2.3.1 Algorithms

Revision sheet

1 Algorithmic Complexity

There are two different measures used to measure the efficiency of an algorithm.

1.1 Space Complexity

This is looking at how much memory the algorithm uses when it is running.

1.2 Time Complexity

This is looking ay how many steps the algorithm takes to reach its goal. They can take different amounts of time depending on the data that has been inputted into them, so we tend to look a three different cases: best, average and worst case.

2 Big O Notation

Big Order Notation is a way of expressing the time complexity or performance of an algorithm. It looks at how many lines of code are executed. There are a number of different possible complexity values. They are listed below in the order best to worst.

2.1 Constant

$O(1)$
Algorithm always executes in the same amount of time regardless of the size of the dataset.

2.2 Logarithmic

$O(\log_n)$
Algorithm which halves the data set with each pass, efficient with large data sets, increases execution time at a slower rate than the rate at which the data set size increases.

2.3 Linear

$O(n)$
Algorithm whose performance declines as the data set grows, reduces efficiency with increasingly large data set.

2.4 Loglinear

$O(N \log_n)$
Algorithm that divide a data set but can be solved using concurrency on independent divided lists.

2.5 Polynomial

$O(N^2)$
Algorithm whose performance is proportional to the size of the data set, efficiency significantly reduces with increasing large data sets.

2.6 Exponential

$O(2^n)$
Algorithm that doubles with each addition to the data set in each pass, very inefficient.

2.7 Complexities for different algorithms

V = vertices and E = edges.

Algorithms	Time Complexity			Space Complexity
	Best	Average	Worst	
Linear Search	$O(1)$	$O(n)$	$O(n)$	-
Binary Search Array	$O(1)$	$O(\log_n)$	$O(\log_n)$	-
Binary Search Tree	$O(1)$	$O(\log_n)$	$O(n)$	-
Hashing	$O(1)$	$O(1)$	$O(n)$	-
Breadth-first or Depth-First graph	$O(1)$	$O(V+E)$	$O(V^2)$	-
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(N \log_n)$	$O(N \log_n)$	$O(N \log_n)$	
Quick Sort	$O(N \log_n)$	$O(N \log_n)$	$O(n^2)$	$O(\log_n)$

3 Algorithms For The Main Data Structures

3.1 Stacks

For the examples shown below, the stack is implemented using a fixed size array.

3.1.1 isEmpty

This checks to see if the stack is empty.

```
function isEmpty
    if top == -1 then
        return True
    else
        return False
    endif
endfunction
```

3.1.2 isFull

This checks to see if the stack is full.

```
function isFull
    if top == maxSize then
        return True
    else
        return False
    endif
endfunction
```

3.1.3 push

This pushes (*adds*) a new item to the top of the stack.

```
procedure push(item)
    if isFull then
        print("Stack is full")
    else
        top = top + 1
        s(top)= item
    endif
endprocedure
```

3.1.4 pop

This removes the top item from the stack.

```
function pop
    if isEmpty then
        print("Stack is empty")
    else
        item = s(top)
        top = top - 1
    return item
    endif
endfunction
```

3.1.5 peek

This looks at the top item in the stack, without removing it.

```
function peek
    if isEmpty then
        print("Stack is empty")
    else
        return s(top)
    end if
end function
```

3.2 Queues

The queue in the follow examples is a circular queue implemented using an array.

3.2.1 initialise

This initialises the queue.

```
procedure initialise
    front = 0
    rear = -1
    size = 0
    maxSize = size of array
endprocedure
```

3.2.2 isEmpty

This checks to see if the queue is empty

```
function isEmpty
    if size == 0 then
        return True
```

```
    else
        return False
    endif
endfunction
```

3.2.3 isFull

This checks to see if the queue is full.

```
function isFull
    if size == maxSize then
        return True
    else
        return False
    endif
endfunction
```

3.2.4 enqueue

This enqueues (*adds*) a new item to the queue.

```
procedure enqueue(newItem)
    if isFull then
        print("Queue full")
    else
        rear = (rear + 1) MOD maxSize
        q[rear] = newItem
        size = size + 1
    endif
endprocedure
```

3.2.5 dequeue

```
function dequeue
    if isEmpty then
        print("Queue empty")
        item = Null
    else
        item = q[front]
        front = (front + 1) MOD maxSize
        size = size - 1
    endif
    return item
endfunction
```

3.3 Trees

Trees can be stored either using an array of records or using an object-oriented approach (each node is an object).

3.3.1 In-order Traversal

This is a recursive function which first traverses the left sub-tree then the root node then the right sub-tree.

```
procedure inorderTraverse(p)
    if tree[p].left != -1 then
        inorderTraverse(tree[p].left)
    endif
    print(tree[p].data)
    if tree[p].right != -1 then
        inorderTraverse(tree[p].right)
    endif
endprocedure
```

3.3.2 Post-order Traversal

This is a recursive function which first traverses the left sub-tree then the right sub-tree then the root node.

```
procedure postorderTraverse(p)
    if tree[p].left != -1 then
        postorderTraverse(tree[p].left)
    endif
    if tree[p].right != -1 then
        postorderTraverse(tree[p].right)
    endif
    print(tree[p].data)
endprocedure
```

3.3.3 Pre-order Traversal

This is a recursive function that first traverses the root node, then the left sub-tree then the right sub-tree.

```
procedure preorderTraverse(p)
    print(tree[p].value)
    if tree[p].left != -1 then
        preorderTraverse(tree[p].left)
    endif
    if tree[p].right != -1 then
        preorderTraverse(tree[p].right)
    endif
endprocedure
```

3.4 Linked Lists

The linked list in the following example is implemented using an array of record data structures, it could also be implemented in an object oriented approach.

3.4.1 Implementation of a Linked List

```
NodeRecord = RECORD
    data: String
    next: Pointer
ENDRECORD

head = Null  // Empty list
```

3.4.2 Traversing

This loops through all the elements in the linked list and prints then one by one.

```
function traverse(head)
    current = head
    WHILE current != Null  // Continue to end of list
        PRINT(current.data)
         current = current.next  // Get pointer to the next record
    ENDWHILE
endfunction
```

3.4.3 Adding a new node

This adds a new node to the front of the linked list.

```
PROCEDURE insert_at_front(head, data)

    // Create the new node
    new_node = NodeRecord
    new_node.data = data
    new_node.next = Null

    // Insert at front of list
    IF head == Null THEN
        head = new_node
    ELSE
        new_node.next = head
            head = new_node
    ENDIF
ENDPROCEDURE
```

This adds a new node to the middle of the linked list to maintain the fact that the linked list is ordered.

```
PROCEDURE insert_in_order(head, data)
    // Create the new node
    new_node = NodeRecord()
    new_node.data = data
    new_node.next = None

    // Insert into position
    current = head

    IF current == None THEN  // No nodes in list
        head = new_node
    ELSEIF current.data >= new_node.data THEN  // New node is head of list
        // Change pointers
        new_node.next = my_list.head
        head = new_node
    ELSE
        // Find the point of insertion
        WHILE (current.next != Null and current.next.data < new_node.data)
            current = current.next
        ENDWHILE
```

```
        // Change pointers
        new_node.next = current.next
        current.next = new_node
    ENDIF
ENDPROCEDURE
```

3.4.4 Deleting A Node

This deletes a node from the middle of the linked list and adjusts the pointers accordingly.

```
PROCEDURE delete(head, data):
    current = head
    IF current.data == data THEN  // Deleting item at head of list
        head = current.next
    ELSE
        WHILE current.next.data != data
            current = current.next
        ENDWHILE
        current.next = current.next.next  // Swap pointers
    ENDIF
ENDPROCEDURE
```

4 Standard Algorithms

4.1 Bubble Sort

```
PROCEDURE bubble_sort(items)

    // Initialise variables
    num_items = LEN(items)
    temp = 0
    pass_number = 1
    swapped = True

    // Continue while swaps have been made and there are more passes to evaluate
    WHILE swapped == True AND pass_number <= num_items - 1
        swapped = False
            FOR index = 0 TO num_items - pass_number
                   // Check if items are out of order
            IF items[index] > items[index + 1] THEN
                    // Swap items
                    temp = items[index]
                    items[index] = items[index + 1]
                    items[index + 1] = temp
                swapped = True
            ENDIF
            pass = pass + 1
        NEXT index
    ENDWHILE
ENDPROCEDURE
```

4.2 Insertion Sort

```
PROCEDURE insertion_sort(items)
```

```
    // Initialise the variables
    num_items = LEN(items)

    // Repeat for each item in the list, starting at second item
    FOR index = 1 TO num_items - 1
        item_to_insert = items[index]  // Get the value of the next item to insert
        previous = index - 1  // Get position of previous item

        // Repeat while there are previous items to check and the value of the
        ↪  previous item is higher that the value to insert
        WHILE previous >= 0 and items[previous] > item_to_insert
            items[previous + 1] = items[previous]  // Move previous item up one
            ↪  place
            previous = previous - 1  // Get position of the next previous item
        ENDWHILE

        // Copy value of the item to insert into the correct position
        items[previous + 1] = item_to_insert
     NEXT index
ENDPROCEDURE
```

4.3 Merge Sort

```
PROCEDURE merge (merged, list_1, list_2)

    index_1 = 0  // list_1 current position
    index_2 = 0  // list_2 current position
    index_merged = 0  // Merged current position

    // While there are still items to merge
    WHILE index_1 < LEN(list_1) AND index_2 < LEN(list_2)

        // Find the lowest of the two items being compared
        // and add it to the new list
        IF list_1[index_1] < list_2[index_2] THEN
            merged[index_merged] = list_1[index_1]
            index_1 = index_1 + 1
        ELSE
            merged[index_merged] = list_2[index_2]
            index_2 = index_2 + 1
        ENDIF
        index_merged = index_merged + 1
    ENDWHILE

    // Add to the merged list any remaining data from list_1
    WHILE index_1 < len(list_1)
        merged[index_merged] = list_1[index_1]
        index_1 = index_1 + 1
        index_merged = index_merged + 1
    ENDWHILE

    // Add to the merged list any remaining data from list_2
    WHILE index_2 < len(list_2)
        merged[index_merged] = list_2[index_2]
```

```
        index_2 = index_2 + 1
        index_merged = index_merged + 1
    ENDWHILE
ENDPROCEDURE
```

4.4 Quick Sort

```
FUNCTION quick_sort(items, start, end)
    IF start >= end THEN  // Base case
        RETURN
    ENDIF

    pivot_value = items[start]  // Set pivot value to first item in partition
    low_mark = start + 1  // Set to second value in partition
    high_mark = end  // Set to last value on partition
    finished = False

    // Repeat until low and high values have been swapped as needed
    WHILE finished == False
        WHILE low_mark <= high_mark AND items[low_mark] <= pivot_value
            low_mark = low_mark + 1  // Increment lowmark
        ENDWHILE
        WHILE low_mark <= high_mark and items[high_mark] >= pivot_value
            high_mark = high_mark - 1  // Decrement highmark
        ENDWHILE
        IF low_mark < high_mark THEN  // Swap values at lowmark and highmark
            temp = items[low_mark]
            items[low_mark] = items[high_mark]
            items[high_mark] = temp
        ELSE
            finished = True
        ENDIF
    ENDWHILE

    // Swap pivot value and value at highmark
    temp = items[start]
    items[start] = items[high_mark]
    items[high_mark] = temp

   // Recursive call on left partition
   quick_sort(items, start, high_mark - 1)

   // Recursive call on right partition
   quick_sort(items, high_mark + 1, end)

ENDFUNCTION
```

4.5 Dijkstra's algorithm

```
COST = 0  // Used to index cost
PREVIOUS = 1  // Used to index previous node

FUNCTION dijkstras_shortest_path(graph, start_node)
```

```
    // Initialise visited and unvisited lists
    unvisited = {}  // Declare unvisited list as empty dictionary
    visited = {}  // Declare visited list as empty dictionary

    // Add every node to the unvisited list
    FOREACH key IN graph
        // Set distance to infinity and previous to Null value
        unvisited[key] = [infinity, NULL]
    NEXT key
    // Set the cost of the start node to 0
    unvisited[start_node][COST] = 0

    // Repeat the following steps until unvisited list is empty
    finished = False
    WHILE finished == False
        IF LEN(unvisited) == 0 THEN
            finished = True  // No nodes left to evaluate
        ELSE
            // Get unvisited node with lowest cost as current node
            current_node = get_minimum(unvisited)
            // Examine neighbours
            FOREACH neighbour IN graph[current_node]
                // Only check unvisited neighbours
                IF neighbour NOT IN visited THEN
                    // Calculate new cost
                    cost = unvisited[current_node][COST] +
                    ↪  graph[current_node][neighbour]
                    // Check if new cost is less
                    IF cost < unvisited[neighbour][COST] THEN
                        unvisited[neighbour][COST] = cost  // Update cost
                        unvisited[neighbour][PREVIOUS] = current_node  // Update
                        ↪  previous
                    ENDIF
                ENDIF
            NEXT neighbour
            // Add current node to visited list
            visited[current_node] = unvisited[current_node]
            // Remove from unvisited list
            DEL(unvisited[current_node])
        ENDIF
    ENDWHILE
    RETURN visited
ENDFUNCTION
```

4.6 A* Algorithm

```
G_SCORE = 0  // Used to index g-score
F_SCORE = 1  // Used to index f-score
PREVIOUS = 2  // Used to index previous

FUNCTION a_star(graph, start_node, target_node)

    visited = {}  // Declare visited list as empty dictionary
    unvisited = {}  // Declare unvisited list as empty dictionary
```

```
    // Add every node to the unvisited list
    FOREACH key IN graph
        unvisited[key] = [infinity , infinity , Null]
    NEXT key

    // Update values for start node in unvisited list
    h_score = heuristic(start_node)
    unvisited[start_node] = [0, h_score, NULL]

    // Repeat until there are no nodes in the unvisited list
    finished = False
    WHILE finished == False
        IF LEN(unvisited) == 0 THEN  // No nodes left to evaluate
            finished = True
        ELSE
            // Get node with lowest f-score from open list
            current_node = get_minimum(unvisited)
            IF current_node == target_node THEN
                finished = True
                // Copy data to visited list
                visited[current_node] = unvisited[current_node]
            ELSE
                // Examine neighbours
                FOREACH neighbour IN graph[current_node]
                    // Only check unvisited neighbours
                    IF neighbour NOT IN visited THEN
                        // Calculate new g-score
                        new_g_score = unvisited[current_node][G_SCORE] +
                        ↪  graph[current_node][neighbour]
                        // Check if new g-score is
                        ↪  less
                        IF new_g_score < unvisited[neighbour][G_SCORE] THEN
                            unvisited[neighbour][G_SCORE] = new_g_score
                            unvisited[neighbour][F_SCORE] = new_g_score +
                            ↪  heuristic(neighbour)
                            unvisited[neighbour][PREVIOUS] = current_node
                        ENDIF
                    ENDIF
                NEXT neighbour

                // Add current node to visited list
                visited[current_node] = unvisited[current_node]
                // Remove from unvisited list
                DEL(unvisited[current_node])
            ENDIF
        ENDIF
    ENDWHILE

    // Return final visited list
    RETURN visited
ENDFUNCTION
```

4.7 Binary Search

```
FUNCTION binary_search(items, search_item)

    index = -1  // Initialise variables
    found = False
    first = 0
    last = LEN(items) - 1

    // Repeat while there are more items to check and the item has not yet been
    ↪ found
    WHILE first <= last AND found = False
        midpoint = (first + last) DIV 2
        IF items[midpoint] == search_item THEN
            index = midpoint
            found = True  // Set flag so loop terminates
        ELSEIF items[midpoint] < search_item THEN
            first = midpoint + 1  // Focus on the right half of the list
        ELSE
            last = midpoint - 1  // Focus on the left half of the list
        ENDIF
    ENDWHILE
    RETURN index

ENDFUNCTION
```

4.8 Linear Search

```
FUNCTION linear_search(items, search_item)

    // Initialise variables
    index = -1
    current = 0
    found = False

    // Repeat while there are still items to examine and the item has not yet been
    ↪ found
    WHILE current < LEN(items) AND found == False
        IF items[current] == search_item THEN
                    index = current
                    found = True  // Set flag to terminate loop
        ENDIF
        current = current + 1
    ENDWHILE

    // Return the position of the search_item or -1 if not found
    RETURN index

ENDFUNCTION
```