Unit 2 | 2.3.1 | Algorithms

Specification & learning objectives

A Level	Specification point description
2.3.1a	Analysis and design of algorithms for a given situation
2.3.1b	The suitability of different algorithms for a given task and data set, in terms of execution time and space
2.3.1c	Measures and methods to determine the efficiency of different algorithms, Big O notation. (Constant, linear, polynomial, exponential, and logarithmic complexity)
2.3.1d	Comparison of the complexity of algorithms
2.3.1e	Algorithms for the main data structures, (Stacks, queues, trees, linked lists, depth-first (post-order) and breadth-first traversal of trees)
2.3.1f	Standard algorithms (Bubble sort, insertion sort, merge sort, quick sort, Dijkstra's shortest path algorithm, A* algorithms, binary search and linear search)

Resources

PG Online textbook page ref: 184-203,209-221,328-363

Hodder textbook page ref: 49-82

CraignDave videos for SLR 26

Unit 2 | 2.3.1 | Algorithms

Key question: Which data structures and their operations are used for common algorithms?

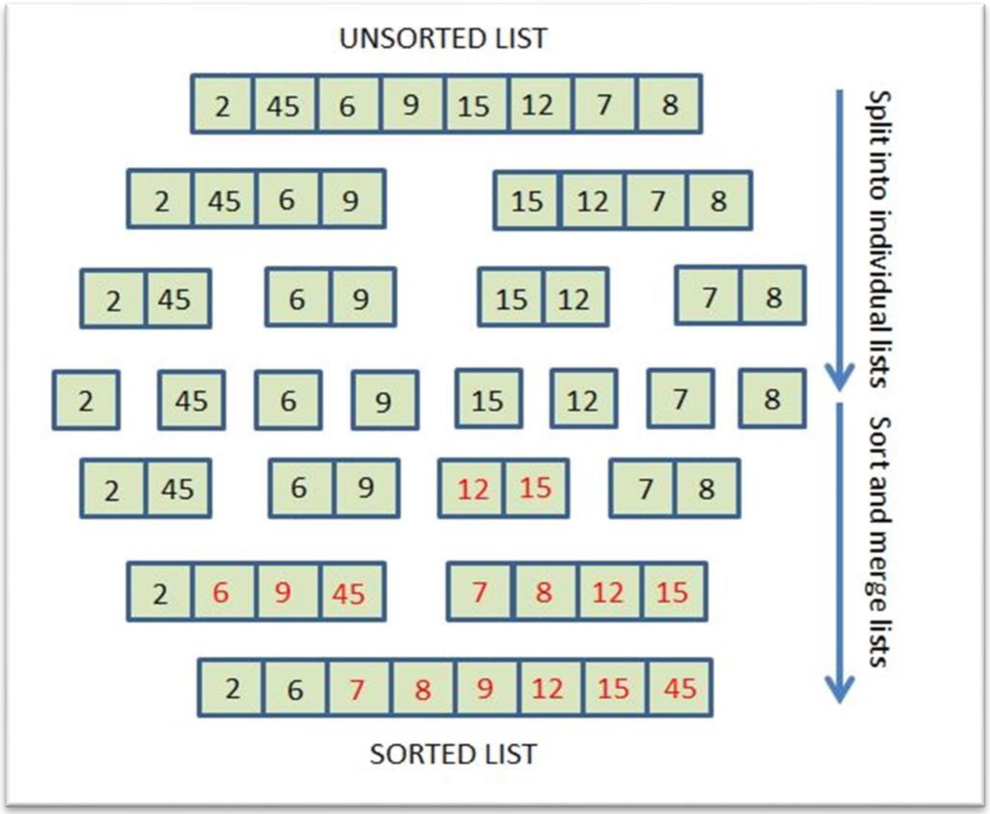
In its most basic form, an algorithm is a set of detailed step-by-step instructions to complete a task.

Algorithms are often grouped into different categories like search, sorting, and compression algorithms. Further, algorithms can be described by the approach it takes to complete a task, such as recursive, backtracking, divide and conquer, greedy, and brute force.

Algorithms are often paired with data structures, though they are fundamentally different. Data Structures are methods of storing data so that an algorithm can perform operations on it easily.

Some common examples of data structures are arrays, stacks, queues, linked lists, trees, graphs, hash tables, and heaps.

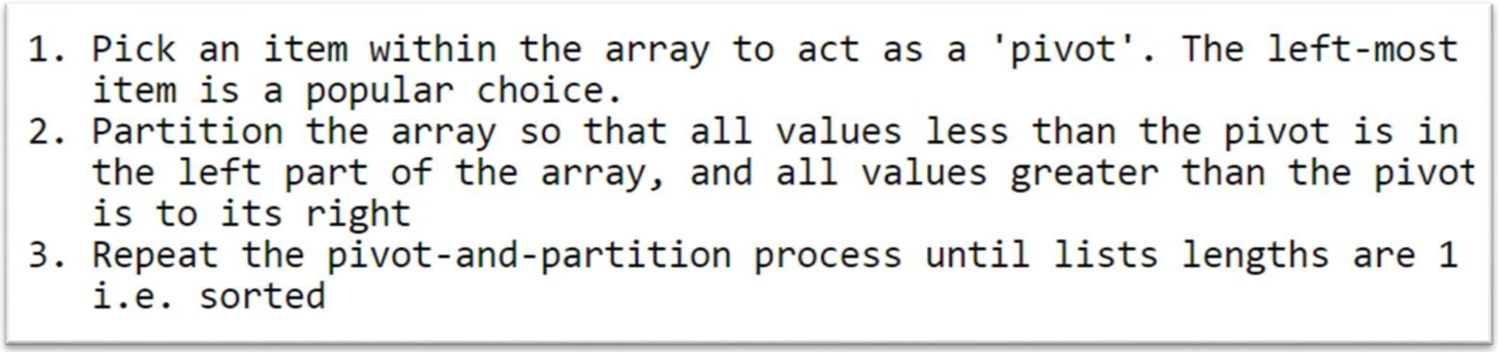
Key question: How does a merge sort work?



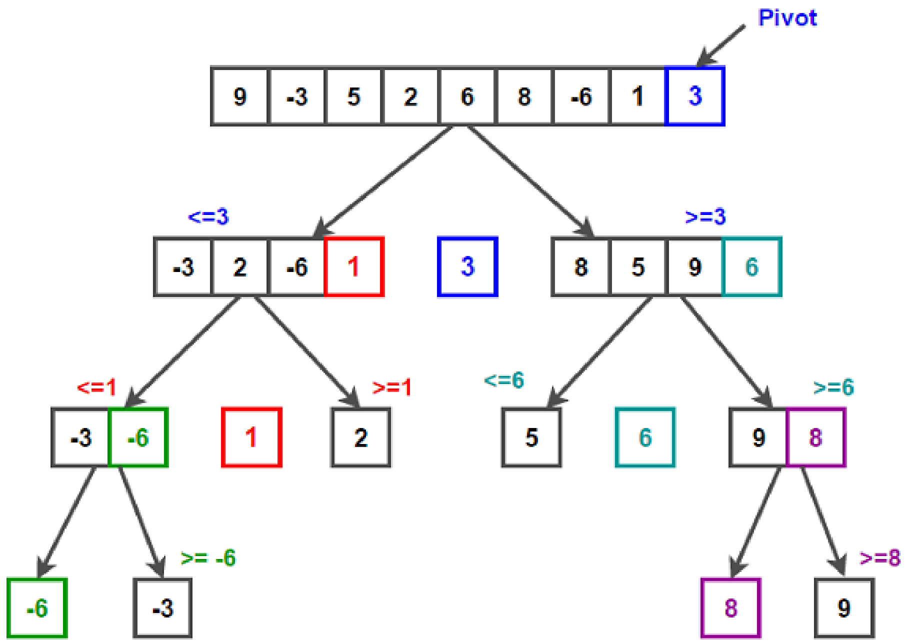
```
Let there be an unsorted list

WHILE there remains an unsorted list

 Function: Split each list in two   # this is a recursive call
 IF every list is 1 or less in length THEN exit

END WHILE

# at this stage every list is either 1 or zero in length
# which means by definition they are sorted

# Now merge and sort each list pair

WHILE there is more than one list
    Function: Sort and combine lists # this is a recursive call
END WHILE
```

Key question: How does a quicksort work?

This is a very fast way of sorting medium to large lists.

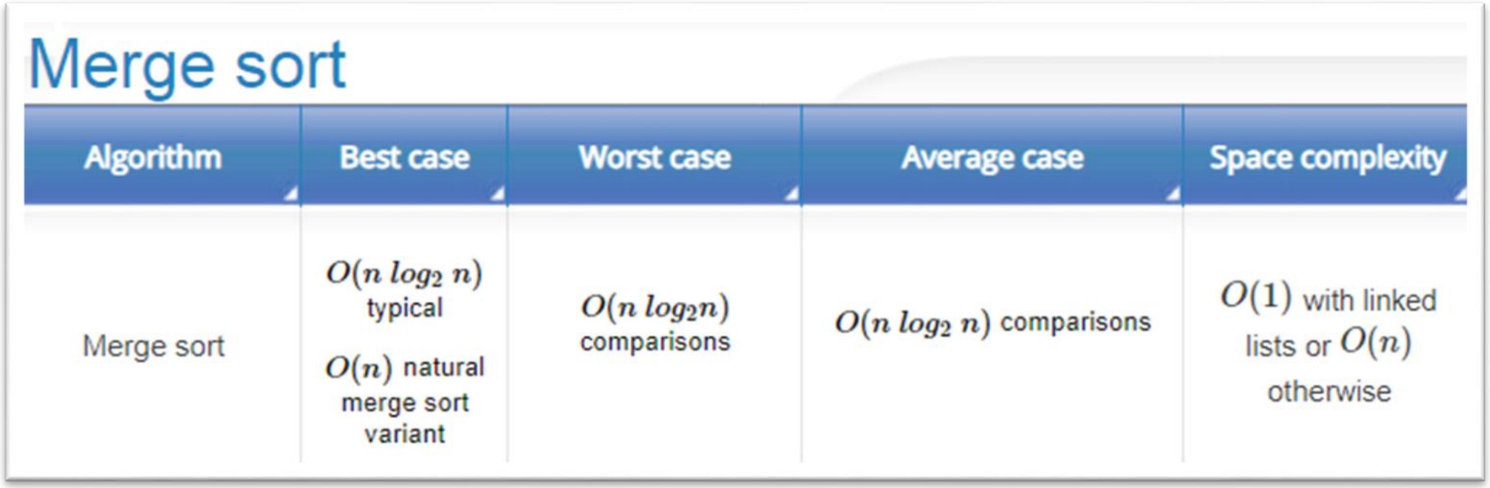


```
1. Pick an item within the array to act as a 'pivot'. The left-most
   item is a popular choice.
2. Partition the array so that all values less than the pivot is in
   the left part of the array, and all values greater than the pivot
   is to its right
3. Repeat the pivot-and-partition process until lists lengths are 1
   i.e. sorted
```



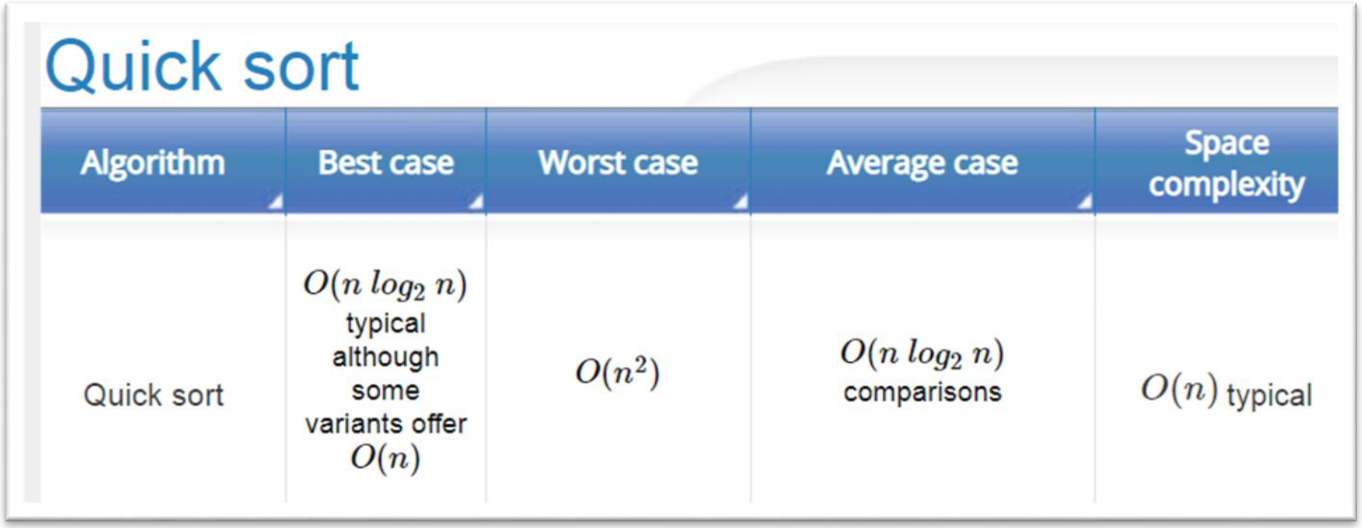
This is very similar to the merge sort procedure except the pivot can be applied to any item in the unsorted list.

Comparison



Merge sort

Algorithm	Best case	Worst case	Average case	Space complexity
Merge sort	$O(n\ log_2\ n)$ typical $O(n)$ natural merge sort variant	$O(n\ log_2 n)$ comparisons	$O(n\ log_2\ n)$ comparisons	$O(1)$ with linked lists or $O(n)$ otherwise



Quick sort

Algorithm	Best case	Worst case	Average case	Space complexity
Quick sort	$O(n\ log_2\ n)$ typical although some variants offer $O(n)$	$O(n^2)$	$O(n\ log_2\ n)$ comparisons	$O(n)$ typical

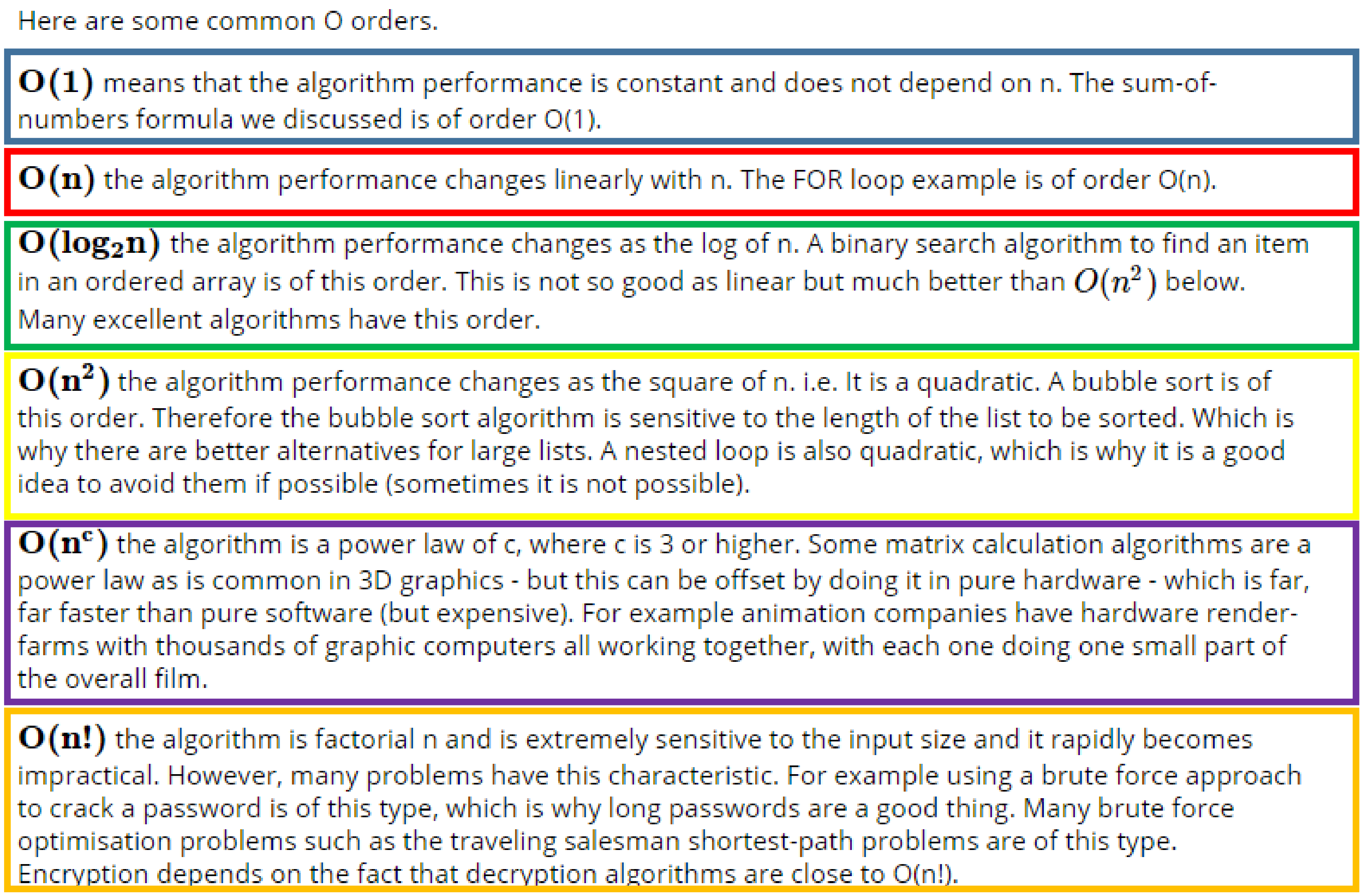
Key question: How is Big O notation used to describe the complexity of algorithms?

Algorithms are often judged and compared based on their efficiency and the resources they require. One of the most common ways to evaluate an algorithm is to look at its time complexity through a method called Big O Notation.

Big O notation is a way to describe the speed or complexity of an algorithm, and shows the worst case number of operations for a given input size. It's important to understand the possible run time for different algorithms, especially when working with large or growing data sets. Big O notation makes it easier to choose the right algorithm for each task.

Unit 2 | 2.3.1 | Algorithms

Key question: How is Big O notation used to describe the complexity of algorithms?



Here are some common O orders.

$\mathbf{O(1)}$ means that the algorithm performance is constant and does not depend on n. The sum-of-numbers formula we discussed is of order O(1).

$\mathbf{O(n)}$ the algorithm performance changes linearly with n. The FOR loop example is of order O(n).

$\mathbf{O(\log_2 n)}$ the algorithm performance changes as the log of n. A binary search algorithm to find an item in an ordered array is of this order. This is not so good as linear but much better than $O(n^2)$ below. Many excellent algorithms have this order.

$\mathbf{O(n^2)}$ the algorithm performance changes as the square of n. i.e. It is a quadratic. A bubble sort is of this order. Therefore the bubble sort algorithm is sensitive to the length of the list to be sorted. Which is why there are better alternatives for large lists. A nested loop is also quadratic, which is why it is a good idea to avoid them if possible (sometimes it is not possible).

$\mathbf{O(n^c)}$ the algorithm is a power law of c, where c is 3 or higher. Some matrix calculation algorithms are a power law as is common in 3D graphics - but this can be offset by doing it in pure hardware - which is far, far faster than pure software (but expensive). For example animation companies have hardware render-farms with thousands of graphic computers all working together, with each one doing one small part of the overall film.

$\mathbf{O(n!)}$ the algorithm is factorial n and is extremely sensitive to the input size and it rapidly becomes impractical. However, many problems have this characteristic. For example using a brute force approach to crack a password is of this type, which is why long passwords are a good thing. Many brute force optimisation problems such as the traveling salesman shortest-path problems are of this type. Encryption depends on the fact that decryption algorithms are close to O(n!).

Unit 2 | 2.3.1 | Algorithms

An important class of algorithms in computer science deals with this question:
If there are a several ways to arrive at a goal, and there are choices to be made along the way, what are the best set of choices and how do we find that optimal solution?

This is called a path-finding problem and it happens in many fields. For example:

- **Navigation**- What is the best path to get from A to B along a transport network?
- **The internet**- What is the most efficient route to get a packet of data to its destination?
- **Speech recognition** - a computer needs to parse a spoken command with many possibilities.
- **Image recognition** - being able to classify/recognise an image from a huge set of possibilities.
- **Artificial Intelligence** - training neural nets to work out an optimal strategy e.g playing chess.
- **Robotics 2D** - controlling their path from A to B.
- **Robotics 3D** - finding the optimal path for a robotic arm to move in 3D space.
- **Gaming** - path finding and NPC control in game worlds.
- **Finance** - help make optimal invest
- **Military** - unmanned drones can us
- **Social network analysis** - friendship networks, influencers, meme propagation.

Two examples of these Algorithms are Dijkstra's shortest path & A*…

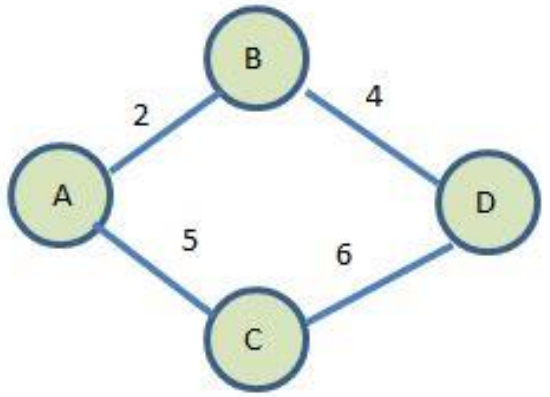
Key question: How does Dijkstra's shortest path algorithm work?

This algorithm is also called the **Uniform Cost Search** algorithm because no heuristic $h(n)$ is involved with finding the optimal path. The cost expression of any node is:

$$f(n)=g(n)$$

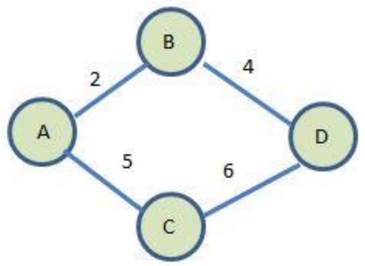
Where $g(n)$ is the cost of getting from the start to node n.

When n is the target, then the path that offers the smallest $f(n)$ is the best path to take. The Dijkstra algorithm is trying to find the A,b,d path below as that is the lowest cost path from A to D.



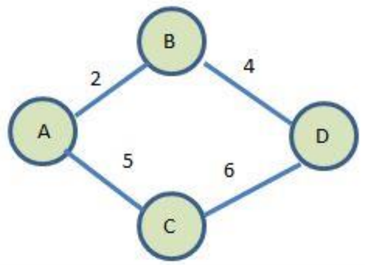
Path	f(d)
A,c,d	11
A,b,d	6

Dijkstra's shortest path algorithm – Algorithm.



Path	f(d)
A,c,d	11
A,b,d	6

1. Assign a cost of zero to the start node
2. Assign a cost of infinity to every other node
3. Assign every node to an unvisited set
4. From the start node, visit every one of its neighbours
5. If the cost of getting to that neighbour (the edge value) is less than the current stored cost, replace it with the lower cost (as we started with infinity, the neighbouring nodes from the start will always be the edge cost)
6. If all its neighbours have been visited, remove the node from the unvisited list
7. From the nodes just visited, find the one with the least current cost
8. Now visit its neighbours that are still in the unvisited set
9. The cost of each node is the sum of the edge value plus the cost of the prior node
10. Repeat steps 5 to 9 until all nodes have been visited
11. Identify the least cost path in this last iteration



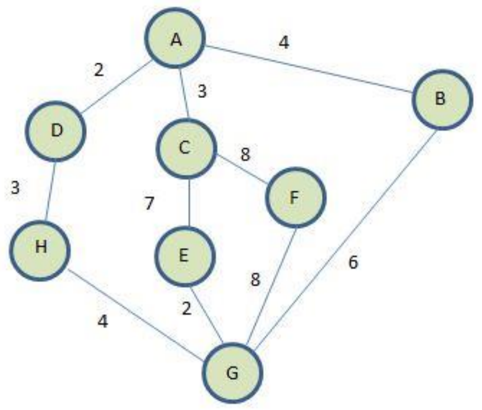
Path	f(d)
A,c,d	11
A,b,d	6

Dijkstra's shortest path algorithm – Pseudocode.

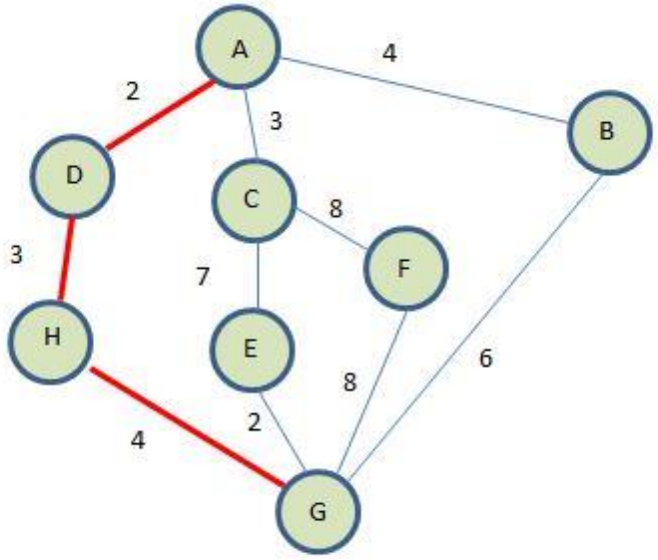
```
function Dijkstra (graph, start, target)  # it is given the graph to work with and the start node
                                          # and the target node
   Create an unvisited empty set Q # this will store all unvisited vertices
   Create an empty array cost_so_far[]  # this will hold vertex costs
   Create a came_from list {} # identifies which prior node of each node had the least cost

   For each vertex in graph
        append v to set Q   # every vertex is initially in Q
   end for

   For each vertex v in Q   # initialise the costs array
          cost_so_far[v] = infinity
   end for
   cost_so_far[start] = 0  # the start node is set to cost zero

   while Q is not empty   # loop until every node has been visited
      u = the vertex with the least cost in Q  # this is start on first iteration
                                               # becasue is has cost zero
      for each neighbour v of u still in Q
              newcost = cost[u] + edge cost (u to v) #cost of getting to u
                                                     # plus the edge cost u to v
              if newcost < costs[v] then   # a new lower cost path has been found to v
                 costs[v] = newcosts       # update the cost of v
                 came_from[v] = u          # this is on the optimal path so far
       end for
       remove u from Q

    end while
    # No more nodes to visit
    iterate through the came_from list in reverse order # this is the best path found
```

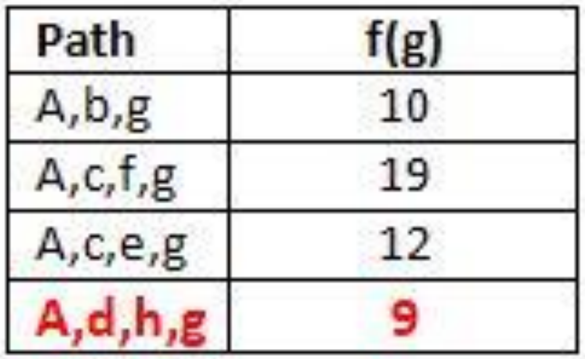
Dijkstra’s shortest path algorithm – Worked example.

For this small graph, we can use the brute force method i.e. work out every path's *f(g)* and determine which one is the smallest.



Each path's *f(g)* is the sum of the edges along that path. We want to show that Dijkstra can find it. In this case the least cost path is a,d,h,g.





Path	f(g)
A,b,g	10
A,c,f,g	19
A,c,e,g	12
A,d,h,g	**9**

Key question: How does the A* algorithm work?

Dijkstra's algorithm does not make any guesses (heurisitc) as to which is the best path to the target. And because of this, it can find the best path from any node to any other node in a reasonable amount of steps.

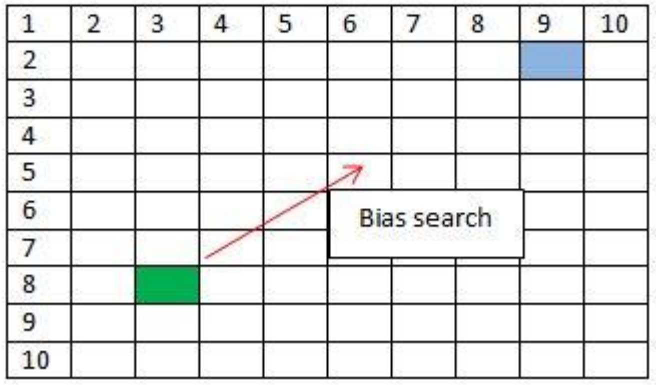
However, if a guess ***is*** made, then it may result in getting to the answer faster (i.e. fewer steps) in other words an algorithm that does this…

$$f(n)=g(n)+h(n)$$

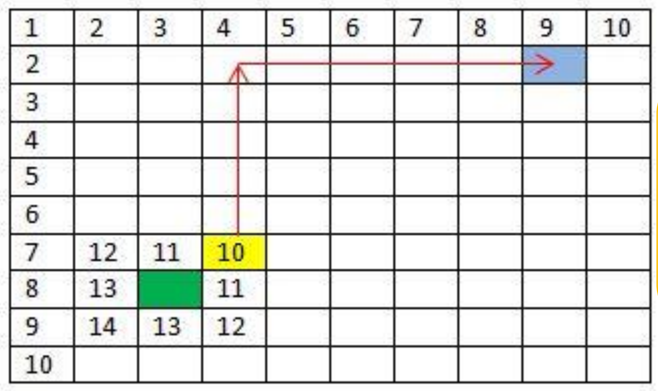
…may be better than just $f(n)=g(n)$.

This is what A-star does, it follows the Dijkstra algorithm but now includes includes a heuristic h(n) that biases the search towards the target.

Key question: How does the A* algorithm work?



For example getting from the green node to the blue node below is clearly better if the algorithm favoured nodes closer to the target than those further away.



The diagram below shows the horizontal and vertical distance of each of green's neighbour from the blue node. Sometimes this is called the Manhattan heurisitic.

For example the yellow node has heurisitc 10 as it is 5 vertical and 5 horizontal from blue. Every other node around green has a higher heurisitc than this, therefore the A star algorithm will use this node first. Then it recalculates its neighbours values, which again biases the search towards the top right.

Key question: How does the A* algorithm work? – Algorithm.

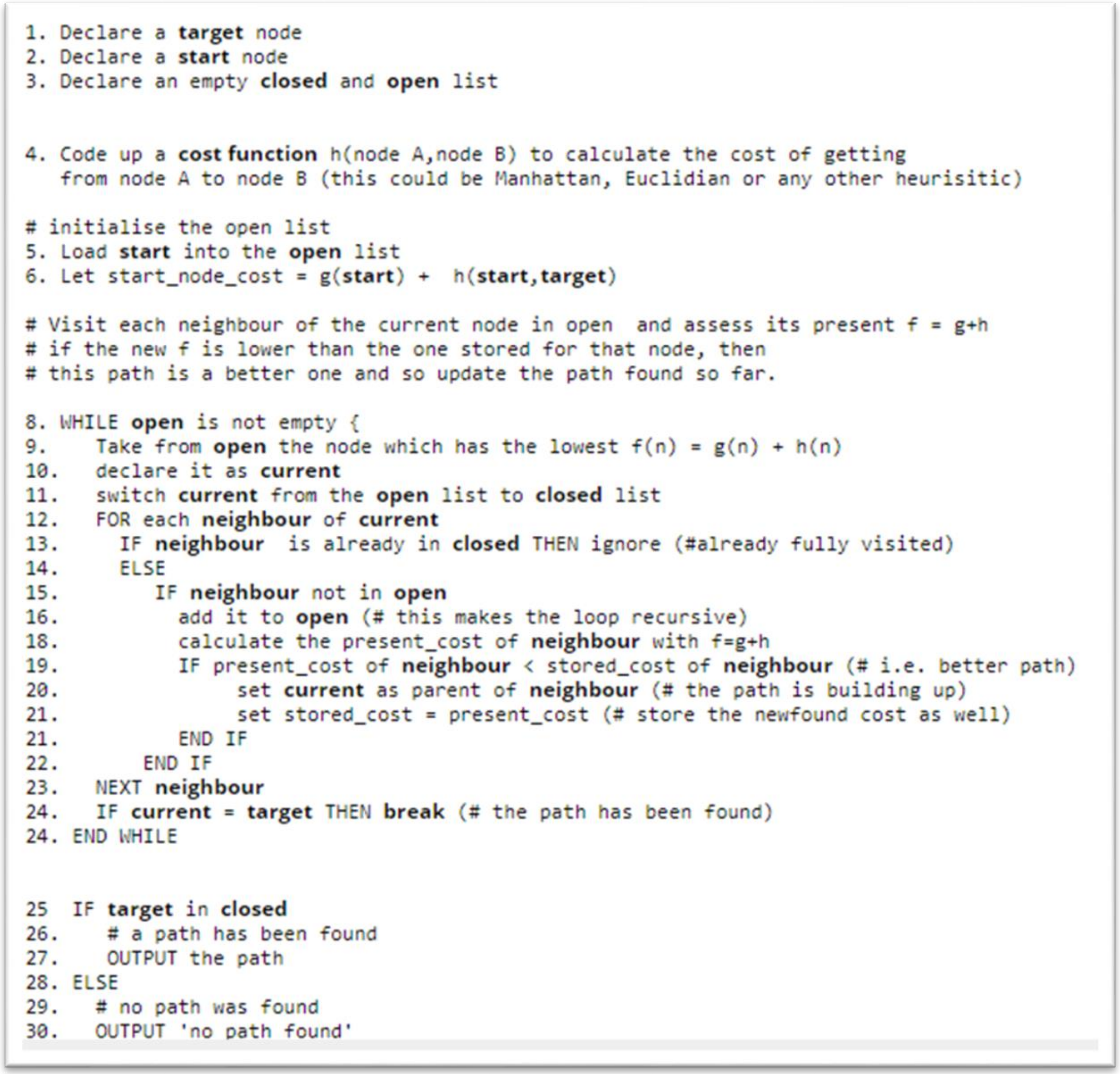
There are various ways to code up the algorithm. This one is making use of two lists: open and closed as well as a heuristic function 'h'.
The total cost of getting from the present node to the target is f = g + h where g is the same one as the Dijkstra algorithm.

Open: keeps a list of the current nodes' neighbours that have not been visited yet. It starts with only the start node.

Closed: keeps a list of the current nodes' neighbours that have been fully visited. It starts out as empty and it should contain the target if a path was found.

H(current node, target node) : a heuristic function. This could be the Manhattan, Euclidean or any other relevant calculation.

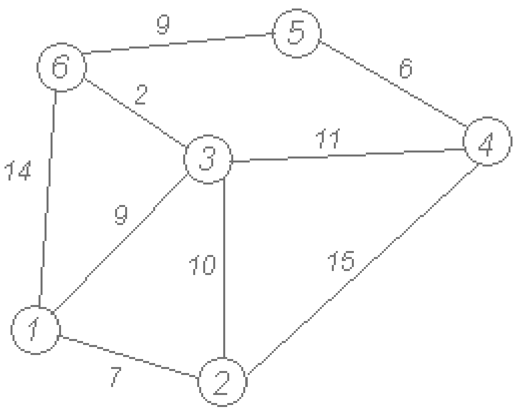
Key question: How does the A* algorithm work? – Algorithm.

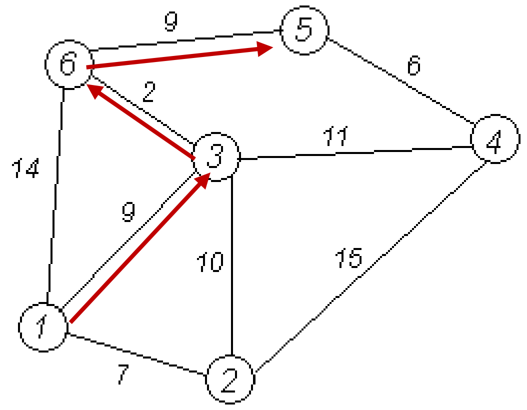


```
1. Declare a target node
2. Declare a start node
3. Declare an empty closed and open list


4. Code up a cost function h(node A,node B) to calculate the cost of getting
   from node A to node B (this could be Manhattan, Euclidian or any other heurisitic)

# initialise the open list
5. Load start into the open list
6. Let start_node_cost = g(start) +  h(start,target)

# Visit each neighbour of the current node in open  and assess its present f = g+h
# if the new f is lower than the one stored for that node, then
# this path is a better one and so update the path found so far.

8. WHILE open is not empty {
9.    Take from open the node which has the lowest f(n) = g(n) + h(n)
10.   declare it as current
11.   switch current from the open list to closed list
12.   FOR each neighbour of current
13.     IF neighbour  is already in closed THEN ignore (#already fully visited)
14.     ELSE
15.        IF neighbour not in open
16.          add it to open (# this makes the loop recursive)
18.          calculate the present_cost of neighbour with f=g+h
19.          IF present_cost of neighbour < stored_cost of neighbour (# i.e. better path)
20.               set current as parent of neighbour (# the path is building up)
21.               set stored_cost = present_cost (# store the newfound cost as well)
21.          END IF
22.       END IF
23.   NEXT neighbour
24.   IF current = target THEN break (# the path has been found)
24. END WHILE


25  IF target in closed
26.    # a path has been found
27.    OUTPUT the path
28. ELSE
29.   # no path was found
30.   OUTPUT 'no path found'
```

Typical exam questions

The graph in Fig.1 illustrates a path between nodes.





1. Using Dijkstra's shortest path algorithm, show the shortest path between nodes 1 and 5. You may use a table to illustrate your answer. **[6]**

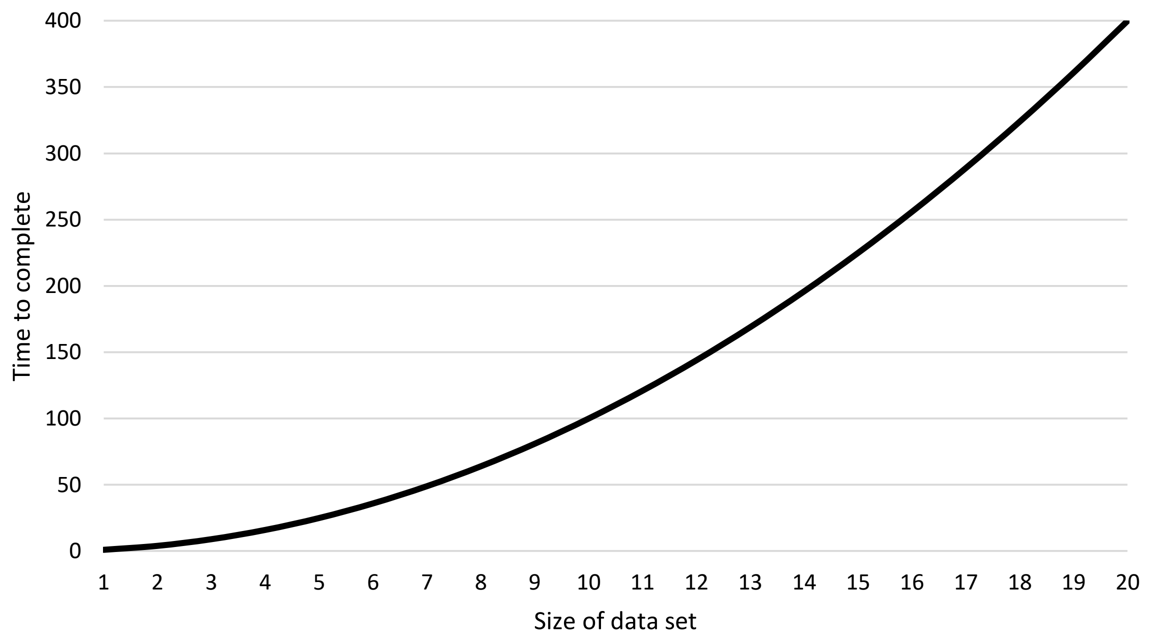
Vertex	Shortest distance from 1	Previous vertex
1	Inf	
2	7	1
3	9	1
4	20	3
5	20	6
6	11	3

Shortest path is 1 ➔ 3 ➔ 6 ➔ 5

Typical exam questions

2. This algorithm has a polynomial (quadratic) complexity of O(n^2)

Draw a graph of the number of nodes against the time to complete to illustrate the efficiency of the algorithm. **[2]**



Unit 2 | 2.3.1 | Algorithms

Target: ☐ Overall grade: ☐

Minimum expectations & learning outcomes

☐	Terms 234-243 from your A Level Key Terminology should be included and formatted.
☐	You must provide a worked example and pseudocode for each of the following algorithms and data structures: Tree traversals algorithms.
☐	Bubble sort, insertion sort, merge sort, quick sort, Dijkstra's shortest path algorithm, A* algorithms, binary search and linear search.
☐	You must comment on the time and space complexity of each algorithm.
☐	Answer the exam questions.

Feedback

Breadth	Depth	Presentation	Understanding
☐ All	☐ Analysed	☐ Excellent	☐ Excellent
☐ Most	☐ Explained	☐ Good	☐ Good
☐ Some	☐ Described	☐ Fair	☐ Fair
☐ Few	☐ Identified	☐ Poor	☐ Poor

Comment & action required

Unit 2 | 2.3.1 | Algorithms

Reflection & Revision checklist

Confidence	Clarification
☹ 😐 ☺	Candidates need to be able to write algorithms using flow charts, pseudocode and program code.
☹ 😐 ☺	Candidates need to be able to follow the code as shown in the OCR pseudocode guide but are not expected to write code in syntax.
☹ 😐 ☺	Candidates' code is not expected to be syntactically perfect but must use appropriate structures and techniques.
☹ 😐 ☺	Candidates need to understand that there are a range of possible solutions to a task, and that these algorithms may be different in respect to their execution time and the amount of memory they make use of.
☹ 😐 ☺	Candidates need to be able to compare different algorithms for a given data set and demonstrate an understanding of which is more efficient in terms of speed and/or memory.
☹ 😐 ☺	Candidates need to be able to compare the use of one or more algorithms against several different data sets, to determine how they will differ in their use of memory and speed of execution.
☹ 😐 ☺	Candidates need to understand how the efficiency of an algorithm is measured using Big O notation.
☹ 😐 ☺	Candidates need to understand the meaning of constant, linear, polynomial, exponential and logarithmic complexity. They need to be able to recognise and draw each of these complexities of using a graph and be able to read and write the notation.
☹ 😐 ☺	Candidates need to know the best- and worst-case complexities for the searching and sorting methods.
☹ 😐 ☺	Candidates need to understand the difference between best case, average case and worst-case complexities and how and why these can differ for an algorithm.
☹ 😐 ☺	Candidates need to have an understanding of the situations where queues, stacks, trees etc. can be used and be able to recommend and justify their use In specific scenarios or programs.
☹ 😐 ☺	Candidates need to have an understanding of a stack as a dynamic data structure.
☹ 😐 ☺	Candidates need to be able to add and remove items to a stack.
☹ 😐 ☺	Candidates need to be able read, trace and write code to implement a stack structure (including adding and removing items).
☹ 😐 ☺	Candidates need to understand how a stack can be implemented using a different data structure, such as a static array.
☹ 😐 ☺	Candidates need to have an understanding of a queue as a dynamic data structure.
☹ 😐 ☺	Candidates need to be able to add and remove data to/from a queue.

Unit 2 | 2.3.1 | Algorithms

Reflection & Revision checklist

Confidence	Clarification
☹ 😐 ☺	Candidates need to be able to read, trace and write code to implement a queue structure (including adding and removing items).
☹ 😐 ☺	Candidates need to understand how a queue can be implemented using a different data structure, such as a static array.
☹ 😐 ☺	Candidates need to have an understanding of a tree structure, both binary and multi branch trees.
☹ 😐 ☺	Candidates need to be able to add and remove data to/from a tree.
☹ 😐 ☺	Candidates need to be able to read, trace and write code to implement a tree structure (including adding and removing items).
☹ 😐 ☺	Candidates need to understand how a tree can be implemented using a different data structure, such as a linked list.
☹ 😐 ☺	Candidates need to understand why and how trees are traversed.
☹ 😐 ☺	Candidates need to understand how a depth-first (post-order) traversal works and be able to perform the traversal on a tree.
☹ 😐 ☺	Candidates need to be able to read, trace and write code for a post-order traversal.
☹ 😐 ☺	Candidates need to understand how a breadth-first traversal works and be able to perform the search on a tree.
☹ 😐 ☺	Candidates need to be able to read, trace and write code for a breadth-first traversal on a tree.
☹ 😐 ☺	Candidates' code is not expected to be syntactically perfect but must use appropriate structures and techniques.
☹ 😐 ☺	Candidates need to have an understanding of a linked list as a dynamic data structure.
☹ 😐 ☺	Candidates need to be able to add, remove and search for data to/from/in a linked list.
☹ 😐 ☺	Candidates need to be able to read, trace and write code to implement a linked list (including adding, removing and search for items).
☹ 😐 ☺	Candidates need to have an understanding of the need for searching and sorting algorithms.
☹ 😐 ☺	Candidates need to have an understanding of pre-conditions required to perform a specific algorithm.
☹ 😐 ☺	Candidates need to understand how a bubble sort works and be able to perform a bubble sort on a set of data.
☹ 😐 ☺	Candidates need to be able to read, trace and write code to perform a bubble sort.
☹ 😐 ☺	Candidates need to understand how a merge sort works and be able to perform a merge sort on a set of data.

Reflection & Revision checklist

Confidence	Clarification
☹ 😐 ☺	Candidates need to be able to read, trace and write code to perform a merge sort.
☹ 😐 ☺	Candidate need to understand how a quick sort works and be able to perform a quick sort on a set of data.
☹ 😐 ☺	Candidates need to be able to read, trace and write code to perform a quick sort.
☹ 😐 ☺	Candidates need to understand how Dijkstra’s shortest path algorithm works.
☹ 😐 ☺	Candidates need to be able to calculate the shortest path in a graph or tree using Dijkstra’s shortest path algorithm.
☹ 😐 ☺	Candidates need to be able to read and trace code that performs Dijkstra’s shortest path algorithm.
☹ 😐 ☺	Candidates need to understand how the A* algorithm works.
☹ 😐 ☺	Candidates need to be able to calculate the shortest path in a graph or tree using the A* algorithm.
☹ 😐 ☺	Candidates need to be able to read and trace code that performs the A* algorithm.
☹ 😐 ☺	Candidates need to understand how a binary search works and be able to perform a binary search on a set of data.
☹ 😐 ☺	Candidates need to be able to read, trace and write code to perform a binary search.
☹ 😐 ☺	Candidates need to understand how a linear search works and be able to perform a linear search on a set of data.
☹ 😐 ☺	Candidates need to be able to read, trace and write code to perform a linear search.