AP Computer Science B

Java Object-Oriented Programming [Ver. 2.0]

Unit 5: Algorithm Study

WEEK 12: CHAPTER 16 ALGORITHMS (PART 2: SORTING)

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Array.sort()
- Insertion Sort/Selection Sort/Bubble Sort/Merge Sort/Quick Sort)

Arrays.sort

LECTURE 1

The Arrays.sort Method

- Since sorting is frequently used in programming, Java provides several overloaded sort methods for sorting an array of int, double, char, short, long, and float in the java.util.Arrays class. For example, the following code sorts an array of numbers and an array of characters.

```
double[] numbers = {6.0, 4.4, 1.9, 2.9, 3.4, 3.5};
java.util.Arrays.sort(numbers);

char[] chars = {'a', 'A', '4', 'F', 'D', 'P'};
java.util.Arrays.sort(chars);
```

If AP test want you to write a sorting program, DO NOT USE THIS !!!

If they allow you to use it, that’s fine.

Array Class
java.util.Arrays class (sorting)

- The java.util.Arrays class contains various static methods for sorting arrays, searching arrays, comparing arrays, filling array elements, and returning a string representation of the array. These methods are overloaded for all primitive types.

```
double[] numbers = {6.0, 4.4, 1.9, 3.4, 3.5} ;

java.util.Arrays.sort(numbers);

 java.util.Arrays.parallelsort(numbers);

char[] chars = {‘a’,’A’,’4’,’F’,’D’,’P’} ;

java.util.Arrays.sort(chars);

 java.util.Arrays.parallelsort(chars);
```

Sorting I: Insertion Sort

LECTURE 2

Insertion Sort

int[] myList = {2, 9, 5, 4, 8, 1, 6}; // Unsorted

The insertion sort algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.

Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.	2	9	5	4	8	1	6
Step2: The sorted sublist is {2, 9}. Insert 5 into the sublist.	2	9	5	4	8	1	6
Step 3: The sorted sublist is {2, 5, 9}. Insert 4 into the sublist.	2	5	9	4	8	1	6
Step 4: The sorted sublist is {2, 4, 5, 9}. Insert 8 into the sublist.	2	4	5	9	8	1	6
Step 5: The sorted sublist is {2, 4, 5, 8, 9}. Insert 1 into the sublist.	2	4	5	8	9	1	6
Step 6: The sorted sublist is {1, 2, 4, 5, 8, 9}. Insert 6 into the sublist.	1	2	4	5	8	9	6
Step 7: The entire list is now sorted.	1	2	4	5	6	8	9

Insertion Sort

int[] myList = {2, 9, 5, 4, 8, 1, 6}; // Unsorted

2	9	5	4	8	1	6

2	9	5	4	8	1	6

2	5	9	4	8	1	6

2	4	5	9	8	1	6

2	4	5	8	9	1	6

1	2	4	5	8	9	6

1	2	4	5	6	8	9

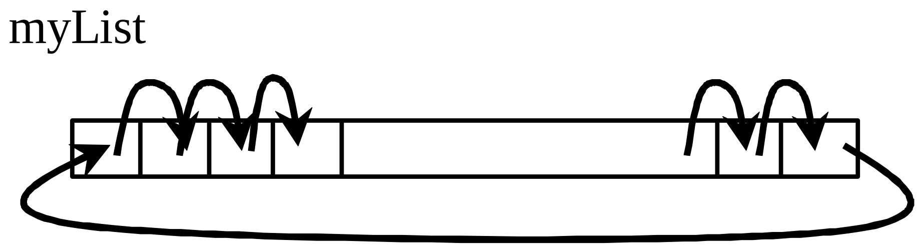
Shifting Elements (Right Shifting)

Borrowed from APCSA Chapter 7: Array Processing I

```
double temp = myList[myList.length-1]; // Retain the last element

// Shift elements left
for (int i = myList.length-2; i >=0; i--) {
  myList[i + 1] = myList[i];
}

// Move the last element to fill in the first position
myList[0] = temp;
```



How to Insert?

The insertion sort algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
list	2	5	9	4			

Step 1: Save 4 to a temporary variable currentElement

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
list	2	5		9			

Step 2: Move list[2] to list[3]

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
list	2		5	9			

Step 3: Move list[1] to list[2]

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
list	2	4	5	9			

Step 4: Assign currentElement to list[1]

From Idea to Solution

```
for (int i = 1; i < list.length; i++) {
  insert list[i] into a sorted sublist list[0..i-1] so that
  list[0..i] is sorted
}
```

```
list[0]

list[0] list[1]

list[0] list[1] list[2]

list[0] list[1] list[2] list[3]

list[0] list[1] list[2] list[3] ...
```

From Idea to Solution

```
for (int i = 1; i < list.length; i++) {
  insert list[i] into a sorted sublist list[0..i-1] so that
  list[0..i] is sorted
}
```

Expand

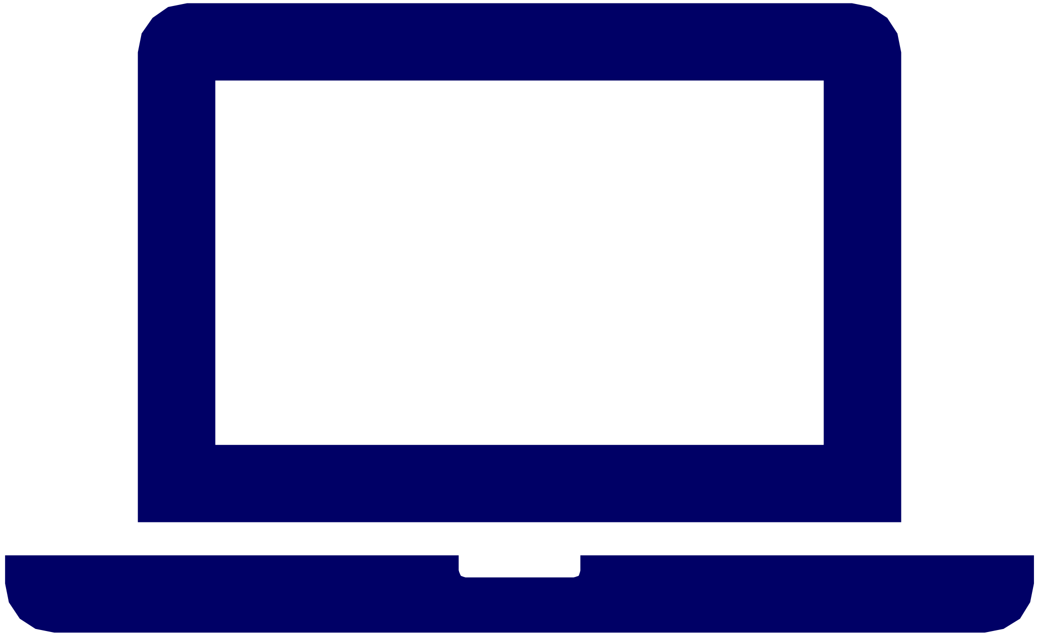
```
double currentElement = list[i];
int k;
for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {
  list[k + 1] = list[k];
}
// Insert the current element into list[k + 1]
list[k + 1] = currentElement;
```

Analyzing Insertion Sort

- The insertion sort algorithm presented in **InsertionSort.java**, sorts a list of values by repeatedly inserting a new element into a sorted partial array until the whole array is sorted. At the ***kth*** iteration, to insert an element to a array of size ***k***, it may take ***k*** comparisons to find the insertion position, and ***k*** moves to insert the element. Let ***T(n)*** denote the complexity for insertion sort and *c* denote the total number of other operations such as assignments and additional comparisons in each iteration. So,

$$T(n) = 2 + c + 2 \times 2 + c \ldots + 2 \times (n-1) + c = n^2 - n + cn$$

- Ignoring constants and smaller terms, the complexity of the selection sort algorithm is O(n^2).



Demonstration Program

INSERTIONSORT.JAVA

Sorting II: Selection Sort

LECTURE 3

Sorting Arrays

- Sorting, like searching, is also a common task in computer programming. Many different algorithms have been developed for sorting. This section introduces two simple, intuitive sorting algorithms: *selection sort* and *insertion sort*.

Selection Sort

- Selection sort finds the smallest number in the list and places it first.
- It then finds the smallest number remaining and places it second, and so on until the list contains only a single number.

Select 1 (the smallest) and swap it with 2 (the first) in the list

swap

2 9 5 4 8 1 6

Select 2 (the smallest) and swap it with 9 (the first) in the remaining list

swap

1 9 5 4 8 2 6

The number 1 is now in the correct position and thus no longer needs to be considered.

Select 4 (the smallest) and swap it with 5 (the first) in the remaining list

swap

1 2 5 4 8 9 6

The number 2 is now in the correct position and thus no longer needs to be considered.

5 is the smallest and in the right position. No swap is necessary

1 2 4 5 8 9 6

The number 6 is now in the correct position and thus no longer needs to be considered.

Select 6 (the smallest) and swap it with 8 (the first) in the remaining list

swap

1 2 4 5 8 9 6

The number 5 is now in the correct position and thus no longer needs to be considered.

Select 8 (the smallest) and swap it with 9 (the first) in the remaining list

swap

1 2 4 5 6 9 8

The number 6 is now in the correct position and thus no longer needs to be considered.

Since there is only one element remaining in the list, sort is completed

1 2 4 5 6 8 9

The number 8 is now in the correct position and thus no longer needs to be considered.

From Idea to Solution

```
for (int i = 0; i < list.length; i++)
{
  select the smallest element in list[i..listSize-1];
  swap the smallest with list[i], if necessary;
  // list[i] is in its correct position.
  // The next iteration apply on list[i..listSize-1]
}
```

From Idea to Solution

```
list[0] list[1] list[2] list[3] ...           list[10]
list[0] list[1] list[2] list[3] ...           list[10]
list[0] list[1] list[2] list[3] ...           list[10]
list[0] list[1] list[2] list[3] ...           list[10]
list[0] list[1] list[2] list[3] ...           list[10]
                                ...
list[0] list[1] list[2] list[3] ...           list[10]
```

```
for (int i = 0; i < listSize; i++)
{
  select the smallest element in list[i..listSize-1];
  swap the smallest with list[i], if necessary;
  // list[i] is in its correct position.
  // The next iteration apply on list[i..listSize-1]
}
```

Expand

```
double currentMin = list[i];
[redacted]
for (int j = i+1; j < list.length; j++) {
  if (currentMin > list[j]) {
    currentMin = list[j];
    [redacted]
  }
}
```

```
for (int i = 0; i < listSize; i++)
{
  select the smallest element in list[i..listSize-1];
  swap the smallest with list[i], if necessary;
  // list[i] is in its correct position.
  // The next iteration apply on list[i..listSize-1]
}
```

Expand

```
double currentMin = list[i];
int currentMinIndex = i;
for (int j = i; j < list.length; j++) {
  if (currentMin > list[j]) {
    currentMin = list[j];
    currentMinIndex = j;
  }
}
```

```
for (int i = 0; i < listSize; i++)
{
  select the smallest element in list[i..listSize-1];
  swap the smallest with list[i], if necessary;
  // list[i] is in its correct position.
  // The next iteration apply on list[i..listSize-1]
}
```

Expand

```
    if (currentMinIndex != i) {
      list[currentMinIndex] = list[i];
      list[i] = currentMin;
    }
```

Wrap it in a Method

```
/** The method for sorting the numbers */
public static void selectionSort(double[] list) {
  for (int i = 0; i < list.length; i++) {
    // Find the minimum in the list[i..list.length-1]
    double currentMin = list[i];
    int currentMinIndex = i;
    for (int j = i + 1; j < list.length; j++) {
      if (currentMin > list[j]) {
        currentMin = list[j];
        currentMinIndex = j;
      }
    }

    // Swap list[i] with list[currentMinIndex] if necessary;
    if (currentMinIndex != i) {
      list[currentMinIndex] = list[i];
      list[i] = currentMin;
    }
  }
}
```

Analyzing Selection Sort

- SelectionSort.java, finds the largest number in the list and places it last. It then finds the largest number remaining and places it next to last, and so on until the list contains only a single number. The number of comparisons is *n-1* for the first iteration, *n-2* for the second iteration, and so on. Let *T(n)* denote the complexity for selection sort and *c* denote the total number of other operations such as assignments and additional comparisons in each iteration. So,

$$T(n) = (n-1) + c + (n-2) + c \ldots + 2 + c + 1 + c = \frac{n^2}{2} - \frac{n}{2} + cn$$

- Ignoring constants and smaller terms, the complexity of the selection sort algorithm is $O(n^2)$.

Quadratic Time

An algorithm with the $O(n^2)$ time complexity is called a *quadratic algorithm*. The quadratic algorithm grows quickly as the problem size increases. If you double the input size, the time for the algorithm is quadrupled. Algorithms with two nested loops are often quadratic.

Demonstration Program

SELECTIONSORT.JAVA

Demo Program:

SelectionSort.java

- InsertSort.java’s core is **circular shifting**.
- SelectionSort.java’s core is **finding mimum and swap**.

Sorting III: Bubble Sort

LECTURE 4

Bubble Sort

A bubble sort sorts the array in multiple phases. Each pass successively swaps the neighboring elements if the elements are not in order.

- The bubble sort algorithm makes several passes through the array. On each pass, **successive neighboring pairs are compared**. If a pair is in decreasing order, its values are **swapped**; otherwise, the values remain unchanged.
- The technique is called a bubble sort or sinking sort, because the smaller values gradually “bubble” their way to the top and the larger values sink to the bottom. After the first pass, the last element becomes the largest in the array. After the second pass, the second-to-last element becomes the second largest in the array. This process is continued until all elements are sorted.

Bubble Sort

(a) 1st pass	(b) 2nd pass	(c) 3rd pass	(d) 4th pass	(e) 5th pass
2 9 5 4 8 1	2 5 4 8 1 9	2 4 5 1 8 9	2 4 1 5 8 9	1 2 4 5 8 9
2 5 9 4 8 1	2 4 5 8 1 9	2 4 5 1 8 9	2 1 4 5 8 9	
2 5 4 9 8 1	2 4 5 8 1 9	2 4 1 5 8 9		
2 5 4 8 9 1	2 4 5 1 8 9			
2 5 4 8 1 9				

Bubble sort time: $O(n^2)$

$$(n-1)+(n-2)+...+2+1=\frac{n^2}{2}-\frac{n}{2}$$

From idea to Solution

```
for (int k = 1; k<list.length; k++){
 // Perform the kth pass
 for (int i = 0; i<list.length-k; i++){
 if (list[i] > list[i + 1])
 swap list[i] with list[i + 1];
 }
}
```

Improved Bubble Sort Algorithm

```
boolean needNextPass = true;
for (int k = 1; k < list.length && needNextPass; k++) {
 // Array may be sorted and next pass not needed
 needNextPass = false;
 // Perform the kth pass
  for (int i = 0; i < list.length - k; i++) {
      if (list[i] > list[i + 1]) {  swap list[i] with list[i + 1];
         needNextPass = true; // Next pass still needed
      }
   }
}
```

Time Complexity

In the best case, the bubble sort algorithm needs just the first pass to find that the array is already sorted—no next pass is needed. Since the number of comparisons is $n - 1$ in the first pass, the best-case time for a bubble sort is $O(n)$.

In the worst case, the bubble sort algorithm requires $n - 1$ passes. The first pass makes $n - 1$ comparisons; the second pass makes $n - 2$ comparisons; and so on; the last pass makes 1 comparison. Thus, the total number of comparisons is:

$$(n-1) + (n-2) + \cdots + 2 + 1$$

$$= \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$

Therefore, the worst-case time for a bubble sort is $O(n^2)$.

Demonstration Program

BUBBLESORT.JAVA

Demo Program:

BubbleSort.java

- BubbleSort.java's Core: **Successive Comparison and Swap**.

Sorting IV: Merge Sort

LECTURE 5

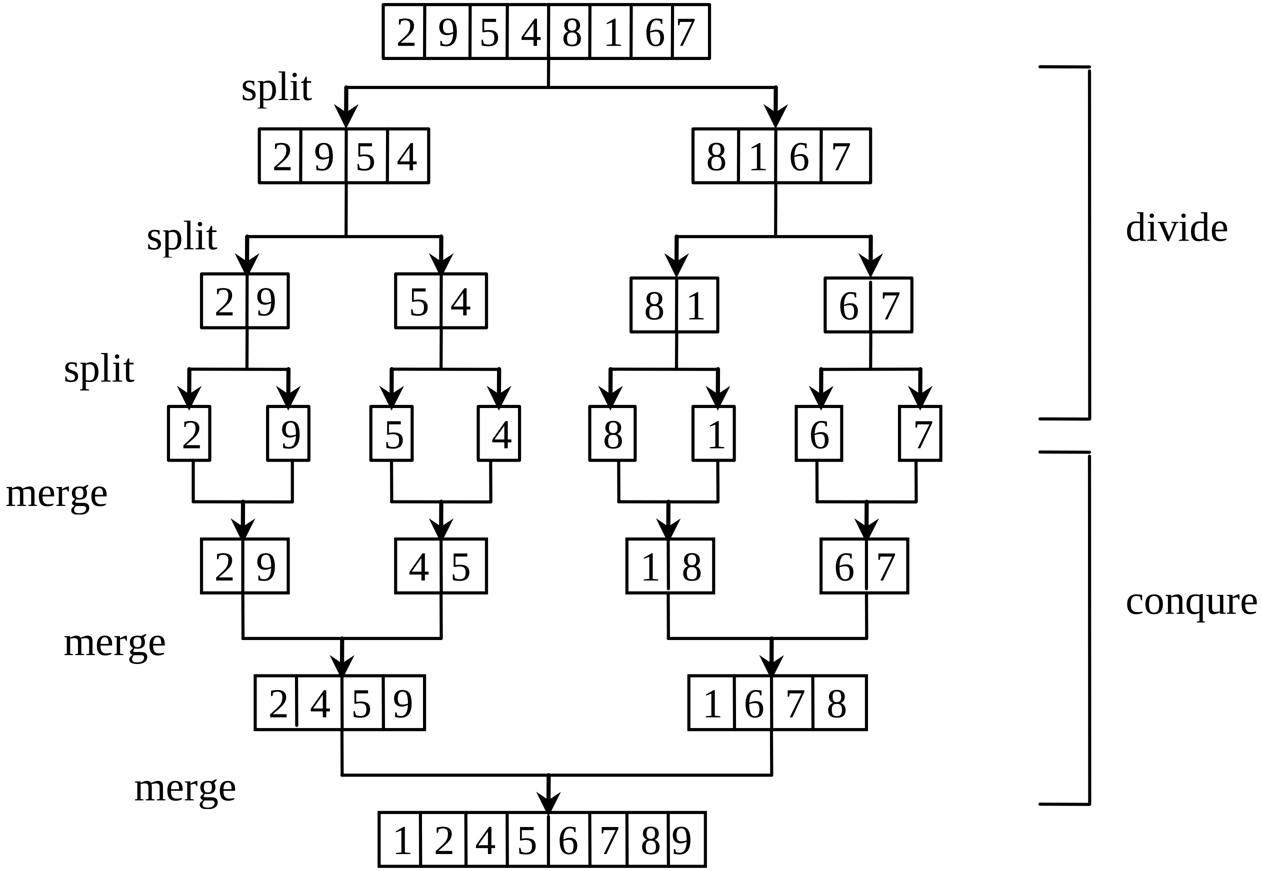
Merge Sort

The merge sort algorithm can be described recursively as follows: The algorithm divides the array into two halves and applies a merge sort on each half recursively. After the two halves are sorted, merge them.

Merge Sort Algorithm

```
public static void mergeSort(int[] list) {
    if (list.length > 1) {
        int low = 0, high = list.length-1;
        int mid = (low+high)/2;
        mergeSort(list[0 ... mid+ 1]);
        mergeSort(list[mid + 1 ... list.length]);
        merge list[0 ... mid+ 1] with
        list[mid + 1 ... list.length];
    }
} // First half will be a little bit longer (if length is odd)
```

Merge Sort



2 9 5 4 8 1 6 7
split
2 9 5 4
8 1 6 7
split
2 9
5 4
8 1
6 7
split
2
9
5
4
8
1
6
7
merge
2 9
4 5
1 8
6 7
merge
2 4 5 9
1 6 7 8
merge
1 2 4 5 6 7 8 9
divide
conqure

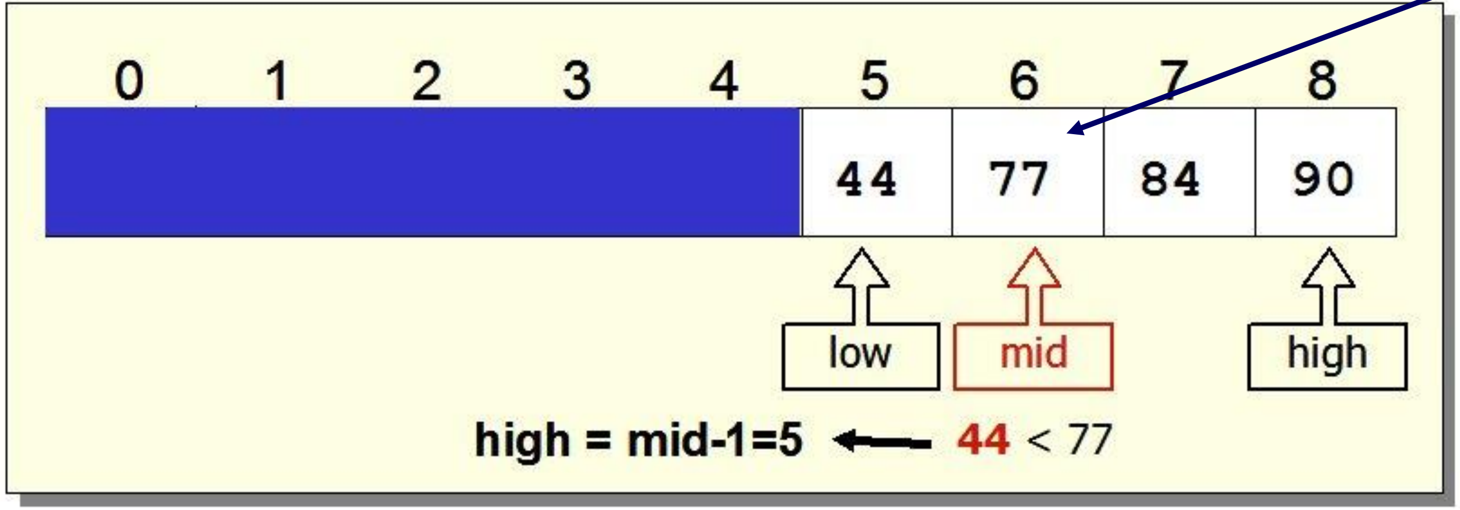
We use the same low, mid, high system as the Binary Search

	low	high	mid
#1	0	8	4
#2	5	8	6

search(44)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

mid belongs to first half



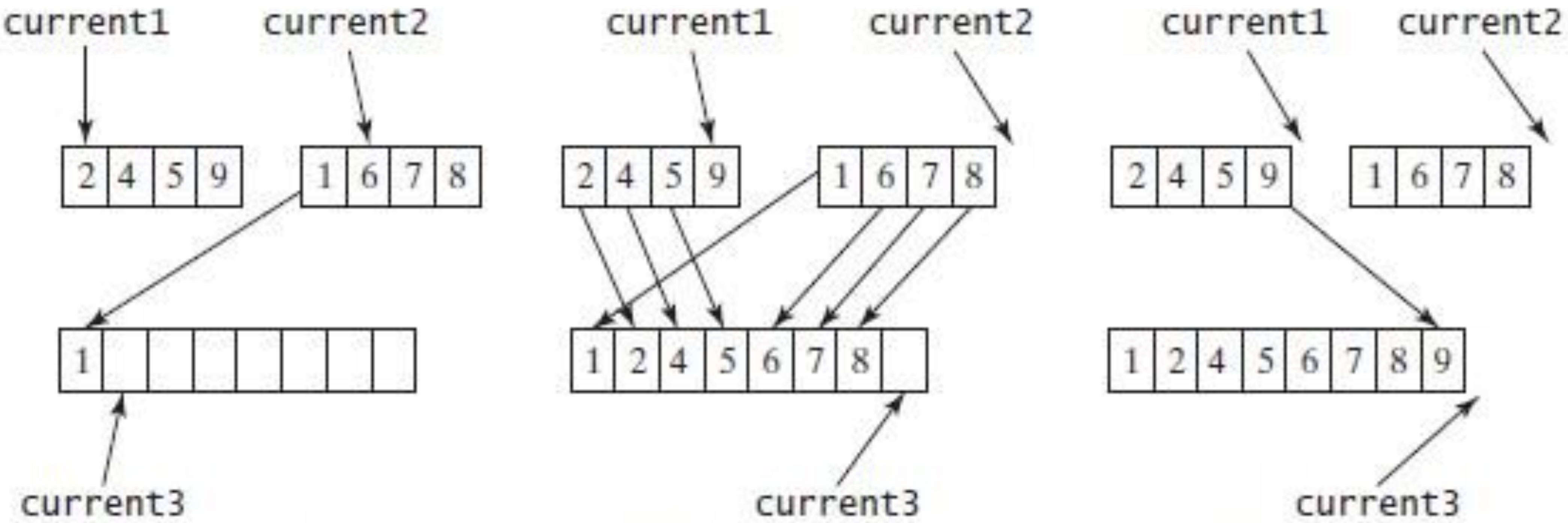
Top Level mergeSort Method

```java
public static void mergeSort(int[] list) {
  if (list.length > 1) {
    // Merge sort the first half
    int low = 0, high = list.length-1;
    int mid = (low+high)/2;
    int[] firstHalf = new int[mid+1];
    System.arraycopy(list, 0, firstHalf, 0, mid+1);
    mergeSort(firstHalf);

    // Merge sort the second half
    int secondHalfLength = list.length - (mid+1);
    int[] secondHalf = new int[secondHalfLength];
    System.arraycopy(list, mid+1,
      secondHalf, 0, secondHalfLength);
    mergeSort(secondHalf);

    // Merge firstHalf with secondHalf into list
    merge(firstHalf, secondHalf, list);
  }
}
```

Merge



(a) After moving 1 to temp

(b) After moving all the elements in list2 to temp

(c) After moving 9 to temp

merge Method

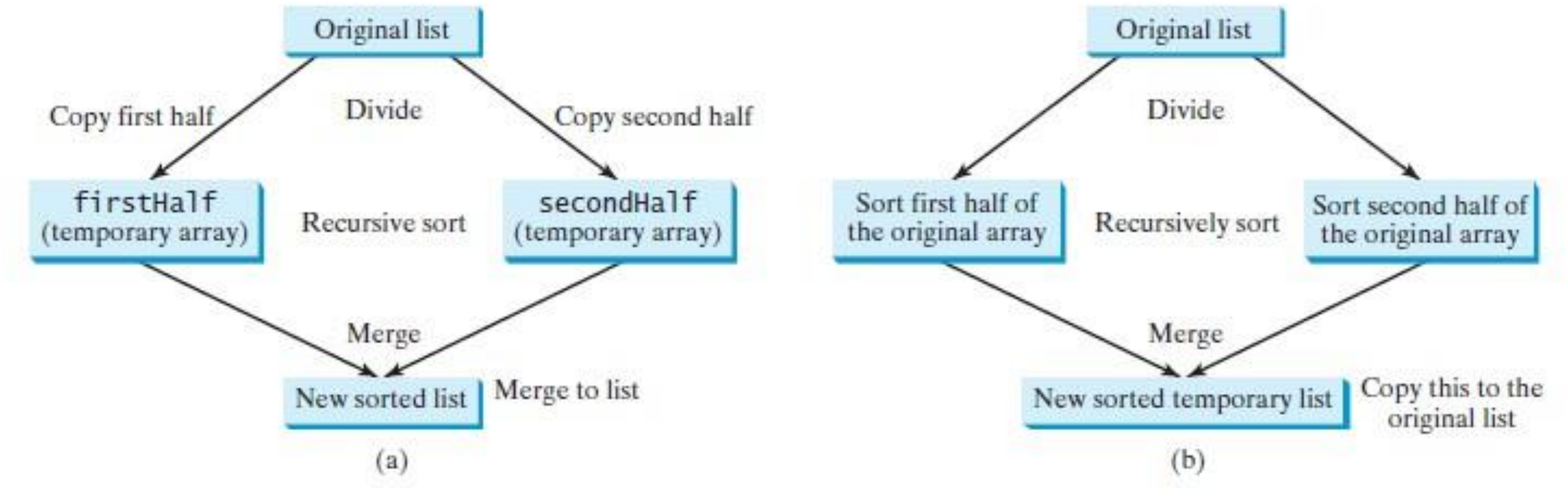
```java
/** Merge two sorted lists */
public static void merge(int[] list1, int[] list2, int[] temp) {
  int current1 = 0; // Current index in list1
  int current2 = 0; // Current index in list2
  int current3 = 0; // Current index in temp

  while (current1 < list1.length && current2 < list2.length) {
    if (list1[current1] < list2[current2])
      temp[current3++] = list1[current1++];
    else
      temp[current3++] = list2[current2++];
  }

  while (current1 < list1.length)
    temp[current3++] = list1[current1++];

  while (current2 < list2.length)
    temp[current3++] = list2[current2++];
}
```

Data Structures



Merge Sort Time

Let T(n) denote the time required for sorting an array of n elements using merge sort. Without loss of generality, assume n is a power of 2. The merge sort algorithm splits the array into two subarrays, sorts the subarrays using the same algorithm recursively, and then merges the subarrays. So,

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + mergetime$$

Merge Sort Time

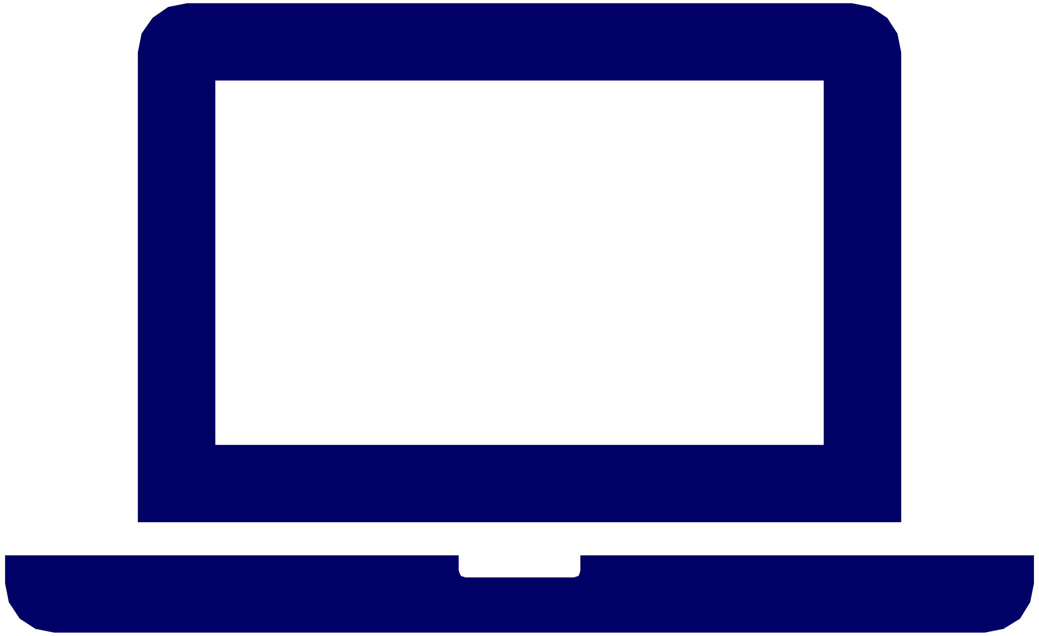
The first *T(n/2)* is the time for sorting the first half of the array and the second *T(n/2)* is the time for sorting the second half. To merge two subarrays, it takes at most *n-1* comparisons to compare the elements from the two subarrays and *n* moves to move elements to the temporary array. So, the total time is *2n-1*. Therefore,

$$T(n) = 2T(\frac{n}{2}) + 2n - 1 = 2(2T(\frac{n}{4}) + 2\frac{n}{2} - 1) + 2n - 1 = 2^2 T(\frac{n}{2^2}) + 2n - 2 + 2n - 1$$

$$= 2^k T(\frac{n}{2^k}) + 2n - 2^{k-1} + \ldots + 2n - 2 + 2n - 1$$

$$= 2^{\log n} T(\frac{n}{2^{\log n}}) + 2n - 2^{\log n - 1} + \ldots + 2n - 2 + 2n - 1$$

$$= n + 2n \log n - 2^{\log n} + 1 = 2n \log n + 1 = O(n \log n)$$



Demonstration Program

MERGESORT.JAVA

Demo Program:

MergeSort.java

- MergeSort’s core is Divide and merge on integration.
- Arrays.sort uses merge sort algorithm. Use extra memory space to cut time.

Sorting V: Quick Sort

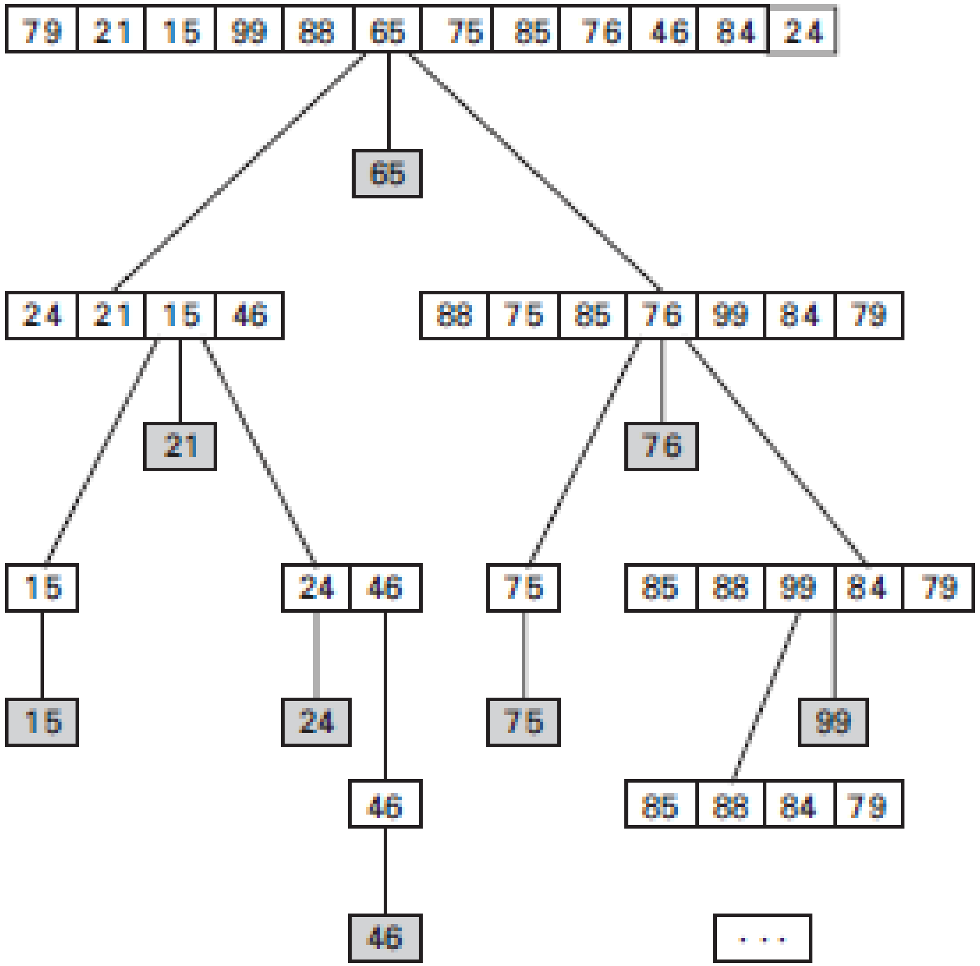
(Non-AP Topic)

LECTURE 6

Quick Sort

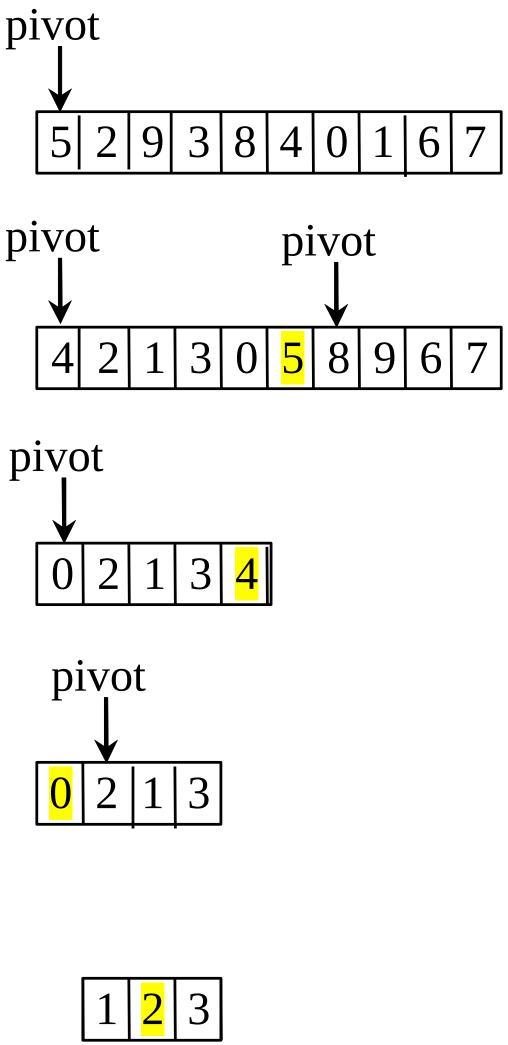
Quick sort, developed by C. A. R. Hoare (1962), works as follows: The algorithm selects an element, called the **pivot**, in the array. Divide the array into two parts such that all the elements in the first part are less than or equal to the pivot and all the elements in the second part are greater than the pivot. Recursively apply the quick sort algorithm to the first part and then the second part.

Conceptual Idea of Quick Sort



Izquierda: 24, 21, 15, 46
Pivote: 65
Derecha: 88, 75, 85, 76, 99, 84, 79

Quick Sort



(a) The original array

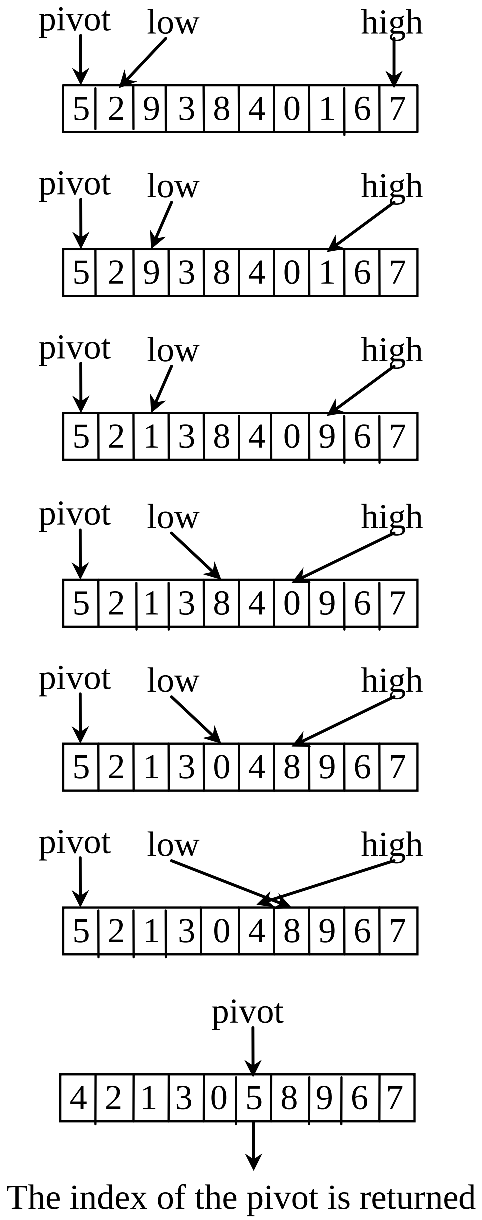
(b)The original array is partitioned

(c) The partial array (4 2 1 3 0) is partitioned

(d) The partial array (0 2 1 3) is partitioned

(e) The partial array (2 1 3) is partitioned

Partition



(a) Initialize pivot, low, and high

(b) Search forward and backward

(c) 9 is swapped with 1

(d) Continue search

(e) 8 is swapped with 0

(f) when high < low, search is over

(g) pivot is in the right place

Top-down Partition and Sort

```
private static void quickSort(int[] list, int first, int last) {
   if (last > first) {  // put use the first element as pivot.
     int pivotIndex = partition(list, first, last); //relocate first
     quickSort(list, first, pivotIndex - 1); // sort first part
     quickSort(list, pivotIndex + 1, last); // sort second part
   }
 }
```

Partition Algorithm

```
private static int partition(int[] list, int first, int last) {
 (1)  int pivot = list[first]; int low = first + 1; int high = last;  //Set the pivot, low and high
 (2)   while (high > low) {
          //  Search forward from left
          while (low <= high && list[low] <= pivot) low++;  // stop when low hit a bigger one
          // Search backward from right
          while (low <= high && list[high] > pivot) high--; // stop when high hit a smaller one
          // Swap two elements in the list
          if (high > low) { int temp = list[high]; list[high] = list[low]; list[low] = temp; }
        } // when stopped, high <= low.  Anything above high is bigger than pivot
 (3) while (high > first && list[high] >= pivot) high--;  // find the right pivot location
 (4)  // Swap pivot with list[high]
          if (pivot > list[high]) { list[first] = list[high]; list[high] = pivot; return high; } // pivot it not high
          else { return first; } // pivot is first
}
```

Quick Sort Time

To partition an array of *n* elements, it takes *n* comparisons and *n* moves in the worst case. So, the time required for partition is *O(n)*.

In the worst case, each time the pivot divides the array into one big subarray with the other empty. The size of the big subarray is one less than the one before divided. The algorithm requires time:

$$(n-1)+(n-2)+\ldots+2+1=O(n^2)$$

Worst-Case Time

In the worst case, each time the pivot divides the array into one big subarray with the other empty. The size of the big subarray is one less than the one before divided. The algorithm requires:

$$(n-1)+(n-2)+\ldots+2+1=O(n^2)$$

Best-Case Time

In the best case, each time the pivot divides the array into two parts of about the same size. Let *T(n)* denote the time required for sorting an array of elements using quick sort. So,

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + n$$

Average-Case Time

On the average, each time the pivot will not divide the array into two parts of the same size nor one empty part. Statistically, the sizes of the two parts are very close. So the average time is *O(logn)*. The exact average-case analysis is beyond the scope of this book.

Demonstration Program

QUICKSORT.JAVA

Demo Program:

QuickSort.java

- Quick Sort is based on pick a pivot, throws bigger one to its right and smaller one to its left. **Pivot and Throw.**