OCR A-Level Computer Science Spec Notes
2.3 Algorithms

2.3.1 Algorithms

(a) Analysis and design of algorithms for a given situation

Algorithms: Set of instructions that complete a task when execute

- Algorithms run by computers are called **'programs'**
- Scale algorithms by:
    - The **time** it takes for the algorithm to complete
    - The **memory/resources** the algorithm needs. '**space**'.
    - **Complexity (Big O notation)**

(b) The suitability of different algorithms for a given task and data set, in terms of execution time and space

There are **different suitable algorithm**s for **each task**

- **Space efficiency:**
    - The measure of how much memory (**space**) the algorithm takes as its input (**N**) is scaled up
    - Space **increases linearly** with N
    - Code space is **constant/data space** is also **constant**
- **Time efficiency**
    - Measure of how much **time** it takes to **complete an algorithm** as its input (**N**) increases
    - Time increases **linearly** with N
    - **Sum of numbers = n(n+1)/2**
- **Big O notation**
    - Refer to ((c) Measures and methods to determine the efficiency of algorithms (Big O) notation (constant, linear, polynomial, exponential and logarithmic complexity))

(c) Measures and methods to determine the efficiency of algorithms (Big O) notation (constant, linear, polynomial, exponential and logarithmic complexity)

(Big O) notation

- Shows **highest order component** with any constants removed to evaluate the **complexity** and **worst-case scenario** of an **algorithm**.
- Shows how **time increases** as **data size increases** to show **limiting behaviour**.

Big O Notation

- **O(1) – Constant complexity** e.g. printing first letter of string.
- **O(n) – Linear complexity** e.g. finding largest number in list.
- **O(kn) – Polynomial complexity** e.g. bubble sort.
- **O(k^n) – Exponential complexity** e.g. travelling salesman problem.
- **O(logn) – Logarithmic complexity** e.g. binary search

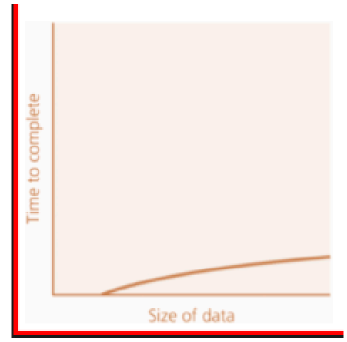
(d) Comparison of the complexity of algorithms

Complexity

- Complexity is a measure of how much time, **memory space** or **resources** needed for an algorithm **increases** as the **data size** it works on **increases**.
- Represents the **average complexity** in **Big-O notation**.
- Big-O notation just shows the **highest order component** with any **constants removed**.
- Shows the **limiting behaviour** of an algorithm to classify its complexity.
- Evaluates the **worst case scenario** for the **algorithm**.

Types of Complexity

Complexity	Description	Graph
Constant complexity O(1)	- **Time taken** for an algorithm stays the **same** regardless of the **size** of the data set - **Example:** Printing the first letter of a string. No matter how big the string gets it won't take longer to display the first letter.	
Linear complexity O(n)	- This is where the **time taken** for an algorithm **increases proportionally** or at the **same rate** with the **size of the data set**. - Example: Finding the largest number in a list. If the list size doubles, the time taken doubles.	
Polynomial complexity O(kn) (where k>=0)	- This is where the time taken for an **algorithm increases proportionally to n** to the **power** of a **constant**. - Bubble sort is an example of such an algorithm.	
Exponential complexity O(k^n) (where k>1)	- This is where the time taken for an **algorithm increases exponentially** as the data set **increases**. - **Travelling Salesman Problem** = example algorithm. - The inverse of **logarithmic growth**. - Does not scale up well when **increased** in **number** of **data items**.	

Logarithmic complexity O(log n)	- This is where the time taken for an algorithm **increases logarithmically** as the **data set increases**. - As **n increases**, the **time taken increases** at a **slower rate**, e.g. Binary search. - The **inverse of exponential growth**. - **Scales up well** as does not **increase significantly** with the **number of data items**.	

(e) Algorithms for the main data structures (stacks, queues, trees, linked lists, depth-first (post-order) and breadth-first traversal of trees)

Data Structures	Description	Algorithm

Stack PUSH	- When a data item is **added** to the **top** of a stack	PROCEDURE AddToStack (item):
		    IF top == max THEN
		      stackFull = True
		    ELSE
		      top = top + 1
		      stack[top] = item
		    ENDIF
		  ENDPROCEDURE

Stack POP	- When a data item is **removed** from the **top** of a stack	PROCEDURE DeleteFromStack (item):
		    IF top == min THEN
		      stackEmpty = True
		    ELSE
		      stack[top] = item
		      top = top - 1
		    ENDIF
		  ENDPROCEDURE

Queue PUSH	- When a data item is **added** to the **back** of a queue	PROCEDURE AddToQueue (item):
		    IF ((front - rear) + 1) == max THEN
		      queueFull = True
		    ELSE
		      rear = rear - 1
		      queue[rear] = item
		    ENDIF
		  ENDPROCEDURE

Queue POP	- When a data item is **removed** from the **front** of a queue	PROCEDURE DeleteFromQueue (item):
		    IF front == min THEN
		      queueEmpty = True
		    ELSE
		      queue[front] = item
		      front = front + 1
		    ENDIF
		  ENDPROCEDURE

Linked List (Output in Order)	- When the contents of a linked list are **displayed in order**	`FUNCTION OutputLinkedListInOrder ():`
		`Ptr = start value`
		`REPEAT`
		`Go to node(Ptr value)`
		`OUTPUT data at node`
		`Ptr = value of next item Ptr at node`
		`UNTIL Ptr = 0`
		`ENDFUNCTION`

Linked List (Add item to list)	- When a data item is added **anywhere** on a **linked list**	`FUNCTION SearchForItemInLinkedList ():`
		`Ptr = start value`
		`REPEAT`
		`Go to node(Ptr value)`
		`IF data at node == search item`
		`OUTPUT AND STOP`
		`ELSE`
		`Ptr = value of next item Ptr at node`
		`ENDIF`
		`UNTIL Ptr = 0`
		`OUTPUT data item not found`
		`ENDFUNCTION`

Tree Traversal	Description	Algorithm

Depth first (post-order)	- Visit **all nodes** to the **left of the root node**
	- Visit **right**
	- Visit **root node**
	- Repeat **three points for each node visited**
	- Depth first isn't **guaranteed** to find the **quickest solution** and possibly **may never find the solution** if no precautions to revisit **previously visited states**.
		`FUNCTION dfs(graph, node, visited):`
		`markAllVertices (notVisited)`
		`createStack()`
		`start = currentNode`
		`markAsVisited(start)`
		`pushIntoStack(start)`
		`WHILE StackIsEmpty() == false`
		`popFromStack(currentNode)`
		`WHILE allNodesVisited() == false`
		`markAsVisited(currentNode)`
		`//following sub-routine pushes all nodes connected to`
		`//currentNode AND that are unvisited`
		`pushUnvisitedAdjacents()`
		`ENDWHILE`
		`ENDWHILE`
		`ENDFUNCTION`

Breadth first	- Visit **root node**
	- Visit all **direct subnodes (children)**
	- Visit all **subnodes of first subnode**
	- Repeat **three points** for each **subnode visited**
	- Breadth first requires **more memory** than Depth first search.
	- It is **slower** if you are looking at **deep parts** of the tree.
		<pre>FUNCTION bfs(graph, node):
		   markAllVertices (notVisited)
		   createQueue()
		   start = currentNode
		   markAsVisited(start)
		   pushIntoQueue(start)
		   WHILE QueueIsEmpty() == false
		     popFromQueue(currentNode)
		     WHILE allNodesVisited() == false
		       markAsVisited(currentNode)
		       //following sub-routine pushes all nodes connected to
		       //currentNode AND that are unvisited
		       pushUnvisitedAdjacents()
		     ENDWHILE
		   ENDWHILE
		 ENDFUNCTION</pre>

(f) Standard algorithms (bubble sort, insertion sort, merge sort, quick sort, Dijkstra's shortest path algorithm,A* algorithm, binary search and linear search)

Sort	Description	Algorithm

Bubble Sort	- Is **intuitive** (easy to understand and program) but **inefficient**.
	- Uses a **temp element**.
	- Moves through the data in the **list repeatedly** in a **linear way**
	- Start at the **beginning** and **compare** the **first item** with the **second**.
	- If they are out of order, **swap them** and set a **variable swapMade true**.
	- Do the same with the **second and third item**, **third and fourth**, and so on until the **end of the list**.
	- When, at the end of the list, **if swapMade is true**, change it to **false** and **start again**; otherwise, If it is **false**, the **list is sorted** and the **algorithm stops**.
		<pre>PROCEDURE (items):
		    swapMade = True
		    WHILE swapMade == True
		      swapMade = False
		      position = 0
		      FOR position = 0 TO length(list) - 2
		        IF items[position] > items[position + 1] THEN
		          temp = items[position]
		          items[count] = items[count + 1]
		          items[count + 1] = temp
		          swapMade = True
		        ENDIF
		      NEXT position
		    ENDWHILE
		    PRINT(items)
		  ENDPROCEDURE</pre>

Insertion Sort	- Works by **dividing** a list into **two parts**: **sorted and unsorted** - Elements are inserted **one by one** into their **correct position** in the **sorted section** by **shuffling them left** until they are **larger** than the item to the **left** of them until all items in the list are **checked**. - **Simplest sort algorithm** - **Inefficient** & takes longer for **large sets of data**	PROCEDURE InsertionSort (list): item = length(list) FOR index = 1 TO item - 1 currentvalue = list[index] position = index WHILE position > 0 AND list[position - 1] > currentvalue list[position] = list[position - 1] position = position - 1 ENDWHILE list[position] = currentvalue NEXT index ENDPROCEDURE

Merge Sort	- Works by splitting **n data items** into **n sublists one item big**.
	- These lists are then **merged** into **sorted lists two items big**, which are **merged into lists four items big**, and so on until there is **one sorted list**.
	- Is a **recursive algorithm** = require **more memory space**
	- Is **fast** & **more efficient** with **larger volumes** of data to sort.
		PROCEDURE MergeSort (listA, listB):
		a = 0
		b = 0
		n = 0
		WHILE length(listA) > 1 AND length(listB) > 1
		IF listA(a) < listB(b) THEN
		newlist(n) = listA(a)
		a = a + 1
		ELSE
		newlist(n) = listB(b)
		b = b + 1
		ENDIF
		n = n + 1
		ENDWHILE
		WHILE length(listA) > 1
		newlist(n) = listA(a)
		a = a + 1
		n = n + 1
		ENDWHILE
		WHILE length(listB) > 1
		newlist(n) = listB(b)
		b = b + 1
		n = n + 1
		ENDWHILE
		ENDPROCEDURE

Quick Sort	- Uses **divide and conquer** - Picks an item as a **'pivot'**. - It then creates two **sub-lists**: those **bigger** than the pivot and those **smaller.** - The same process is then applied **recursively/iteratively** to the **sub-lists** until all items are **pivots**, which will be in the **correct order**. - Alternative **method uses two pointers**. - **Compares** the numbers at the **pointers** and swaps them if they are in the **wrong order**. - Moves **one pointer at a time**. - **Very quick** for **large sets of data**. - Initial arrangement of **data affects the time taken**. - **Harder to code**.	PROCEDURE QuickSort (list, leftPtr, rightPtr): leftPtr = list[start] rightPtr = list[end] WHILE leftPtr! != rightPtr WHILE list[leftPtr] < list[rightPtr] AND leftPtr! != rightPtr leftPtr = leftPtr + 1 ENDWHILE temp = list[leftPtr] list[leftPtr] = list[rightPtr] list[rightPtr] = temp WHILE list[leftPtr] < list[rightPtr] AND leftPtr! != rightPtr rightPtr = rightPtr - 1 ENDWHILE temp = list[leftPtr] list[leftPtr] = list[rightPtr] list[rightPtr] = temp ENDWHILE ENDPROCEDURE

Path Algorithms	Description	Algorithm

Dijkstra's shortest path algorithm	- Finds the **shortest path** between **two nodes** on a graph.
	- It works by keeping **track of the shortest distance** to each **node** from the **starting node**.
	- It **continues** this until it has **found the destination node**.
		FUNCTION Dijkstra ():
		start node distance from itself = 0
		all other nodes distance from start node = infinity
		WHILE destination node = unvisited
		current node = closest unvisited node to A // initially this will be A itself
		FOR every unvisited node connected to current node:
		distance = distance to current node + distance of edge to unvisited node
		IF distance < currently recorded shortest distance THEN
		distance = new shortest distance
		NEXT connected node
		current node = visited
		ENDWHILE
		ENDFUNCTION

<table>
<tr>
<td>A* algorithm</td>
<td>

- **Improvement** on **Dijkstra's algorithm**.
- **Heuristic approach** to estimate the **distance** to the **final node**, = **shortest path** in **less time**
- Uses the **distance** from the **start node plus** the **heuristic estimate** to the **end node**.
- Chooses which **node** to **take next** using the **shortest distance + heuristic**.
- All **adjoining nodes** from this **new node are taken**.
- Other **nodes** are **compared again in future checks**.
- Assumed that this **node** is a **shorter distance**.
- **Adjoining nodes** may **not** be **shortest path** so may need to **backtrack to previous nodes**.

</td>
<td>
<pre>
FUNCTION AStarSearch ():
    start node = current node
    WHILE destination node = unvisited
    FOR each open node directly connected to the current node
      Add to the list of open nodes.
      g = distance from the start
      h = heuristic estimate of the distance left
      f = g + h
    NEXT connected node
    current node = unvisited node with lowest value
    ENDWHILE
  ENDFUNCTION
</pre>
</td>
</tr>
</table>

<table>
<tr>
<th>Search Type</th>
<th>Description</th>
<th>Algorithm</th>
</tr>
<tr>
<td>Binary Search Recursive</td>
<td>

- Requires the list to be **sorted in order** to allow the **appropriate items to be discarded**.
- It involves checking the item in the **middle** of the **bounds of the space being searched.**
- It the **middle item is bigger** than the item we are looking for, it becomes the **upper bound.**
- If it is s**maller than the item we are looking for**, it

</td>
<td>
<pre>
FUNCTION BinaryS (list, value, leftPtr, rightPtr):
    IF rightPtr < leftPtr THEN
      RETURN error message
    ENDIF
    mid = (leftPtr + rightPtr)/2)
    IF list[mid] > value THEN
      RETURN BinaryS (list, value, leftPtr, mid-1)
    ELSEIF list[mid] < value THEN
      RETURN BinaryS (list, value, mid+1, rightPtr)
    ELSE
      RETURN mid
  ENDFUNCTION
</pre>
</td>
</tr>
</table>

Binary Search Iterative	becomes the **lower bound.**
	- Repeatedly discards and **halves** the list at **each step** until the **item** is found.
	- Is usually faster in a **large set of data** than **linear search** because **fewer items** are checked so is more **efficient for large files**.
	- Doesn't benefit from **increase in speed** with **additional processors**.
	- Can perform better on **large data sets** with **one processor** than **linear search with many processors**.
		`FUNCTION BinaryS (list, value, leftPtr, rightPtr):`
		`  Found = False`
		`  IF rightPtr < leftPtr THEN`
		`    RETURN error message`
		`  ENDIF`
		`  WHILE Found == False`
		`    mid = (leftPtr + rightPtr)/2)`
		`    IF list[mid] > value THEN`
		`      rightPtr = mid - 1`
		`    ELSEIF list[mid] < value THEN`
		`      leftPtr = mid + 1`
		`    ELSE`
		`      Found = True`
		`    ENDIF`
		`  ENDWHILE`
		`  RETURN mid`
		`ENDFUNCTION`

Linear Search	- Start at the **first location** and check each **subsequent location** until the **desired item is found** or the **end of the list is reached**.
	- Does not need an **ordered list** and **searches through all items** from the **beginning one by one**.
	- Generally performs much better than binary search if the **list is small** or if the item being searched for is **very close to the start of the list**
	- Can have **multiple processors** searching **different areas** at the same time.
	- Linear search **scales very** with **additional processors.**
		`FUNCTION LinearS (list, value):`
		`  Ptr = 0`
		`  WHILE Ptr < length(list) AND list[Ptr] != value`
		`    Ptr = Ptr + 1`
		`  ENDWHILE`
		`  IF Ptr >= length(list) THEN`
		`    PRINT("Item is not in the list")`
		`  ELSE`
		`    PRINT("Item is at location "+Ptr)`
		`  ENDIF`
		`ENDFUNCTION`

Summary

	Worst Case	Best Case
Bubble Sort	n^2	n
Insertion Sort	n^2	n
Merge Sort	n log n	n log n
Quick Sort	n^2	n log n
Binary Search	log_2 (n)	1
Linear Search	n	1