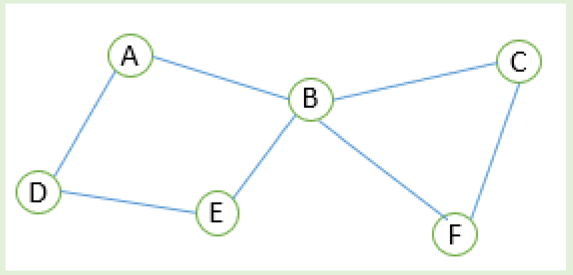
Algorithms

Traversing Graphs

We can use depth first traversal or breadth first traversal to traverse a graph.

Graph used in examples to follow

A	[D, B]
B	[A, E, C,F]
C	[B, F]
D	[A, E]
E	[D, B]
F	[B, C]



Breadth First Traversal

Breadth first traversal starts at a node and explores all the neighbour nodes before moving onto the next level of nodes. A breadth first traversal uses an iterative approach. A typical application of a breadth first traversal is for determining the shortest path of an unweighted graph

```
breadth_first_traversal(node)
 queue = []
 visited = []
 queue.append(node)
 visited.append(node)

 while queue is not empty
  node = queue.pop(0)
  print (node, end = " ")
  for i in graph[node]:
   if i not in visited
    queue.append(i)
    visited.append(i)

graph={'A':['D','B'],\
'B':['A','E','C','F'], 'C': ['B','F'],\
'D': ['A','E'], 'E':['D','B'],'F':['B','C']}

breadth_first_traversal("A")
```

Trace Table

Node	i	Output	Visited	Queue
A			[A]	[A]
		A		[]
	D		[A,D]	[D]
	B		[A,D,B]	[D,B]
D		D		[B]
	A			
	E		[A,D,B,E]	[B,E]
B		B		[E]
	A			
	E			
	C		[A,D,B,E]	[E,C]
	F		[A,D,B,E,C,F]	[E,C,F]
		E		[C,F]
		C		[F]
		F		[]

Depth First Traversal

Depth first traversal starts at a node and traverses along each path as far as it goes before backtracking to the next branch. Depth first traversal uses recursion An application of a depth first traversal is for navigating a maze.

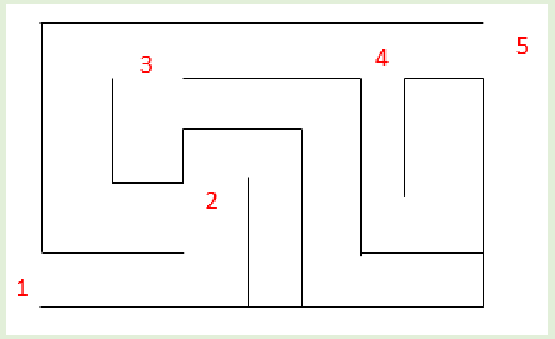
```
# Uses recursive calls
depth_first_traversal(node)
 visited.append(node)
 for i in graph[node]:
  if i not in visited
   depth_first_traversal(i)

# Graph represented as an adjacency list
graph={"A":["D","B"], "B":["A","E","C","F"],\
"C": ["B","F"], "D": ["A","E"],\
"E":["D","B"],"F":["B","C"]}

# Create a list of visited nodes,
# set to false to begin with
visited = []
depth_first_traversal("A")
```

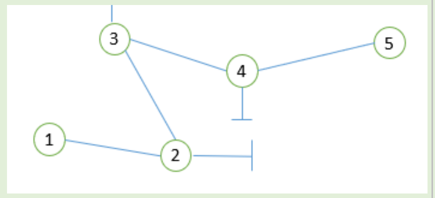
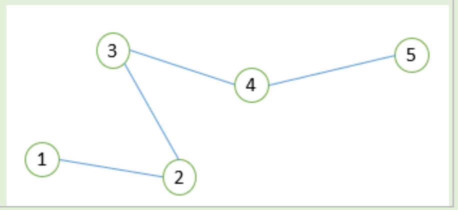
Trace Table

Call	Node	i	Visited
			[]
1	A		[A]
2	D	D	[A,D]
		A	
3	E	E	[A,D,E]
		D	
4	B	B	[A,D,E,B]
		A	
		E	
5	C	C	[A,D,E,B,C]
		B	
6	F	F	[A,D,E,B,C,F]

Navigating a maze with depth first traversal

Nodes are placed at the start and end points as well as at locations where there are alternative paths



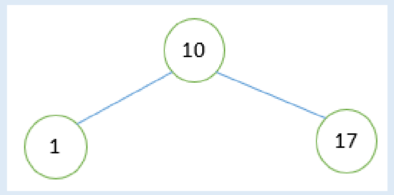
Graph representation of maze with dead ends	Graph representation of maze without dead ends
	

Tree-traversal

There are three ways of traversing a binary tree:

- Pre-order tree traversal
- Post-order tree traversal
- In-order tree traversal

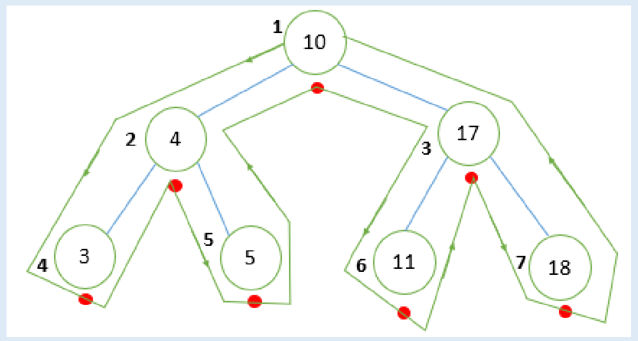
When traversing a tree we start at the root node. We can then visit the node (that is obtain the value of the node), traverse left or traverse right. The order in which we visit, traverse left or traverse right depends on the traversal method that we use.



	Pre-order traversal	Post-order traversal	In-order traversal
Order	1.Visit Node 2.Left Traversal 3.Right Traversal	1.Left Traversal 2.Right Traversal 3.Visit Node	1.Left Traversal 2.Visit Node 3.Right Traversal
Example	10, 1, 17	1, 17, 10	1, 10, 17
Example Application	Prefix Notation, Copying a tree	Reverse Polish Notation	Ordering a sequence of numbers, binary tree search

In-order traversal

```
in_order_traversal(node):
    if tree_left[node] != -1:
      in_order_traversal(tree_left[node])
    print(values[node])
    if tree_right[node] != -1:
      in_order_traversal(tree_right[node])
# node_index[1,2,3,4,5,6,7]
values=[10,4,17,3,5,11,18]
tree_left=[2,4,6,-1,-1,-1,-1]
tree_right=[3,5,7,-1,-1,-1,-1]
in_order_traversal(1)
```



Sequence output: 3,4,5,10,11,17,18

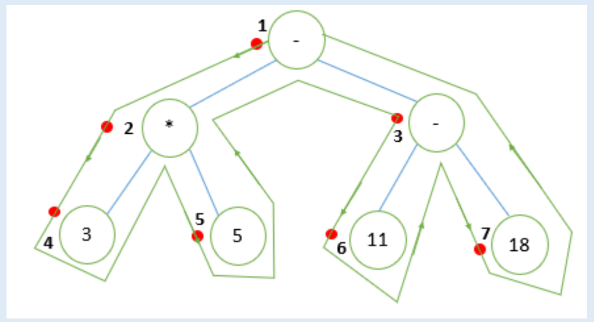
Trace table

node	Value [node]	Tree_left [node]	Tree_right [node]	Output
1	10	2	3	
2	4	4	5	
4	3	-1	-1	3
2	4			4
5	5	-1	-1	5
1	10			10
3	17	6	7	
6	11	-1	-1	11
3	17			17
7	18	-1	-1	18

Pre-order traversal

```
pre_order_traversal(node):
    print(values[node])
    if tree_left[node] != -1:
      pre_order_traversal(tree_left[node])
    if tree_right[node] != -1:
      pre_order_traversal(tree_right[node])
values=["+","-","*",2,4,6,7]
tree_left=[2,4,6,-1,-1,-1,-1]
tree_right=[3,5,7,-1,-1,-1,-1]
pre_order_traversal(1)
```

Sequence output: - * 3 5 - 11 18



Trace Table

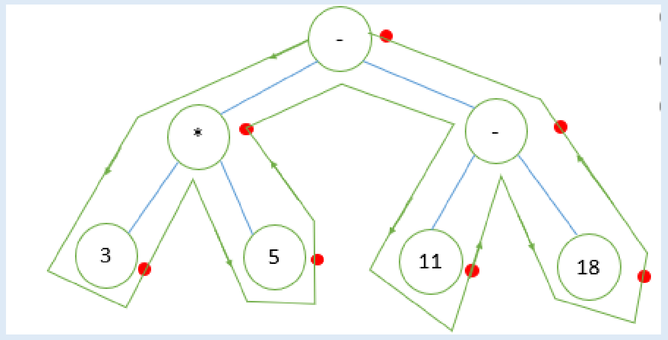
node	Value [node]	Tree_right [node]	Tree_left [node]	Output
1	+	3	2	+
2	-	5	4	-
4	2	-1	-1	2
2	-	5	4	
5	4	-1	-1	4
1	+	3	2	
3	*	7	6	*
6	6	-1	-1	6
7	7	-1	-1	7

Post-Order Traversal

```
post_order_traversal(node):
    if tree_left[node] != -1:
      post_order_traversal(tree_left[node])
    if tree_right[node] != -1:
      post_order_traversal(tree_right[node])
    print(values[node])
values=["+","-","*",2,4,6,7]
tree_left=[2,4,6,-1,-1,-1,-1]
tree_right=[3,5,7,-1,-1,-1,-1]
```

```
post_order_traversal(1)
```

Sequence output: 3 5 * 11 18 - -



Trace Table

Call	node	Value [node]	Tree_right [node]	Tree_left [node]	Output
1	1	+	3	2	
2	2	-	5	4	
3	4	2	-1	-1	2
2	2	-	5	4	
4	5	4	-1	-1	4
2	2	-	5	4	-
5	3	*	7	6	
6	6	6	-1	-1	6
5	3	*	7	6	
7	7	7	-1	-1	7
5	3	*	7	6	*
1	1	+	3	2	+

Reverse Polish Notation

Infix Notation
we are all familiar infix notation where the operators appear between the operands (ie the numbers) that you want to apply the operator to.

Reverse Polish Notation (Postfix)
RPN uses postfix notation where the operators follow the operand. Using infix notation to add two numbers we get:

```
<operand> <operator> <operand>   3 + 4
```

In RPN (postfix notation) this becomes

```
<operand> <operand> <operator>   3 4 +.
```

Many interpreters and compiler automatically convert between infix notation to postfix notation, so there is no requirement to write code using the less familiar postfix notation.

Advantages of Postfix
- Simpler for computer to evaluate
- Do not need brackets
- Operators appear in correct order for computation
- No need for order of precedence of operators, so there are fewer operations

RPN Algorithm
1. Go through each character in the postfix expression from left to right
2. If character is a number then push number onto the stack
3. Otherwise if the character is an operator (+,-,/ ,X) then pop the top 2 numbers from the stack
4. Evaluate the 2 numbers using the operator
5. Push result back onto the stack

Worked example: Solve the following expression: 5 3 1 + − 6 ×

Stack at each step

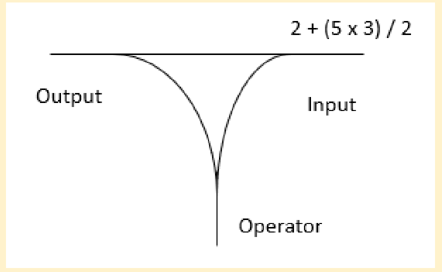
1	2	3	4	5	8	9
5	3	1	4	1	6	6
	5	3	5		1	
		5				
~~5~~31+-6x	531+-6x	~~531~~+-6x	531+-6x	531+-6x	~~531+-6~~x	~~531+-6x~~
Push 5 onto stack	Push 4 onto stack	Push 1 onto stack	Pop 1,3 Evaluate 1+3=4 Push result on stack	Pop 4,5 Evaluate 5-4=1 Push result on stack	Push 6 onto stack	Pop 6,1 Evaluate 6x1=6 Push result on stack

Answer is 6
Infix expression (5-(1+3))x6

Convert from Infix to Postfix notation

Step 1	Add Brackets	(3 + ((5 x 3) / (7 − 4)))
Step 2	Write out the operands with spaces	3 5 3 7 4
Step 3	Starting with the inner most brackets move the operator to after the operands from between the operands	3 5 3 x 7 4 – *3 + (15 / 3)* 3 5 3 x 7 4 / *3 + 5* **3 5 3 x 7 4 /+** *8*

Alternative Shunting Yard Algorithm to Convert from Infix to Postfix notation



Worked Example: Convert the following expression to RPN: 2 + (5 x 3) / 2

Symbol	Action	Output queue	Operator stack
2	Push operand onto output queue	2	
+	Push operator onto operator stack	2	+
5	Push operand onto output queue	2 5	+
x	Push operand onto operator stack, x has higher precedence than +	2 5	x +
3	Push operand onto output queue	2 5 3	x +
/	Pop stack to output, x has same precedence as / Push on operator stack, / has higher precedence than +	2 5 3 x 2 5 3 x	+ / +
2	Push operand onto output queue	2 5 3 x 2	/ +
	Pop whole stack onto output queue	**2 5 3 x 2 / +**	

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.
- The time complexity of the linear search algorithm is O(n).

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
```

Worked example: given the following vales for `listOfItems` and `itemSearch` we have the following trace table

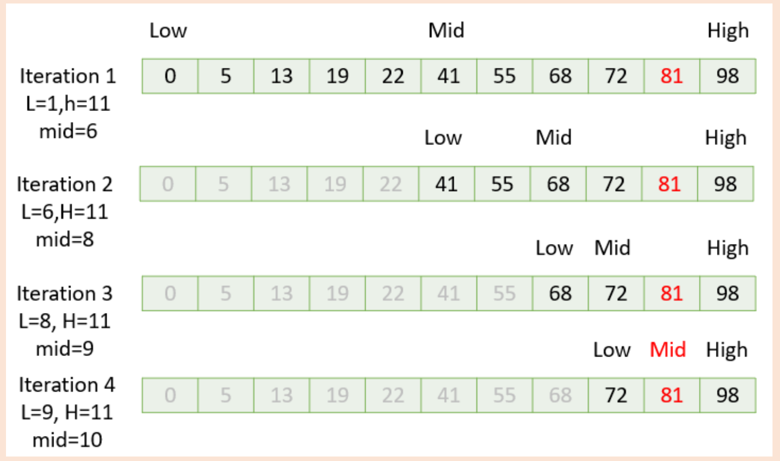
```
listOfItems ←[6,3,9,1,2]
itemSearch ← 1
```

i	x	pos	found	itemSearch	listOfItems[i]	OUTPUT
0	5	-1	False	1	6	
1					*3*	
2					*9*	
3					*1*	
4		4	True			4

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.
- The time complexity of the binary search algorithm is O(log n)

Example: Binary search in operation to find 81



Pseudocode

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

Worked example: given the following vales for `arr` and `val` we have the following trace table

```
val ← 81
A ← [0,5,13,19,22,41,55,68,72,81,98]
```

mid	high	low	A[mid]	A[high]	A[low]
6	11	1	41	98	0
8	*11*	*6*	*68*	*98*	*41*
9	*11*	*8*	*72*	*98*	*68*
10	*11*	*9*	*81*	*98*	*72*

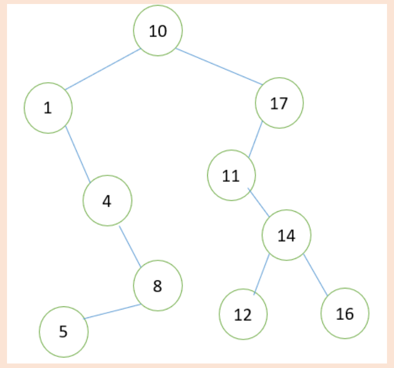
Linear search versus binary search

	Advantages	Disadvantages
Linear Search	• Very simple algorithm and easy to implement • No sorting required • Good for short lists	• slow because it searchers through the whole list • 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

Binary Tree Search

- Binary tree search can be coded using a recursive algorithm or iterative algorithm. We are going to present the recursive binary tree search algorithm here.
- The time complexity of the binary tree search algorithm is the same as that for the binary search algorithm: O(log N).
- It can be applied to any values: number or letters that are ordered in the binary tree numerically or alphabetically.

Example sorted binary tree



Python implementation using lists

```
def binaryTreeSearch(node,searchItem)
 path.append(values[node])
 if values[node] == searchItem:
  return "Value in Tree. Path: "+str(path)
 elif values[node] < searchItem:
  if treeRight[node] == -1:
   return "Value not in Tree"
 return binaryTreeSearch(treeRight[node],searchItem)
 elif values[node] > searchItem:
  if treeLeft[node] == -1:
   return "Value not in Tree"
  return binaryTreeSearch(treeLeft[node],searchItem)

path = []
#  node[0,1,2,3,4,5,6,7,8,9]
values = [10,1,17,4,11,8,14,5,12,16]
treeLeft = [1,-1,4,-1,-1,7,8,-1,-1,-1]
treeRight=[2,3,-1,5,6,-1,9,-1,-1,-1]
print(binaryTreeSearch(0, 5))
```

Tracing

Call num	*Call*	*Output*	*Return*
1	*BinarySearchTree(10,5)*	*10*	
2	*BinarySearchTree(1,5)*	*1*	
3	*BinarySearchTree(4,5)*	*4*	
4	*BinarySearchTree(8,5)*	*8*	
5	*BinarySearchTree(5,5)*	*5*	*5*

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.
- The time complexity of the algorithm is O(n^2)

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

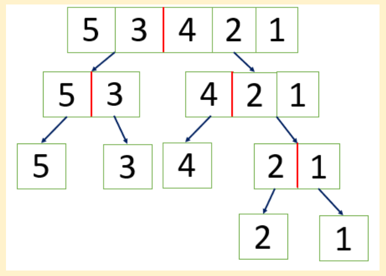
```
A ← [5,3,4,1,2]
sorted ← False
WHILE not sorted
 sorted ← True
 FOR i TO LEN(A)-1:
  IF A[i] > A[i+1]:
   temp ← A[i]
   A[i] ← A[i+1]
   A[i+1] ← temp
   sorted ← False
  ENDIF
 ENDFOR
ENDWHILE
OUTPUT A
```

sorted	i	A[i]	A[i+1]	temp	A
False					5,3,4,1,2
True	0	5	3	5	
False		3	5		3,5,4,1,2
	1	5	4	5	
		4	5		3,4,5,1,2
	2	5	1	5	
		1	5		3,4,1,5,2
	3	5	2	5	
		2	5		3,4,1,2,5
True	0	3	4		
False	1	4	1	4	
		1	4		3,1,4,2,5
	2	4	2	4	
		2	4		3,1,2,4,5
	3	4	5		
True	0	3	1	3	
		1	3		1,3,2,4,5
False	1	3	2	3	
		2	3		1,2,3,4,5
	2	3	4		
	3	4	5		
True	0	1	2		
	2	2	3		
	3	3	4		
	4	4	5		

Merge Sort

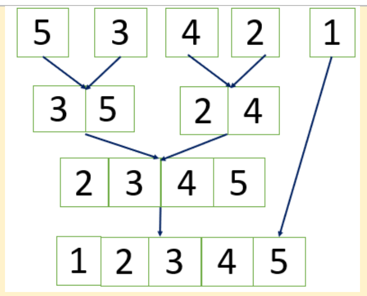
- Merge sort is a type of divide and conquer algorithm.
- There are two steps: divide and combine
- Merge sort works by dividing the unsorted list sublists. It keeps on doing this until there is 1 item in each list.
- Pairs of sublists are combined into an ordered list containing all items in the two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

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 Pseudocode

```
SUBROUTINE MergeSort(List, Start, End)
 IF Start < End THEN
  Mid ← (Start + End) DIV 2
  List1 ← MergeSort(List, Start, Mid)
  List2 ← MergeSort(List, Mid + 1, End)
  List3 ← []
  WHILE LEN(List1) > 0 AND LEN(List2) > 0
   IF List1[1] > List2[1] THEN
    APPEND List2[1] TO List3
    POP List2[1] FROM List2
   ELSE
    APPEND List1[1] TO List3
    POP List1[1] FROM List1
   ENDIF
  ENDWHILE
  WHILE LEN(List1) > 0
   APPEND List1[1] TO List3
   POP List1[1] FROM List1
  ENDWHILE
  WHILE LEN(List2) > 0
   APPEND List2[1] TO List3
   POP List2[1] FROM List2
  ENDWHILE
  RETURN List3
 ELSE
  List4 ← []
  APPEND List[Start] To List4
  RETURN List4
ENDSUBROUTINE
```

Tracing the code

```
L=[5,3,4,1,2]
MergeSort(L,1,5)
```

Call	Start	End	Mid	List returned
1	1	5	3	
2	1	3	2	
3	1	2	1	
4	1	1		[5]
3	1	2	1	
5	2	2		[3]
3	1	2	1	[3,5]
2	1	3	2	
6	3	3		[4]
2	1	3	2	[3,4,5]
1	1	5	3	
7	4	5	4	
8	4	4		[1]
7	4	5	4	
9	5	5		[2]
7	4	5	4	[1,2]
1	1	5	3	[1,2,3,4,5]

Merge sort Versus Bubble sort

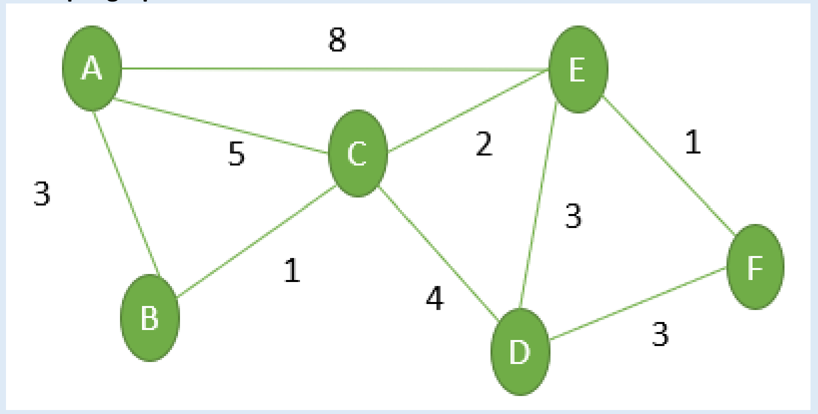
	Advantages	Disadvantages
Bubble sort	Very simple and robust algorithm	Can be slow particularly for long lists. As the number of items increases the time taken for the algorithm to run increases dramatically.
Merge sort	Much faster than bubble sort especially when the number of elements is large	More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

Optimisation algorithms

Dijkstra's shortest path algorithm

- The purpose of Dijkstra's algorithm finds the shortest path between nodes / verticies in a weighted graph.
- Selects the unvisited node with the shortest path.
- Calculates the distance to each unvisited neighbor
- Updates the distance of each unvisited neighbor if smaller
- Once all neighbours have been visited mark node as visited

Example graph



Example: find the shortest route between node A and F

Node	A	B	C	D	E	F
A	0_A	3_A	5_A	∞	8_A	∞
B		3_A	4_B	∞	8_A	∞
C			4_B	8_C	6_C	∞
E				8_C	6_C	7_E
F				8_C		7_E
D				8_C		

Start at node A because it is the unvisited node with the shortest distance to node A. The distance to each unvisited neighbor is 3 and 5 for B and C respectively. B has the shortest distance to node A so this is the next unvisited node we select. At B there is only 1 neighbor (C). The distance is updated because the route A-B-C (4) has less cost that the route A-C (5). E is the next unvisited node with the shortest distance and is has as neighbours D and F. F has the less cost out the two and is then selected as the next unvisited node. The shortest route is A-C-E-F.

Dijkstra Pseudocode

```
Q ← []
distance ← []
previous_node ← []
FOR i ← 1 TO NUMBER_OF_VERTICIES
 Append i to Q
 Append 100 to distance
 Append -1 to previous_node
ENDFOR
distance[1] ← 0
WHILE LEN(Q) != 0
 u ← Q[1]
 Pop u from Q
 FOR v  in Q
  IF matrix[u][v] > 0:
   a=distance[u] + matrix[u][v]
   IF a< distance[v]
    distance[0]=a
    previous_node[v]=u
   ENDIF
  ENDIF
 ENDFOR
ENDWHILE
```

Trace table Given the following matrix

u/v	1	2	3	4
1	0	2	5	3
2	0	0	1	0
3	0	0	0	0
4	0	0	0	0

q	u	v	a	Distance				Previous_node			
1, 2, 3, 4				100	100	100	100	-1	-1	-1	-1
				0							
2, 3, 4	1	2	2		2				1		
		3	5			5				1	
		4	3				3				1
3,4	2	3	3			3				2	
4	3										
-	4										