Vietnam National University of HCMC
International University
School of Computer Science and Engineering

Data Structures and Algorithms
★ Advance Sorting ★

Dr Vi Chi Thanh - vcthanh@hcmiu.edu.vn
https://vichithanh.github.io

SCAN ME

Week by week topics (*)

1. Overview, DSA, OOP and Java
2. Arrays
3. Sorting
4. Queue, Stack
5. List
6. Recursion

Mid-Term

7. Advanced Sorting
8. Binary Tree
9. Hash Table
10. Graphs
11. Graphs Adv.

Final-Exam

10 LABS

Today objectives

- Shell sort
- Partitioning
- Quick sort
- Radix sort

Shell sort

Introduction

- Based on insertion sort
- Is good for medium-size arrays
- Faster than $O(N^2)$ – selection, insertion
- Is recommended to use in first place for any sorting project.

Review insertion sort

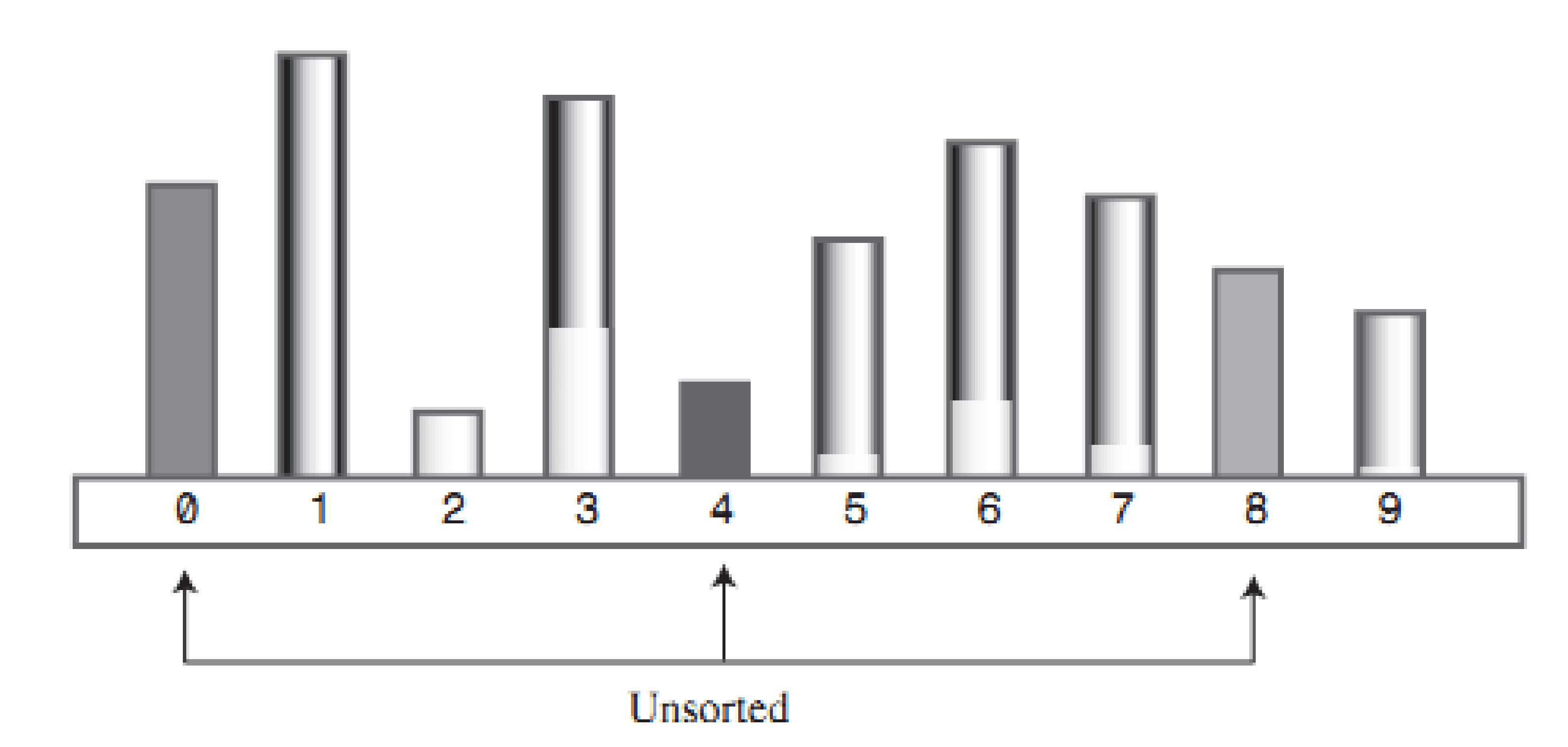
- Sort the following array

100	34	51	61	73	0

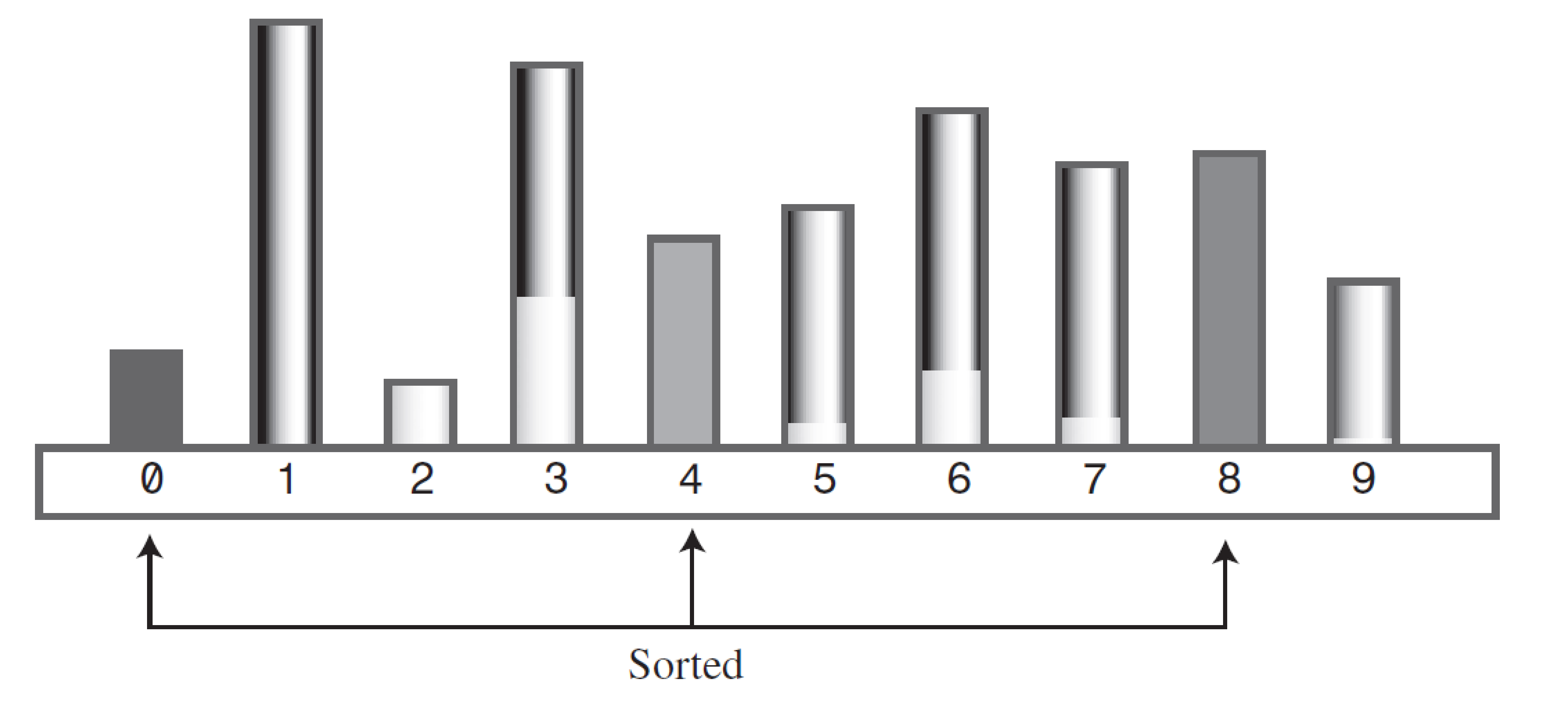
- How many copies have been made?
  - → To many copies
  - → can be improved

N-sorting

- Insertion sort widely spaced elements
- *Increment*: spacing between elements (h)

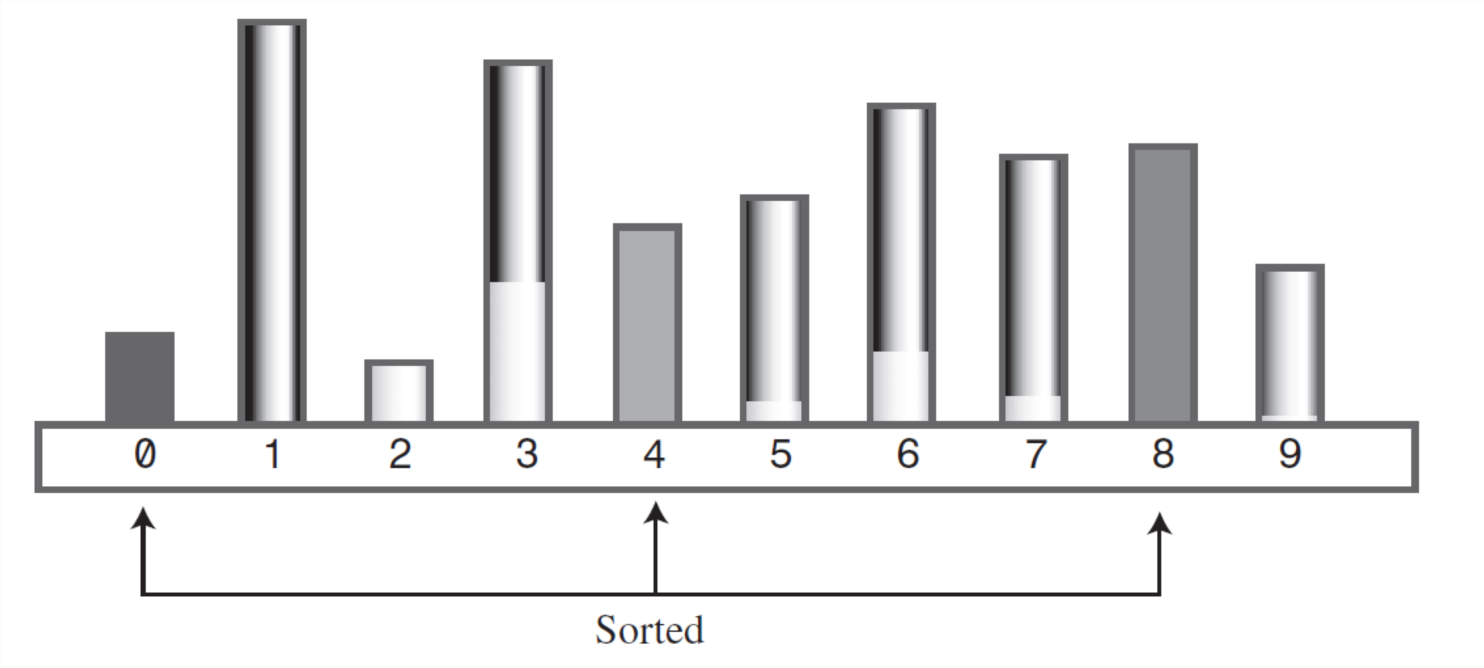


4-sorting



4-sorting

- Array is though of as 4 subarrays:
  - (0, 4, 8), (1, 5, 9), (2, 6), (3, 7)



4-sorted arrays

- All sub-arrays are sorted
- No item is more than 3 cells from where it should be (in our case)
  - → “almost” sorted
  - → is the secret of the Shellsort.
- Continue with the 1-sorting (insertion sort)

Animation

- https://opendsa-server.cs.vt.edu/embed/shellsortAV

Diminishing gap

- For array of 10 elements:
  - 4-sort then 1-sort
- For array of 1000 elements?
  - 364-sort, 121 sort, 40-sort, 13-sort, 4-sort and then 1-sort
- What is the interval sequence or gap sequence?
- How would you calculate it?

Knuth gap sequence

h = 3 * h + 1

- First value: **1**
- Apply the formula until

h > size of array

- Example:
  - Generate the gap sequence for 1100-element array

Knuth gap sequence

- What is the next gap?

h = (h - 1) / 3

- Until h = 1

Implementation

- Find the initial value of h (gap)

```
int h = 1;                          // find initial value of h
while(h <= nElems/3)
   h = h*3 + 1;                     // (1, 4, 13, 40, 121, ...)
```

```
while(h>0)                          // decreasing h, until h=1
   {
                                    // h-sort the file
   for(outer=h; outer<nElems; outer++)
      {
      temp = theArray[outer];
      inner = outer;
                                    // one subpass (eg 0, 4, 8)
      while(inner > h-1 && theArray[inner-h] >=  temp)
         {
         theArray[inner] = theArray[inner-h];
         inner -= h;
                }
             theArray[inner] = temp;
             }  // end for
          h = (h-1) / 3;              // decrease h
          }  // end while(h>0)
       }  // end shellSort()
```

Other interval sequence

- Original paper:
  - h = h / 2
  - Not the best approach: sometimes degenerates $O(N^2)$ running time
- Another variation:
  - h = h /2.2 (i.e., n = 100 has h = 45, 20, 9, 4, 1)
- Another possibility (Flamig):
  - h < 5 → h = 1
  - h = (5 * h - 1)/11

Efficiency of Shell sort

- Range from
  - O $(N^{3/2})$ down to $O(N^{7/6})$
  - → Better than simple sort

TABLE 7.2 Estimates of Shellsort Running Time

O() Value	Type of Sort	10 Items	100 Items	1,000 Items	10,000 Items
N^2	Insertion, etc.	100	10,000	1,000,000	100,000,000
$N^{3/2}$	Shellsort	32	1,000	32,000	1,000,000
$N*(\log N)^2$	Shellsort	10	400	9,000	160,000
$N^{5/4}$	Shellsort	18	316	5,600	100,000
$N^{7/6}$	Shellsort	14	215	3,200	46,000
$N*\log N$	Quicksort, etc.	10	200	3,000	40,000

Quick sort

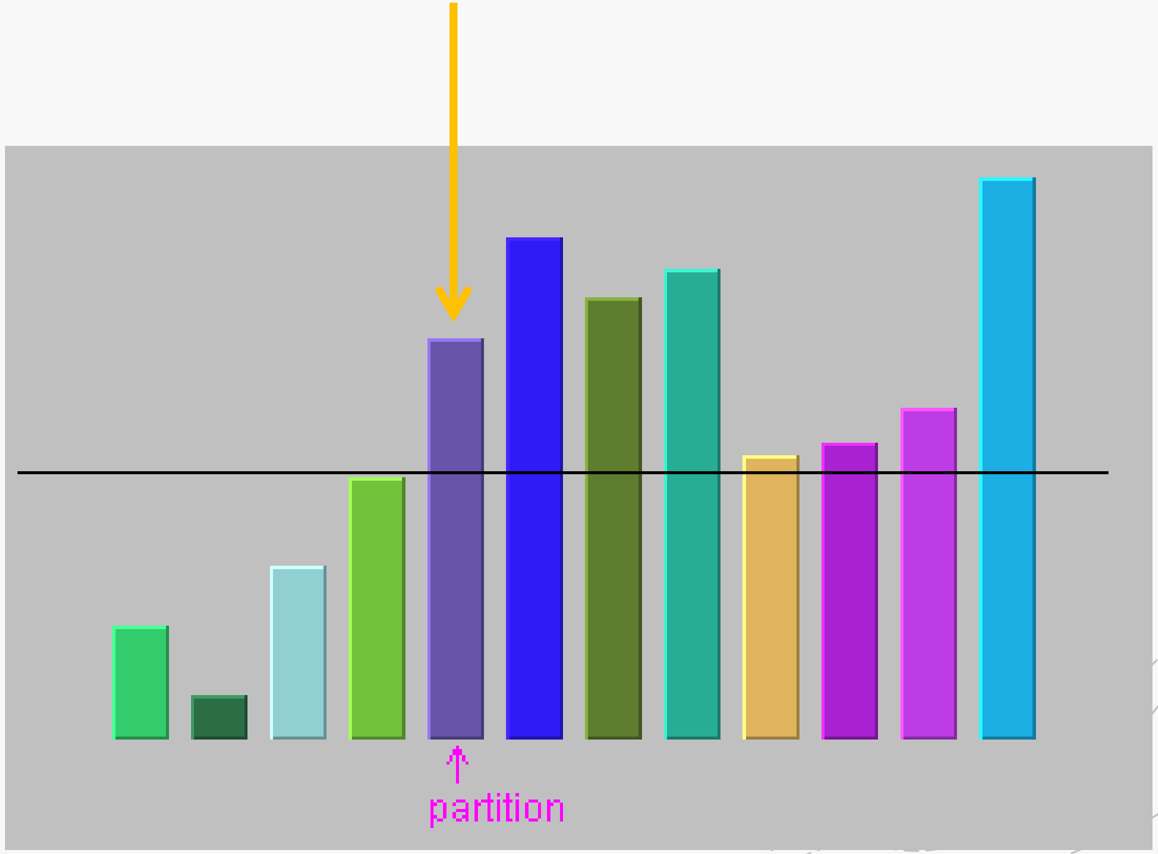
- Efficient, general-purpose sorting algorithm
- Developed by British computer scientist Tony Hoare in 1959
- Example of Divide and Conquer algorithm
- Two phases
  - **Partition phase**: Divides the work into half
  - **Sort phase**: Conquers the halves!

Partitioning – Introduction

- Is the underlying mechanism of Quick sort
- Is a useful operation
- **Partition data** : divide data into 2 groups
  - Based on a ***pivot value***
  - > pivot value
  - <= pivot value

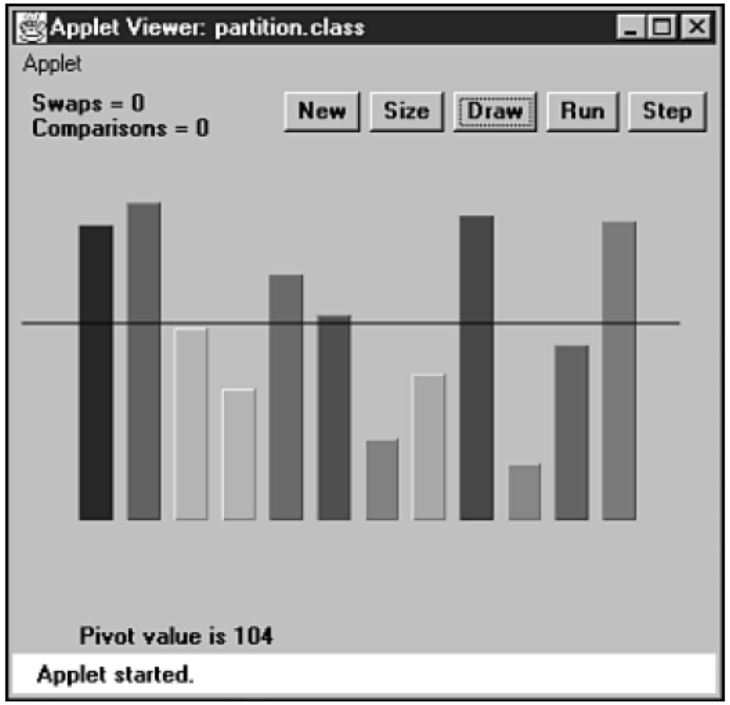
Partition

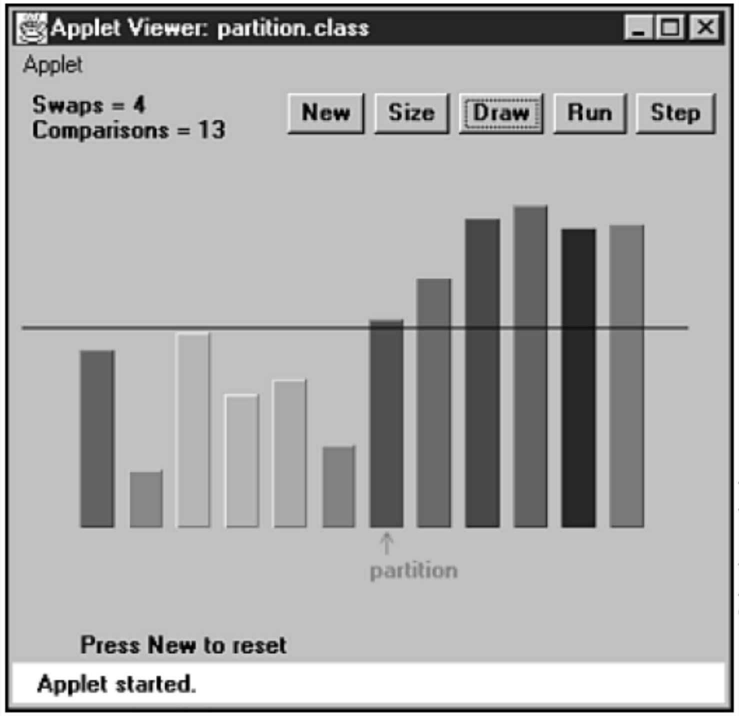
- Partition:
  - leftmost item of right sub-array
  - is returned from the partitioning method
  - Indicate where the division is



Implementation

- Find an item **(a)**
  - in the left, pointed by **leftPtr**
  - and bigger than pivot
- Find an item **(b)**
  - in the right, pointed by **rightPtr**
  - and smaller than pivot
- Swap them
- Repeat until two pointers meet





Implementation

- Input: an array with
  - Index of left-most item
  - Index of right-most item
  - Pivot value
- Output:
  - Partitioned array
  - Index of the partition element (where the division is)

- Example: partition this array
  - [5, 10, 3, 8, 6, 9, 2]
  - Pick a pivot value (i.e., 7)

https://liveexample.pearsoncmg.com/liang/animation/web/QuickSortPartition.html

Implementation – Find (a), (b)

```
public int partitionIt(int left, int right, long pivot)
   {
   int leftPtr = left - 1;            // right of first elem
   int rightPtr = right + 1;          // left of pivot
   while(true)
      {
      while(leftPtr < right &&        // find bigger item
            theArray[++leftPtr] < pivot)
         ;  // (nop)

      while(rightPtr > left &&        // find smaller item
            theArray[--rightPtr] > pivot)
         ;  // (nop)
      if(leftPtr >= rightPtr)         // if pointers cross,
         break;                       //    partition done
      else                            // not crossed, so
         swap(leftPtr, rightPtr);     //    swap elements
      }  // end while(true)
   return leftPtr;                    // return partition
```

Efficiency of Partition

- Two pointers start from two ends of array
- Move toward each other
- When they meet, partition is complete

→ **O(N)**

Quick sort

Introduction

- Most popular sorting algorithm
- Is the fastest (in most of the cases)
- On average: **O(N*logN)**

Main idea

- Partition an array into two sub-arrays
- Then call itself **recursively** to quicksort each of these sub-arrays

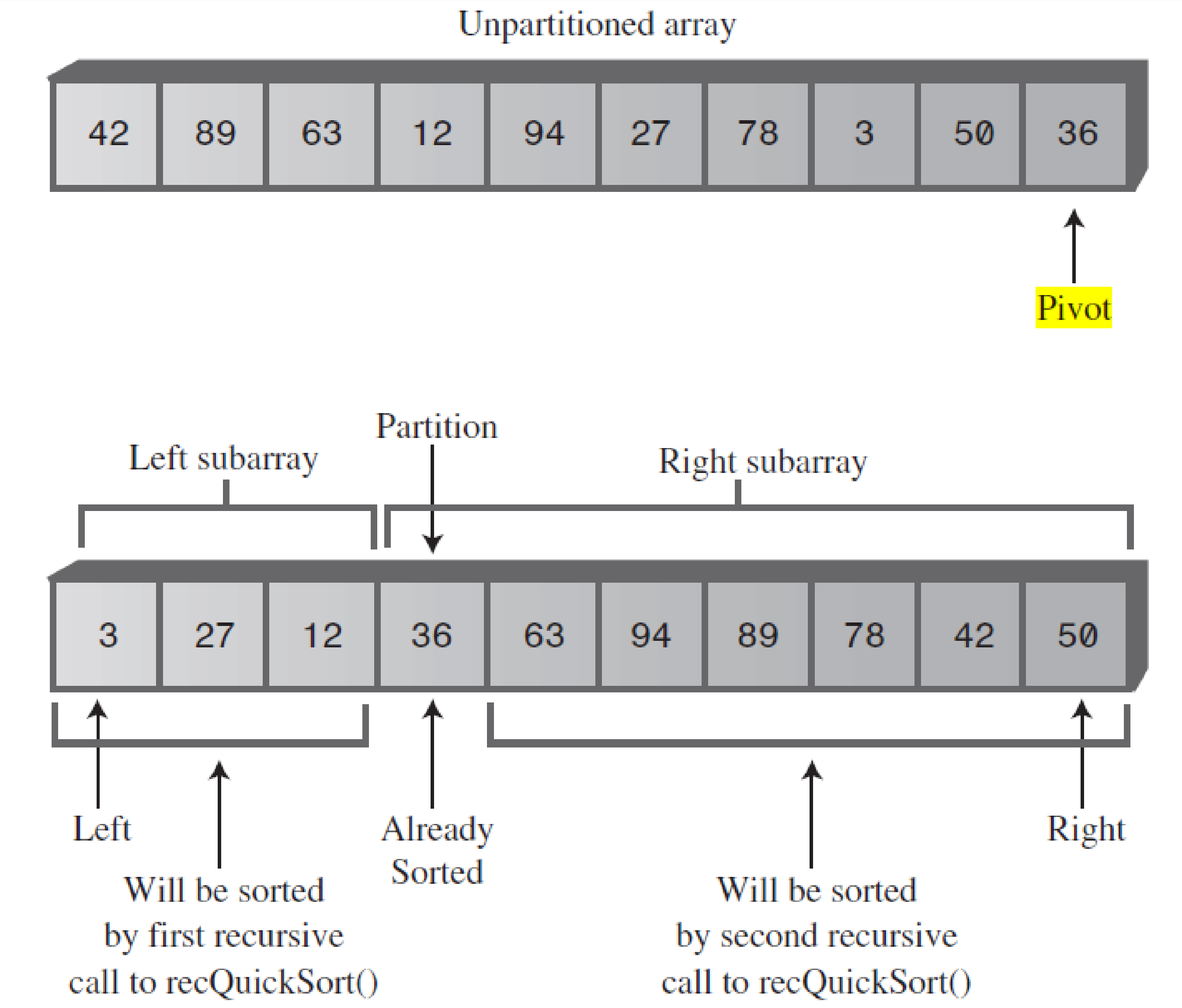
Implementation

```
public void recQuickSort(int left, int right)
   {
   if(right-left <= 0)          // if size is 1,
       return;                  //    it's already sorted
   else                         // size is 2 or larger
      {
                                           // partition range
      int partition = partitionIt(left, right);
      recQuickSort(left, partition-1);   // sort left side
      recQuickSort(partition+1, right);  // sort right side
      }
   }
```

Main idea

- Three steps:
  1. Partition the array or subarray into left (smaller keys) and right (larger keys) groups.
  2. Call ourselves to sort the left group.
  3. Call ourselves again to sort the right group.

After first partitioning



Choosing a Pivot value

- Should be the value of an actual data item
- Can pick at random place in array
  - For our algorithm: the rightmost item
- After partition,
  - Partition item is at BOUNDARY between left and right subarray
    - Swap the pivot item with partition item.
  - The pivot item will be in its FINAL position

Update the implementation

```
public void recQuickSort(int left, int right)
   {
   if(right-left <= 0)                  // if size <= 1,
       return;                          //    already sorted
   else                                 // size is 2 or larger
      {
      long pivot = theArray[right];      // rightmost item
                                         // partition range
      int partition = partitionIt(left, right, pivot);
      recQuickSort(left, partition-1);   // sort left side
      recQuickSort(partition+1, right);  // sort right side
      }
   }  // end recQuickSort()
```

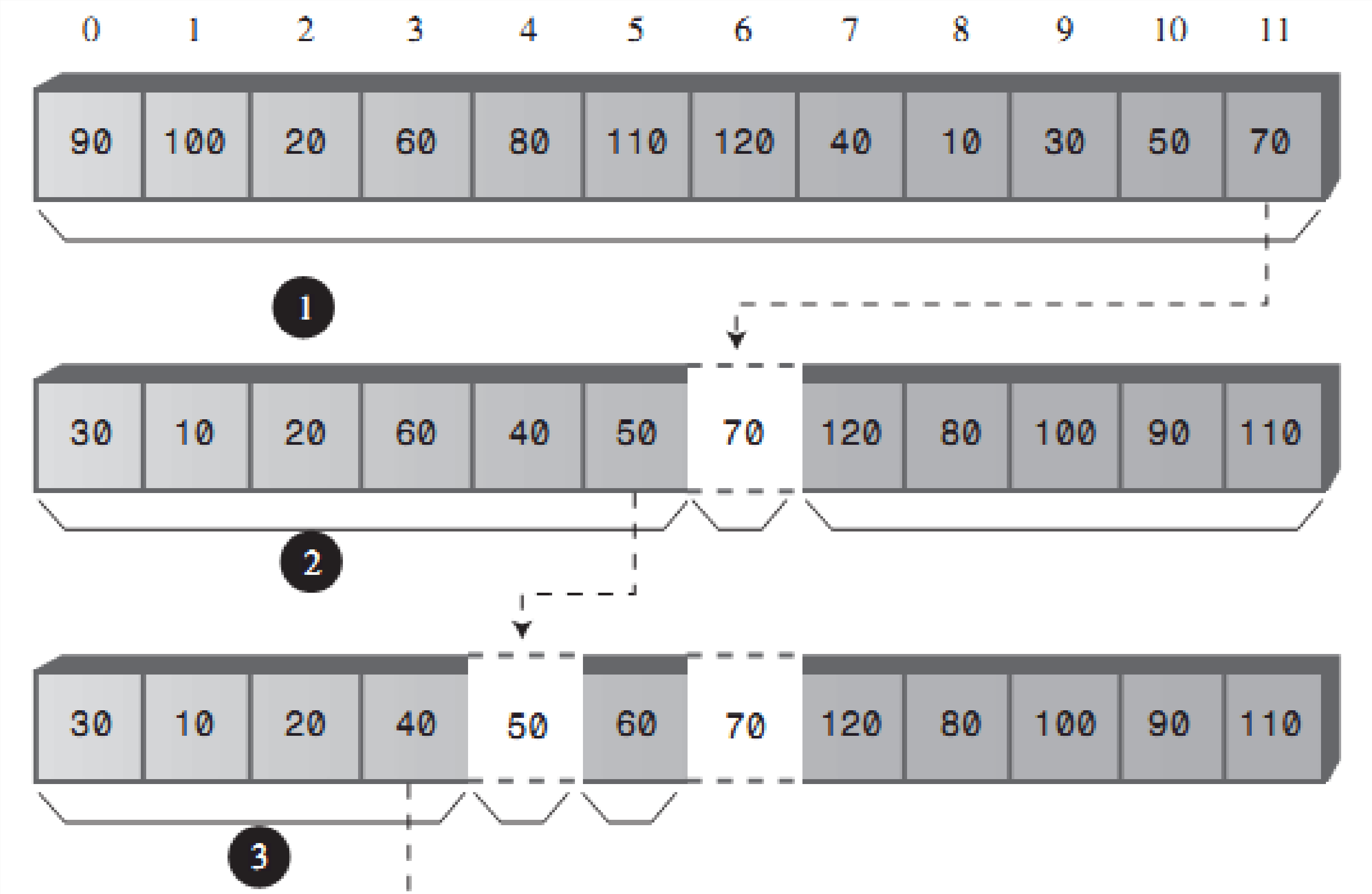
```
public int partitionIt(int left, int right, long pivot)
   {
   int leftPtr = left-1;              // left    (after ++)
   int rightPtr = right;              // right-1 (after --)
   while(true)
      {                               // find bigger item
      while( theArray[++leftPtr] < pivot )
         ;  // (nop)
                                      // find smaller item
      while(rightPtr > 0 && theArray[--rightPtr] > pivot)
         ;  // (nop)

      if(leftPtr >= rightPtr)         // if pointers cross,
         break;                       //    partition done
      else                            // not crossed, so
         swap(leftPtr, rightPtr);     //    swap elements
      }  // end while(true)
   swap(leftPtr, right);              // restore pivot
   return leftPtr;                    // return pivot location
   }  // end partitionIt()
```

The improvement

- Do not need to check for the end of array in while loop
  - ~~leftPrt < right~~

Step-by-step sort



https://opendsa-server.cs.vt.edu/embed/quicksortAV

Degenerate to $O(N^2)$

- The pivot divides the list into two sublists of size 0 and n-1

Degenerate to $O(N^2)$

- Ideally, pivot should be the MEDIAN of the items
- The worst case: after partition, we have
  - 1 element & N-1 elements
- → Increase the number of recursive call
- → Slow
- → Stack overflow
- → Need better approach for selecting pivot

Quick sort

with Median-Of-Three Partioning

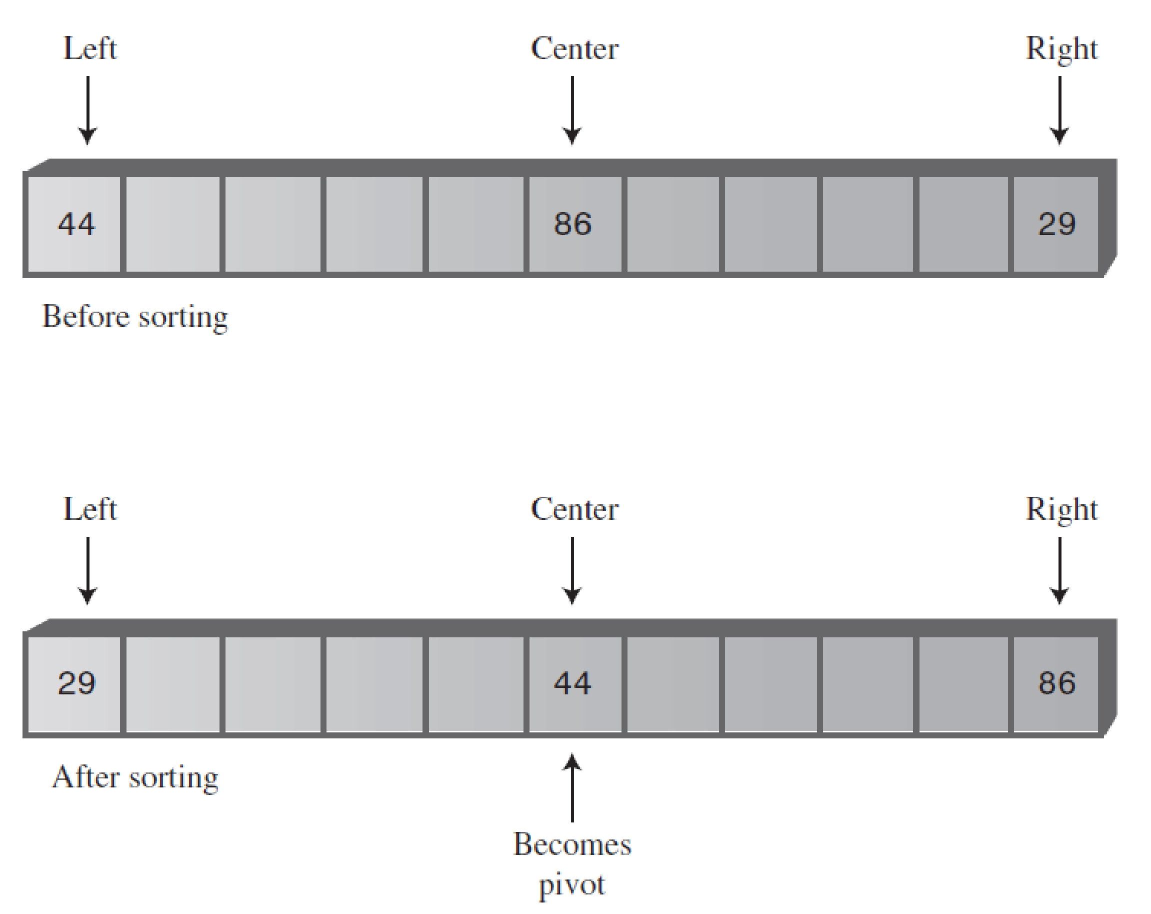
Median-Of-Three Partitioning

- Ideally, examine all items → Median
- Compromise solution:

Median of (Left, Right, Center)

- In addition, sort Left, Right and Center

Median-Of-Three Partitioning



Implementation

```
long median = medianOf3(left, right);
int partition = partitionIt(left, right, median);
recQuickSort(left, partition-1);
recQuickSort(partition+1, right);
```

MedianOf3

```
public long medianOf3(int left, int right)
   {
   int center = (left+right)/2;
                                       // order left & center
   if( theArray[left] > theArray[center] )
      swap(left, center);
                                       // order left & right
   if( theArray[left] > theArray[right] )
      swap(left, right);
                                       // order center & right
   if( theArray[center] > theArray[right] )
      swap(center, right);

   swap(center, right-1);              // put pivot on right
   return theArray[right-1];           // return median value
   }  // end medianOf3()
```

Partition (p. 349)

```
public int partitionIt(int left, int right, long pivot)
   {
   int leftPtr = left;                // right of first elem
   int rightPtr = right - 1;          // left of pivot
```

```
swap(leftPtr, right-1);               // restore pivot
```

Cutoff point

- This version can use only if array size > 3
- If not, sort manually or use insertion sort

```
int size = right-left+1;
if(size <= 3)                        // manual sort if small
   manualSort(left, right);
else                                 // quicksort if large
   {
```

Efficiency of Quick sort

- O(N * logN)
- Is a divide-and-conquer algorithm

Radix Sort

Radix sort

- Radix Sort is a clever and intuitive little sorting algorithm. Radix Sort puts the elements in order by comparing the **digits of the numbers**. We will explain with an example.

Radix Sort

- Consider the following scheme
  - Given the numbers

    16 31 99 59 27 90 10 26 21 60 18 57 17

  - If we first sort the numbers based on their last digit only, we get:

    90 10 60 31 21 16 26 27 57 17 18 99 59

  - Now sort according to the first digit:

    10 16 17 18 21 26 27 31 57 59 60 90 99

Radix Sort

- Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists.

Radix Sort

- **Thus, consider the following algorithm:**
- Suppose we are sorting decimal numbers
- Create an array of 10 queues
- For each digit, starting with the least significant
  - Place the $\boldsymbol{i^{th}}$ number into the bin corresponding with the current digit
  - Remove all digits in the order they were placed into the bins in the order of the bins

Radix Sort

- Suppose that two n-digit numbers are equal for the first **m** digits:

$$a = a_n a_{n-1} a_{n-2} \cdots a_{n-m+1} \mathbf{a_{n-m}} \cdots a_1 a_0$$

$$b = a_n a_{n-1} a_{n-2} \cdots a_{n-m+1} \mathbf{b_{n-m}} \cdots b_1 b_0$$

  where $\mathbf{a_{n-m}} < \mathbf{b_{n-m}}$

- For example, 103574 < 103892 because 1 = 1, 0 = 0, 3 = 3 but 5 < 8
- Then, on iteration $n - m$, a will be placed in a lower bin than b
- When they are taken out, a will precede b in the list

Radix Sort

- For all subsequent iterations, ***a*** and ***b*** will be placed in the same bin, and will therefore continue to be taken out in the same order
- Therefore, in the final list, ***a*** must precede ***b***

Example

- Sort the following decimal numbers:

  86 198 466 709 973 981 374 766 473 342

- First, interpret 86 as 086

Example

- Next, create an array of 10 queues:

0				
1				
2				
3				
4				
5				
6				
7				
8				
9				

Example

- Push according to the 3rd digit:

086 198 466 709 973 981 374 766 473 342

0				
1	981			
2	342			
3	973	473		
4	374			
5				
6	086	466	766	
7				
8	198			
9	709			

and dequeue: 98**1** 34**2** 97**3** 47**3** 37**4** 08**6** 46**6** 76**6** 19**8** 70**9**

Example

- Enqueue according to the 2nd digit:

981 342 973 473 374 086 466 766 198 709

0	709			
1				
2				
3				
4	342			
5				
6	466	766		
7	973	473	374	
8	981	086		
9	198			

and dequeue: 7**0**9 3**4**2 4**6**6 7**6**6 9**7**3 4**7**3 3**7**4 9**8**1 0**8**6 1**9**8

Example

- Enqueue according to the 1st digit:

709 342 466 766 973 473 374 981 086 198

0	086			
1	198			
2				
3	342	374		
4	466	473		
5				
6				
7	709	766		
8				
9	973	981		

and dequeue: **0**86 **1**98 **3**42 **3**74 **4**66 **4**73 **7**09 **7**66 **9**73 **9**81

Example

- The numbers

086 198 342 374 466 473 709 766 973 981

are now in order

- The next example uses the binary representation of numbers, which is even easier to follow

Java code

- Google is the magic!
- Some examples to read:
  - https://www.geeksforgeeks.org/radix-sort/
  - https://www.javatpoint.com/radix-sort
  - https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_radix_sort.htm
  - https://www.programiz.com/dsa/radix-sort

Complexity of Radix sort

- The time complexity of radix sort is given by the formula:
- **T(n) = O(d*(n+b))**
- d: the number of digits in the given list
- n: the number of elements in the list
- b: the base or bucket size used, which is normally base 10 for decimal representation.

Practice

- ***QuickSort1App.java***
- ***QuickSort2App.java***
- ***QuickSort3App.java***
- Add counters for the number of comparisons, swaps, and recursive calls, and display them after sorting.
- Compute the average number of comparisons, swaps, and recursive calls over 100 runs.

Practice

- ***ShellSortApp.java***
- Generate an array of 50 random elements
- Run the code to “shell sort” the array
- For each change of ***h***
  - Print out ***h*** value
  - Print out the array

Vietnam National University of HCMC
International University
School of Computer Science and Engineering

THANK YOU

Dr Vi Chi Thanh - vcthanh@hcmiu.edu.vn
https://vichithanh.github.io

SCAN ME