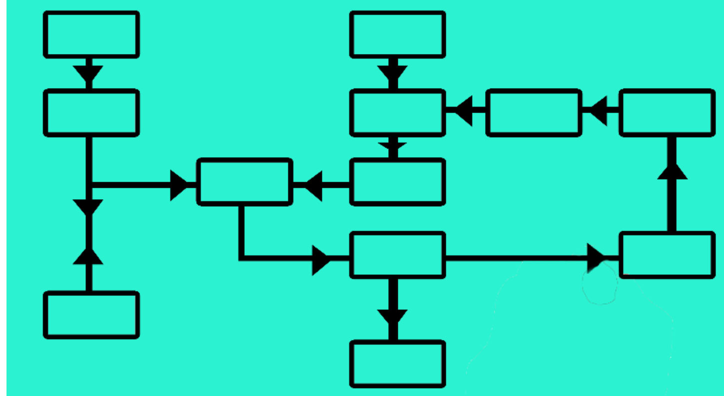
Unit 2.3 Algorithms

Algorithms
- A set of instructions used to solve a set problem.
- Inputs must be clearly defined.
- Must always produce a valid output.
- Must be able to handle invalid inputs.
- Must always reach a stopping condition.
- Must be well-documented for reference.
- Must be well-commented.

Designing Algorithms
- The priority for an algorithm is to achieve the given task.
- The second priority is to reduce time and space complexity.
- There may be a conflict between space and time complexity and the requirements and situation for an algorithm will dictate which is more important.
- To reduce space complexity, make as many changes on the original data as possible. Do not create copies.
- To reduce time complexity, reduce the number of loops.



Queues
- FIFO (First in first out)
- Often an array.
- The front pointer marks the position of the first element.
- The back pointer marks the position of the next available space.

Queue Functions
- Check size `size()`
- Check if empty `isEmpty()`
- Return top element (but don't remove) `peek()`
- Add to the queue `enqueue(element)`
- Remove element at the front of the queue and return it `dequeue()`

Big-O Notation
- 0(1) - Consistent time complexity - The amount of time is not affected by the number of inputs.
- 0(n) - Linear time complexity - The amount of time is directly proportional to the number of inputs.
- 0(nn) - Polynomial time complexity - The amount of time is directly proportional to the number of inputs to the power of n.
- 0(2n) - Exponential time complexity - The amount of time will double with every additional input.
- 0(log n) - Logarithmic time complexity - The amount of time will increase at a smaller rate as the number of inputs increases.

Sorting Algorithms
- Places elements into a logical order.
- Usually numerical or alphabetical.
- Usually in ascending order.
- Can be set to work in descending order.

Bubble Sort
- Compares elements and swaps as needed.
- Compares element 1 to element 2.
- If they are in the wrong order, they are swapped.
- This process is repeated with 2 and 3, 3 and 4, and so on until the end of the list is reached.
- This process must be repeated as many times as there are elements in the array.
- Each repeat is referred to as a "pass".
- Can be modified to improve efficiency by using a flag to indicate if a swap has occurred during the pass.
- If no swaps are made during a pass the list must be in the correct order and so the algorithm stops.
- A slow algorithm.
- Time complexity of 0(n2)

Insertion Sort
- Places elements into a sorted list.
- Starts at element 2 and compares it with the element directly to its left.
- When compared, elements are inserted into the correct position in the list.
- This repeats until the last element is inserted into the correct position.
- In the 1st iteration 1 element is sorted, in the 2nd iteration 2 are sorted etc.
- Time complexity of 0(n2)

Merge Sort
- A divide and conquer algorithm.
- Formed of a Merge and MergeSort function.
- MergeSort divides the input into two parts.
- It then recursively calls MergeSort on each part until their length is 1.
- Merge is called.
- Merge puts the groups of elements back together in a sorted order.
- You will not be asked about the detailed implementation of this algorithm but do need to know how it works.
- It is more efficient than bubble and merge sort.
- It has a worst case time of O(n log n)

Quick Sort
- Selects an element and divides the input around it.
- Often selects the central element, which is known as the pivot.
- Elements smaller than the pivot are listed to its left.
- Larger elements are listed to its right.
- The process is repeated recursively.
- Slow.
- Time complexity of O(n2)

Searching Algorithms
- Used to locate an element within a data structure.
- Many different forms exist.
- Each is suited to different purposes and data structures.

Linear Search
- Most basic search algorithm.
- Works through the elements one at a time until the requested element is found.
- Does not need data to be sorted.
- Easy to implement.
- Not very efficient.
- Time Complexity is 0(n)

Binary Search
- Only works with sorted data.
- Finds the middle element, then decides on which side of the data the requested element is.
- The unneeded half is discarded and the process repeats until either the requested element is found or it is determined that the requested element does not exist.
- A very efficient algorithm.
- Time Complexity is 0(log n)

Stacks
- FILO (First In Last Out)
- Often an array.
- Uses a single pointer (the top pointer) to track the top of the stack.
- The top pointer is initialised at -1, with the first element being 0, the second 1 and so on.

Stack Functions
- Check size `size()`
- Check if empty `isEmpty()`
- Return top element (but don't remove) `peek()`
- Add to the stack `push(element)`
- Remove top element from the stack and return it `pop()`

Space Complexity
- The amount of storage space the algorithm takes up.
- Does not have a defined notation.
- Copying data increases the storage used.
- Storage space is expensive so this should be avoided.

Linked Lists
- Contains several nodes.
- Each node has a pointer to the next item in the list.
- For node N, N.next will access the next item.
- The first node is the head.
- The last node is the tail.
- Searched using a linear search.

Time Complexity
- How much time an algorithm needs to solve a problem.
- Measured using big-o notation.
- Shows the amount of time taken relative to the number of inputs.
- Allows the required time to be predicted.

Logarithms
- The inverse of an exponential.
- An operation which determines how many times a certain number is multiplied by itself to reach another number.
- x y = log(x)
- 1 (20) 0
- 8 (23) 3
- 1024 (210) 10

Path Finding Algorithms

Dijkstra's Algorithm
- Finds the shortest path between two points.
- The problem is depicted as a weighted graph.
- Nodes represent the items in the scenario such as places.
- Edges connect the nodes together.
- Each edge has a cost.
- The algorithm will calculate the best way, known as the least cost path, between two nodes.

A* Algorithm
- Provides a faster solution than Dijkstra's Algorithm to find the shortest path between two nodes.
- Uses a heuristic element to decide which node to consider when choosing a path.
- Unlike Dijkstra's Algorithm, A* only looks for the shortest path between two nodes, instead of the shortest path from the start node to all other nodes.

Trees
- Consists of nodes and edges.
- Cannot contain cycles.
- Edges are not directed.
- Can be traversed using depth first or breadth first.
- Both methods can be implemented recursively.

Depth First (Post Order) Traversal
- Moves as far as possible through the tree before backtracking.
- Uses a stack.
- Moves to the left child node wherever possible.
- Will use the right child node if no left child node exists.
- If there are no child nodes, the current node is used.
- the algorithm then backtracks to the next node moving right.

Breadth First
- Starts from the left.
- Visits all children of the starting node.
- Then visits all nodes directly connected to each of these nodes in turn.
- Continues until all nodes have been visited.