PROGRAMMING AND DATA STRUCTURES

SORTING ALGORITHMS

HOURIA OUDGHIRI

FALL 2021

OUTLINE

- Sorting problem
- Types of sorting solutions
- Sorting algorithms
- Comparison of sorting algorithms

STUDENT LEARNING OUTCOMES

At the end of this chapter, you should be able to:

- List the types and categories of sorting
- Implement different sorting algorithms
- Evaluate the complexity of the sorting algorithms
- Compare the sorting algorithms

Sorting problem

- Given a list of data items, arrange the list in an ascending or descending order
- Commonly used in computer science to organize objects based on one specific criterion
- Allows using the efficient binary search algorithm
- Sorting is more complex than searching

Sorting types

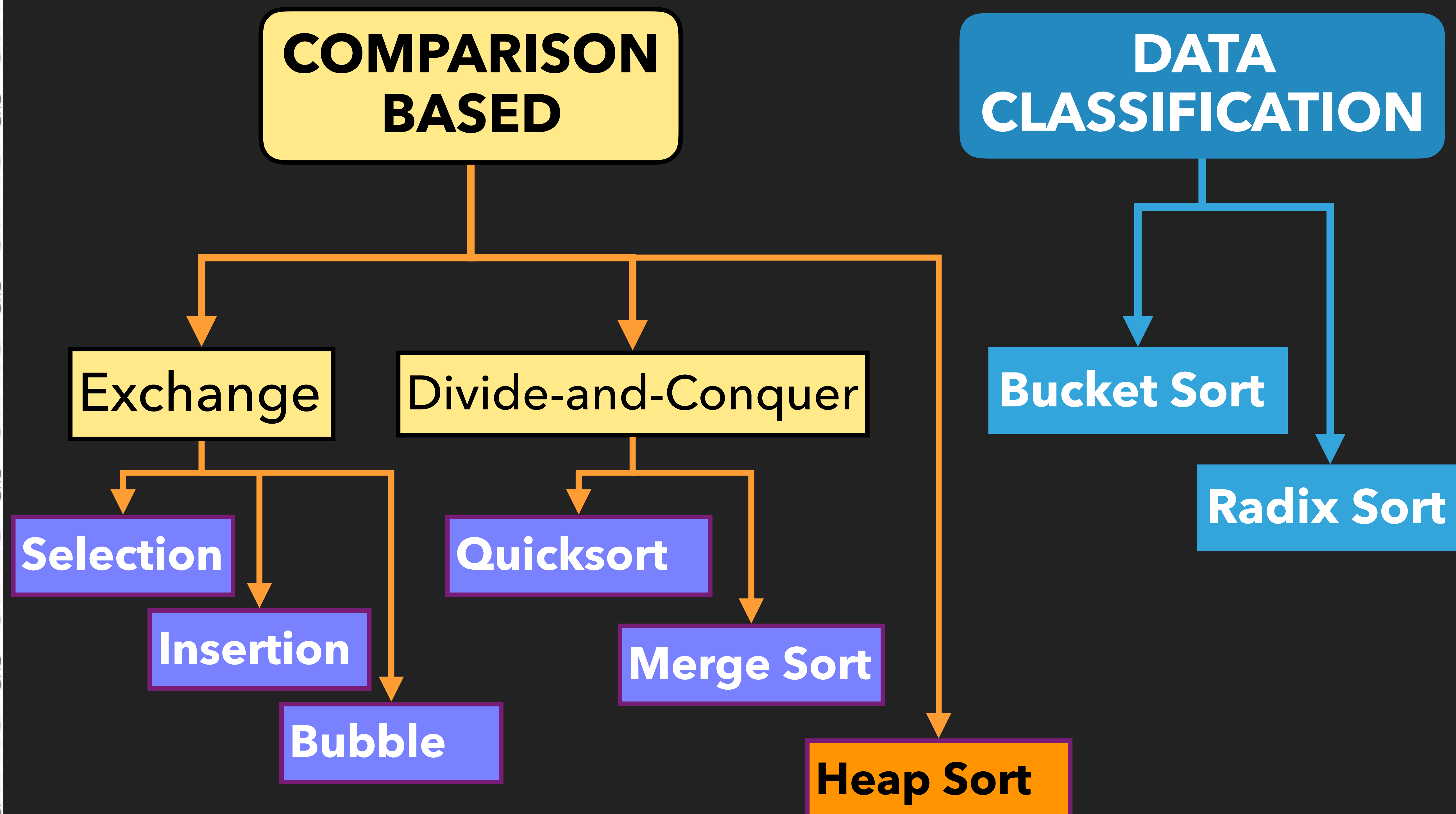
INTERNAL

Data is in memory
(RAM)

EXTERNAL

Data is in secondary memory
(Hard disk: Large Files)

Sorting categories



Selection Sort

67	33	21	84	49	50	75	min = 21
21	33	67	84	49	50	75	min = 33
21	33	67	84	49	50	75	min = 49
21	33	49	84	67	50	75	min = 50
21	33	49	50	67	84	75	min = 67
21	33	49	50	67	84	75	min = 75
21	33	49	50	67	75	84	

Selection Sort Algorithm

```
Algorithm selectionSort
 for every element i (N size of the list)
  find the smallest element from i to N-1
  swap element i with the smallest element
End
```

Selection Sort Algorithm

```
public static void selectionSort(int[] list) {
   int minIndex;
   for (int i=0; i<list.length-1; i++) {
      int min = list[i];
      minIndex = i;
      for (int j=i; j<list.length; j++){
         if (list[j] < min){
             min = list[j];
             minIndex = j;
         }
      }
      int temp = list[i];
      list[i] = list[minIndex];
      list[minIndex] = temp;
   }
}
```

selectionSort.java

Selection Sort Algorithm

- Analyzing Selection Sort Complexity

```
Iteration1(outer loop)
    (n)iterations(inner loop)

Iteration 2 (outer loop)
    (n-1) iterations (inner loop)

Iteration k (outer loop)
    (n-k+1) iterations (inner loop)

Iteration n-1 (outer loop)
    1 iteration (inner loop)

1 + 2 + … + (n) = n(n+1)/2
```

Selection Sort: $O(n^2)$ – Quadratic growth

Selection Sort Algorithm

```
public static <E extends Comparable<E>>
                    void selectionSort(E[] list) {
    int minIndex;
    for (int i=0; i<list.length-1; i++) {
      E min = list[i];
      minIndex = i;
      for (int j=i; j<list.length; j++){
        if (list[j].compareTo(min) < 0){
            min = list[j];
            minIndex = j;
        }
      }
      E temp = list[i];
      list[i] = list[minIndex];
      list[minIndex] = temp;
    }
}
```

selectionSort.java

Insertion Sort

- At each iteration, insert one element in the sorted partial list

Insertion Sort

67	33	21	84	49	50	75	Insert 33 at 0
33	67	21	84	49	50	75	Insert 21 at 0
21	33	67	84	49	50	75	Insert 84 at 3
21	33	67	84	49	50	75	Insert 49 at 2
21	33	49	67	84	50	75	Insert 50 at 3
21	33	49	50	67	84	75	Insert 75 at 5
21	33	49	50	67	75	84	

Insertion Sort Algorithm

```
Algorithm insertionSort

   for every element i

     insert element i in the sorted list
      (0 to i)

   end for

End
```

Insertion Sort Algorithm

insertionSort.java

```
public static void insertionSort(int[] list) {
 for (int i=1; i<list.length; i++) {
  //Insert element i in the sorted sub-list
  int currentVal = list[i];
  int j = i;
  while (j > 0 && currentVal < (list[j - 1]))
  {
   // Shift element (j-1) into element (j)
   list[j] = list[j - 1];
   j--;
  }
  // Insert currentVal at position j
  list[j] = currentVal;
 }
}
```

Insertion Sort Algorithm

- Analyzing Insertion Sort Complexity

```
Iteration1(outer loop)
    (1)iteration(inner loop)

Iteration 2 (outer loop)
    (2) iterations (inner loop)

Iteration k (outer loop)
    (k) iterations (inner loop)

Iteration n-1 (outer loop)
    n-1 iterations (inner loop)

1 + 2 + … + (n-1) = n(n-1)/2
```

Insertion Sort: $O(n^2)$ – Quadratic growth

Bubble Sort

- At each iteration, exchange out of order pairs of elements until all elements are sorted
- Pushing the largest element to the end of the list

Bubble Sort

67 33 21 84 49 50 75

Bubble Sort

67 33 21 84 49 50 75 Out of order - swap

33 67 21 84 49 50 75

Bubble Sort

```
67 33 21 84 49 50 75

33 67 21 84 49 50 75

33 67 21 84 49 50 75
```

Out of order - swap

```
33 21 67 84 49 50 75
```

Bubble Sort

```
67 33 21 84 49 50 75
33 67 21 84 49 50 75
33 67 21 84 49 50 75
33 21 67 84 49 50 75
33 21 67 84 49 50 75
33 21 67 84 49 50 75
```

In order - No swap

Bubble Sort

```
67 33 21 84 49 50 75
33 67 21 84 49 50 75
33 67 21 84 49 50 75
33 21 67 84 49 50 75
33 21 67 84 49 50 75
33 21 67 84 49 50 75
33 21 67 49 84 50 75
```

Out of order - swap

Bubble Sort

```
67 33 21 84 49 50 75
33 67 21 84 49 50 75
33 67 21 84 49 50 75
33 21 67 84 49 50 75
33 21 67 84 49 50 75
33 21 67 84 49 50 75
33 21 67 49 84 50 75
33 21 67 49 84 50 75
33 21 67 49 50 84 75
```

Out of order - swap

Bubble Sort

```
67 33 21 84 49 50 75
33 67 21 84 49 50 75
33 67 21 84 49 50 75
33 21 67 84 49 50 75
33 21 67 84 49 50 75
33 21 67 84 49 50 75
33 21 67 49 84 50 75
33 21 67 49 84 50 75
33 21 67 49 50 84 75
33 21 67 49 50 84 75
33 21 67 49 50 75 84
```

Pass 1 completed

Out of order - swap

Bubble Sort

Pass 2

33 21 67 49 50 75 84 Out of order - swap

21 33 67 49 50 75 84

Bubble Sort

Pass 2

33 21 67 49 50 75 84

21 33 67 49 50 75 84

21 **33 67** 49 50 75 84 In order - No swap

Bubble Sort

Pass 2

```
33 21 67 49 50 75 84
21 33 67 49 50 75 84
21 33 67 49 50 75 84
21 33 67 49 50 75 84   Out of order - swap
21 33 49 67 50 75 84
```

Bubble Sort

Pass 2

```
33 21 67 49 50 75 84
21 33 67 49 50 75 84
21 33 67 49 50 75 84
21 33 67 49 50 75 84
21 33 49 67 50 75 84
21 33 49 67 50 75 84
21 33 49 50 67 75 84
```

Out of order - swap

Bubble Sort

Pass 2

```
33 21 67 49 50 75 84
21 33 67 49 50 75 84
21 33 67 49 50 75 84
21 33 67 49 50 75 84
21 33 49 67 50 75 84
21 33 49 67 50 75 84
21 33 49 50 67 75 84
21 33 49 50 67 75 84
```

In order - No swap

Bubble Sort

Pass 2

```
33 21 67 49 50 75 84
21 33 67 49 50 75 84
21 33 67 49 50 75 84
21 33 67 49 50 75 84
21 33 49 67 50 75 84
21 33 49 67 50 75 84
21 33 49 50 67 75 84
21 33 49 50 67 75 84
```

Pass 2 completed

Bubble Sort

Pass 3

21 33 49 50 67 75 84 In order - No swap

21 33 49 50 67 75 84 In order - No swap

21 33 49 50 67 75 84 In order - No swap

21 33 49 50 67 75 84 In order - No swap

Pass 3 completed - No swaps - List sorted

Bubble Sort Algorithm

```
Algorithm BubbleSort
  sorted = false
  last = N-1 (N size of the array)
  while (not sorted)
    sorted = true
    for i=0 to last-1
       if(list[i] > list[i+1])
          swap(list[i], list[i+1])
          sorted = false
       end if
    end for
    last = last - 1;;
  end while
End
```

Bubble Sort Method

bubbleSort.java

```
public static void bubbleSort(int[] list) {
 boolean sorted = false;
 for (int k=1;
      k < list.length && !sorted; k++) {
   sorted = true;
   for (int i=0; i<list.length-k; i++) {
     if (list[i] > list[i+1]) {
        // swap
        int temp = list[i];
        list[i] = list[i+1];
        list[i+1] = temp;
        sorted = false;
     }
   }
}
```

Bubble Sort Algorithm

✦ Analyzing Bubble Sort Complexity

```
Iteration1(outer loop)
    (n-1)iteration(inner loop) to push the max

Iteration 2 (outer loop)
    (n-2) iterations (inner loop)

Iteration k (outer loop)
    (n-k) iterations (inner loop)

Iteration n-1 (outer loop)
    1 iterations (inner loop)

1 + 2 + … + (n-1) = n(n-1)/2
```

Bubble Sort: O(n²) - Quadratic growth

Comparison of Sorting Algorithms

Algorithm	Complexity	Performance Analysis
Selection Sort	$O(n^2)$	Simple Redundant Processing **Worst**: Same performance all the time
Bubble Sort	$O(n^2)$	Complex Inefficient for large sets **Worst**: Reversed list
Insertion Sort	$O(n^2)$	Lower overhead than selection and bubble sort **Worst**: Reversed list

Merge Sort

- Use divide-and-conquer strategy
- Split the list in halves recursively until obtaining lists with one element
- Merge the lists back in order

Merge Sort

67 33 21 84 49 50 75

[67 33 21] [84 49 50 75] Split

[67] [33 21] [84 49] [50 75] Split

[33] [21] [84] [49] [50] [75] Split

Merge Sort

67	33	21	84	49	50	75

67	21 33	49 84	50 75	Merge

21 33 67	49 50 75 84	Merge

21 33 49 50 67 75 84	Merge

Merge Sort Algorithm

```
Algorithm MergeSort (recursive)

   Split array in two halves
   MergeSort the first half
   MergeSort the second half
   Merge the two sorted halves

End
```

Merge Sort method

mergeSort.java

```
public static void mergeSort(int[] list) {
  if (list.length > 1) { // ==1: base case
   int[] firstHalf = new int[list.length/2];
   int[] secondHalf = new int[list.length -
                              list.length/2];
   System.arraycopy(list, 0, firstHalf, 0,
                    list.length/2);
   System.arraycopy(list,  list.length/2,
                    secondHalf, 0,
                  list.length-list.length/2);
   mergeSort(firstHalf);
   mergeSort(secondHalf);
   merge(firstHalf, secondHalf, list);
  }
}
```

Merge Sort method

mergeSort.java

```
public static void merge(int[] list1, int[] list2,
                         int[] list) {
    int list1Index = 0;
    int list2Index = 0;
    int listIndex = 0;
    while( list1Index < list1.length &&
           list2Index < list2.length) {
      if (list1[list1Index] < list2[list2Index])
        list[listIndex++] = list1[list1Index++];
      else
        list[listIndex++] = list2[list2Index++];
    }
    while(list1Index < list1.length)
      list[listIndex++] = list1[list1Index++];
    while(list2Index < list2.length)
      list[listIndex++] = list2[list2Index++];
  }
```

Merge Sort Algorithm

- Analyzing Merge Sort time Complexity

```
Splitting the array in halves
   (log n) iterations (n/2^k = 1)

Merging halves
   (n) iterations (worst case)
```

Merge Sort: O(n log n) - Log Linear

Merge Sort Algorithm

- Analyzing Merge Sort space Complexity

```
Splitting the array in halves
 n/2, n/4, n/8, …

Total number of elements
    (n/2 + n/4 + n/8 + …) ≃ n


Merge Sort space complexity: O(n)
```

Quick Sort

- Use divide-and-conquer strategy
- Divide the list in two partially sorted parts using a pivot
  - Part 1: all elements less than the pivot
  - Part 2: all elements greater than the pivot
- Repeat `quickSort` recursively on each part

Quick Sort

67 33 21 84 49 50 75 pivot = 67

50 33 21 49	67	84 75
List1		List2

Quick Sort

67 33 21 84 49 50 75   pivot = 67

$\boxed{50\ 33\ 21\ 49}$ $\boxed{67}$ $\boxed{84\ 75}$

List1 List2

$\boxed{49\ 21\ 33}$ $\boxed{50}$ $\boxed{67}$ $\boxed{75}$ $\boxed{84}$

List3 List4

pivot 1 = 50

pivot 2 = 84

Quick Sort

67 33 21 84 49 50 75

50 33 21 49		67		84 75
List1				List2

49 33 21	50	67	75	84
List3			List4	

21 33	49	50	67	75	84
List5					

pivot 3 = 49

Quick Sort

21	33	49	50	67	75	84

pivot 4 = 21

List7

21	33	49	50	67	75	84

21 33 49 50 67 75 84 Sorted list

Quick Sort

```
Algorithm QuickSort (recursive)
   Select a pivot
   Partition array in two parts
   Part1: Elements less than the pivot
   Part2: Elements greater than the pivot
   QuickSort(part1)
   QuickSort(part2)
End
```

Quick Sort

quickSort.java

```
public static void quickSort(int[] list) {
    quickSort(list, 0, list.length-1);
}

public static void quickSort(int[] list,
                      int first, int last) {
    if (last > first) {
      int pivotIndex = partition(list, first, last);
      quickSort(list, first, pivotIndex-1);
      quickSort(list, pivotIndex+1, last);
    }
}
```

Quick Sort

quickSort.java

```
public static int partition(int list[],
                            int first, int last){
   int pivot;
   int index, pivotIndex;
   pivot = list[first];// pivot is the first element
   pivotIndex = first;
   for (index = first + 1;
        index <= last; index++)
     if (list[index] < pivot){
        pivotIndex++;
        swap(list, pivotIndex, index);
     }
   swap(list, first, pivotIndex);
   return pivotIndex;
}
```

Quick Sort

- Analyzing Quick Sort Complexity

```
 Partitioning the array in ~ halves
    (log n) iterations (average)

 Arranging elements around the pivot
    (n) iterations - worst case


Quick Sort: average case
    O(n log n) - Log Linear
```

Quick Sort

- Analyzing Quick Sort Complexity

```
Partitioning the array not in halves
   (n) iterations - worst case

Arranging elements around the pivot
   (n) iterations - worst case
```

Quick Sort: worst case – $O(n^2)$–Quadratic

Quick Sort

- How to select the pivot?
- Original data is random, pivot = first element
- Original data partially sorted, use "middle of the three" rule (median of first, middle, and last)

Heap Sort

- Uses a binary heap (complete binary tree)
- Data is inserted in the heap then removed in order

Heap Sort

- Sorting using the heap data structure
  - Add the data to be sorted to the heap using **`add()`** method
  - Remove the elements from the heap using **`remove()`** method (elements are removed in descending order)

Heap Sort

heapSort.java

```
public static <E extends Comparable<E>>
 void heapSort(E[] list) {
    Heap<E> heap = new Heap<>();
    for(int i=0; i<list.length; i++){
       heap.add(list[i]);
    }
    for (int i=list.length-1; i>=0; i--) {
      list[i] = heap.remove();
    }
 }
```

Heap Sort

- Analyzing Heap Sort Time Complexity

add() method – (***O(log n)***)
    Trace path from a leaf to root

remove() method – (***O(log n)***)
    Trace path from the root to a leaf

add() and remove() are called **n** times to create the heap and to get the sorted data – (***O(n)***)

Heap Sort: Worst case
    O(n log n) – Log Linear

Heap Sort

- Analyzing Heap Sort Space Complexity

```
 Heap with n nodes required to sort the
array list


Heap Sort space complexity: O(n)
```

Merge/Quick/Heap Sort

	Merge Sort	**Quick Sort**	**Heap Sort**
Type	Divide and Conquer Recursive	Divide and Conquer Recursive	Complete Binary Tree
Main task	Merging *O(n)*	Partitioning *O(n)*	Adding/removing nodes to/from heap
Time Complexity	*Worst case: O(n logn)*	Average: *O(n logn)* Worst case: $O(n^2)$	Worst case: *O(n logn)*
Space Complexity	Temporary arrays O(n)	No additional space	Heap data structure O(n)

Summary

- Quadratic sorting algorithms
  - Selection - Insertion - Bubble
- Log Linear sorting algorithms
  - Merge sort - Quick sort - Heap sort
- Other sorting algorithms (~linear)
  - Bucket and Radix sort

Summary

- Sorting algorithms are general - work for any type of data
- Sorting criterion defined in method **`compareTo()`** or **`compare()`**
- Comparison based sorting cannot perform better than $O(n \log n)$

Summary

- Can we do sorting without comparisons?
- Can sorting be performed in less than ***O(n log n)***?
- If data is classified based of its own value with no comparisons - Bucket Sort / Radix Sort

Summary

	Quadratic	Log Linear			Linear		External Merge Sort
Sorting Algorithm	**Selection Insertion Bubble Sort**	**Merge Sort**	**Quick Sort**	**Heap Sort**	**Bucket Sort (t buckets)**	**Radix Sort (d digits)**	
Type	Exchange	Divide and Conquer		Binary Tree	Data Classification		Divide and Conquer
Time	*$O(n^2)$*	*$O(n \log n)$*	*$O(n \log n)$ to $O(n^2)$*	*$O(n \log n)$*	**$O(n+t)$**	**$O(d.n)$**	*$O(n \log n)$*
Space	No additional space	Require temporary array **$O(n)$**	No additional space	Heap **$O(n)$**	Require buckets		Require temporary files **$O(n)$**