Advanced Sorting

TRAN THANH TUNG

Content

- Shell sort
- Partitioning
- Quick 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

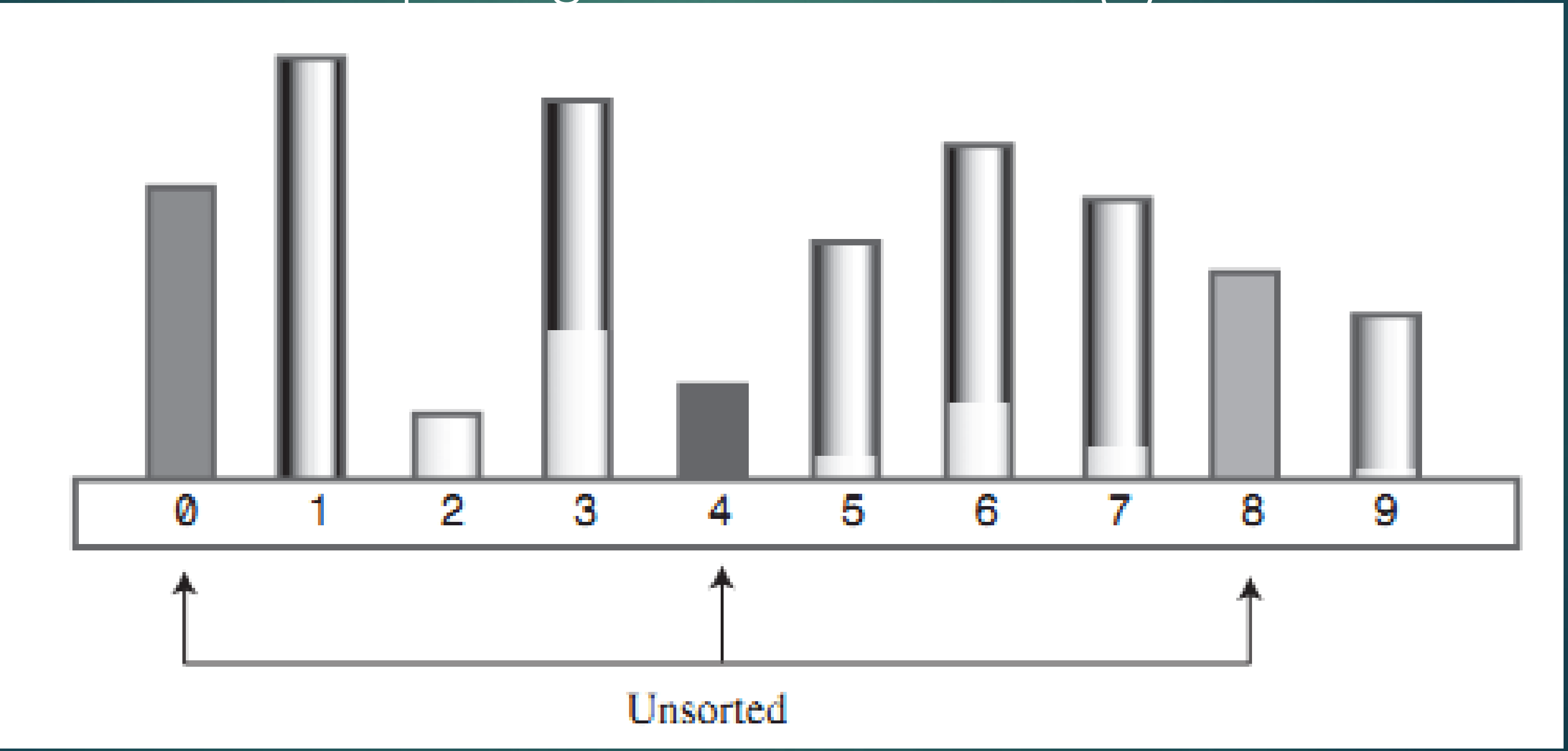
- Sort the following array

100	**34**	**51**	**61**	**73**	**0**

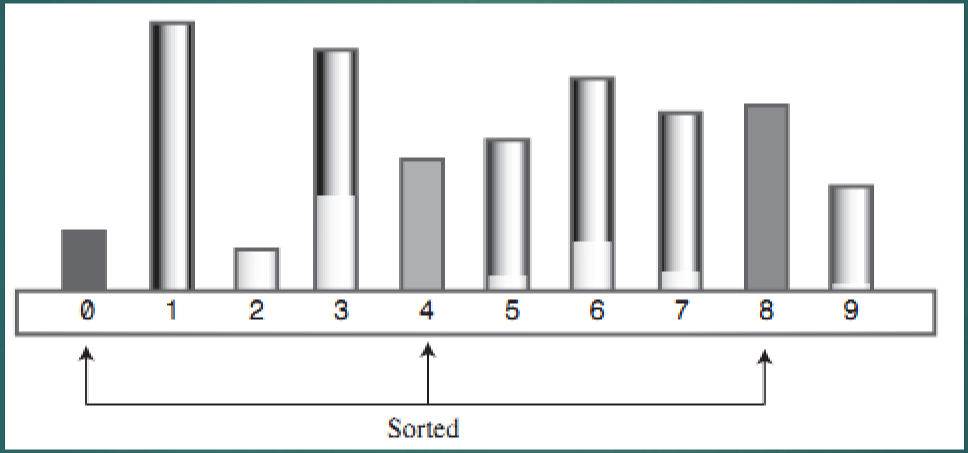
- How many copies have been made?
- → To many copies
- → can be improved

N-sorting

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

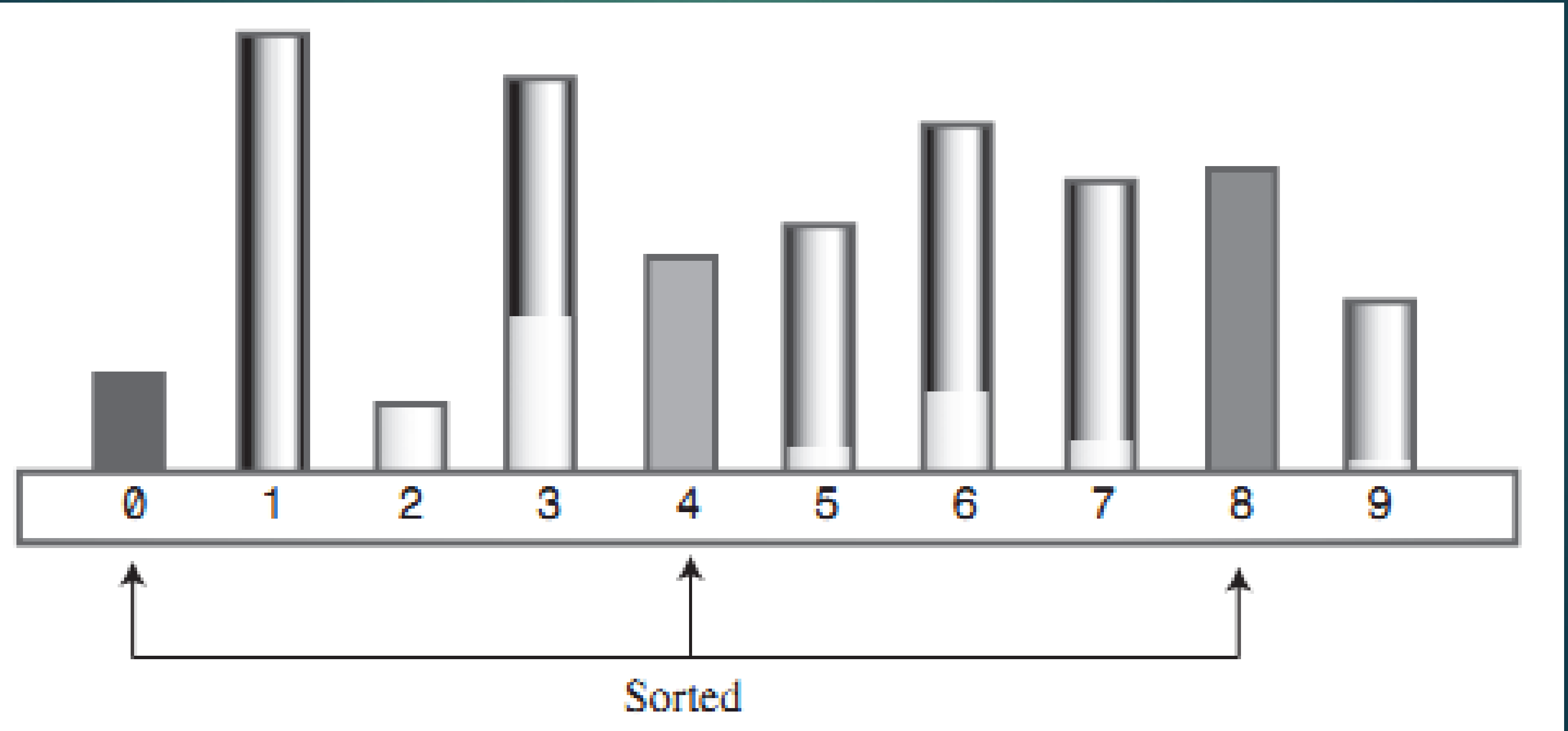


4-sorting



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 2 cells from where it should be (in our case)
- → “almost” sorted
- Continue with the 1-sorting (insertion sort)

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
- → 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

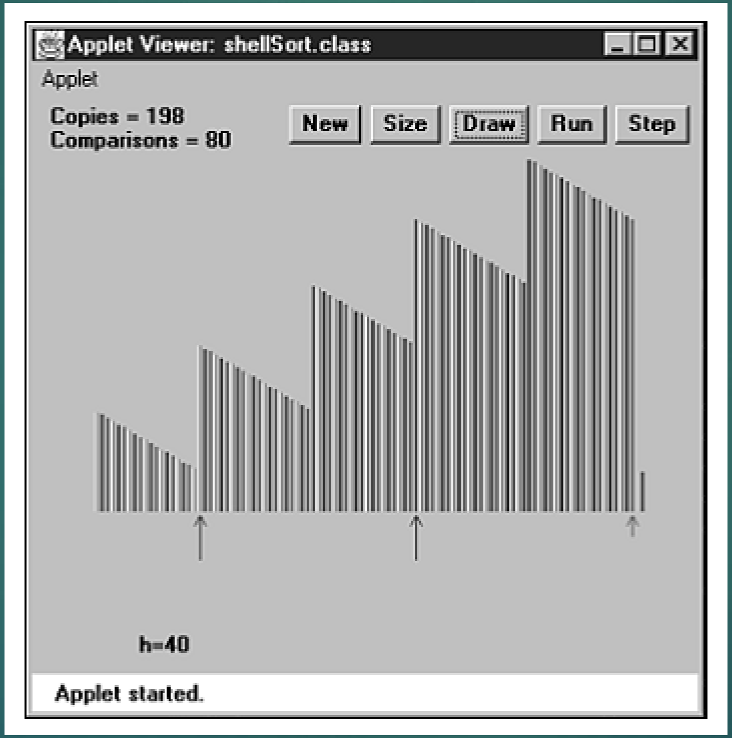
Knuth gap sequence

- What is the next gap?

$$h = (h - 1) / 3$$

- Until h = 1

Shell sort applet



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;                        // decre
   }  // end while(h>0)
```

```
for(out=1; out<nElems; out++)
   {
   long temp = a[out];            //
   in = out;
   while(in>0 && a[in-1] >= temp
      {
      a[in] = a[in-1];
```

Other interval sequence

- $h = h / 2$ (original paper)
- $h = h / 2.2$ (original paper)

- $h < 5 \rightarrow h = 1$
- $h = (5 * h - 1)/11$

Efficiency of Shell sort

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

Partitioning

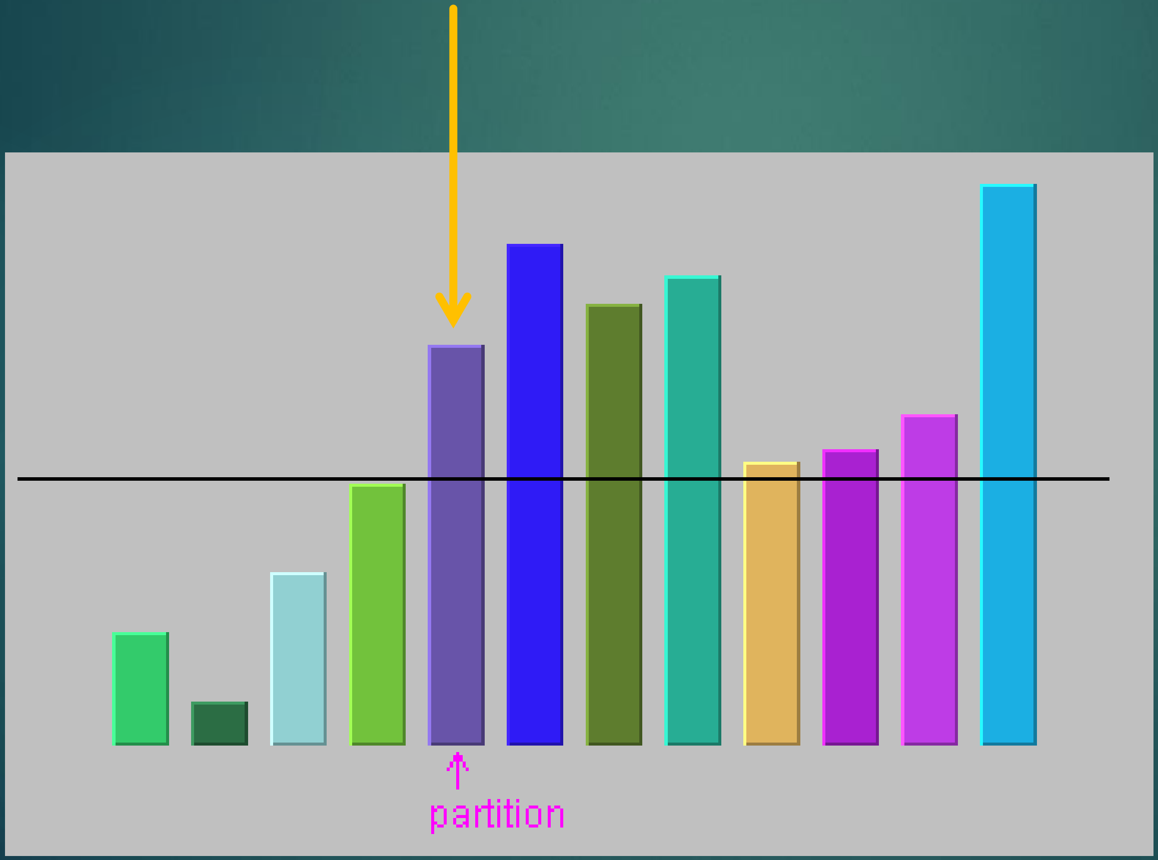
Introduction

- Is the underlying mechanism of Quick sort
- Is a useful operation

- Partition data : divide data into 2 groups
  - > pivot value
  - <= pivot value

Partition

Partition: Leftmost item of right sub-array



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 – 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)
```

Implement - Swap


```
   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)

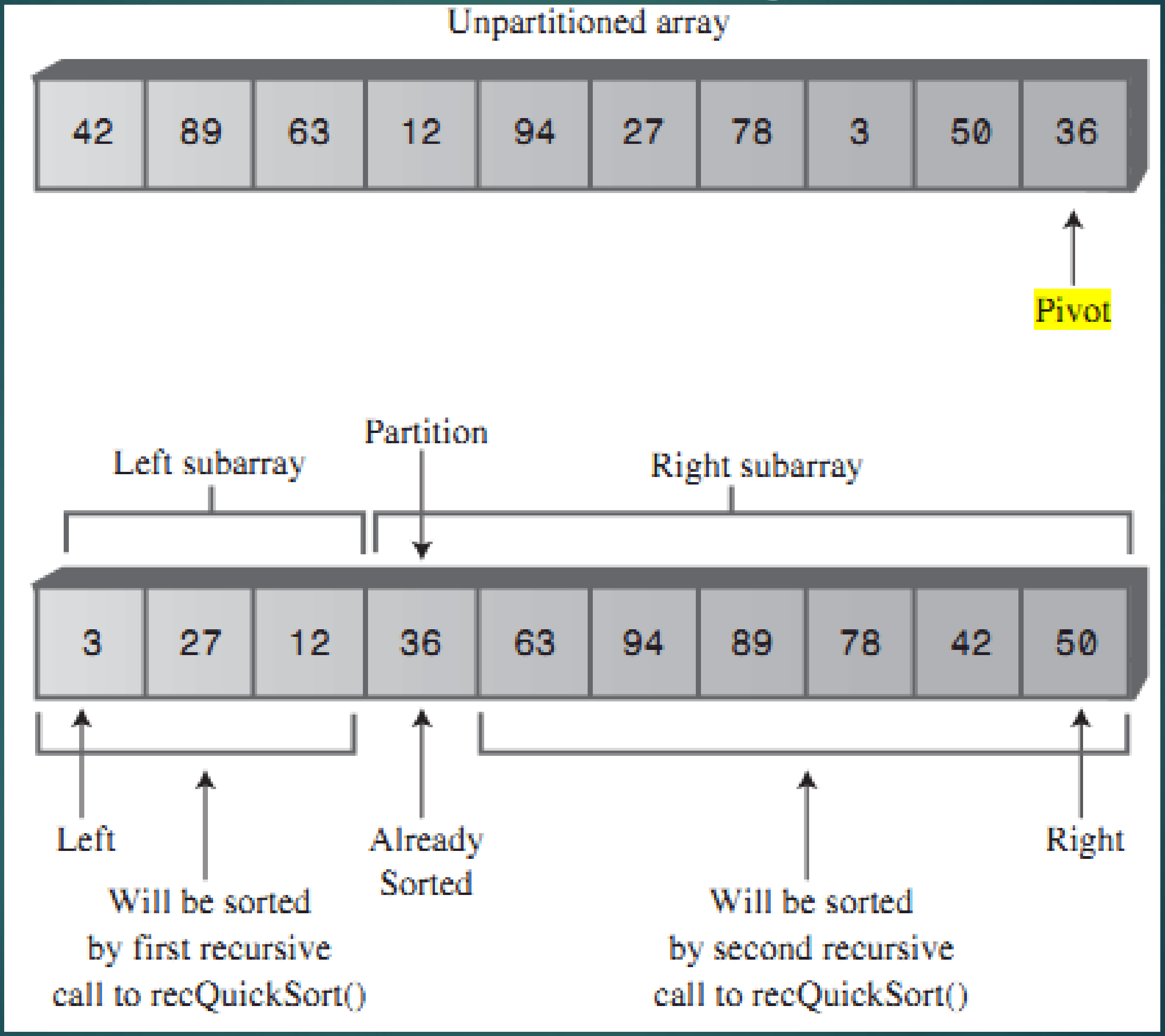
Main idea

- 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
      }
   }
```

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,
  - IF it is at BOUNDARY between left and right subarray
    - Swap the pivot item with partition item.
  - THEN the pivot item will be in its FINAL position

Update the implementation

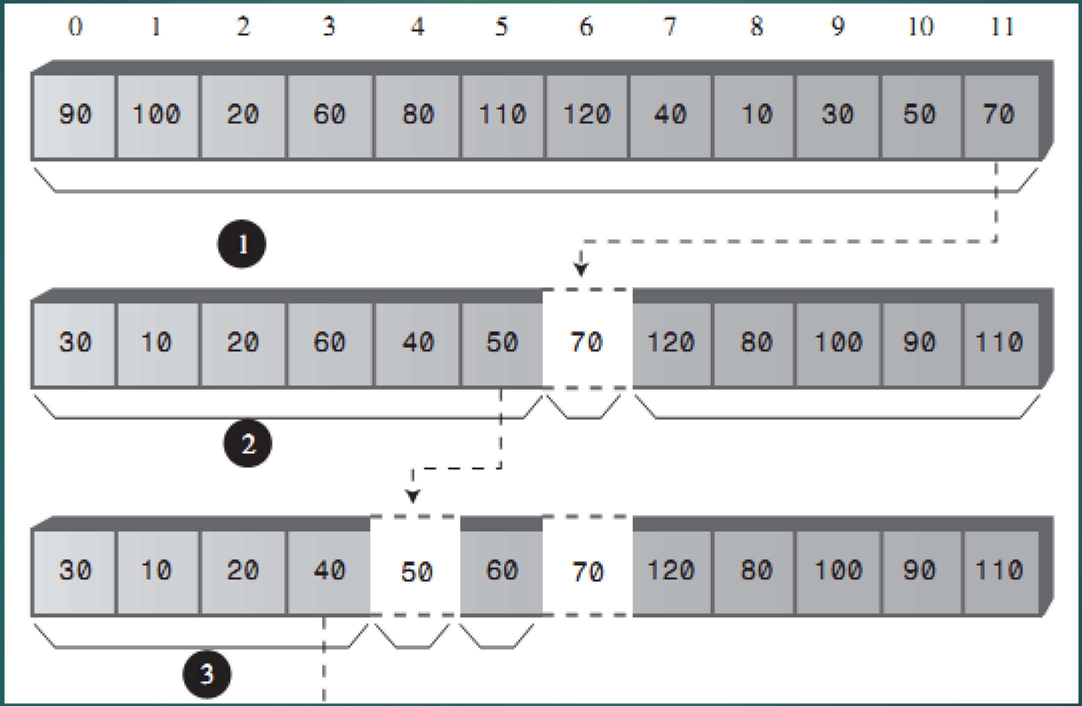

```
public void recQuickSort(int left, int right)
   {
   if(right-left <= 0)              // if size <= 1,
       return;                      //    already sorted
   else                             // size is 2 or larger
      {
      long pivot = theArray[right];      // rightmost item
         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

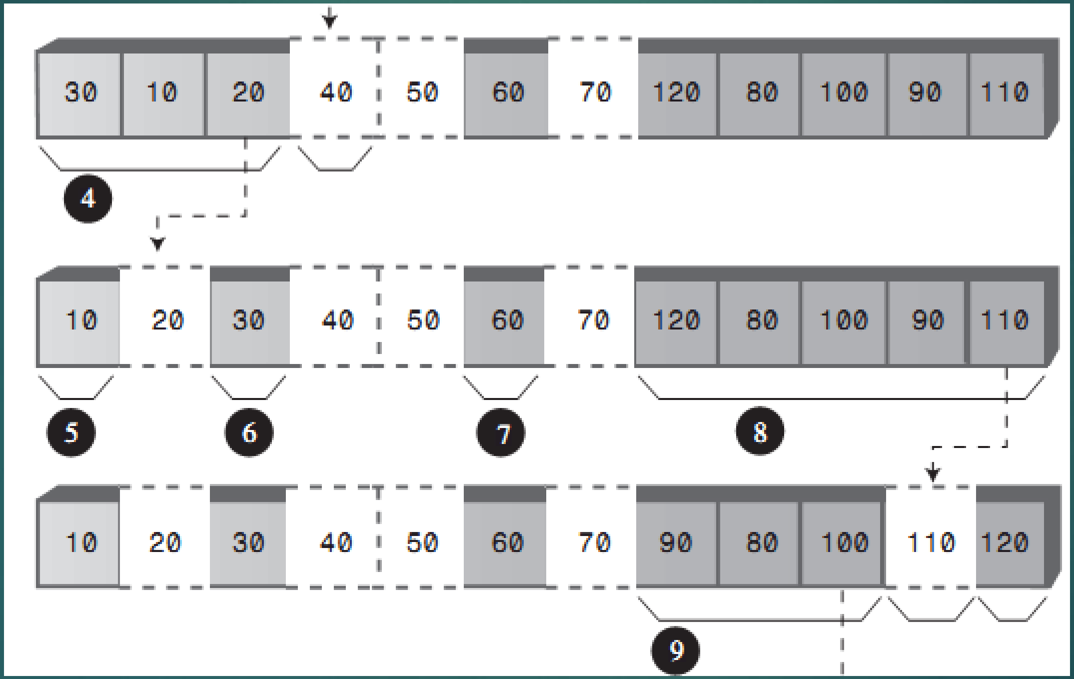
```java
public int partitionIt(int left, int right, long pivot)
   {
   int leftPtr = left-1;                 // left    (after ++)
   int rightPtr = right;                 // right-1 (after --)
   while(true)
      {                                  // find bigger item
      while( theArray[++leftPtr]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

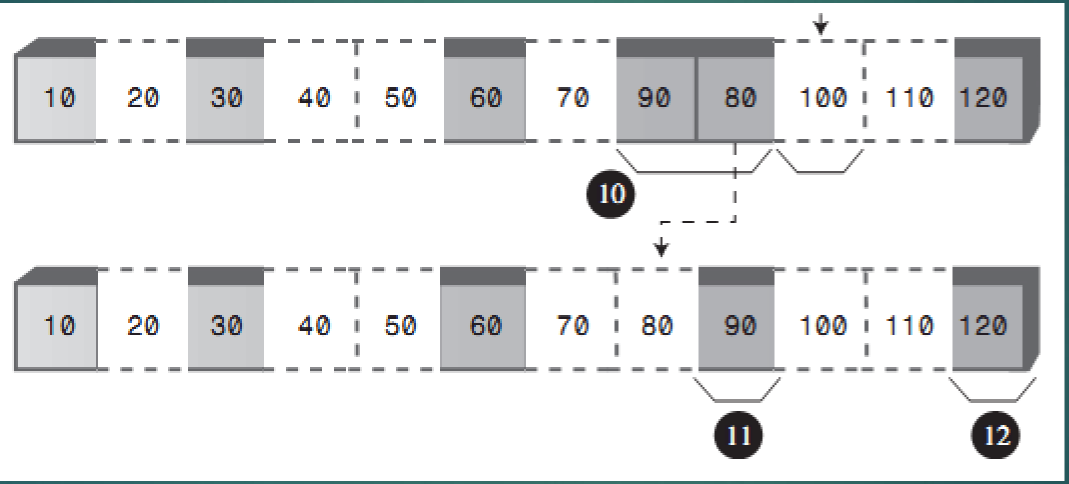
The improvement

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

Step-by-step sort




30
10
20
40
50
60
70
120
80
100
90
110
4
10
20
30
40
50
60
70
120
80
100
90
110
5
6
7
8
10
20
30
40
50
60
70
90
80
100
110
120
9


10
20
30
40
50
60
70
90
80
100
110
120
10
10
20
30
40
50
60
70
80
90
100
110
120
11
12

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

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
```

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