Component 02

Algorithms and programming

Sorting Algorithms and Searching Algorithms

BUBBLE SORT

Bubble sort is a brute force and iterative sorting algorithm where each adjacent item in the array is compared. If the item on the right is less than the item on the left they are swapped. The last element of the array will be in the correct place.

Process

1) Set swapMade variable to True.
2) While swapMade is True, make passes through the list:
   - for all items list.LENGTH-2 (all but the last item):
     - compare the first and the second item in the list:
       - if the first item is greater than the second item, then they are not in order – swap them and set swapMade variable to True; or
       - if the first item is less than the second item, then they are in order; repeat the comparison process until the end of the list is reached;
     - when the end of the list is reached, begin another pass.
3) On the final pass, all of the items should be sorted and therefore swapMade will remain False and the algorithm will finish.

Pseudocode

```
FUNCTION bubbleSort(list)
   swapMade = True
   WHILE swapMade == True
      swapMade = False
      FOR i=0 TO list.LENGTH – 2  // iterate through all but the last item
         IF list[i] > list[i+1] THEN
            temp = list[i+1]
            list[i+1] = list[i]
            list[i] = temp
            swapMade = True
         ENDIF
      NEXT i
   ENDWHILE
   RETURN list
ENDFUNCTION
```

The algorithm can be improved by decrementing the number of items to be inspected on each pass, as these missed items are assumed to be sorted.

Process

1) Set swapMade variable to True.
2) Set passes variable to list.LENGTH-2 (all but the last item)
3) While swapMade is True, make passes through the list:
   - for all items, but the last item, in the list:
     - compare the first and the second item in the list:
       1. if the first item is greater than the second item, then they are not in order – swap them and set swapMade variable to True; or
       2. if the first item is less than the second item, then they are in order; repeat the comparison process until the end of the list is reached;
     - when the end of the list is reached, begin another pass;
   - decrement the passes variable.
4) On the final pass, all of the items should be sorted and therefore swapMade will remain False and the algorithm will finish.

BUBBLE SORT

Pseudocode

```
FUNCTION bubbleSort(list)
   swapMade = True
   passes = list.LENGTH-2    // iterate through all but the last item
   WHILE swapMade == True
      swapMade = False
      FOR i=0 TO passes
         IF list[i] > list[i+1] THEN
            temp = list[i+1]
            list[i+1] = list[i]
            list[i] = temp
            swapMade = True
         ENDIF
      NEXT i
      passes = passes - 1
   ENDWHILE
   RETURN list
ENDFUNCTION
```

This improvement means that unnecessary passes are not made as the items at the end of the list do not need to be compared since they are sorted.

INSERTION SORT

Insertion sort is an iterative sorting algorithm that works by dividing a list into two parts: a sorted portion; and an unsorted portion. The elements in the list are inserted, one at a time, into their correct position in the sorted portion.

Process

1) Make the first item in the list the sorted portion of the list and the remaining items are the unsorted portion of the list.
2) While there are items in the unsorted list:
    - take the first item in the unsorted list;
    - while there is an item to the left of the first item in the unsorted list:
        - swap with that item;
    - the sorted list is now one item bigger, as the first item of the unsorted list has become a member of the sorted list.

Pseudocode

```
FUNCTION insertionSort(list)
   FOR i=0 TO list.LENGTH-1    // iterate through all but the last item
      currentItem = list[i]    // take the first item in the unsorted list
      position = i    // set the position to the current index value
      // while there are items in the unsorted list
      WHILE position > 0 AND list[position - 1] > currentItem
         // swap the items
         list[position] = list[position - 1]
         position = position - 1
      ENDWHILE
      list[position] = currentItem
   NEXT i
   RETURN list
ENDFUNCTION
```

MERGE SORT

Merge sort is a divide and conquer sorting algorithm where the list recursively partitioned in to halves, until each sublist is of length one, and therefore sorted by definition as the single item is the smallest and the largest in that sublist. The sublists are then sorted and merged into larger sublists until they are recombined into a single sorted list. The implementation of a merge sort is usually recursive as the way it solves the problem is inherently recursive so it naturally lends itself to this style of implementation.

Process

1) The list is split into two halves.
2) For the left half:
    - split the left half into halves until each sublist is of length one.
    - merge the pair of sublists on the left half by repeating this process until all items are in the merged list:
        - comparing the first item in the left half with the first item in the right half;
        - if the item in the left half is less than the item in the right half, add the item from the left half to the merged list and read the next item from the left half;
        - if the item in the right half is less than the item in the left half, add the item from the right half to the merged list and read the next item from the right half;
        - once either list is empty, any remaining items are added to the merged list.
3) For the right half:
    - split the right half into halves until each sublist is of length one.
    - merge the pair of sublists on the right half by repeating this process until all items are in the merged list:
        - comparing the first item in the left half with the first item in the right half;
        - if the item in the left half is less than the item in the right half, add the item from the left half to the merged list and read the next item from the left half;
        - if the item in the right half is less than the item in the left half, add the item from the right half to the merged list and read the next item from the right half;
        - once either list is empty, any remaining items are added to the merged list.
4) Merge the left half and the right half by repeating this process until all items are in the merged list:
    - comparing the first item in the left half with the first item in the right half;
    - if the item in the left half is less than the item in the right half, add the item from the left half to the merged list and read the next item from the left half;
    - if the item in the right half is less than the item in the left half, add the item from the right half to the merged list and read the next item from the right half;
    - once either list is empty, any remaining items are added to the merged list.

MERGE SORT

Pseudocode

```
PROCEDURE mergeSort(list)
   // base case
   IF list.LENGTH > 1 THEN   // if list is not, by definition, sorted
      mid = list.LENGTH DIV 2   // performs integer division to find
                                   the midpoint of the list
      leftHalf = mergeList[:mid]    // left half of list
      rightHalf = mergeList[mid:]    // right half of list

      // recursive case
      mergeSort(leftHalf)    // recursive call for leftHalf
      mergeSort(rightHalf)    // recursive call for rightHalf

      i = 0    // pointer to item in leftHalf (starting at the first item)
      j = 0    // pointer to item in rightHalf (starting at the first item)
      k = 0    // pointer to item in list (starting at the first item)

      // while the first item in the leftHalf is less than the length of the
         length of the leftHalf AND the first item in the rightHalf is less
         than the length of the rightHalf

         i.e. while there are still item in the leftHalf and rightHalf of the
              sublists
      WHILE i < leftHalf.LENGTH AND j < rightHalf.LENGTH
         // if the item at the pointer in the leftHalf is less than the item
            at the pointer in the rightHalf
         IF leftHalf[i]<rightHalf[j] THEN
            // the item at the pointer in leftHalf the becomes is added to
               the list
            list[k] = leftHalf[i]
            // increment the pointer pointing to the item in the leftHalf
            i = i + 1
         ELSE
            // the item at the pointer in rightHalf the becomes is added to
               the list
            list[j] = rightHalf[j]
         ENDIF
         // increment the pointer pointing to the item in the list
         k = k + 1
      ENDWHILE

      WHILE i < leftHalf.LENGTH
         mergeList[k] = leftHalf[i]
         // increment the pointer pointing to the item in the leftHalf
         i = i + 1
         // increment the pointer pointing to the item in the list
         k = k + 1
      ENDWHILE

      WHILE j < rightHalf.LENGTH
         mergeList[k] = rightHalf[j]
         // increment the pointer pointing to the item in the rightHalf
         j = j + 1
         // increment the pointer pointing to the item in the list
         k = k + 1
      ENDWHILE
   ENDIF
ENDPROCEDURE
```

QUICK SORT

Quick sort is a divide and conquer sorting algorithm where a pivot value is used, such as the first item in the list. The remainder of the list is divided into two partitions where: all elements less than the pivot value must be in the first partition; and all elements greater than the pivot value must be in the second partition. The implementation of a quick sort is usually recursive as the way it solves the problem is inherently recursive so it naturally lends itself to this style of implementation.

Process
1) Select a pivot value, sometimes the first item in the list. 2) Locate the two position markers: • set leftMark variable to the index of the second item in the list (after the pivot); and • set rightMark variable to the index of the last item in the list. 3) Move the items which are less than or equal to the pivot to the left-hand side of the pivot and move the items which are greater than the pivot to the right-hand side of the pivot: • compare the pivot to the item at the leftMark and if the item at the leftMark is less than pivot, increment the leftMark (move to the right) – repeat this until the pivot is greater than the item at the leftMark. • compare the pivot to the item at the rightMark and if the item at the rightMark is greater than the pivot, decrement the rightMark (move to the left) – repeat this until the pivot is less than the item at the rightMark. 4) Exchange the item at leftMark with the item at rightMark and repeat stage 3). 5) When rightMark < leftMark, the split point is at the position of the rightMark. 6) Exchange the item at the pivot with the item at the split point. 7) Divide the list at the split point and recursively quick sort each half.

QUICK SORT

Pseudocode

```
FUNCTION partition(list, start, end)
   pivot = list[start]    // set the pivot to point to the first item
   leftMark = start + 1    // set the leftMark to point to the item after the pivot
   rightMark = end    // set the rightMark to point to the last item
   done = False    // the split point has not been found
   WHILE done == False    // while the split point has not been found
      // while the leftMark is less than or equal to the rightMark and the leftMark
         is less than or equal to the pivot
      WHILE leftMark <= rightMark AND list[leftMark] <= pivot
      // increment the leftMark
         leftMark = leftMark + 1
      ENDWHILE
      // while the rightMark is greater than or equal to the leftMark and the
         rightMark is greater than or equal to the pivot
      WHILE rightMark >= leftMark AND list[rightMark] >= pivot
      // decrement the rightMark
         rightMark = rightMark - 1
      ENDWHILE
      // if the pointer have swapped over
      IF rightMark < leftMark THEN
         // the split point has been found
         done = True
      ELSE
         // the split point has not been found so swap the items at the leftMark and
            the rightMark
         temp = list[leftMark]
         list[leftMark] = list[rightMark]
         list[rightMark] = temp
      ENDIF
   ENDWHILE
   // swap the item at the pivot with the item at the rightMark
   temp = list[start]
   list[start] = list[rightMark]
   list[rightMark] = temp
   // return the split point (rightMark)
   RETURN rightMark
ENDFUNCTION

FUNCTION quicksort(list, start, end)
   // base case
   IF start < end THEN
      // partition the list
      split = partition(list, start, end)
      // recursive case - quick sort the right half
      quickSort(list, start, split-1)
      // recursive case - quick sort the left half
      quickSort(list, split+1, end)
   ENDIF
   RETURN list
ENDFUNCTION

list = [9, 5, 4, 15, 3, 8, 11]
sortedList = quicksort(list, 0, list.LENGTH-1)
PRINT(sortedList)
```

COMPARISON

		Bubble Sort	Insertion Sort	Merge Sort	Quick Sort
Time Complexity	Best Case	Linear $O(n)$	Linear $O(n)$	Logarithmic $O(n \log n)$	Logarithmic $O(n \log n)$
	Average Case	Polynomial $O(n^2)$	Polynomial $O(n^2)$	Logarithmic $O(n \log n)$	Logarithmic $O(n \log n)$
	Worst Case	Polynomial $O(n^2)$	Polynomial $O(n^2)$	Polynomial $O(n \log n)$	Polynomial $O(n^2)$
Space Complexity (Auxiliary Worst Case)		Constant $O(1)$	Constant $O(1)$	Linear $O(n)$	Logarithmic $O(\log n)$
Avoiding the worst case time		Sort the list from both directions. Decrement the number of items to be inspected on each pass, as these missed items are assumed to be sorted.	-	Cannot be optimised because it takes $O(\log n)$ to break the array down into the sub lists and then $O(n)$ swaps are made.	Choosing an appropriate pivot value; a common method is to use the median of the leftmark rightmark and middle value of the array as the pivot value. This is recalculated on each recursive call.
Coding difficultly (1 – easiest, 4 – hardest)		1	2	3	4
Evaluation		Slowest but easiest to code.	Polynomial time complexity but reduced to linear if list is almost sorted. Worst case if data is in descending order.	Scales well since logarithmic but requires additional memory space for the merging process. Recursion could lead to a stack overflow if the list is very large.	Generally fastest but dependent on using a pivot which is not close to the smallest or largest elements of the list. Rarely worst case especially if pivot value has been chosen carefully. Does not required additional memory space, operations are completed "in place".
		Number of comparisons is the same. Less swaps made in insertion, thus less writing.			In a partially or fully sorted list, bubble or insertion may actually be better than merge sort and they are simpler to code.

LINEAR SEARCH

Linear search is a brute force and iterative searching algorithm which sequentially checks each element of the list to see if it matches the search criteria until a match is found or until all the elements have been searched.

Process

1) Set found variable to False.
2) Set index variable to 0.
3) While the item has not been found and the index is within range of the list:
    - if the item at the current value of index is equal to the search criteria, set the found variable to True and return the item at the current value of index;
    - else, increment index.
4) If the item is not found, the index will become greater than the length of the list and the while loop will finish, this means that the item has not been found.

Pseudocode

```
FUNCTION linearSearch(list, searchCriteria)
   found = False
   index = 0
   WHILE found = False AND index < list.LENGTH
      IF list[index] == searchCriteria THEN
         found = True
         RETURN list[index]
      ELSE
         index = index + 1
      ENDIF
   ENDWHILE
   RETURN "Item not found"
ENDFUNCTION
```

BINARY SEARCH

Binary search is a divide and conquer iterative searching algorithm which works by repeatedly dividing in half the portion of a list which contains the required data item until there is only one item in the list. This can also be implemented recursively.

Process
1) Set found variable to False. 2) Set lowerBound variable to 0. 3) Set upperBound variable to list.LENGTH-1 (index of the last item). 4) While the item has not been found and the lowerBound is less than or equal to the upperBound: • calculate a midpoint by doing floor division of the sum of lowerBound and upperBound; • if the item at the midpoint is equal to the search criteria, set found to True; • if the item at the midpoint is less than the search criteria, set the lowerBound to midpoint + 1; • if the item at the midpoint is greater than the search criteria, set the upperBound to midpoint – 1. 5) Return the found variable.

Pseudocode

```
FUNCTION binarySearch(list, searchCriteria)
   found = False
   lowerBound = 0
   upperBound = list.LENGTH-1
   WHILE found == False AND lowerBound <= upperBound
      midPoint = (lowerBound + upperBound) DIV 2
      IF list[midPoint] == searchCriteria THEN
         found = True
      ELIF list[midPoint] < searchCriteria THEN
         lowerBound = midPoint + 1
      ELSE
         upperBound = midPoint - 1
   ENDWHILE
   RETURN found
ENDFUNCTION
```

COMPARISON

		Linear Search	Binary Search
Time Complexity	Best Case	Constant O(1)	Constant O(1)
	Average Case	Linear O(n)	Logarithmic O(log n)
	Worst Case	Linear O(n)	Logarithmic O(log n)
Space Complexity (Auxiliary Worst Case)		Constant O(1)	Constant O(1)
Avoiding the worst case time		Cannot be optimised.	Can be implemented recursively. This could be considered the optimum implementation as it is naturally recursive.
Coding difficultly (1 – easiest, 4 – hardest)		1	2
Evaluation		Works on any data set. If the item is the last in the list, then it will take O(n). More suited to smaller data sets.	The data set must be in order. Very efficient even if list is large. If data isn't sorted, consider the complexity of a sorting algorithm.