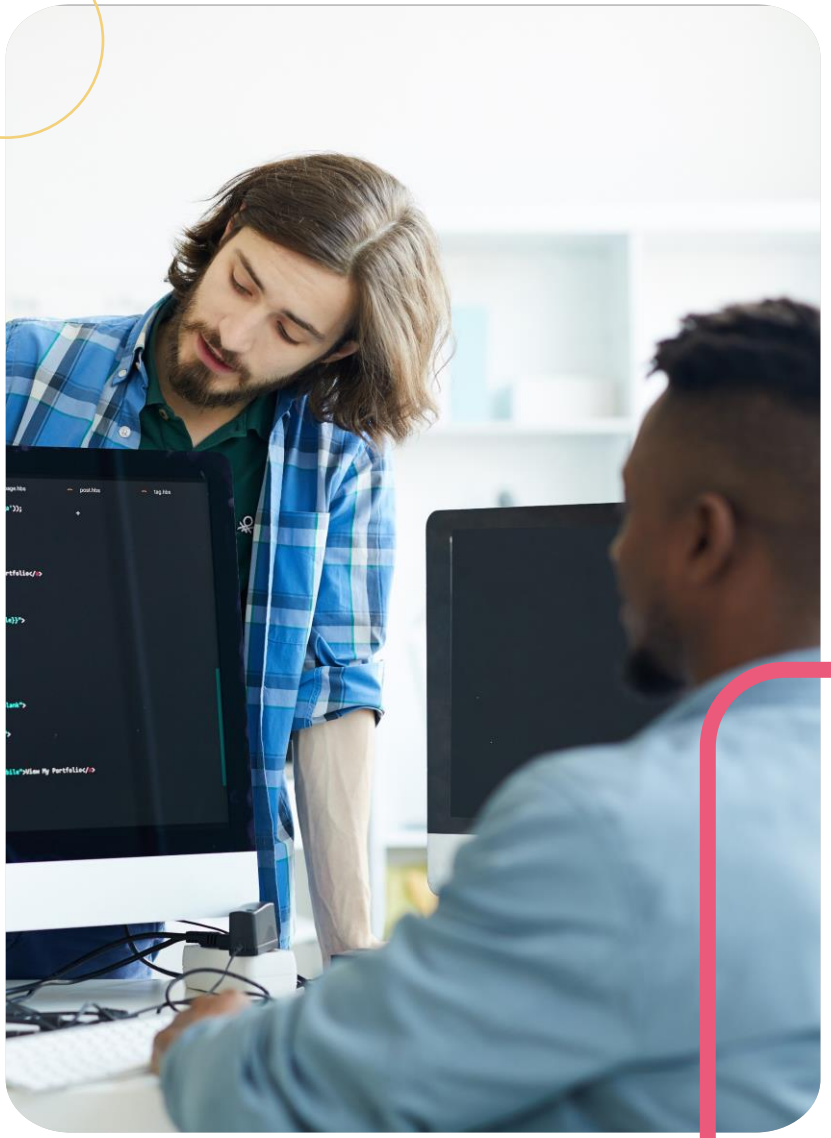
Diploma in

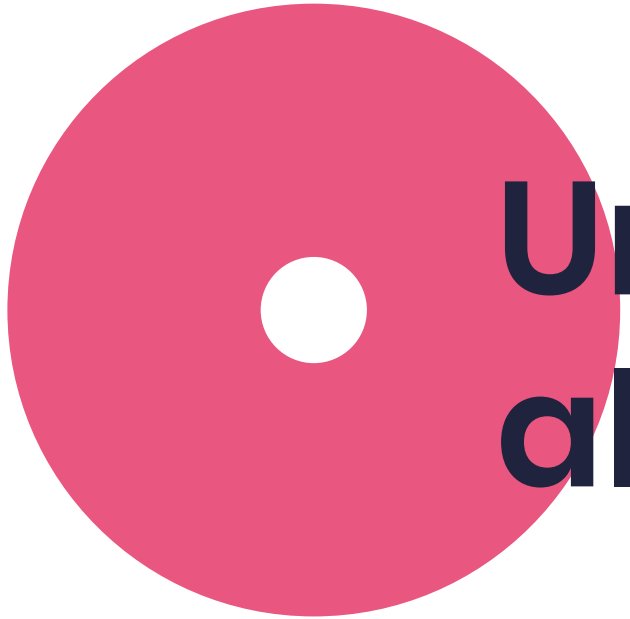
Computer Science

Algorithms



Objectives

- Discover what algorithms really are
- Appreciate the different types of algorithm complexities
- Understand how to search and sort algorithms
- Discuss the attributes of a good algorithm



Understanding algorithms

What is an algorithm?

Step-by-step sequence for doing something

Process be followed in calculations or problem-solving operations

A finite list of instructions

Why study algorithms?

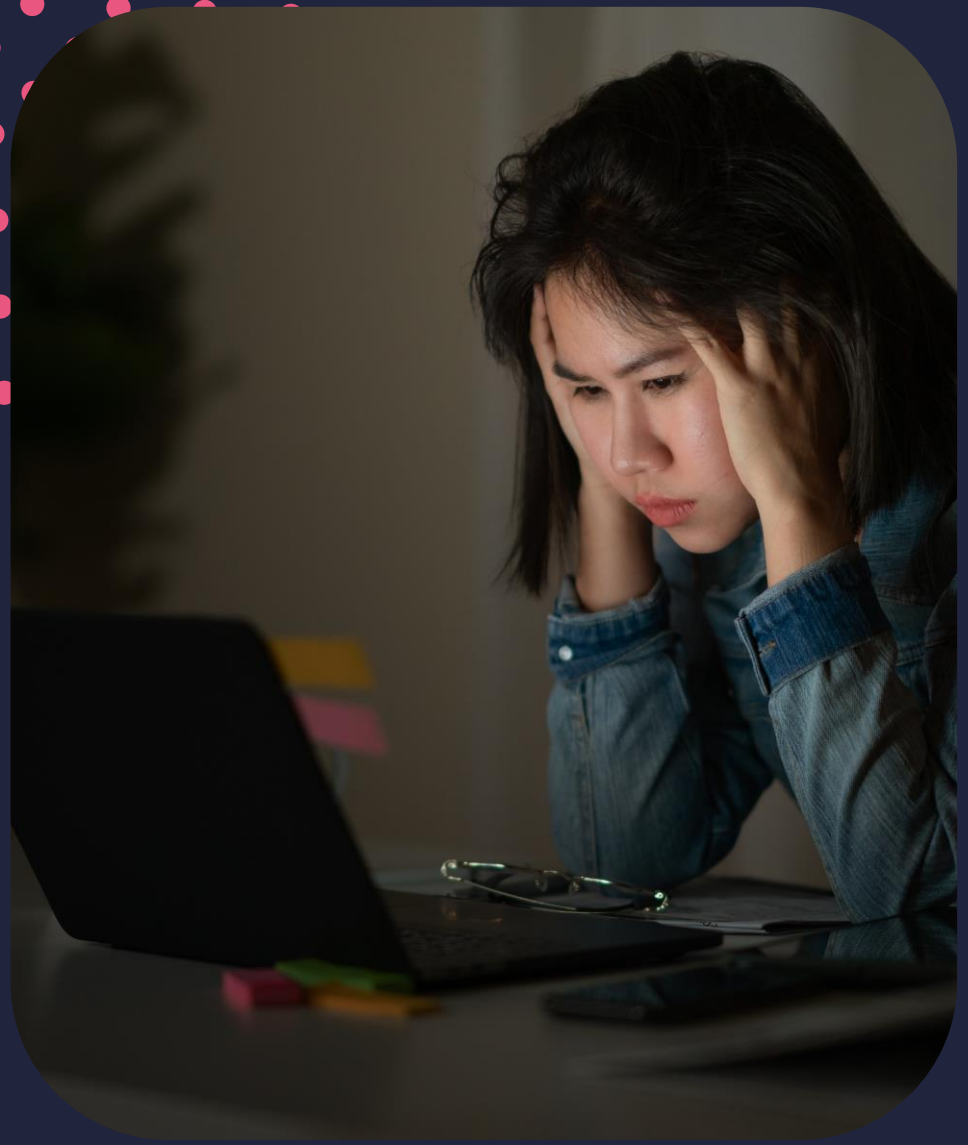
By studying algorithms you will write code with a greater understanding of the underlying logic.

Why use algorithms?

- Need to prove that a proposed solution is viable, can be executed and can terminate
- Need most optimal solution
- Software produced according to proper standards of software development
- Make computers more effective

Did you know?

Proficiency in algorithms is one of the most sought-after skills in the world.



Relationship of an algorithm to a problem

An algorithm is a tool for solving a well-specified computational problem.

Specification is everything when it comes to computing problems.

Algorithm vs problem

- Computer takes input and applies each step to produce output
- Flowchart follows path and tracks input
- An algorithm starts with a **problem statement**

Formulating an algorithm

Consists of three parts:

- **Input:** set of data that algorithm will work on
- **Process:** steps that computer will follow
- **Output:** result of process followed by computer

Tips for formulating an algorithm

- Don’t lose goal in lines of code
- Build in efficiency
- Choose best way to execute task
- Follow the optimal route

Algorithm complexity

A measure which evaluates the order of the count of operations performed by an algorithm as a function of the input data size.

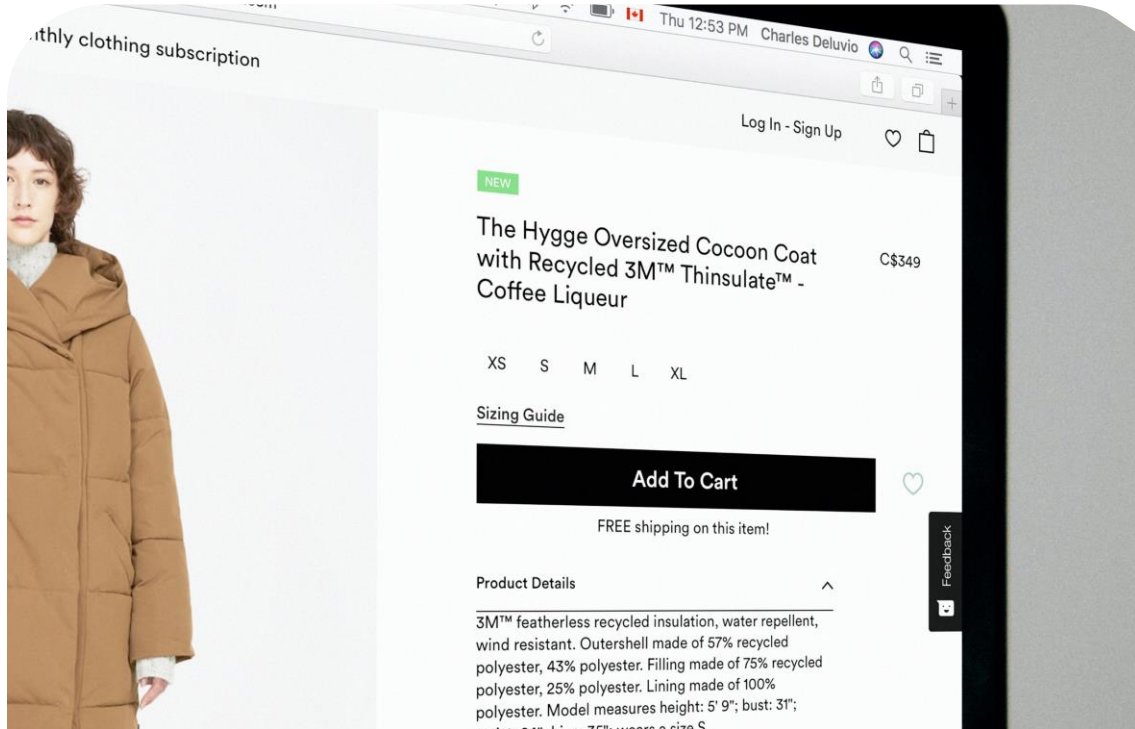
Approximates the number of steps the algorithm will take to produce output, given input of a certain size.

Algorithm complexity example

Which search algorithm takes the least time to complete the search?

Calculate complexities side by side to find best for site - not exact count but rather order of operation count

For example: an order of N^2 operations to process N elements, then $N^2/2$ and $3*N^2$ are of one and the same quadratic order



Types of algorithm complexities

Big O notation

Way of expressing efficiency of algorithms

Measures how fast code will run

Determines pros and cons of an algorithm

Big O notation

O (1) — constant

O (log n) — logarithmic

O(n) — linear

O (n log n) — n log n

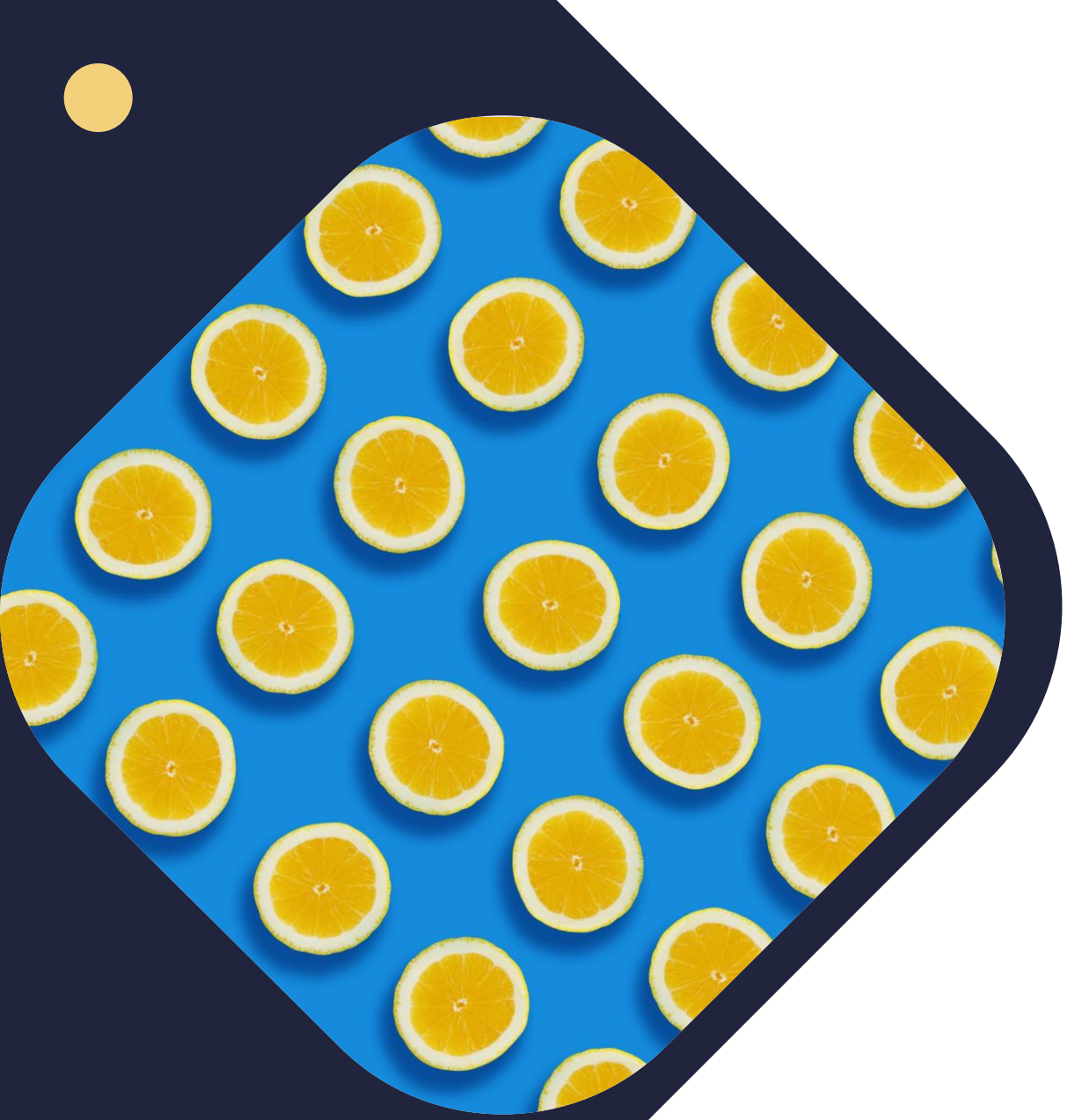
O(n^2) — quadratic

O(n^3) — cubic

O(2^n) — exponential

O(n!) — factorial

Snap
& save



Constant algorithm

- Takes a constant number of steps regardless of the input size

Logarithmic algorithm

- Takes the order of log(N)steps
- Base is often 2

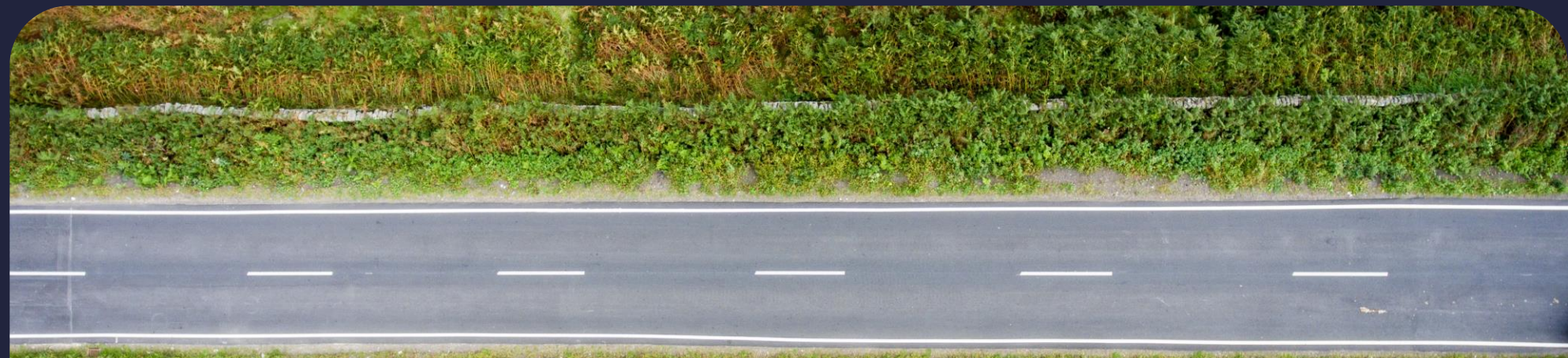
Example: If your algorithm has an input of 5 elements then the algorithm would take $O(\log_2(5))$, which would be approximately 2 steps. Similarly, for N=1,000,000 we would say $O(\log_2(1000000))$, which is approximately 20 steps.

Linear algorithm

Takes approximately same number of steps as the number of elements presented as input

Denoted as O(N)

Example: If you have 100 input elements, it takes approximately 100 steps. If you have 60000 input elements, you have approximately 60000 steps.



Quasilinear algorithm

Denoted as O(n*log(n))

Takes N*log(N) steps for performing a given operation on N elements

Between linear complexity and logarithmic complexity

Example: If you have 1,000 elements, it will take about 10,000 steps.
(set as per other examples)

Quadratic and cubic algorithms

In **quadratic,** takes the order of N^2 numbers of steps

Example: If N = 100, it takes about 10,000 steps.

In **cubic** functions, takes the order of N^3 numbers of steps

Example: If we have 100 elements, it takes about 1,000,000 steps.

Exponential algorithm

- Of the order $O(2^n)$, $O(N!)$, $O(n^k)$
- Escalate very quickly in relation to size of the input

Example: If N is 10, the value of $O(2^N)$ will be 1024.

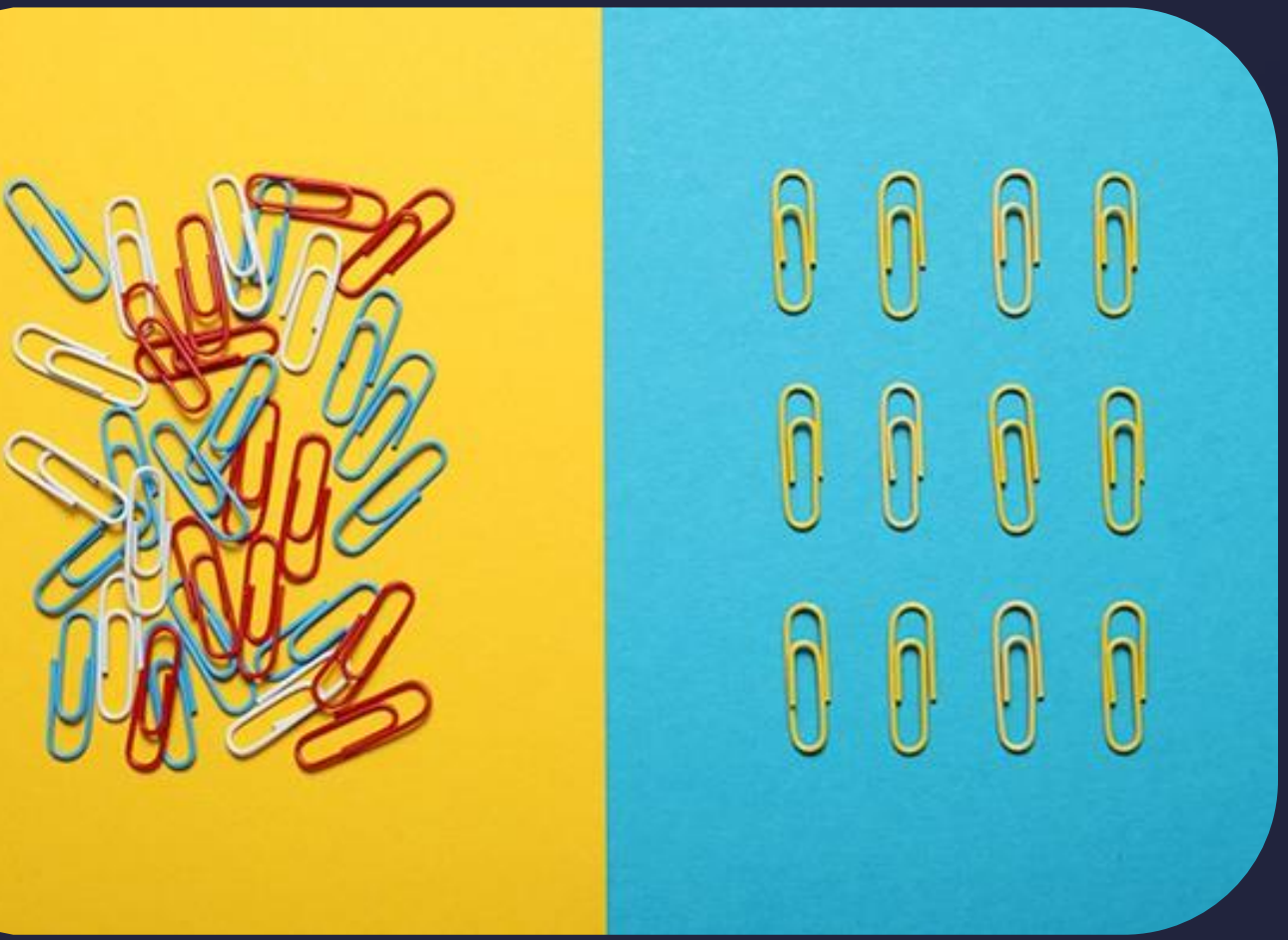
For N=20, $O(2^N)$ will be 1 048 576!

Now if we add just 10 more input elements to the processing queue, the value of $o(2^N)$ will be 1 048 576!

If we bump it to 120, then the value of $O(2^N)$ will be 1.33*10^36!

Summary of types of algorithm complexities

Complexity	Number of steps
Constant	O (1)
Logarithmic	O(log(N))
Linear	O(N)
Linearithmic /quasilinear	O(n*log(n))
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(2^n)$, O(N!), $O(n^k)$, …



Categorisation of algorithms

Best case – algorithm under optimal conditions

Average case – midway between min and max number of elements

Worst case – maximum number of operations executed

Typical computer perspective

Algorithm	10	20	50	100	1,000	10,000	100,000
O (1)	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
O(log(n))	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
O(n)	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
O(n*log(n))	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
$O(n^2)$	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	2 sec.	3-4 min.
$O(n^3)$	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	20 sec.	5.55 hours	231.5 days
$O(2^n)$	< 1 sec.	< 1 sec.	260 days	hangs	hangs	hangs	hangs
O(n!)	< 1 sec.	hangs	hangs	hangs	hangs	hangs	hangs
$O(n^n)$	3-4 min.	hangs	hangs	hangs	hangs	hangs	hangs

Analysing the problem

Algorithms with a constant, logarithmic or linear complexity are fast.

Algorithms of complexity O(n*log(n)) are nearly as fast.

Quadratic algorithms work very well up to several thousand elements.

Cubic algorithms get ugly beyond 1000.

Algorithm	10	20	50	100	1,000	10,000	100,000
O (1)	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
O(log(n))	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
O(n)	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
O(n*log(n))	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
$O(n^2)$	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	2 sec.	3-4 min.
$O(n^3)$	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	20 sec.	5.55 hours	231.5 days
$O(2^n)$	< 1 sec.	< 1 sec.	260 days	hangs	hangs	hangs	hangs
O(n!)	< 1 sec.	hangs	hangs	hangs	hangs	hangs	hangs
$O(n^n)$	3-4 min.	hangs	hangs	hangs	hangs	hangs	hangs

Analysing the problem

Polynomial algorithms are considered to be fast and work well for thousands of elements.

Exponential algorithms do not work well and should be avoided.

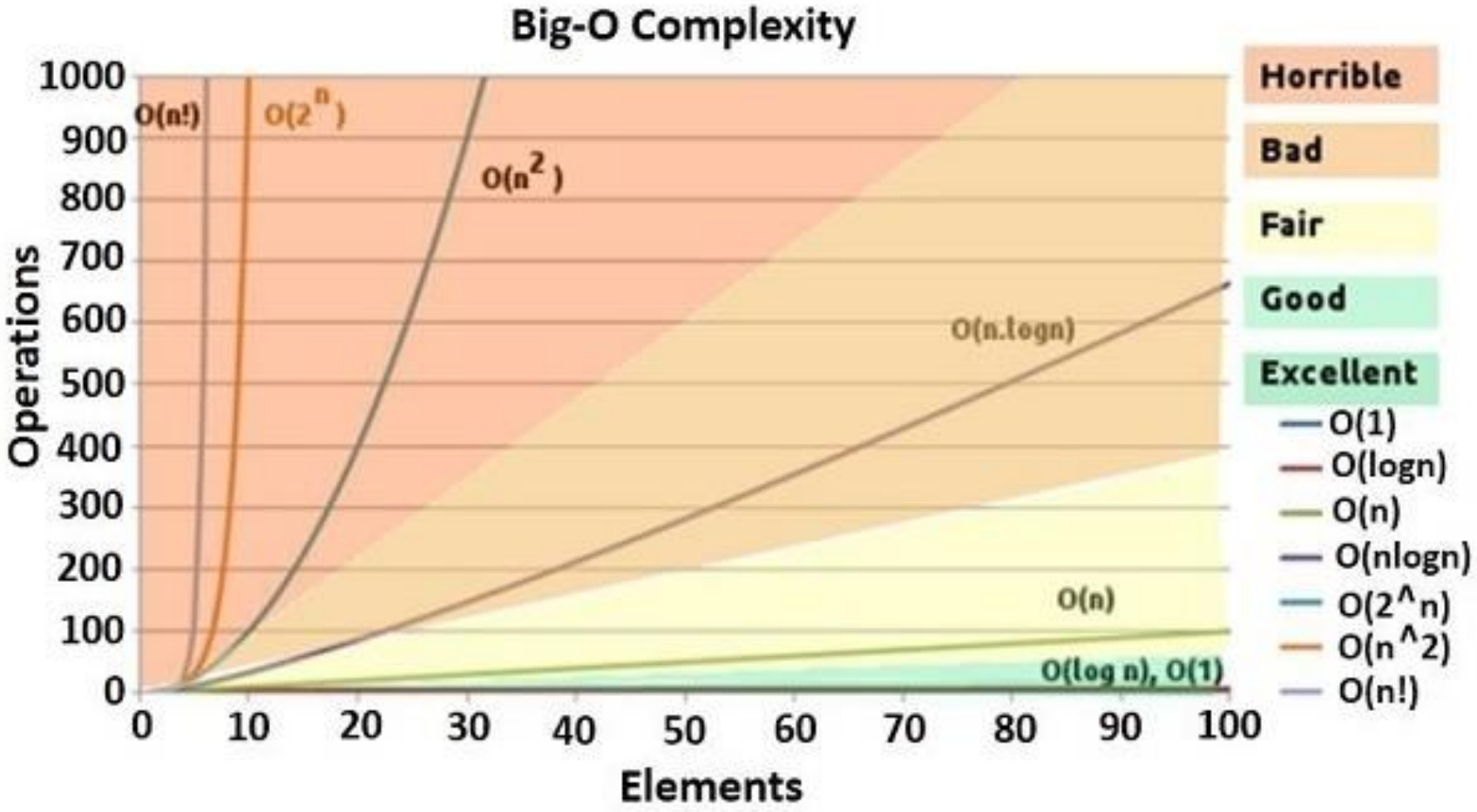
An exponential solution generally does not work.

Algorithm	10	20	50	100	1,000	10,000	100,000
O (1)	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
O(log(n))	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
O(n)	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
O(n*log(n))	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
$O(n^2)$	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	2 sec.	3-4 min.
$O(n^3)$	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	20 sec.	5.55 hours	231.5 days
$O(2^n)$	< 1 sec.	< 1 sec.	260 days	hangs	hangs	hangs	hangs
O(n!)	< 1 sec.	hangs	hangs	hangs	hangs	hangs	hangs
$O(n^n)$	3-4 min.	hangs	hangs	hangs	hangs	hangs	hangs

Did you know?

Research by Google has found that 53% of mobile website visitors will leave if a webpage doesn't load within three seconds.

Summary



www.bigocheatsheet.com

Algorithms in the wild

Recursive algorithms

- Call themselves again with a smaller value as input after processing current input
- Repeats until problem is solved
- **Example:** factorial algorithm

Divide-and-conquer algorithm

- Works in 2 parts
- First divides the problem into two
- Each problems is solved and merged to produce final result
- **Example**: merge sorting algorithm

Dynamic programming algorithms

Recall the results of the past run and using them to find new results.
Breaks a problem into several simple subproblems, stores them for future reference.

Example: Fibonacci number algorithm:
fib(N) = 0 (for n=0)
= 0 (for n=1)
= fib (N-1) +fib(N-2) (for n>1)

Previous number is used to find the next number.

Greedy algorithm

- Attempts to find optimal solution in a set of solutions
- Has five components
- First component is candidate set
- Selection function to help choose best candidate

Greedy algorithm continued

- Feasibility function decides if candidate can be used to find a solution
- Objective function assigns value to a possible solution or partial solution
- Solution function determines when we have found a solution
- **Example**: Dijkstra’s algorithm

Brute force algorithm

Blindly searches for a solution through a set of possible solutions

Example: sequential search

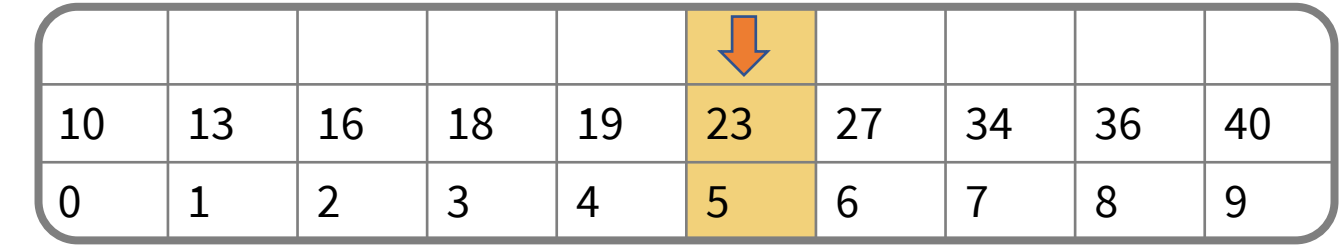
Backtracking algorithm

- Attempts to solve a problem by solving one piece at a time
- If a piece of the problem cannot be solved you backtrack and try another
- Fun fact: sudoku is solved using a backtracking algorithm

Search algorithms

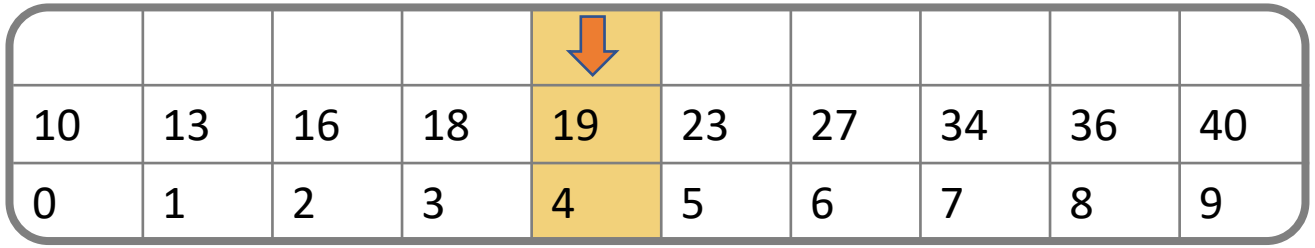
Binary search visualisation

- Compares middle item of the list with search item
- If match, then index of item is returned
- If greater than, item is searched in sub-array to the left of middle item
- Alternatively, item is searched in sub-array to right of middle item
- Repeated until search item is found or no more sub-lists to sort.



```
Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched

  Set lowerBound = 1
  Set upperBound = n

  while x not found
    if upperBound < lowerBound
      EXIT: x does not exist.

    set midPoint = lowerBound + ( upperBound
- lowerBound ) / 2

    if A[midPoint] < x
      set lowerBound = midPoint + 1

    if A[midPoint] > x
      set upperBound = midPoint - 1

    if A[midPoint] = x
      EXIT: x found at location midPoint
  end while

end procedure
```

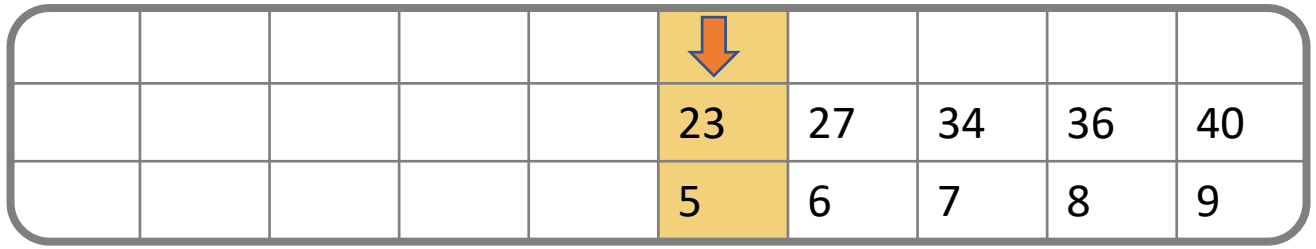
Binary search visualisation



				⇩					
10	13	16	18	19	23	27	34	36	40
0	1	2	3	4	5	6	7	8	9

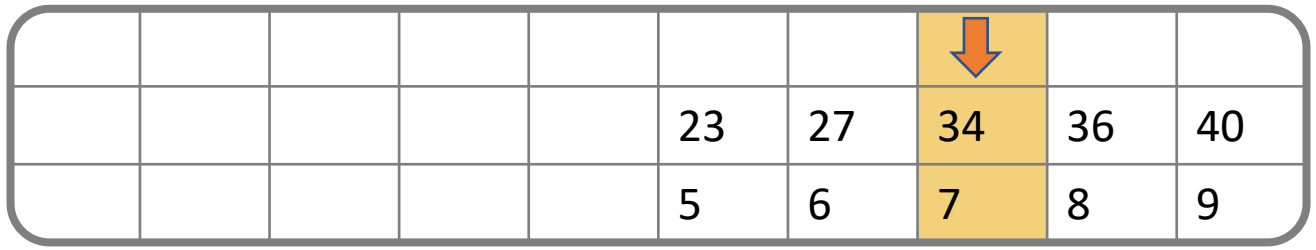
```
Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched

  Set lowerBound = 1
  Set upperBound = n

  while x not found
    if upperBound < lowerBound
      EXIT: x does not exist.

    set midPoint = lowerBound + ( upperBound
- lowerBound ) / 2

    if A[midPoint] < x
      set lowerBound = midPoint + 1

    if A[midPoint] > x
      set upperBound = midPoint - 1

    if A[midPoint] = x
      EXIT: x found at location midPoint
  end while

end procedure
```

Binary search visualisation



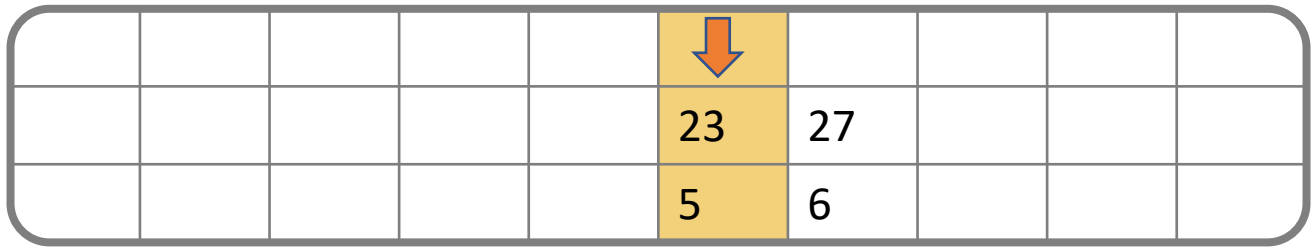
```
Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched

  Set lowerBound = 1
  Set upperBound = n

  while x not found
    if upperBound < lowerBound
      EXIT: x does not exist.

    set midPoint = lowerBound + ( upperBound
- lowerBound ) / 2

    if A[midPoint] < x
      set lowerBound = midPoint + 1

    if A[midPoint] > x
      set upperBound = midPoint - 1

    if A[midPoint] = x
      EXIT: x found at location midPoint
  end while

end procedure
```

Binary search visualisation



```
Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched

  Set lowerBound = 1
  Set upperBound = n

  while x not found
    if upperBound < lowerBound
      EXIT: x does not exist.

    set midPoint = lowerBound + ( upperBound
- lowerBound ) / 2

    if A[midPoint] < x
      set lowerBound = midPoint + 1

    if A[midPoint] > x
      set upperBound = midPoint - 1

    if A[midPoint] = x
      EXIT: x found at location midPoint
  end while

end procedure
```

Binary search visualisation



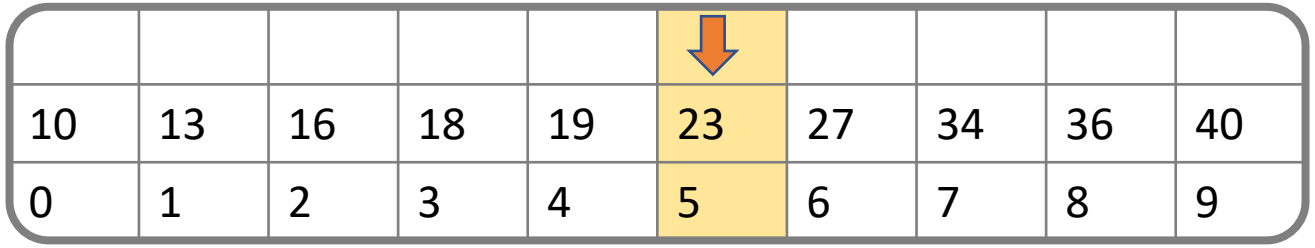
```
Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched

  Set lowerBound = 1
  Set upperBound = n

  while x not found
    if upperBound < lowerBound
      EXIT: x does not exist.

    set midPoint = lowerBound + ( upperBound
- lowerBound ) / 2

    if A[midPoint] < x
      set lowerBound = midPoint + 1

    if A[midPoint] > x
      set upperBound = midPoint - 1

    if A[midPoint] = x
      EXIT: x found at location midPoint
  end while

end procedure
```

Binary search visualisation

					⇩				
10	13	16	18	19	23	27	34	36	40
0	1	2	3	4	5	6	7	8	9

Linear search visualisation

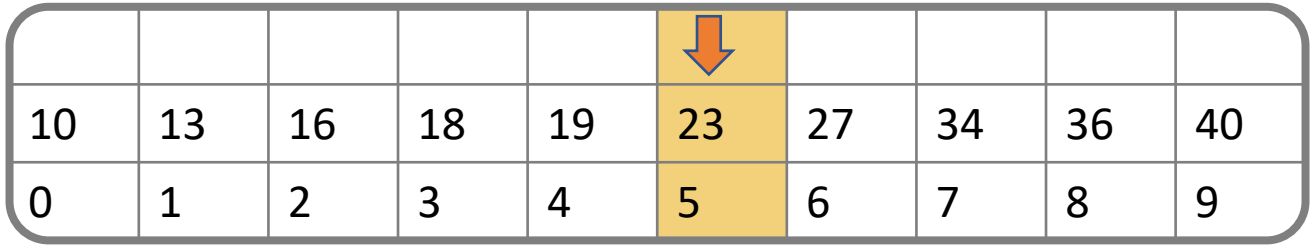
- Go through list item by item until item is found
- Doesn’t need list to be sorted



10	13	16	18	19	23	27	34	36	40
0	1	2	3	4	5	6	7	8	9

Linear search visualisation

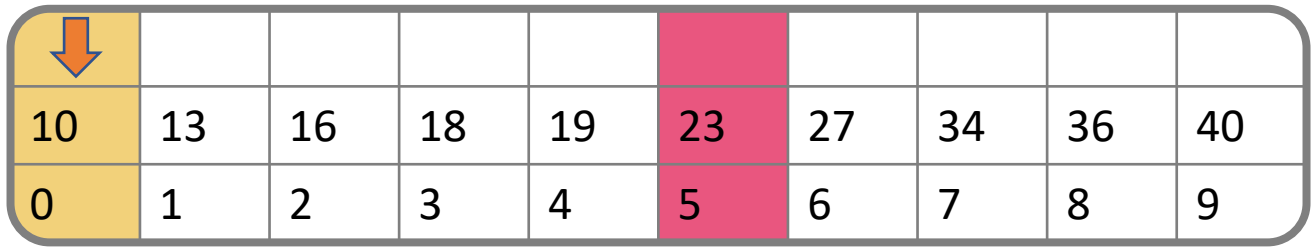
```
procedure linear_search (list, value)

  for each item in the list
    if match item == value
      return the item's location
    end if
  end for

end procedure
```



					⇩				
10	13	16	18	19	23	27	34	36	40
0	1	2	3	4	5	6	7	8	9

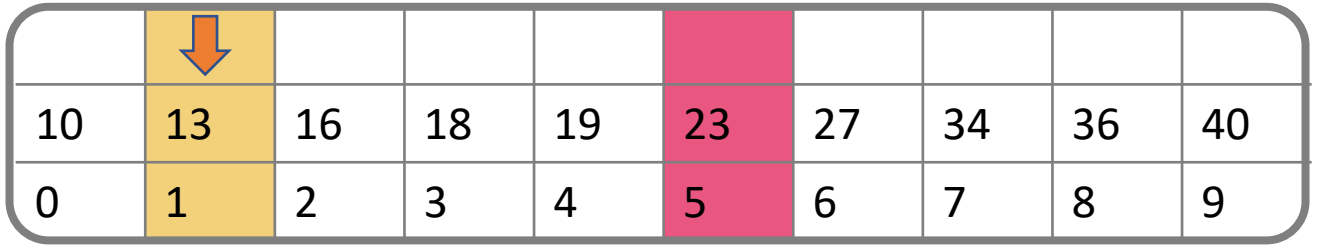
Linear search visualisation

```
procedure linear_search (list, value)

  for each item in the list
    if match item == value
        return the item's location
    end if
  end for

end procedure
```



Linear search visualisation

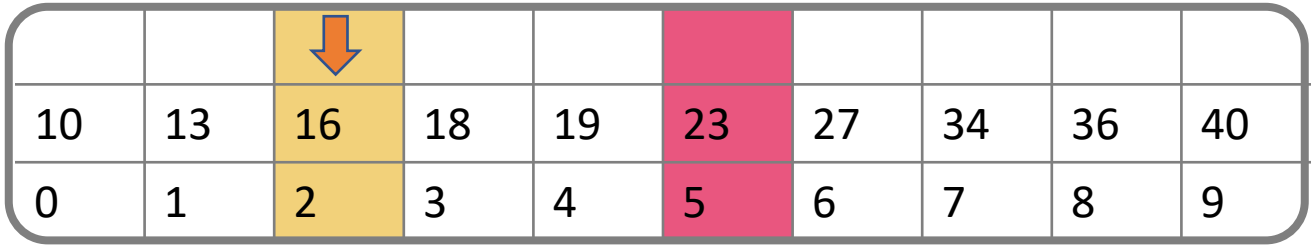
```
procedure linear_search (list, value)

  for each item in the list
    if match item == value
      return the item's location
    end if
  end for

end procedure
```



10	13	16	18	19	23	27	34	36	40
0	1	2	3	4	5	6	7	8	9

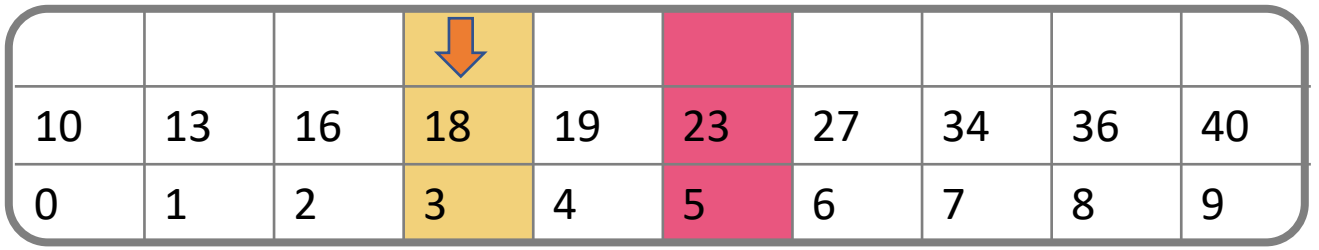
Linear search visualisation

```
procedure linear_search (list, value)

  for each item in the list
    if match item == value
      return the item's location
    end if
  end for

end procedure
```



Linear search visualisation

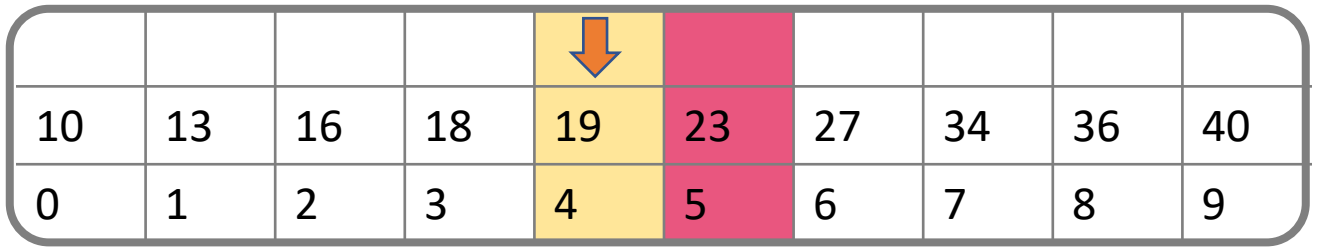
```
procedure linear_search (list, value)

  for each item in the list
    if match item == value
      return the item's location
    end if
  end for

end procedure
```



			⇩						
10	13	16	18	19	23	27	34	36	40
0	1	2	3	4	5	6	7	8	9

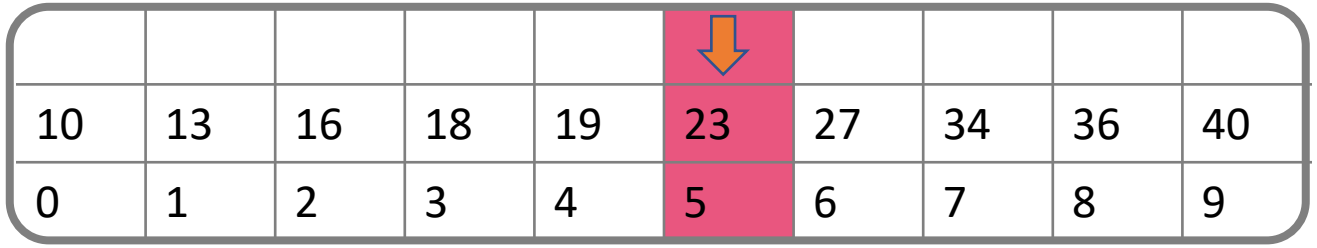
Linear search visualisation

```
procedure linear_search (list, value)

  for each item in the list
    if match item == value
      return the item's location
    end if
  end for

end procedure
```



Linear search visualisation

```
procedure linear_search (list, value)

  for each item in the list
    if match item == value
      return the item's location
    end if
  end for

end procedure
```



10	13	16	18	19	23	27	34	36	40
0	1	2	3	4	5	6	7	8	9

Linear Search (Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print item x Found at index i and go to step 8

Step 7: Print item not found

Step 8: Exit

Sort algorithms

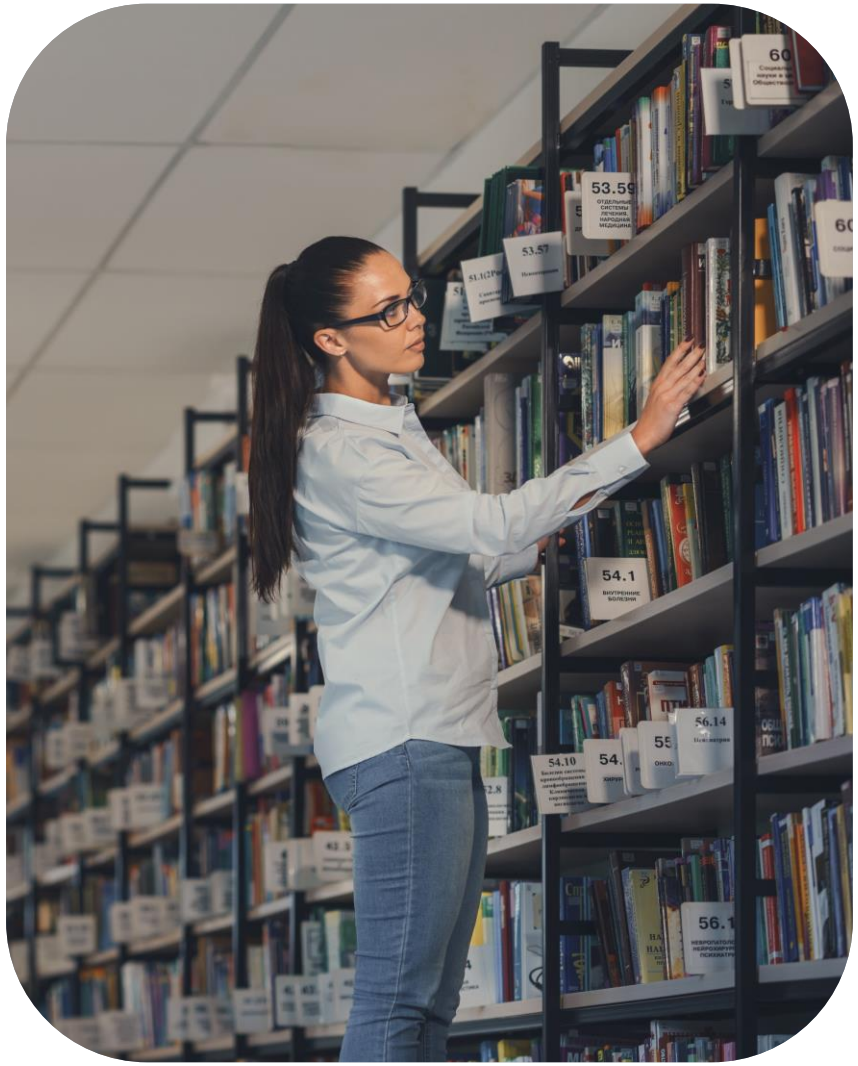
Insertion sort

Selection sort

Quicksort

Bubble sort

Merge sort



Example: Search engine application to search for books in a library system

- Results must be in descending order of relevance
- Create an array
- Store returned books in an array and sort using built-in sort algorithm
- Not always searching is searching and sorting and sorting

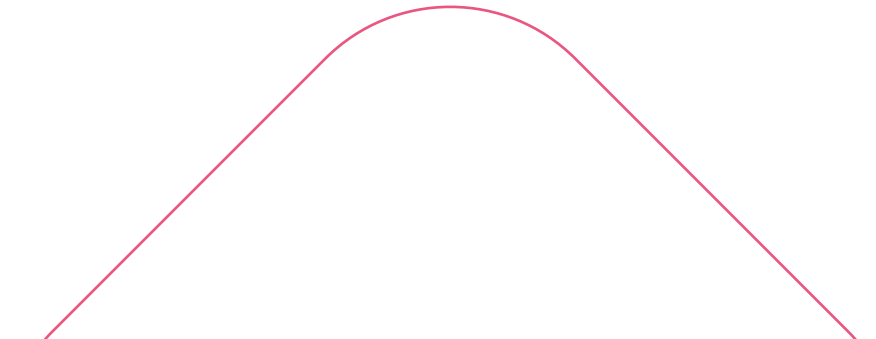
Bubble sort algorithm

- Goes through an entire array
- Compares each neighbouring number
- Then swaps the numbers
- Keeps doing this until the list is in ascending order

Bubble sort algorithm

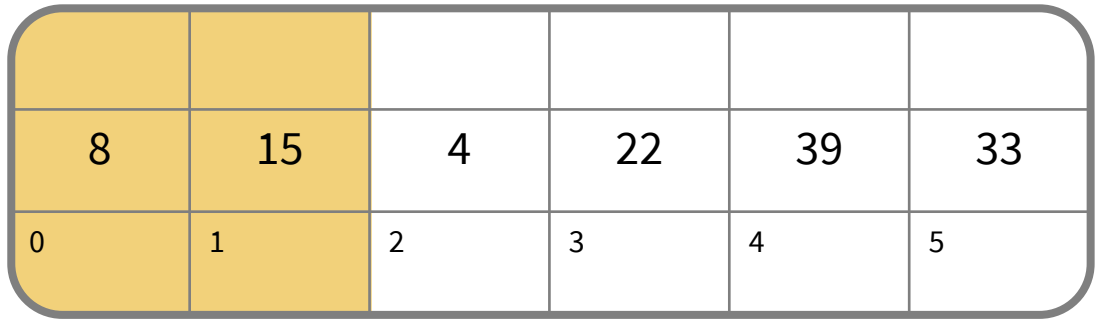
- Based on the concept of comparing pair of adjacent elements
- If elements not in order, they exchange places

15	8	4	22	39	33



```
procedure bubbleSort( list : array of items )
  loop = list.count;
  for i = 0 to loop-1 do:
    swapped = false
    for j = 0 to loop-1 do:
      /* compare the adjacent elements */
      if list[j] > list[j+1] then
        /* swap them */
        swap( list[j], list[j+1] )
        swapped = true
      end if
    end for
    /*if no number was swapped that means
    array is sorted now, break the loop.*/
    if(not swapped) then
      break
    end if
  end for
end procedure return list
```

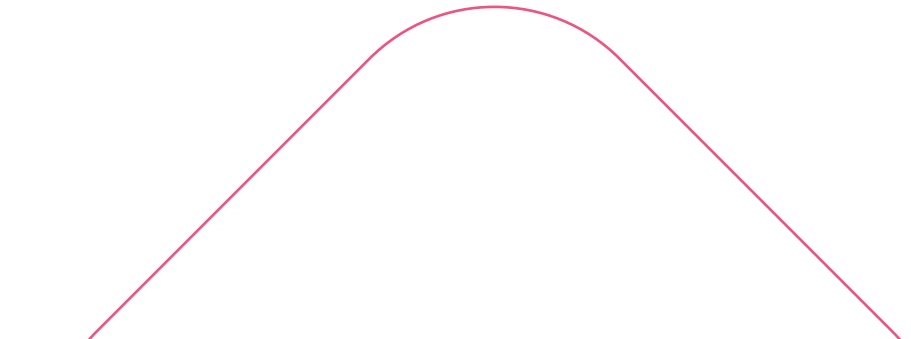
Bubble sort algorithm



```
procedure bubbleSort( list : array of items )
  loop = list.count;
  for i = 0 to loop-1 do:
    swapped = false
    for j = 0 to loop-1 do:
      /* compare the adjacent elements */
      if list[j] > list[j+1] then
        /* swap them */
        swap( list[j], list[j+1] )
        swapped = true
      end if
    end for
    /*if no number was swapped that means
    array is sorted now, break the loop.*/
    if(not swapped) then
      break
    end if
  end for
end procedure return list
```

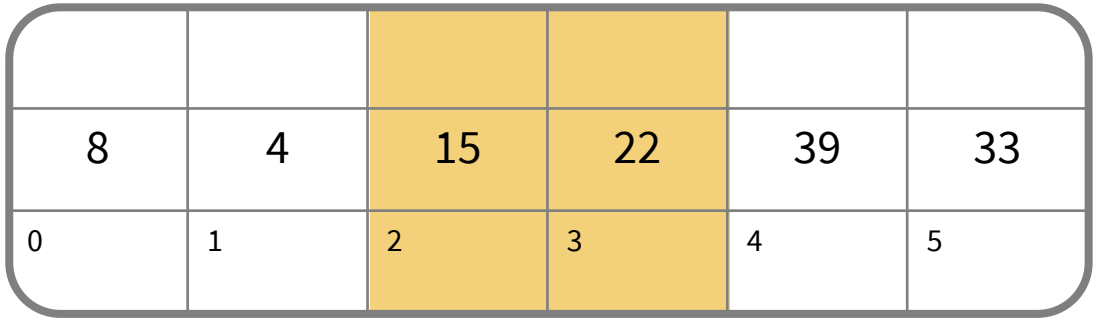
Bubble sort algorithm

8	4	15	22	39	33
0	1	2	3	4	5



```
procedure bubbleSort( list : array of items )
  loop = list.count;
  for i = 0 to loop-1 do:
    swapped = false
    for j = 0 to loop-1 do:
      /* compare the adjacent elements */
      if list[j] > list[j+1] then
        /* swap them */
        swap( list[j], list[j+1] )
        swapped = true
      end if
    end for
    /*if no number was swapped that means
    array is sorted now, break the loop.*/
    if(not swapped) then
      break
    end if
  end for
end procedure return list
```

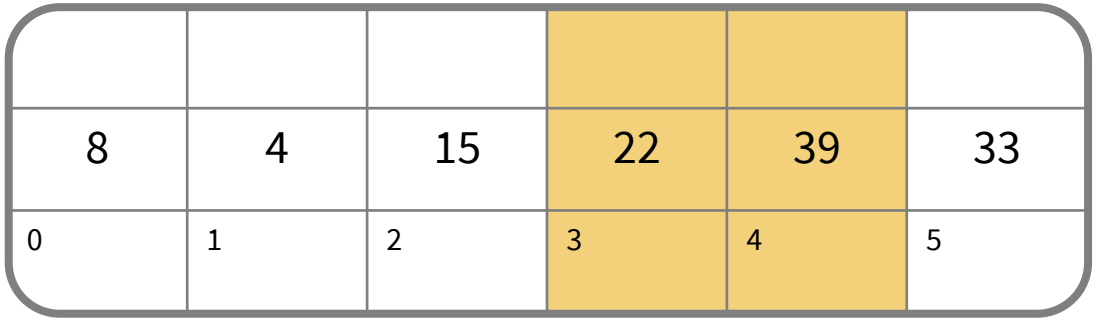
Bubble sort algorithm



8	4	15	22	39	33
0	1	2	3	4	5

```
procedure bubbleSort( list : array of items )
  loop = list.count;
  for i = 0 to loop-1 do:
    swapped = false
    for j = 0 to loop-1 do:
      /* compare the adjacent elements */
      if list[j] > list[j+1] then
        /* swap them */
        swap( list[j], list[j+1] )
        swapped = true
      end if
    end for
    /*if no number was swapped that means
    array is sorted now, break the loop.*/
    if(not swapped) then
      break
    end if
  end for
end procedure return list
```

Bubble sort algorithm



8	4	15	22	39	33
0	1	2	3	4	5

```
procedure bubbleSort( list : array of items )
  loop = list.count;
  for i = 0 to loop-1 do:
    swapped = false
    for j = 0 to loop-1 do:
      /* compare the adjacent elements */
      if list[j] > list[j+1] then
        /* swap them */
        swap( list[j], list[j+1] )
        swapped = true
      end if
    end for
    /*if no number was swapped that means
    array is sorted now, break the loop.*/
    if(not swapped) then
      break
    end if
  end for
end procedure return list
```

Bubble sort algorithm

8	4	15	22	33	39
0	1	2	3	4	5

```
procedure bubbleSort( list : array of items )
  loop = list.count;
  for i = 0 to loop-1 do:
    swapped = false
    for j = 0 to loop-1 do:
      /* compare the adjacent elements */
      if list[j] > list[j+1] then
        /* swap them */
        swap( list[j], list[j+1] )
        swapped = true
      end if
    end for
    /*if no number was swapped that means
    array is sorted now, break the loop.*/
    if(not swapped) then
      break
    end if
  end for
end procedure return list
```

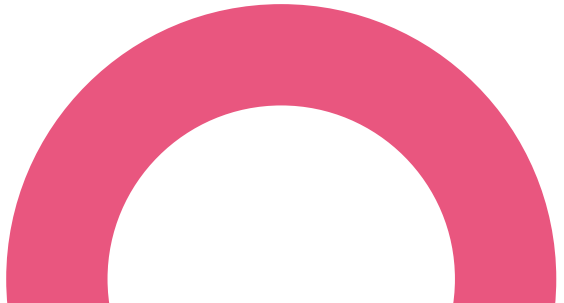
Bubble sort algorithm

4	8	15	22	33	39
0	1	2	3	4	5

```
procedure bubbleSort( list : array of items )
  loop = list.count;
  for i = 0 to loop-1 do:
    swapped = false
    for j = 0 to loop-1 do:
      /* compare the adjacent elements */
      if list[j] > list[j+1] then
        /* swap them */
        swap( list[j], list[j+1] )
        swapped = true
      end if
    end for
    /*if no number was swapped that means
    array is sorted now, break the loop.*/
    if(not swapped) then
      break
    end if
  end for
end procedure return list
```

Bubble sort algorithm

4	8	15	22	33	39
0	1	2	3	4	5



```
procedure bubbleSort( list : array of items )
  loop = list.count;
  for i = 0 to loop-1 do:
    swapped = false
    for j = 0 to loop-1 do:
      /* compare the adjacent elements */
      if list[j] > list[j+1] then
        /* swap them */
        swap( list[j], list[j+1] )
        swapped = true
      end if
    end for
    /*if no number was swapped that means
    array is sorted now, break the loop.*/
    if(not swapped) then
      break
    end if
  end for
end procedure return list
```

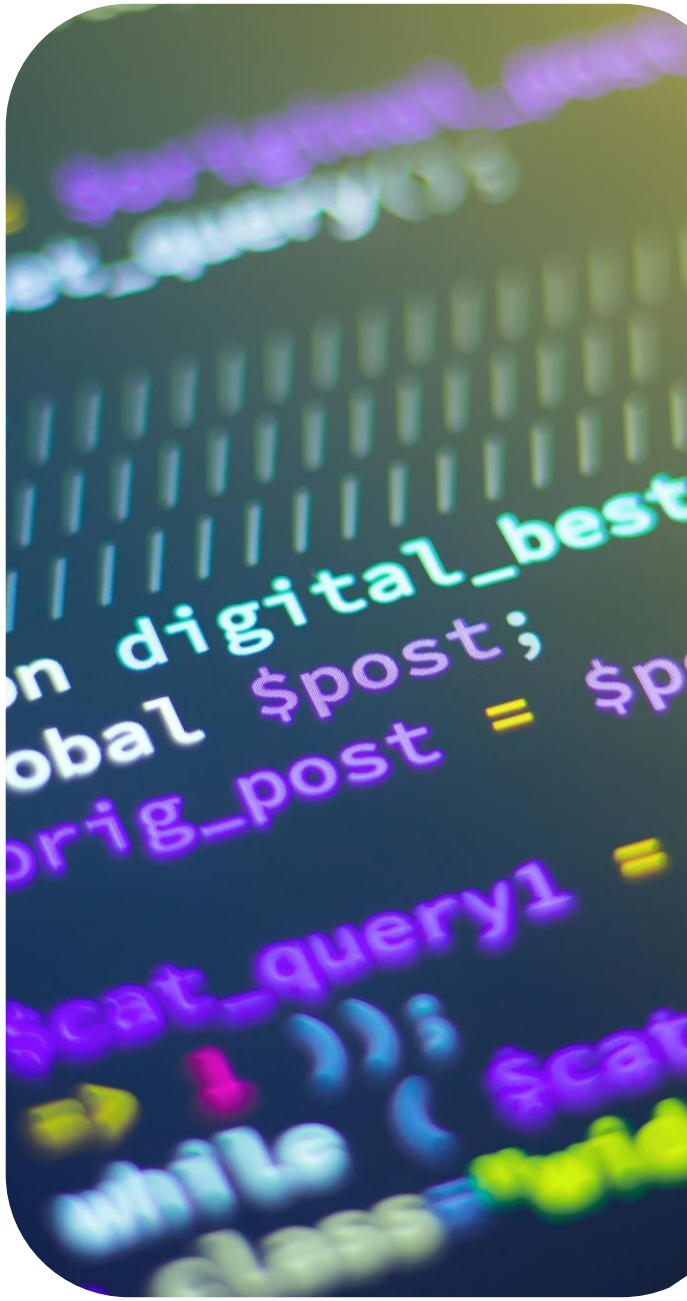
Bubble sort algorithm

4	8	15	22	33	39
0	1	2	3	4	5

Pseudocode to algorithm

Programme must follow pseudocode conventions.

Pseudocode has ground rules that make it universally readable.



Breaking down code

- Smallest bits make the problem easier to solve
- Cuts down on algorithm development time and results in faster turnaround times
- Makes algorithm easier to understand
- Ensures problem is completely solved

Creating good algorithms

The input should be specified.

The output of the algorithm must be specified.

The algorithm must specify every step and specify the order in which the steps will be taken.

Creating good algorithms

The algorithm should be effective.

The algorithm must eventually stop running.

The algorithm should be efficient.

The algorithm should be understandable.

Did you know?

The first algorithm written for a machine was published in 1843 by a woman named Ada Lovelace.