of memory space to run the algorithm.

Complexity

O(n*log(n))— logarithmic. This shows Merge Sort to be substantially faster then the Bubble and Selection Sort.

Insertion Sort - used to **sort** a live list of data.



Advantages

Very high performance in small lists.

This algorithm can work on live list where the data is still coming in.

Disadvantages:

Poor performance with large lists. Not as fast as a merge sort.

Algorithm

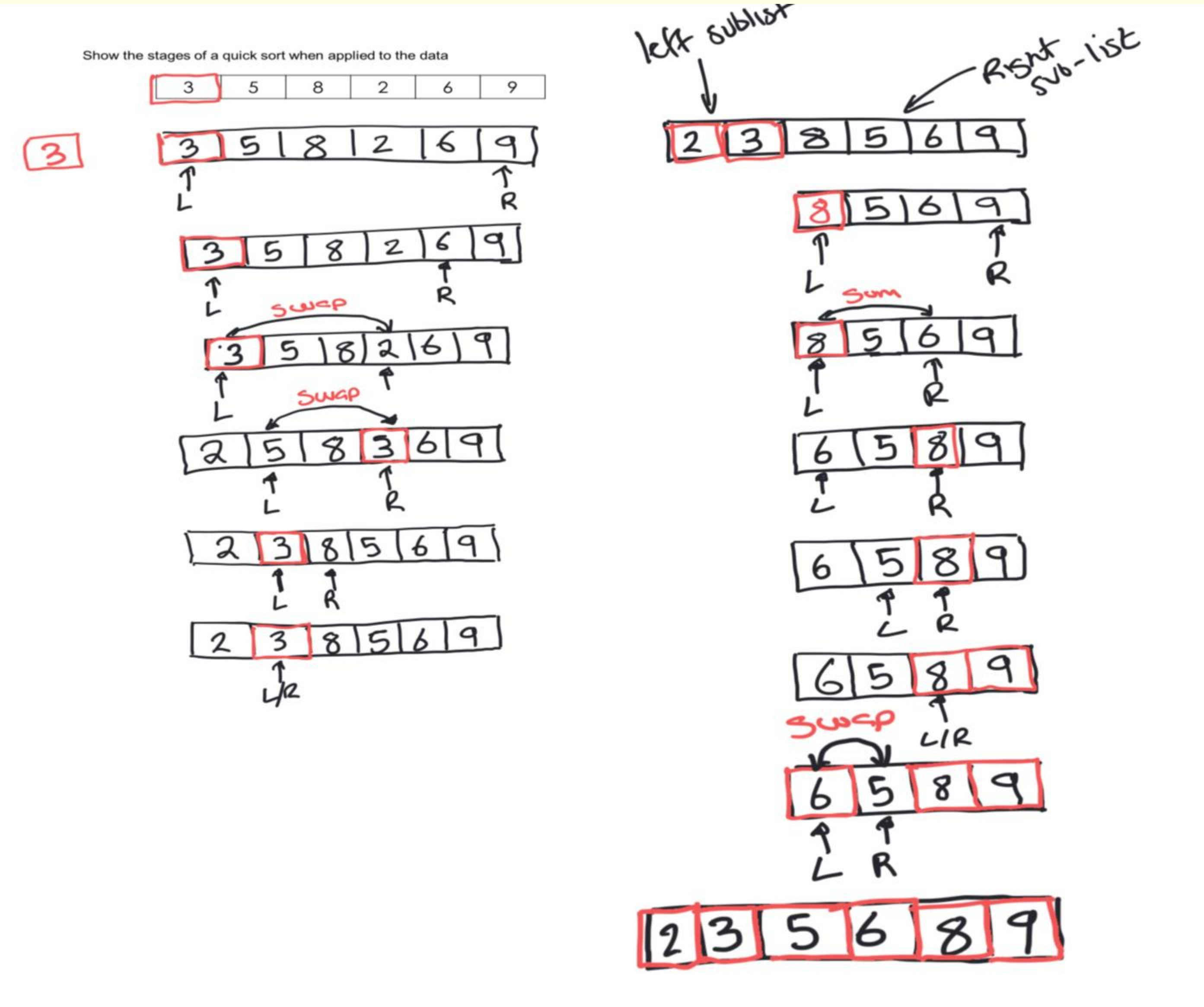
1. Look at the second item in the list
2. Compare it to all items before and insert the item in the correct place
3. Repeat step 2 until you get the end by moving to the next number and placing it into the correct place.

```
FOR position 1 to len(array -1)

    currentValue = array[position]
    pos = position

    WHILE pos > 0 AND array[pos-1] > currentValue:
        array[pos] = array[pos-1]
        pos = pos – 1

    array[pos] = currentValue
```

Complexity : O(n2) Polynomial

Quick Sort

Uses divide-and-conquer



Advantages can be much faster than bubble sort and insertion sort.

Disadvantage: inefficient in terms of memory for very large lists due to recursion (the stack can grow large with all the return addresses, variables etc that have to be stored for each recursive call). Sometimes it can grow too large causing a stack overflow (out of stack memory) error.

Complexity: O(n log(n))— logarithmic.

Algorithm

- Choose a pivot
- Set a left pointer and right pointer
- If current pointer is the right pointer
    - If right pointer data is less than pivot
        - Swap with right pointer data with left pointer data
        - Move left pointer along by 1
        - Set left pointer as current
    - Otherwise
        - Move right pointer back by one
- If current pointer is the left pointer
    - If leftpointer data is greater than pivot
        - Swap with right pointer data with left pointer data
        - Move right pointer back by1
        - Set right pointer as current
    - Otherwise
        - Move left pointer along by 1
- Continue until leftpointer == RightPointer
- Slot the pivot data into the leftpointer
- Repeat steps on the left half and the right half of the list till the entire list is sorted.

Mark scheme Example:

- Uses divide-and-conquer (1)

- Highlight first list element as start pointer, and last list element as end pointer
- Repeatedly compare numbers being pointed to…
- …if incorrect, swap and move end pointer
- …else move start pointer
- Split list into 2 sublists
- Quick sort each sublist
- Repeat until all sublists have only 1 number
- Combine sublists

Big O notation

What is Big O

Evaluate the complexity of the algorithm
Show how the time / memory / resources increase as the data size increases
Evaluate worst case scenario for the algorithm

Time Complexity - How the time scales as data size increases
Space Complexity – how much memory is required

	Notation	Description	Example code	Example use
	$O(1)$	**Constant**. Time does **not change** with input size. Efficient with any data set.	random_num = data_set(x)	Extracting data from any element from an array. Hashing algorithm.

	$O(\log N)$	**Logarithmic**.
		
		Algorithms that halves the data set each time and time grows **very slowly** as input grows.
		
		Because the algorithm halves each time from a large data it starts off with a really large search time then flattens out over time.
			While Found = False And LowerBound <= UpperBound
			MidPoint = LowerBound + (UpperBound - LowerBound) \ 2
			If data_set (MidPoint) = searchedFor Then
			Found = True
			ElseIf data_set (MidPoint) < searchedFor Then
			LowerBound = MidPoint + 1
			Else
			UpperBound = MidPoint - 1
			End If
			End While
				Binary search.

	$O(N)$	**Linear**. Time grows **proportionally** with input size.	For x = 1 To y data_set(x) = counter Next	A loop iterating through a single dimension array. Linear search.
	$O(n \log N)$	**Linearithmic:** The algorithm goes through all the items (**n**), and for each pass you do a small extra amount of work (**log n**).	Merge sort's time comes from **two separate actions**: • Splitting → log n (You keep dividing the list in half) • Merging → n At **each level of splitting**, you merge all elements back together. Every element is looked at **once per level**	Quick sort. Merge sort.
	$O(N^2)$	**Polynomial**. Time grows with the **square** of input size. Significantly reduces efficiency with increasingly large data sets. Deeper nested iterations result in $O(N^3)$, $O(N^4)$ etc. depending on the number of dimensions.	For x = 1 To w For y = 1 To z data_set(x, y) = 0 Next Next	A nested loop iterating through a two dimension array. Bubble sort.
	$O(2^N)$	**Exponential**. Time **doubles** with each extra input. Opposite to logarithmic. Inefficient.	Function fib(x) If x <= 1 Then Return x Return fib(x - 2) + fib(x - 1) End Function	Recursive functions with two calls. Fibonacci number calculation with recursion.

Searching algorithms	Time complexity		
	Best	Average	Worst
Linear search	O(1)	O(n)	O(n)
Binary search array	O(1)	O(log n)	O(log n)
Binary search tree	O(1)	O(log n)	O(n)
Hashing	O(1)	O(1)	O(n)
Breadth/Depth first of graph	O(1)	O(V+E) No. vertices + No. edges	O(V^2)

Sorting algorithms	Time complexity			Space complexity
	Best	Average	Worst	
Bubble sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick sort	O(n log n)	O(n log n)	O(n^2)	O(log n)

n	n^2 (Polynomial)	2^n (Exponential)
1	1	2
10	100	1,024
20	400	1,0485,76
30	900	1,073,741,824