DATA STRUCTURES

WENYE LI
CUHK-SZ

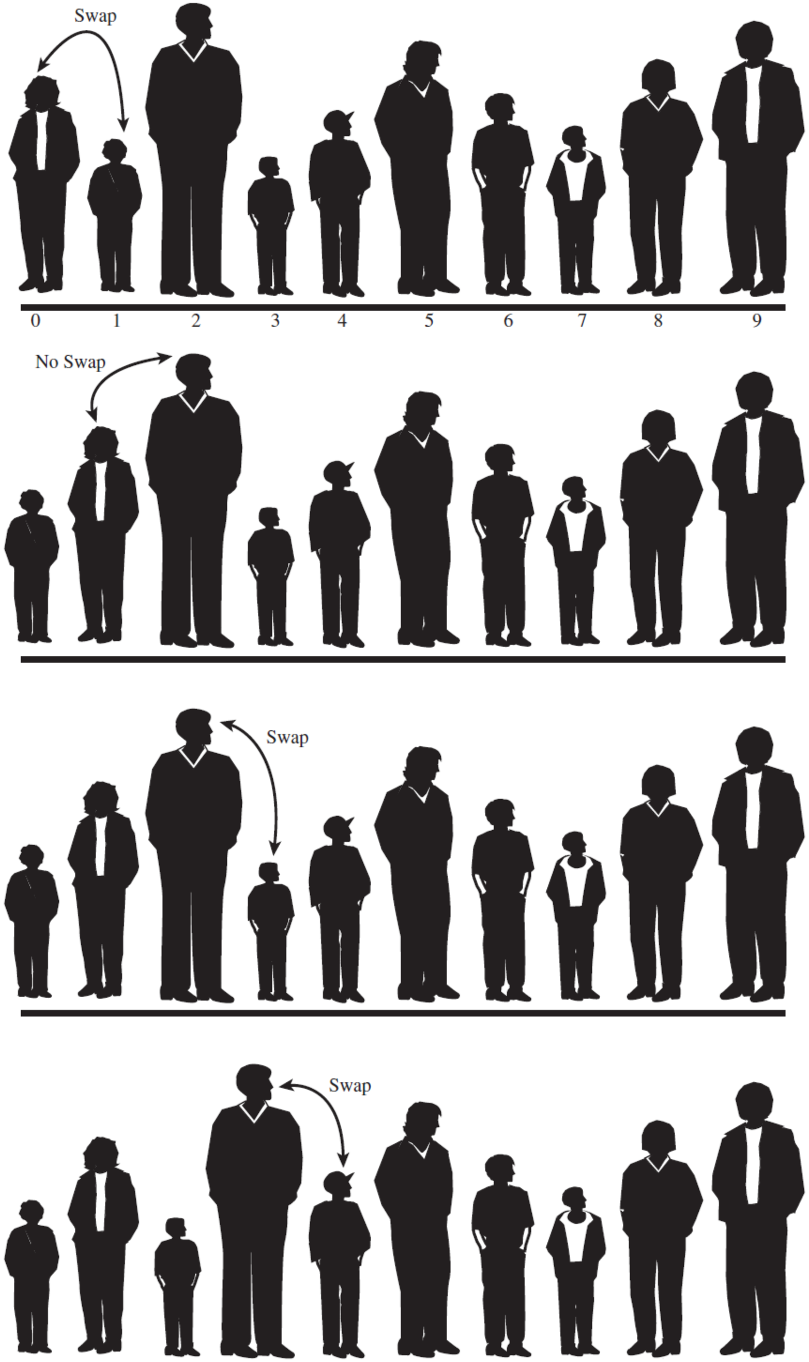
SORTING

- Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order.
- Consider an array $A = \{A_1, A_2, \cdots, A_n\}$,
  - The array is called to be in ascending order if $A_1 \leq A_2 \leq \cdots \leq A_n$,
  - Descending, if $A_1 \geq A_2 \geq \cdots \geq A_n$

SN	Sorting Algorithms	Description
1	Bubble Sort	It is the simplest sort method which performs sorting by repeatedly moving the largest element to the highest index of the array. It comprises of comparing each element to its adjacent element and replace them accordingly.
2	Bucket Sort	Bucket sort is also known as bin sort. It works by distributing the element into the array also called buckets. In this sorting algorithms, Buckets are sorted individually by using different sorting algorithm.
3	Comb Sort	Comb Sort is the advanced form of Bubble Sort. Bubble Sort compares all the adjacent values while comb sort removes all the turtle values or small values near the end of the list.
4	Counting Sort	It is a sorting technique based on the keys i.e. objects are collected according to keys which are small integers. Counting sort calculates the number of occurrence of objects and stores its key values. New array is formed by adding previous key elements and assigning to objects.
5	Heap Sort	In the heap sort, Min heap or max heap is maintained from the array elements deending upon the choice and the elements are sorted by deleting the root element of the heap.
6	Insertion Sort	As the name suggests, insertion sort inserts each element of the array to its proper place. It is a very simple sort method which is used to arrange the deck of cards while playing bridge.
7	Merge Sort	Merge sort follows divide and conquer approach in which, the list is first divided into the sets of equal elements and then each half of the list is sorted by using merge sort. The sorted list is combined again to form an elementary sorted array.
8	Quick Sort	Quick sort is the most optimized sort algorithms which performs sorting in O(n log n) comparisons. Like Merge sort, quick sort also work by using divide and conquer approach.
9	Radix Sort	In Radix sort, the sorting is done as we do sort the names according to their alphabetical order. It is the lenear sorting algorithm used for Inegers.
10	Selection Sort	Selection sort finds the smallest element in the array and place it on the first place on the list, then it finds the second smallest element in the array and place it on the second place. This process continues until all the elements are moved to their correct ordering. It carries running time O(n2) which is worst than insertion sort.
11	Shell Sort	Shell sort is the generalization of insertion sort which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions.

SIMPLE SORTING

- Key idea:
  - Compare two items
  - Swap two items, or copy one item
  - Bubble Sort, Selection Sort, Insertion Sort



Bubble sort: the beginning of the first pass.

Bubble sort: the end of the first pass.

Bubble Sort:

Compare two players.

If the one on the left is taller, swap them.

Move one position right.

When you reach the first sorted player, start over at the left end of the line.

Continue this process until all players are in order.

```java
class ArrayBub {
    private long[] a; // ref to array a
    private int nElems; // number of data items
    public ArrayBub(int max) // constructor
    {
        a = new long[max]; // create the array
        nElems = 0; // no items yet
    }
    public void insert(long value) // put element into array
    {
        a[nElems] = value; // insert it
        nElems++; // increment size
    }
    public void display() // displays array contents
    {
        for(int j = 0; j < nElems; j++) // for each element,
            System.out.print(a[j] + “ “); // display it
        System.out.println(“”);
    }
    public void bubbleSort()
    {
        int out, in;
        for(out = nElems - 1; out > 1; out--) // outer loop (backward)
            for(in = 0; in < out; in++) // inner loop (forward)
                if( a[in] > a[in + 1] ) // out of order?
                    swap(in, in + 1); // swap them
    } // end bubbleSort()
    private void swap(int one, int two)
    {
        long temp = a[one];
        a[one] = a[two];
        a[two] = temp;
    }
} // end class ArrayBub
class BubbleSortApp {
    public static void main(String[] args) {
        int maxSize = 100; // array size
        ArrayBub arr; // reference to array
        arr = new ArrayBub(maxSize); // create the array
        arr.insert(77); // insert 10 items
        arr.insert(99);
        arr.insert(44);
        arr.insert(55);
        arr.insert(22);
        arr.insert(88);
        arr.insert(11);
        arr.insert(00);
        arr.insert(66);
        arr.insert(33);
        arr.display(); // display items
        arr.bubbleSort(); // bubble sort them
        arr.display(); // display them again
    } // end main()
} // end class BubbleSortApp
```

```java
public void bubbleSort()
  {
  int out, in;

  for(out=nElems-1; out>1; out--)   // outer loop (backward)
    for(in=0; in<out; in++)         // inner loop (forward)
      if( a[in] > a[in+1] )         // out of order?
        swap(in, in+1);             // swap them
  }  // end bubbleSort()
```

```
77 99 44 55 22 88 11 0 66 33
0 11 22 33 44 55 66 77 88 99
```

In general, where N is the number of items in the array, there are N-1 comparisons on the first pass, N-2 on the second, and so on. The formula for the sum of such a series is

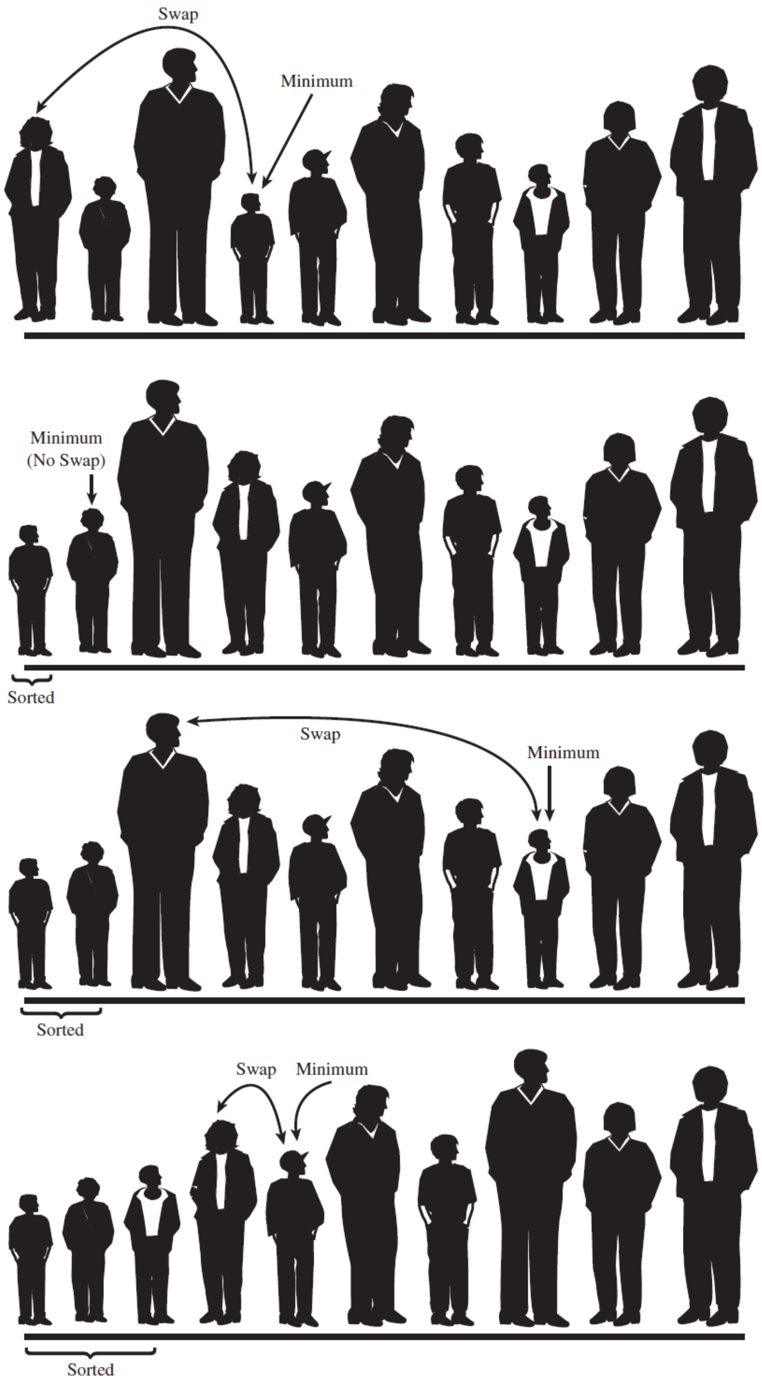
$(N-1) + (N-2) + (N-3) + \dots + 1 = N*(N-1)/2$

$N*(N-1)/2$ is 45 (10*9/2) when N is 10.

Thus, the algorithm makes about $N^2/2$ comparisons (ignoring the –1, which doesn't make much difference, especially if N is large).

There are fewer swaps than there are comparisons because two bars are swapped only if they need to be. If the data is random, a swap is necessary about half the time, so there will be about $N^2/4$ swaps. (Although in the worst case, with the initial data inversely sorted, a swap is necessary with every comparison.)

Both swaps and comparisons are proportional to N^2. Because constants don't count in Big O notation, we can ignore the 2 and the 4 and say that the bubble sort runs in $O(N^2)$ time.



Selection sort on baseball players.

```
public void selectionSort()
   {
   int out, in, min;

   for(out=0; out<nElems-1; out++)   // outer loop
      {
      min = out;                     // minimum
      for(in=out+1; in<nElems; in++) // inner loop
         if(a[in] < a[min] )         // if min greater,
             min = in;               // we have a new min
      swap(out, min);                // swap them
      }  // end for(out)
   }  // end selectionSort()
```

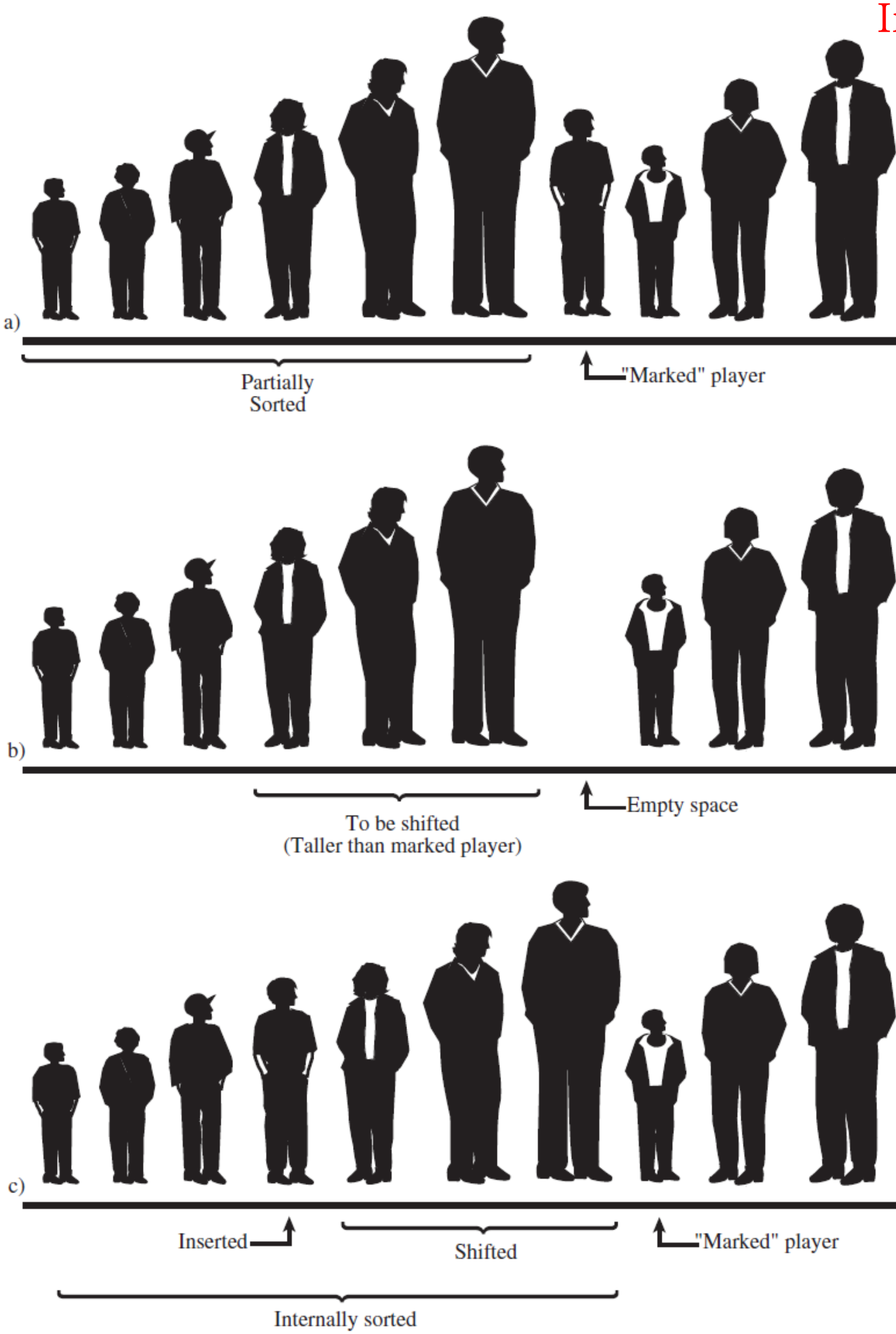
Selection Sort:

Making a pass through all players and picking the shortest one. This shortest player is then swapped with the player on the left end of the line, at position 0. Now the leftmost player is sorted and won't need to be moved again.

The next time you pass down the players, you start at position 1, and, finding the minimum, swap with position 1. This process continues until all players are sorted.

The selection sort performs the same number of comparisons as the bubble sort: N*(N-1)/2. For 10 data items, this is 45 comparisons. However, 10 items require fewer than 10 swaps. With 100 items, 4,950 comparisons are required, but fewer than 100 swaps. For large values of N, the comparison times will dominate, so we would have to say that the selection sort runs in $O(N^2)$ time, just as the bubble sort did. However, it is unquestionably faster because there are so few swaps. For smaller values of N, the selection sort may in fact be considerably faster, especially if the swap times are much larger than the comparison times.

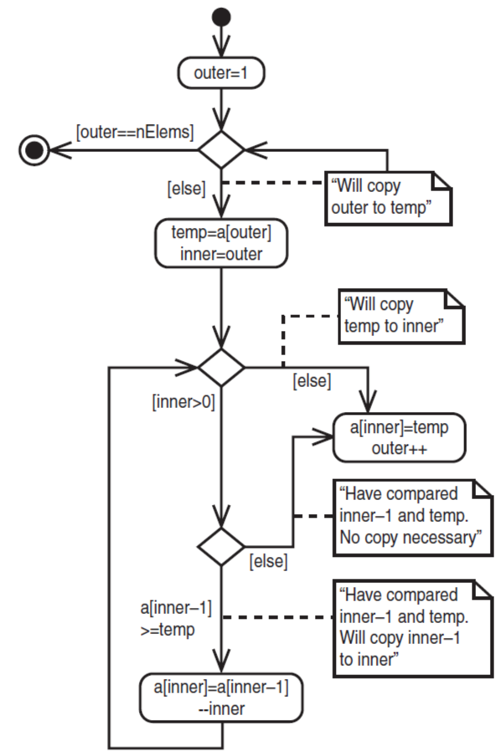
Insertion Sort



The insertion sort on baseball players.

```
public void insertionSort()
  {
  int in, out;

  for(out=1; out<nElems; out++)      // out is dividing line
     {
     long temp = a[out];             // remove marked item
     in = out;                       // start shifts at out
     while(in>0 && a[in-1] >= temp) // until one is smaller,
        {
        a[in] = a[in-1];             // shift item to right
        --in;                        // go left one position
        }
     a[in] = temp;                   // insert marked item
     }  // end for
```



Activity diagram for `insertSort()`.

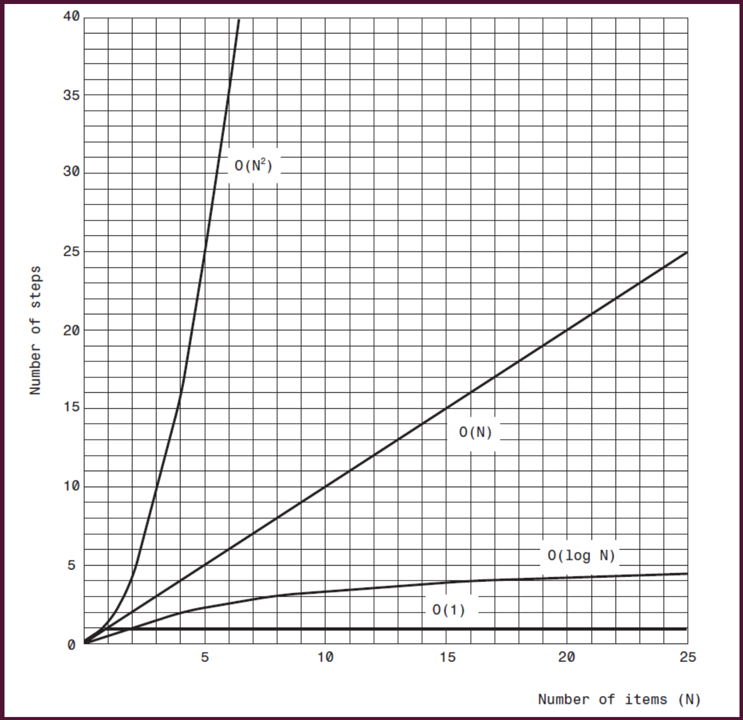
In most cases the insertion sort is the best of the elementary sorts described in this chapter. It still executes in $O(N^2)$ time, but it's about twice as fast as the bubble sort and somewhat faster than the selection sort in normal situations.

SUMMARY OF SIMPLE SORTING

- The sorting algorithms all assume an array as a data storage structure.
- Sorting involves comparing data items in the array and moving them until sorted.
- All execute in $O(N^2)$ time. Nevertheless, some can be substantially faster than others.
- An invariant is a condition that remains unchanged while an algorithm runs.
- The bubble sort is the least efficient, but the simplest, sort.
- The insertion sort is the most commonly used of the $O(N^2)$ sorts.
- A sort is stable if the order of elements with the same key is retained.
- None of the sorts require more than a single temporary variable, in addition to the original array.

ADVANCED SORTING

- Merge Sort, Shell Sort, and Quick Sort
  - Operate much faster than simple sorting



MERGESORT

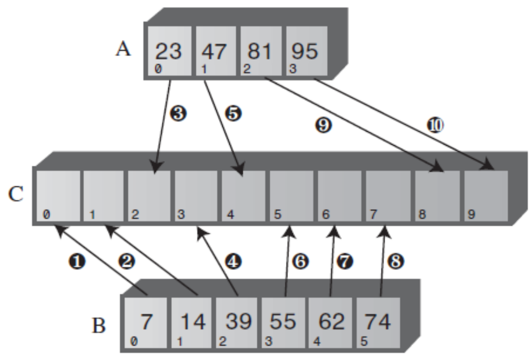
- A much more efficient sorting technique than simple sorting methods, at least in terms of speed.
    - Bubble, Insertion, and Selection sorts take $O(N^2)$ time.
    - Mergesort is $O(N * \log N)$.
    - $N = 10{,}000$: $N^2 =$100,000,000, $N * \log N = 40{,}000$.
    - If sorting this N items required 40 seconds with Mergesort, it would take almost 28 hours for Insertion sort.
    - It's conceptually easier than Quicksort and Shell sort.

- Mergesort requires an additional array, equal in size to the one being sorted.
    - With limited memory, Mergesort won't work.
    - If you have enough space, it's a good choice.

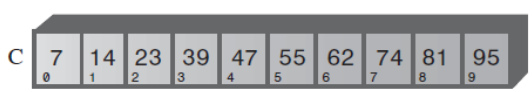
MERGING TWO SORTED ARRAYS

- The heart of Mergesort is the merging of two already-sorted arrays.

- Merging two sorted arrays A and B creates a third array C, that contains all the elements of A and B, also arranged in sorted order.



a) Before Merge



b) After Merge

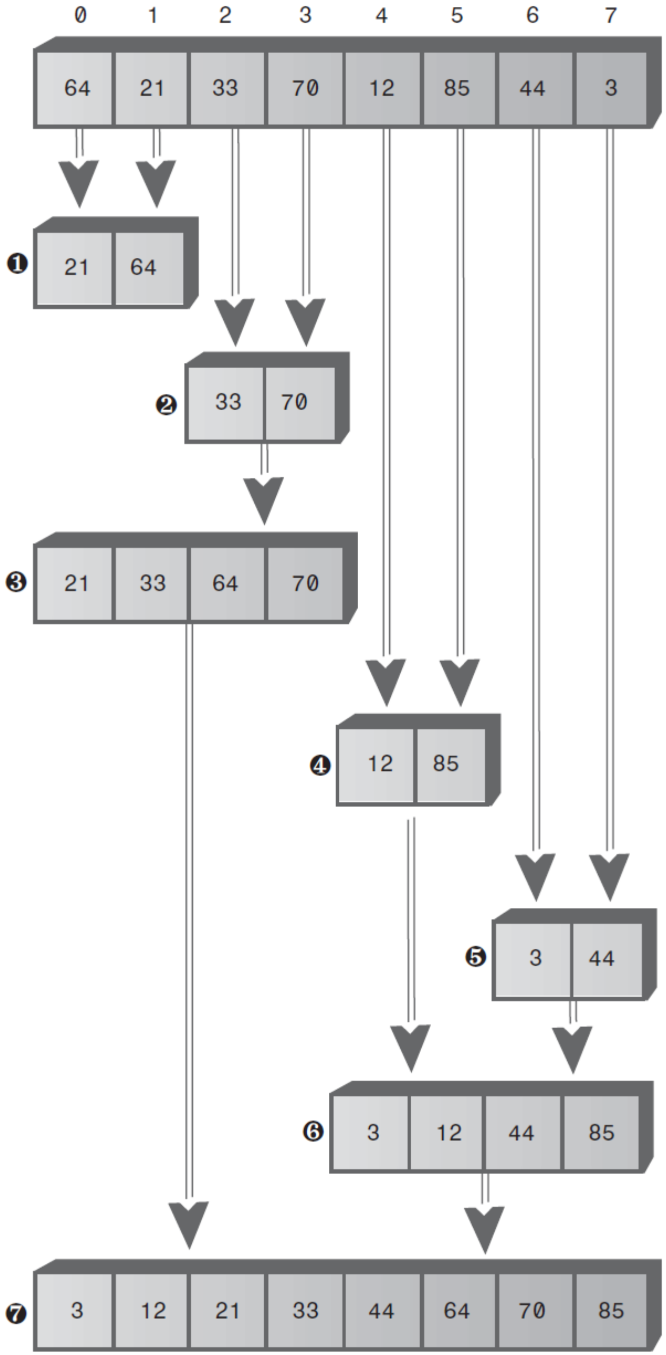
Step	Comparison (If Any)	Copy
1	Compare 23 and 7	Copy 7 from B to C
2	Compare 23 and 14	Copy 14 from B to C
3	Compare 23 and 39	Copy 23 from A to C
4	Compare 39 and 47	Copy 39 from B to C
5	Compare 55 and 47	Copy 47 from A to C
6	Compare 55 and 81	Copy 55 from B to C
7	Compare 62 and 81	Copy 62 from B to C
8	Compare 74 and 81	Copy 74 from B to C
9		Copy 81 from A to C
10		Copy 95 from A to C

```java
// merge.java
// demonstrates merging two arrays into a third
// to run this program: C>java MergeApp
////////////////////////////////////////////////////////////////
class MergeApp
{
    public static void main(String[] args)
    {
        int[] arrayA = {23, 47, 81, 95};
        int[] arrayB = {7, 14, 39, 55, 62, 74};
        int[] arrayC = new int[10];
        merge(arrayA, 4, arrayB, 6, arrayC);
        display(arrayC, 10);
    } // end main()
    //-------------------------------------------------------------
    // merge A and B into C
    public static void merge( int[] arrayA, int sizeA,
                              int[] arrayB, int sizeB,
                              int[] arrayC )
    {
        int aDex = 0, bDex = 0, cDex = 0;
        while(aDex < sizeA && bDex < sizeB) // neither array empty
            if( arrayA[aDex] < arrayB[bDex] )
                arrayC[cDex++] = arrayA[aDex++];
            else
                arrayC[cDex++] = arrayB[bDex++];
        while(aDex < sizeA) // arrayB is empty,
            arrayC[cDex++] = arrayA[aDex++]; // but arrayA isn’t
        while(bDex < sizeB) // arrayA is empty,
            arrayC[cDex++] = arrayB[bDex++]; // but arrayB isn’t
    } // end merge()
    //-------------------------------------------------------------
    // display array
    public static void display(int[] theArray, int size)
    {
        for(int j = 0; j < size; j++)
            System.out.print(theArray[j] + “ “);
        System.out.println(“”);
    }
    //-------------------------------------------------------------
} // end class MergeApp
```

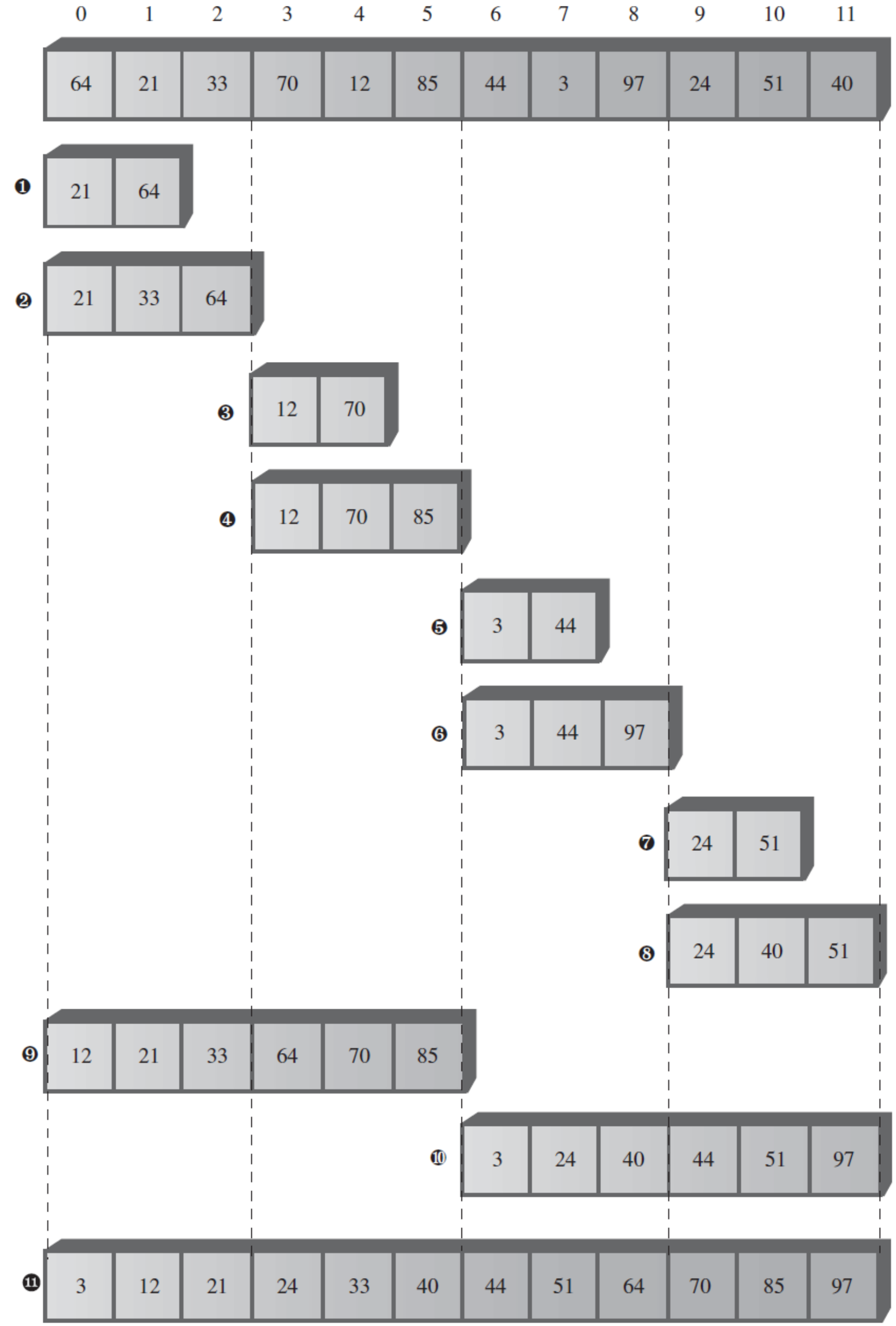
SORTING BY MERGING

- The idea in Mergesort:
  - divide an array in half,
  - sort each half,
  - and then use the merge() method to merge the two halves into a single sorted array.

- How to sort each half?
  - divide the half into two quarters,
  - sort each of the quarters,
  - and merge them to make a sorted half.



Merging larger and larger arrays.



Array size not a power of 2.

```java
// mergeSort.java
// demonstrates recursive merge sort
// to run this program: C>java MergeSortApp
////////////////////////////////////////////////////////////////
class DArray
{
    private long[] theArray; // ref to array theArray
    private int nElems; // number of data items
    //-----------------------------------------------------------
    public DArray(int max) // constructor
    {
        theArray = new long[max]; // create array
        nElems = 0;
    }
    //-----------------------------------------------------------
    public void insert(long value) // put element into array
    {
        theArray[nElems] = value; // insert it
        nElems++; // increment size
    }
    //-----------------------------------------------------------
    public void display() // displays array contents
    {
        for(int j = 0; j < nElems; j++) // for each element,
            System.out.print(theArray[j] + “ “); // display it
        System.out.println(“”);
    }
    //-----------------------------------------------------------
    public void mergeSort() // called by main()
    {
        // provides workspace
        long[] workSpace = new long[nElems];
        recMergeSort(workSpace, 0, nElems - 1);
    }
    //-----------------------------------------------------------
    private void recMergeSort(long[] workSpace, int lowerBound,
                                                int upperBound)
    {
        if(lowerBound == upperBound) // if range is 1,
            return; // no use sorting
        else
        {
            // find midpoint
            int mid = (lowerBound + upperBound) / 2;
            // sort low half
            recMergeSort(workSpace, lowerBound, mid);
            // sort high half
            recMergeSort(workSpace, mid + 1, upperBound);
            // merge them
            merge(workSpace, lowerBound, mid + 1, upperBound);
        } // end else
    } // end recMergeSort()
    private void merge(long[] workSpace, int lowPtr,
                                int highPtr, int upperBound)
    {
        int j = 0; // workspace index
        int lowerBound = lowPtr;
        int mid = highPtr - 1;
        int n = upperBound - lowerBound + 1; // # of items
        while(lowPtr <= mid && highPtr <= upperBound)
            if( theArray[lowPtr] < theArray[highPtr] )
                workSpace[j++] = theArray[lowPtr++];
            else
                workSpace[j++] = theArray[highPtr++];
        while(lowPtr <= mid)
            workSpace[j++] = theArray[lowPtr++];
        while(highPtr <= upperBound)
            workSpace[j++] = theArray[highPtr++];
        for(j = 0; j < n; j++)
            theArray[lowerBound + j] = workSpace[j];
    } // end merge()
    //-----------------------------------------------------------
} // end class DArray
////////////////////////////////////////////////////////////////
class MergeSortApp
{
    public static void main(String[] args)
    {
        int maxSize = 100; // array size
        DArray arr; // reference to array
        arr = new DArray(maxSize); // create the array
        arr.insert(64); // insert items
        arr.insert(21);
        arr.insert(33);
        arr.insert(70);
        arr.insert(12);
        arr.insert(85);
        arr.insert(44);
        arr.insert(3);
        arr.insert(99);
        arr.insert(0);
        arr.insert(108);
        arr.insert(36);
        arr.display(); // display items
        arr.mergeSort(); // merge sort the array
        arr.display(); // display items again
    } // end main()
} // end class MergeSortApp
```

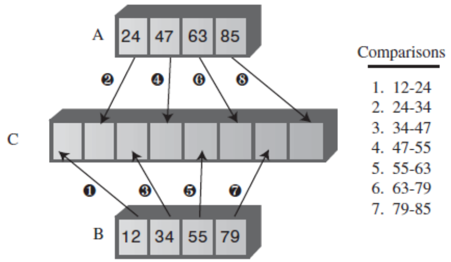
EFFICIENCY OF MERGESORT

- Mergesort runs in $O(N * \log N)$ time. Why?
  - Let's figure out the number of times a data item must be copied and the number times it must be compared with another data item.
  - Assume that copying/comparing are the most expensive operations; that the recursive calls and returns don't add much overhead.

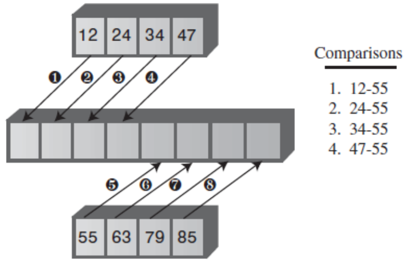
- Example: To sort 8 items requires 3 levels, each of which involves 8 copies. A level means all copies into the same size subarray.
  - In the 1st level, there are four 2-element subarrays;
  - In the 2nd level, there are two 4-element subarrays;
  - In the 3rd level, there is one 8-element subarray.
  - Each level has 8 elements, and there are 3*8 or 24 copies.

Number of Operations When N Is a Power of 2

N	$\log_2 N$	Number of Copies into Workspace ($N*\log_2 N$)	Total Copies	Comparisons Max (Min)
2	1	2	4	1 (1)
4	2	8	16	5 (4)
8	3	24	48	17 (12)
16	4	64	128	49 (32)
32	5	160	320	129 (80)
64	6	384	768	321 (192)
128	7	896	1792	769 (448)



a) Worst-case Scenario



b) Best-case Scenario

Step Number	1	2	3	4	5	6	7	Totals
Number of items being merged (N)	2	2	4	2	2	4	8	24
Maximum comparisons (N-1)	1	1	3	1	1	3	7	17
Minimum comparisons (N/2)	1	1	2	1	1	2	4	12

THANKS