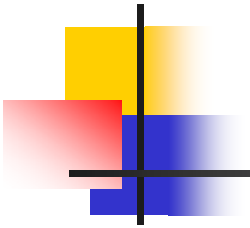
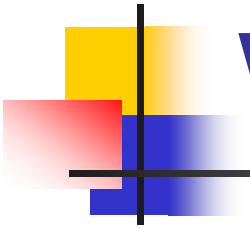


CS1102 – Lecture 8

Sorting

Chapter 10: pages 476 - 510



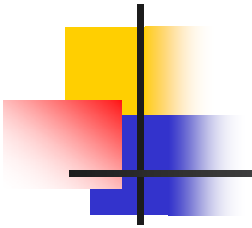
Why study sorting?

- When an input is sorted by some sort key, many problems becomes easy (eg. searching, min, max, k^{th} smallest, …).

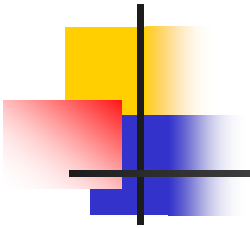
  Q: What is a sort key? Examples?

- Sorting has a variety of interesting algorithmic solutions. These solutions embody many ideas:
  - internal sort vs external sort — Q: What?
  - iterative vs recursive
  - comparison vs non-comparison based — Q: What?
  - divide-and-conquer
  - best/worst/average case bounds

Sorting applications



- uniqueness testing
- deleting duplicates
- frequency counting
- set intersection/union/difference
- efficient searching
- dictionary
- telephone directory
- street directory
- index of book
- author index of conference proceedings
- etc.



Outline

- Iterative sort algorithms (comparison based)
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
- Recursive sort algorithms (comparison based)
  - Merge Sort
  - Quick Sort
- Radix sort (non-comparison based)
- In-place sort
- stable sort
- Comparison of sort algorithms

Note: we only consider sorting data in ascending order

Web Resources

Sorting algorithm animation

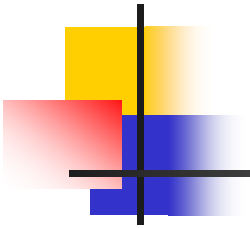
(http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html)

(http://max.cs.kzoo.edu/~abrady/java/sorting/)

(http://en.wikipedia.org/wiki/Sort_algorithm)

(http://www.sorting-algorithms.com/)

(http://search.msn.com/results.aspx?q=sort+algorithm&FORM=SMCRT)



Algo #1: **Selection Sort**

Given an array of *n* items:

(1) Find the largest item.

(2) Swap it with the item at the end of the array.

(3) Go to step 1 by excluding the largest item from the array.

Example:

Selection sort of 5 integers

29	10	14	**37**	**13**
29	10	14	**13**	37
13	10	**14**	29	37
13	**10**	14	29	37
10	13	14	29	37

37 is the largest, swap it with the last one, i.e. **13.**
Q: How to find the largest?

sorted

Code of Selection sort

```
public static void selectionSort (int[] a) {
    for (int i=a.length-1; i>=1; i--) {
        int index = i; // i is the last item position and
                       // index is the largest element position
        // loop to get the largest element
        for (int j=0; j<i; j++) {
            if (a[j] > a[index])
                index = j; // j is the current largest item
        }
        int temp = a[index]; // Swap the largest item a[index]
                             // with the last item a[i]
        a[index] = a[i];
        a[i] = temp;
    }
}
```

Analysis of Selection sort

```
public static void selectionSort(int[] a)
{
  int n = a.length;
  for (int i=n-1; i>=1; i--) {
    int index = i;
    for (int j=0; j<i; j++) {
      if (a[j] > a[index])
        index = j;
    }
    SWAP( ... )
  }
}
```

Number of times the statement. is executed:

- n-1
- n-1
- (n-1)+(n-2)+…+1 = n(n-1)/2
- n-1

Total = $t_1(n-1) + t_2*n*(n-1)/2 = O(n^2)$

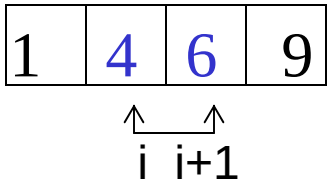
t_1 and t_2 = costs of statements in outer and inner block.

Algo #2: **Bubble Sort**

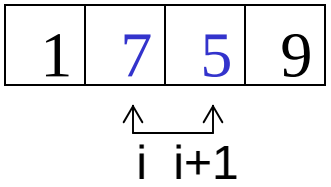
Idea:

- "bubble" down the largest item to the end of the list in each iteration by

  Examining items i and i+1 to see whether they need to be swapped.



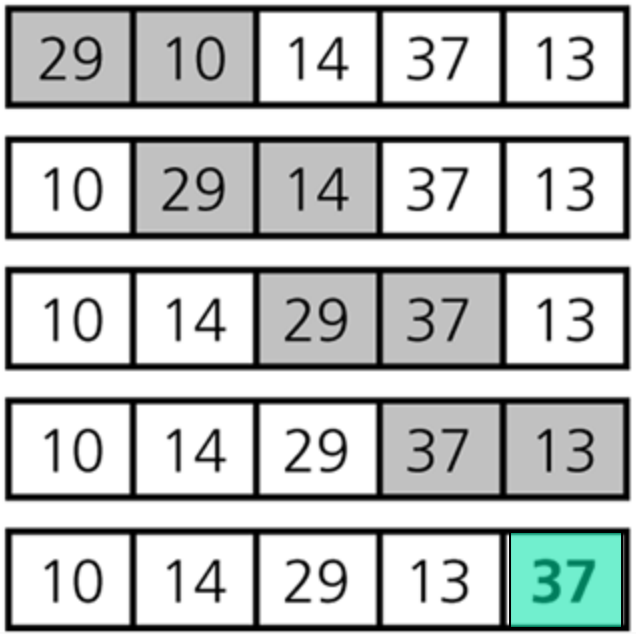
// no need to swap



// out of order, need to swap

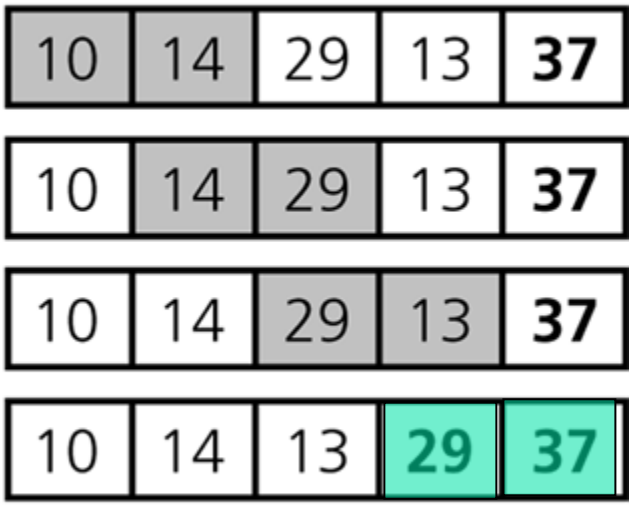
Example: The first two passes of a bubble sort of an array of five integers

(a) Pass 1



At the end of the **pass 1**, the largest item **37** is at the end (bottom).

(b) Pass 2



At the end of the **pass 2**, the second largest item **29** is at the second last.

Code of Bubble Sort

```
public static void bubbleSort (int[] a)
{
    for (int i = 1; i < a.length; i++) {
      for (int j = 0; j < a.length-i; j++) {
          if (a[j] > a[j+1]) {  // the larger item bubbles down (swap)
              int temp = a[j];
              a[j] = a[j+1];
              a[j+1] = temp;
          }
      }
}
```

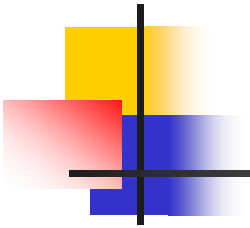
animation

Analysis of Bubble Sort

- 1 iteration of the inner loop (test and swap) requires time bounded by a constant ct
- Two nested loops.
  - outer loop: exactly n-1 iterations
  - inner loops:
    - when i=1, (n-1) iterations
    - when i=2, (n-2) iterations
    - ...
    - when i=(n-1), 1 iterations

```
for (int i = 1; i < a.length; i++) {
    for (int j = 0; j < a.length-i; j++) {
        if (a[j] > a[j+1]) { // (swap)
            int temp = a[j];
            a[j] = a[j+1];
            a[j+1] = temp;
    }
}
```

- Total number of iterations = (n-1)+(n-2)+...+1
  = n(n-1)/2
- Total time is = ct*n(n-1)/2 = $O(n^2)$



Bubble sort is inefficient

Given a sorted input, bubble sort will still take $O(n^2)$ to sort.

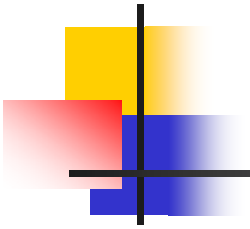
It does not make an effort to check whether the input has been sorted.

Thus it can be improved by using a **flag** (is_sorted) as follow:

Code of Bubble Sort (Improved version)

```
public static void bubbleSort2 (int[] a) {
   for (int i = 1; i < a.length; i++) {
       boolean is_sorted = true; // is_sorted = true if a[] is sorted

       for (int j = 0; j < a.length-i; j++) {
          if (a[j] > a[j+1]) {  // the larger item bubbles up
             int temp = a[j];   // and is_sorted is set to false,
             a[j] = a[j+1];     // i.e. the data was not sorted
             a[j+1] = temp;
             is_sorted = false;
          }
       }
       if (is_sorted) return;   // Q: Why?
   }
}
```

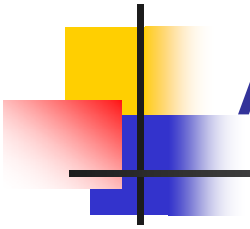
Q: Can it be further improved?



Analysis of Bubble Sort (Improved version)

- Worst-case
  - input is in descending order

    Q: How many outer iterations are needed?
  - running-time remains the same: $O(n^2)$
- Best-case
  - input is already in ascending order
  - the algorithm returns after a single outer-iteration. Why?
  - Running time: $O(n)$



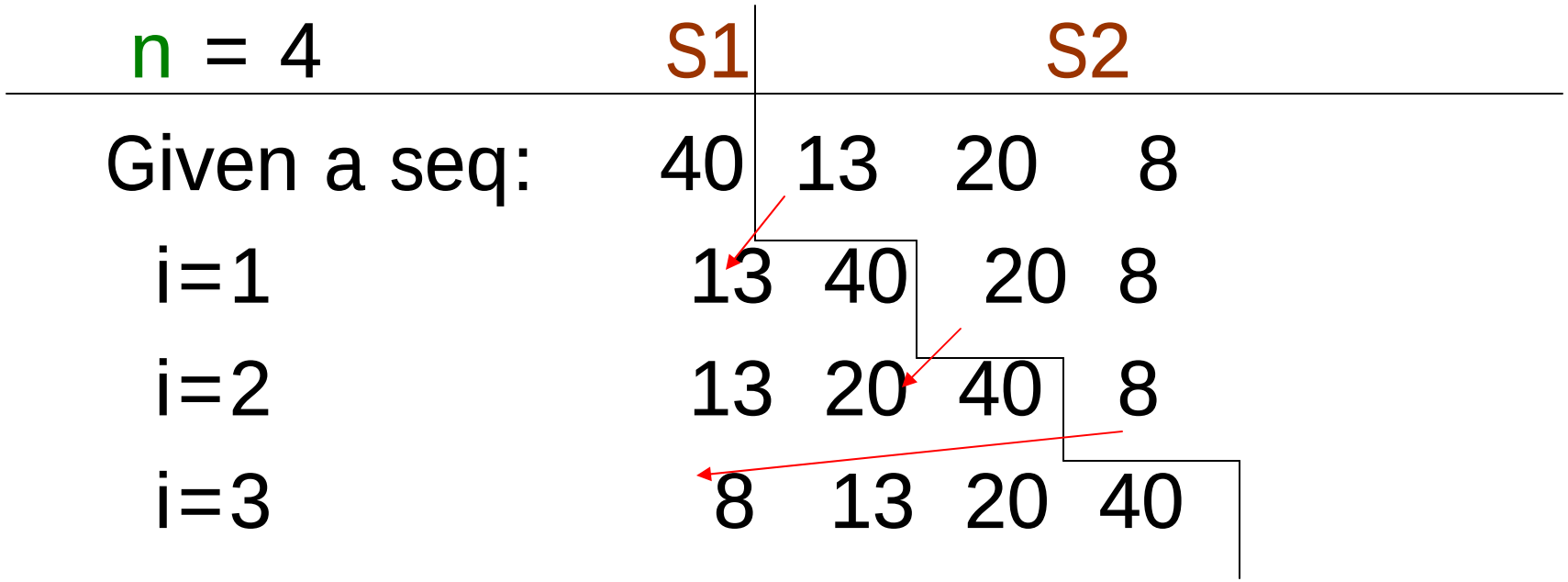
Algo #3: **Insertion Sort**

Idea:

Arranging a hand of poker cards

- Start with one card in your hand
- Pick the next card and insert it into its proper sorted order.
- Repeat previous step for all the rest of the cards

Example of Insertion Sort



- n = no of items to be sorted
- S1 = sub-array sorted so far
- S2 = Elements yet to be processed.

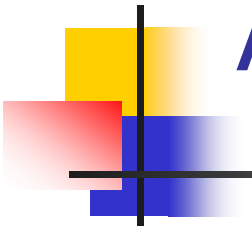
Code of Insertion Sort

```
public static void insertionSort(int[] a) {
    for (int i=1;i<n;i++) { // Q: Why does i start from 1?
        // a[i] is the next data to insert
        int next = a[i];
        // Scan backwards to find a place. Q: Why not scan forwards?
        int j;
                    // Q: Why is j declared here?
        for (j=i-1; j>=0 && a[j]>next; j--)
            a[j+1] = a[j];
        // Now insert the value next after index j at the end of loop
        a[j+1] = next;
    }
}
```

Q: Can we replace the second and third “next” to a[i]?

Analysis of Insertion Sort

- Outer-loop executes exactly n-1 times.
- Number of times inner-loop executed depends on the input:
  - Best-case: the array is already sorted and (a[j] > next) is always false.
    - No shifting of data is necessary.
  - Worst-case: the array is reversely sorted and (a[j] > next) is always true.
    - Need i shifts for i=1 to n-1
    - insertion always occur at the front.
- Therefore, the best-case time is O(n). Q: Why?
- And the worst-case time is $O(n^2)$. Q: Why?



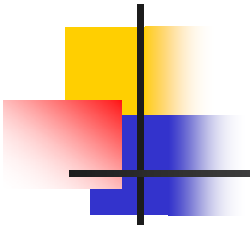
Algo #4: **Merge Sort**

Suppose we only know how to merge two sorted sets of elements into one.

Given an unsorted set of n elements

Since each element is a sorted set, we can repeatedly

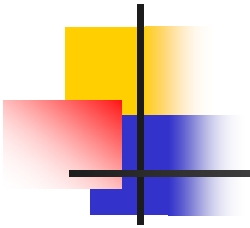
- merge each pair of elements into sets of 2.
- merge each pair of sets of 2 into sets of 4.
- …
- The final step merges 2 sets of n/2 elements to obtain a sorted set.



Divide-and-Conquer

Divide-and-conquer method solves problem by three steps:

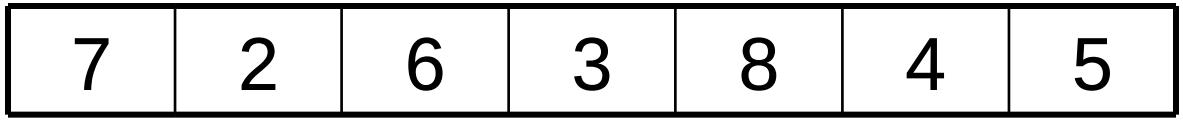
- Divide Step: divide the large problem into smaller problems.
- (Recursively) solve the smaller problems
- Conquer Step: combine the results of the smaller problems to produce the result of the larger problem.



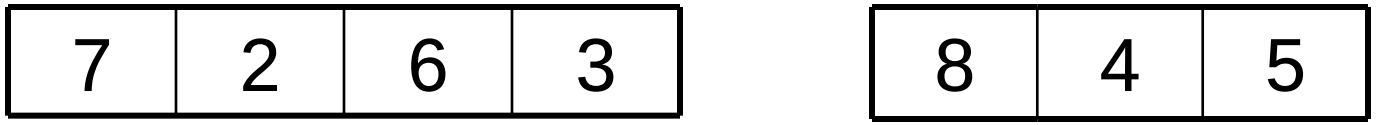
Merge sort Idea

- Mergesort is a divide-and-conquer sorting algorithm
- Divide Step: Divide the array into two (equal) halves
- Recursively sort the two halves
- Conquer Step: Merge the two sorted halves to form a sorted array

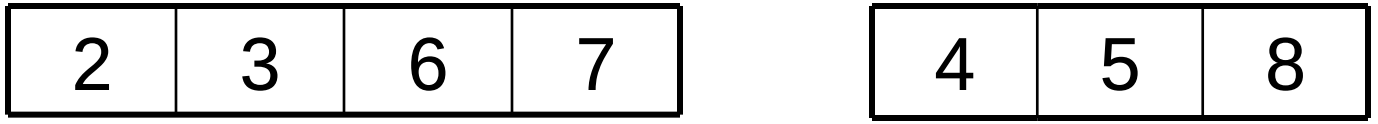
Example of Merge sort



Divide into two halves



Recursively sort the halves



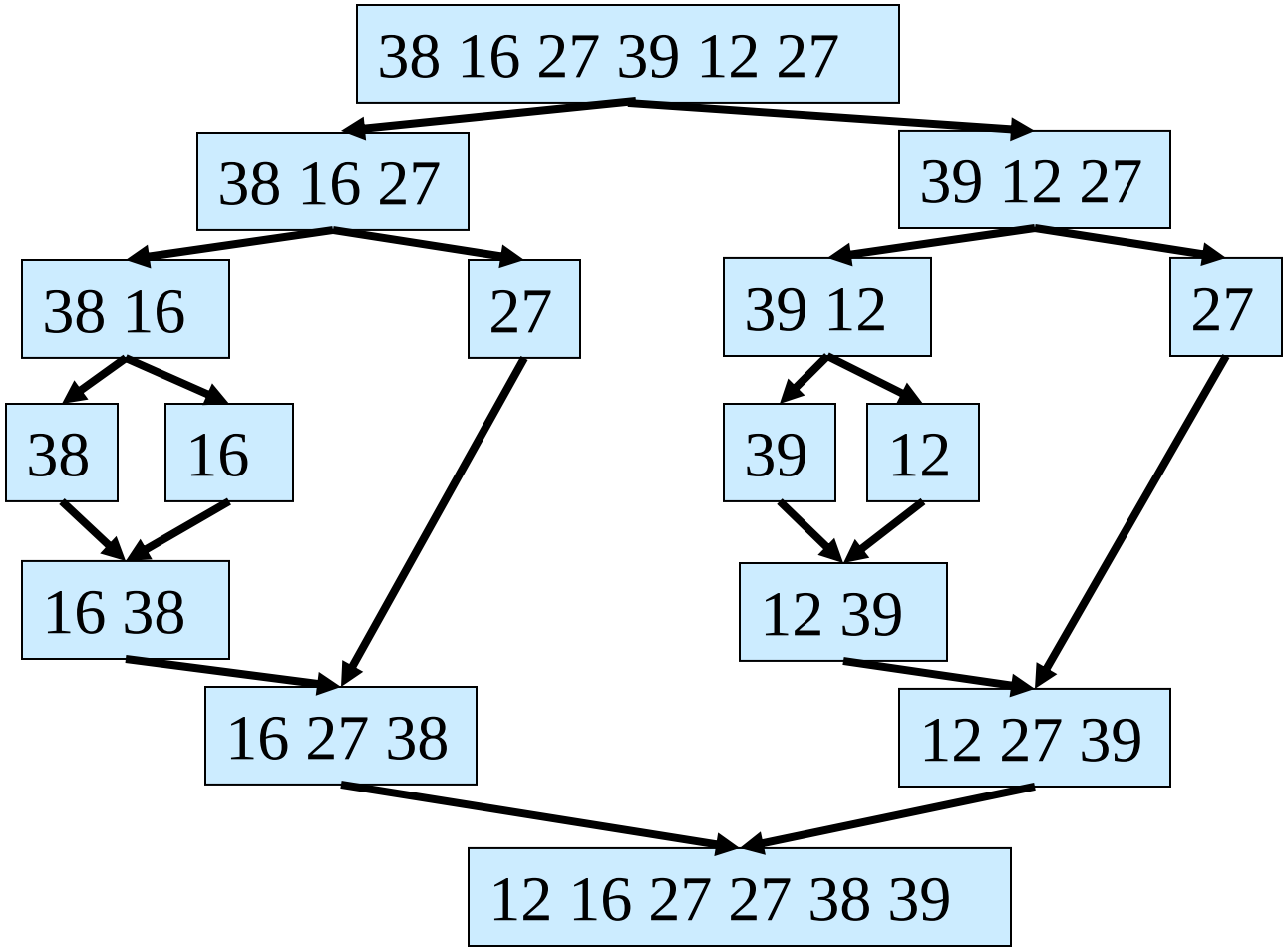
Merge them

Code of Merge sort

```
public static void
        mergeSort(int[] a, int i, int j){
                // to sort data from a[i] to a[j], where i<j
    if (i < j) {  // Q: What if i >= j?
        int mid = (i+j)/2;      // divide
        mergeSort(a,i,mid);     // recursion
        mergeSort(a,mid+1,j);  //
        merge(a,i,mid,j); //conquer – merge a[i..mid]
                          // and a[mid+1..j] back into a[i..j]
    }
}
```

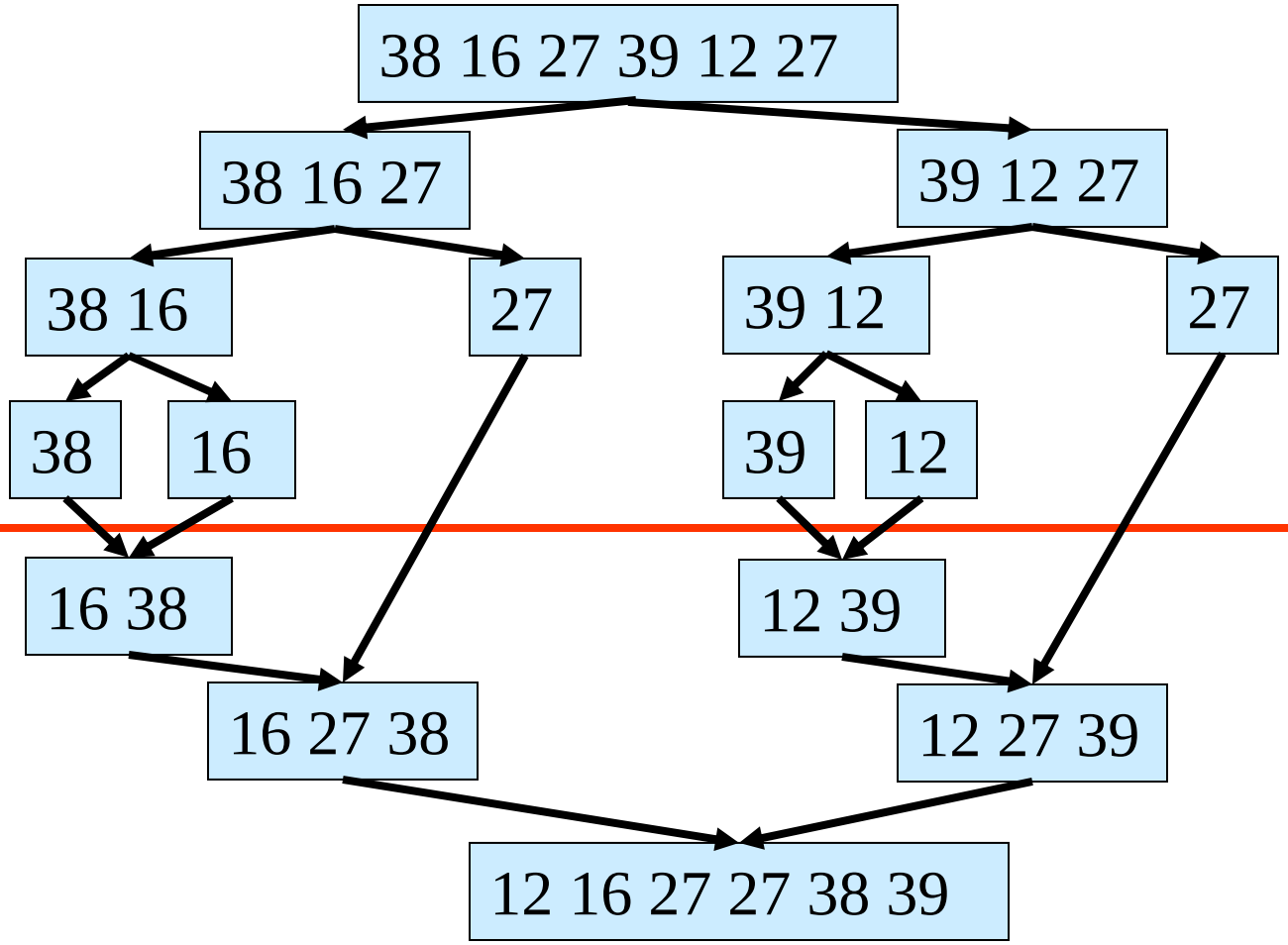
Mergesort of an array of six integers

```
mergeSort(a,i,mid);
mergeSort(a,mid+1,j);
merge(a,i,mid,j);
```



Mergesort of an array of six integers

```
mergeSort(a,i,mid);
mergeSort(a,mid+1,j);
merge(a,i,mid,j);
```



Divide phase

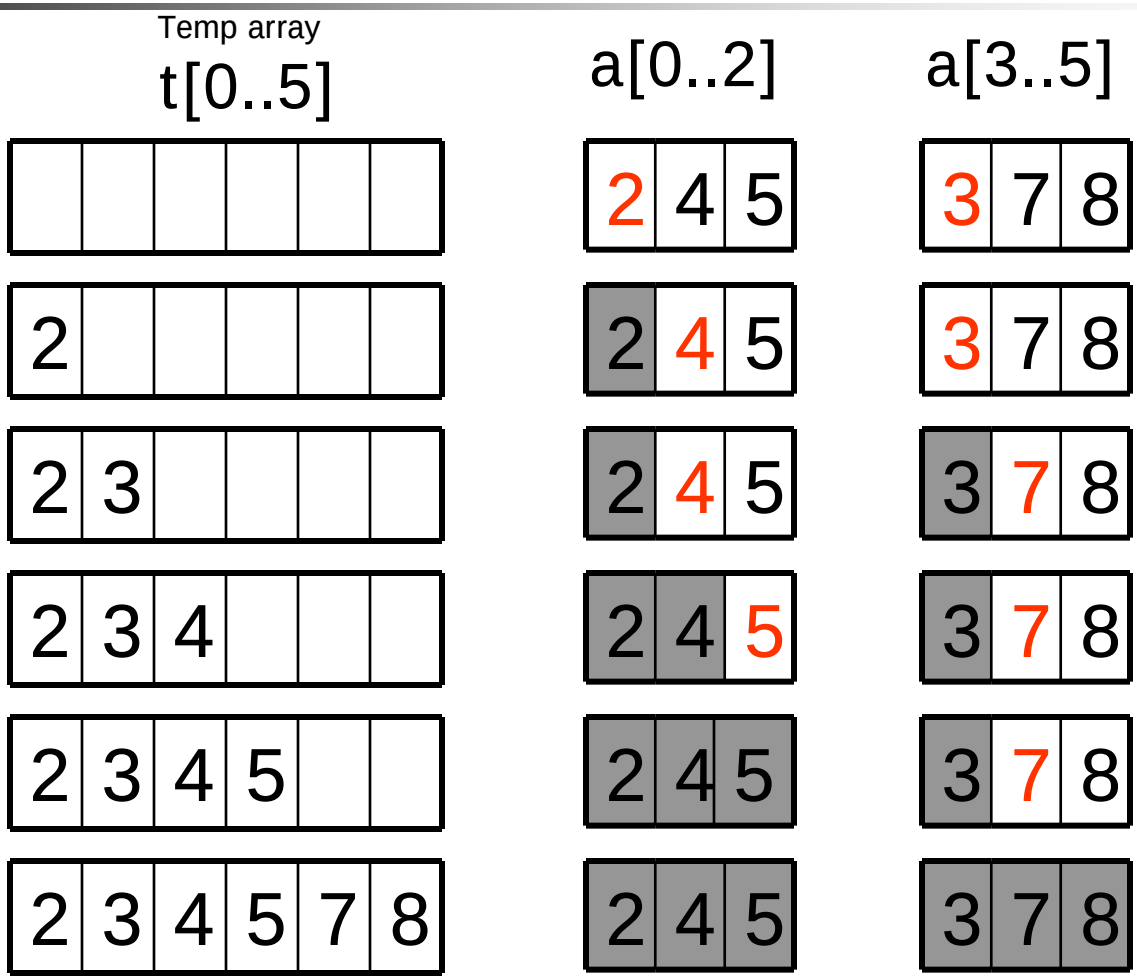
Recursive call to mergesort

Conquer phase

Merge steps

The sorting is done here

How to merge two sorted subarrays?



Merge Algorithm

```
public static void merge
     (int[] a, int i, int mid, int j) {
       // Merges a[i..mid] a[mid+1..j] into a[i..j]
    int[] t = new int[j-i+1];  // t[] temp storage
    int left=i, right=mid+1, it=0;
                  // it = next index to store merged item in t[]
                  // Q: What are left and right?

    while (left<=mid && right<=j) {// output the smaller
        if (a[left] <= a[right])
            t[it++] = a[left++];
        else
            t[it++] = a[right++];
}
```

Merge Algorithm (cont.)

```
    // Copy the remaining elements into t. Q: Why?
    while (left<=mid) t[it++] = a[left++];
    while (right<=j)  t[it++] = a[right++];
      // Q: Will both the above while statements be executed?

      // Copy the result in t back into array a
    for (int k=0;k<t.length;k++)
      a[i+k] = t[k];
}
```

Time analysis for Merge sort

In Merge sort, the bulk of work is done in the merge step

Merge(a, i, mid, j)

Total no. of items = k = (j-i+1)

- Number of comparisons ≤ k-1 **(Q: Why not = k-1?)**
- Number of moves from original array to temporary array = k
- Number of moves from temporary array to original array = k

In total, no. of operations ≤ 3k-1 = O(k)

How many times merge() is called?

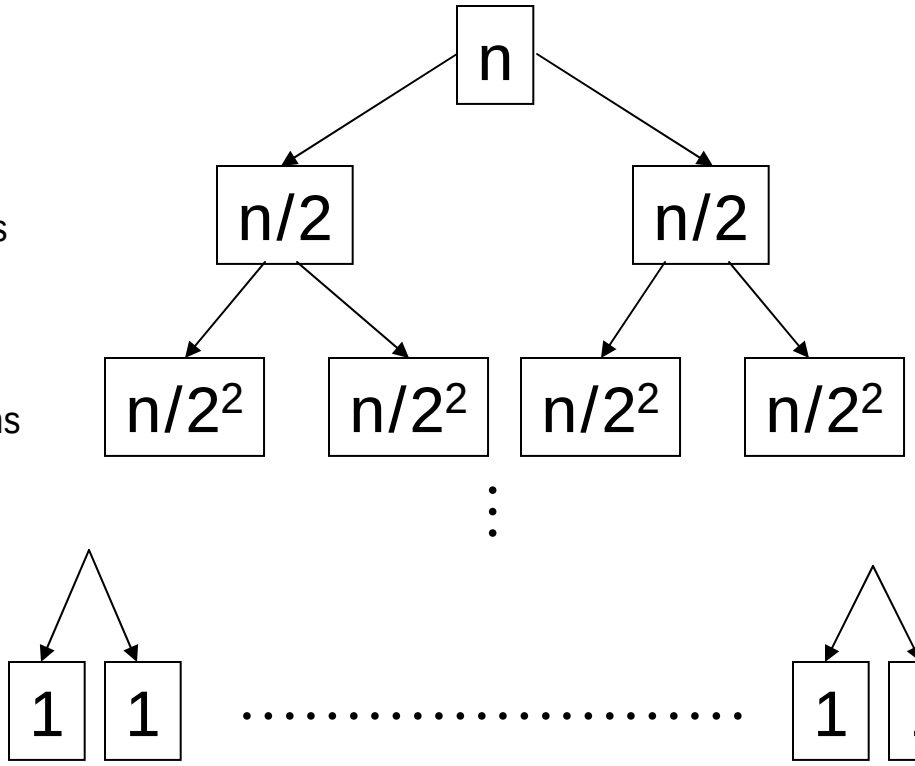
Time analysis for Merge sort

Level 0:
Mergesort n items

Level 1:
2 calls to Mergesort n/2 items

Level 2:
4 calls to Mergesort $n/2^2$ items

Level (log n):
n calls to Mergesort 1 item



Level 0:
0 call to Merge

Level 1:
1 calls to Merge

Level 2:
2 calls to Merge

Level (log n):
$2^{(\log n) - 1} (= n/2)$
calls to Merge

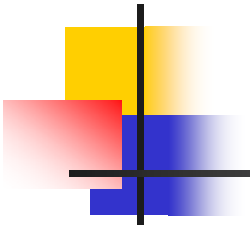
Let k be the maximum level, ie. mergesort 1 item.

$n/(2^k) = 1 \quad => \quad n = 2^k \quad => \quad k = \log n$

Time analysis for Merge sort

- Level 0: 0 call to Merge
- Level 1: 1 calls to Merge with $n/2$ items each, $O(1x\ 2x\ n/2) = O(n)$ time
- Level 2: 2 calls to Merge with $n/2^2$ items each, $O(2x\ 2x\ n/2^2) = O(n)$ time
- Level 3: 2^2 calls to Merge with $n/2^3$ items each, $O(2^2\ x\ 2x\ n/2^3) = O(n)$ time
- …
- Level $(\log n)$: $2^{(\log n)-1} (= n/2)$ calls to Merge with $n/2^{\log n} (= 1)$ item each,

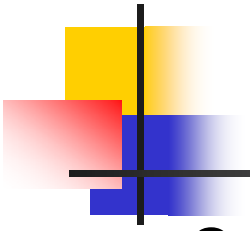
  $O(n/2\ x\ 2\ x\ 1) = O(n)$ time
- In total, running time = $(\log n)*O(n) = O(n \log n)$



Drawbacks of Merge sort

1. Implementation of merge() is not straightforward
2. Requires additional temporary arrays and to copy the merged sets stored in the temporary arrays to the original array.

Algo #5: **Quick Sort**



Quick sort is a divide-and-conquer algorithm.

- Divide Step: Choose a **pivot** item **p** and partition the items of a[i..j] into two parts
  - the items in the first part are smaller than **p** while
  - those in the second part are greater than or equal to **p**.
- Recursively sort the two parts
- Conquer Step: Do nothing! No merging is needed

Note: mergesort spends most of the time in conquer step but very little time in divide step.

Q: How about Quick Sort?

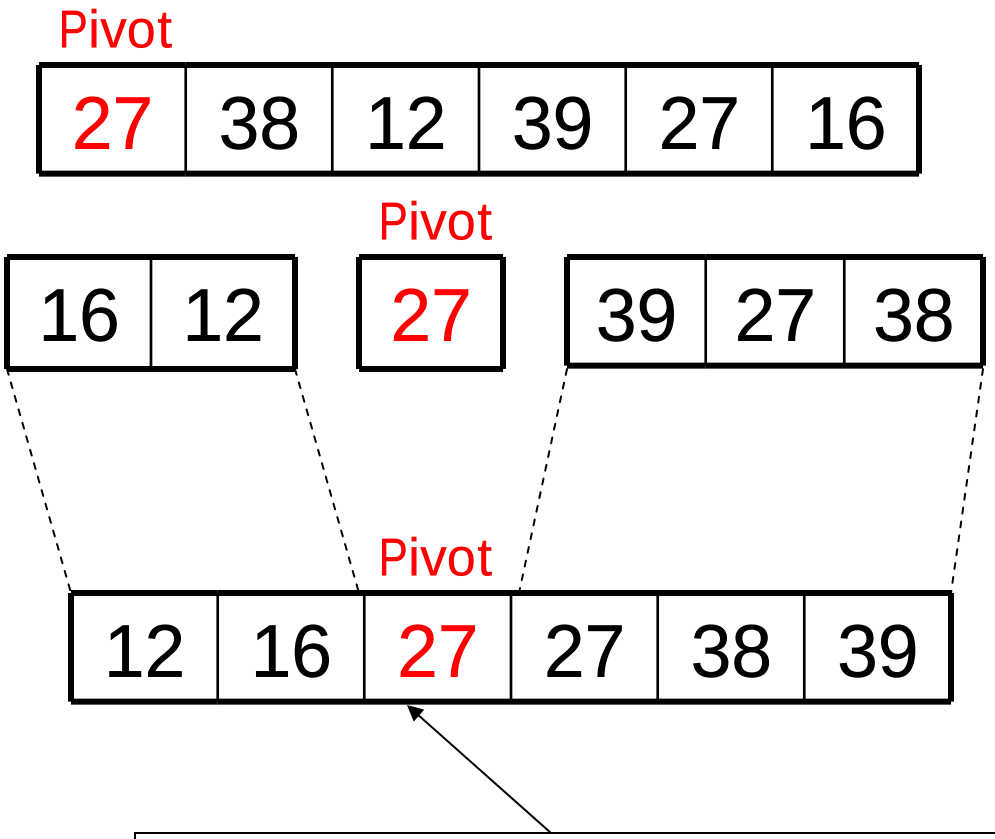
Q: Is it similar to the lecture notes on finding the K^{th} smallest element in the lecture notes on Recursion?

Quick sort Example

Choose the **1st** item as pivot

Partition a[] about the pivot 27

Recursively sort the two parts



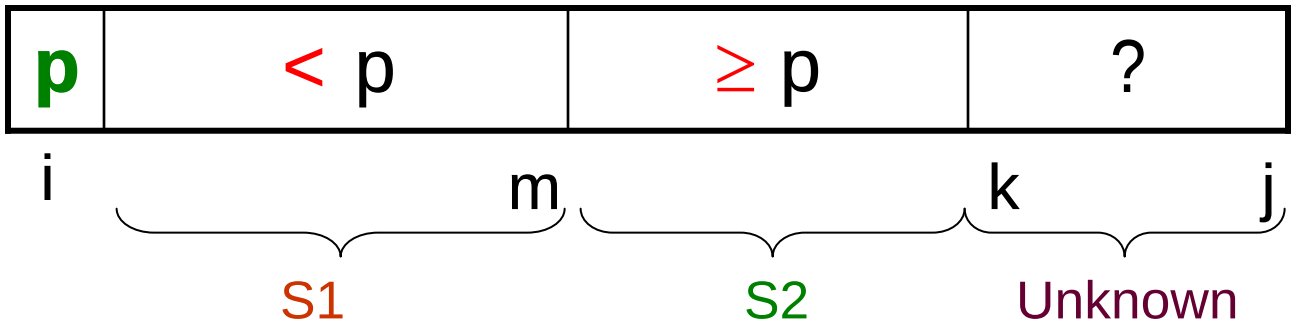
Note that after the partition,
the pivot is moved to its final position!
No merge phase is needed.

Code of Quick sort

```
void Quicksort (int a[], int i, int j){
    if (i < j) {    // Q: What if i >= j ?
        int pivotIdx = partition(a,i,j);
        Quicksort(a,i,pivotIdx-1);
                    // a[pivotIdx] is in its final position
        Quicksort(a,pivotIdx+1,j); }
}
    // No conquer part!  Q: Why?
```

Partition algorithm idea

- To partition a[i..j], we choose a[i] as the **pivot p**.
  Q: Why choose a[i]? Any other choices?
- The remaining items (i.e., a[i+1..j]) are divided into three regions:
  - S1 = a[i+1..m] : items < **p**
  - S2 = a[m+1..k-1] : items ≥ **p**
  - Unknown (unprocessed) = a[k..j] : items yet to be assigned to S1 or S2



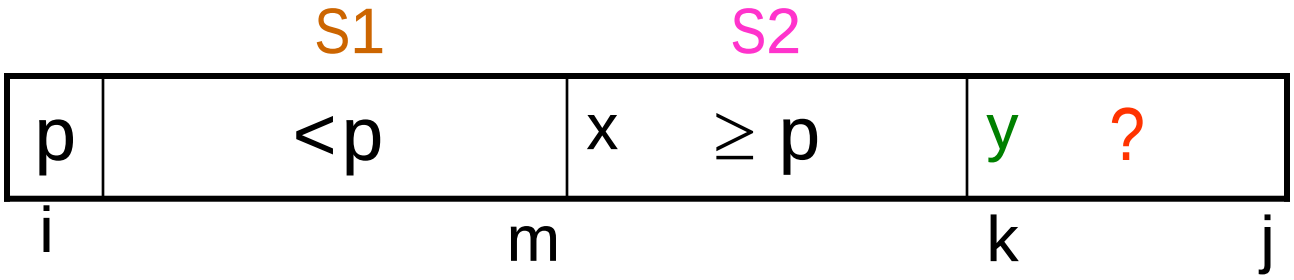
Partition algorithm idea (cont.)

- Initially, regions S1 and S2 are empty. All items excluding **p** are in the unknown region.
- Then, for each item a[k] for k=i+1 to j, in the unknown region, compare a[k] with **p**:
  - If a[k] >= **p**, put a[k] into S2.
  - Otherwise, put a[k] into S1.

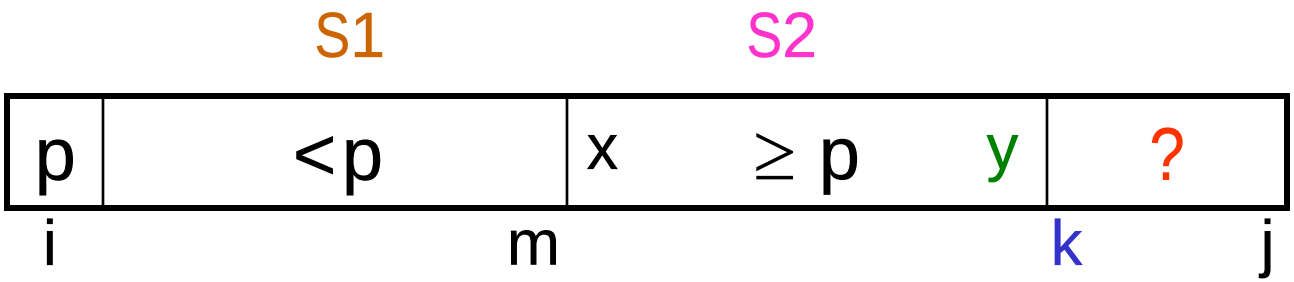
Q: How about if we change “>=“ to “>” in the condition part?

Partition algorithm idea (case 1)

If a[k] =y ≥ p,

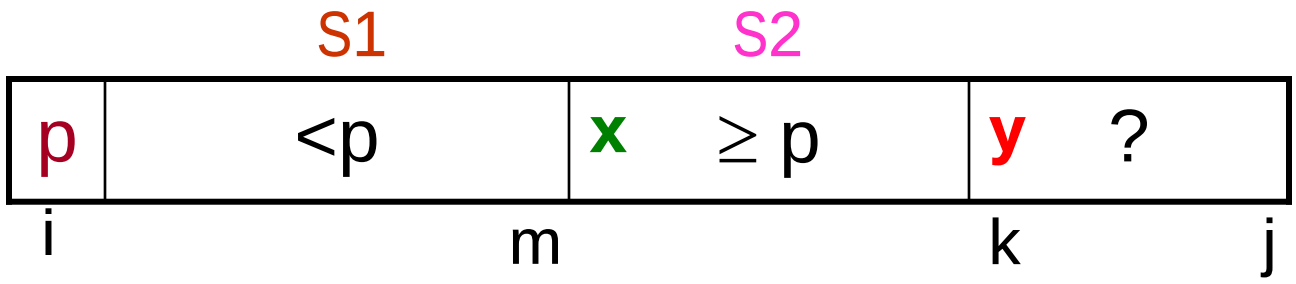


Increment k

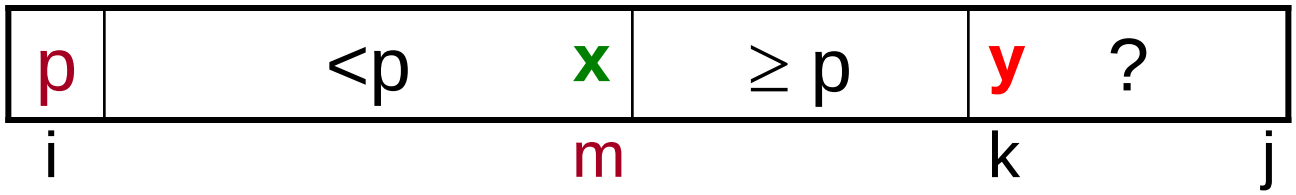


Partition algorithm idea (case 2)

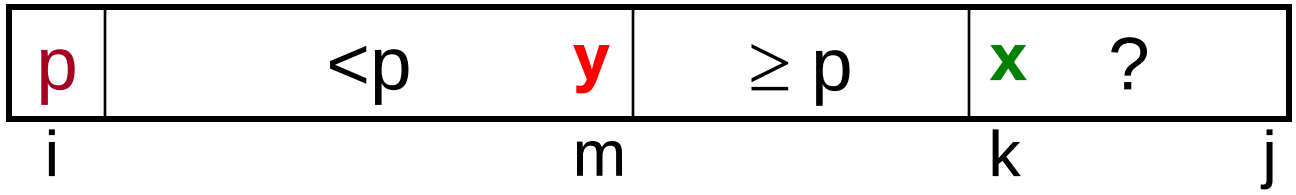
If a[k]=y **<** p



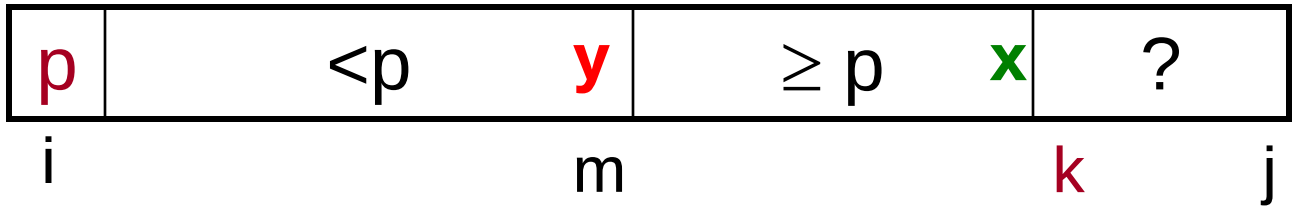
Increment m



Swap x and y

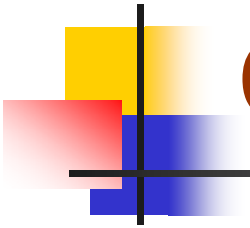


Increment k



Code of Partition algorithm

```
public static int partition (int a[], int i, int j)  {
                                          // partition data items in a=[i..j]
    int p = a[i];                         // p is the pivot, the ith item
    int m = i;                            // Initially S1 and S2 are empty
    for (int k = i+1; k<=j; ++k) {        // process unknown region
        if (a[k] < p) {                   // case 2: put a[k] to S1
           ++m;
           swap (a,k,m);
        } else {                          // case 1:  put a[k] to S2! Do nothing!
          }
    }
    swap (a,i,m);                         // put the pivot at the right place
    return m;                             // m is the pivot final position
}
```

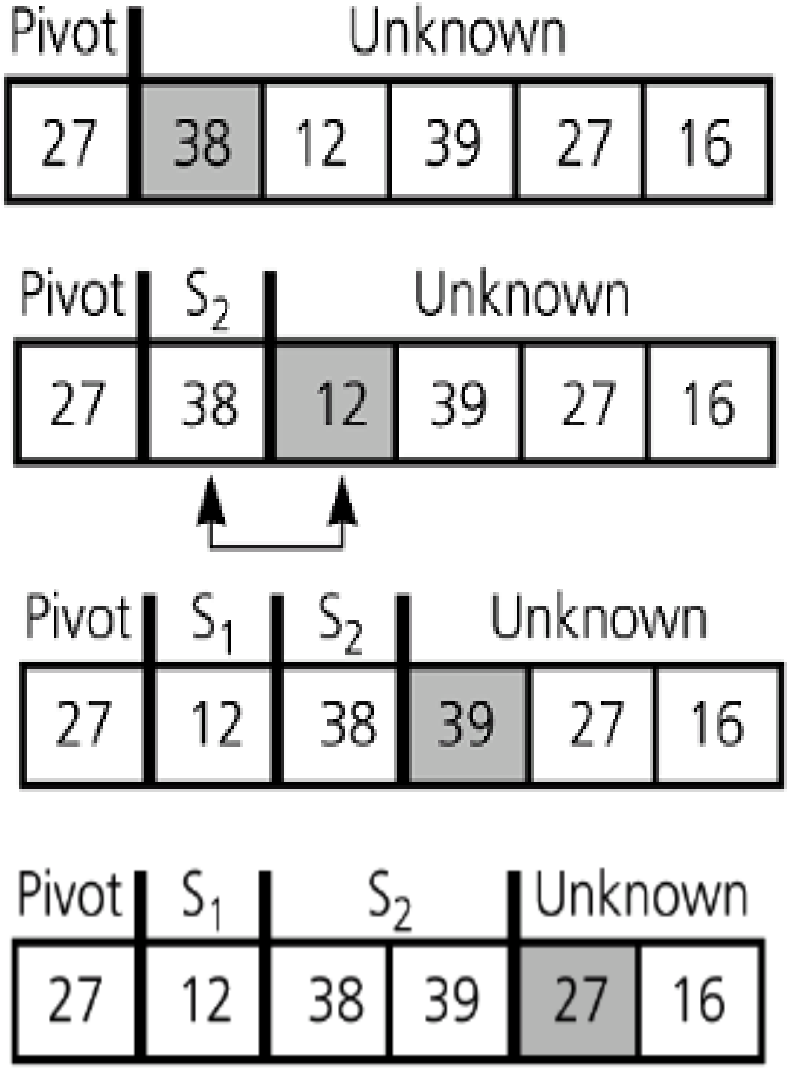


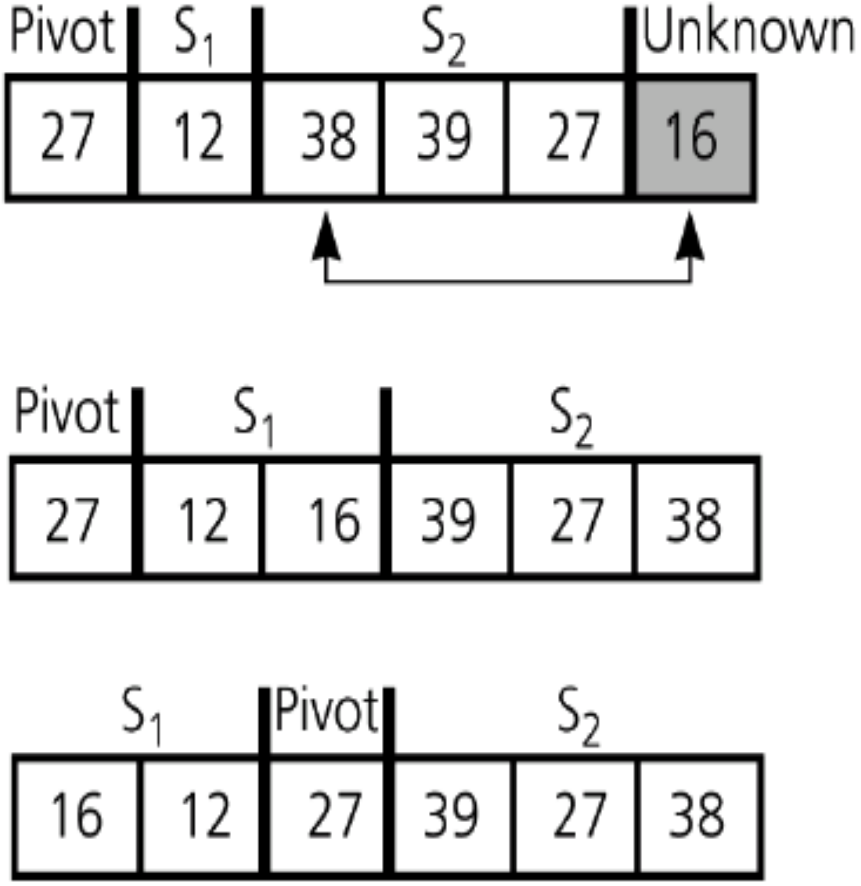
Complexity of partition algorithm

As there is only one for loop and the size of the array is n=j-i+1, so

the complexity is O(n)

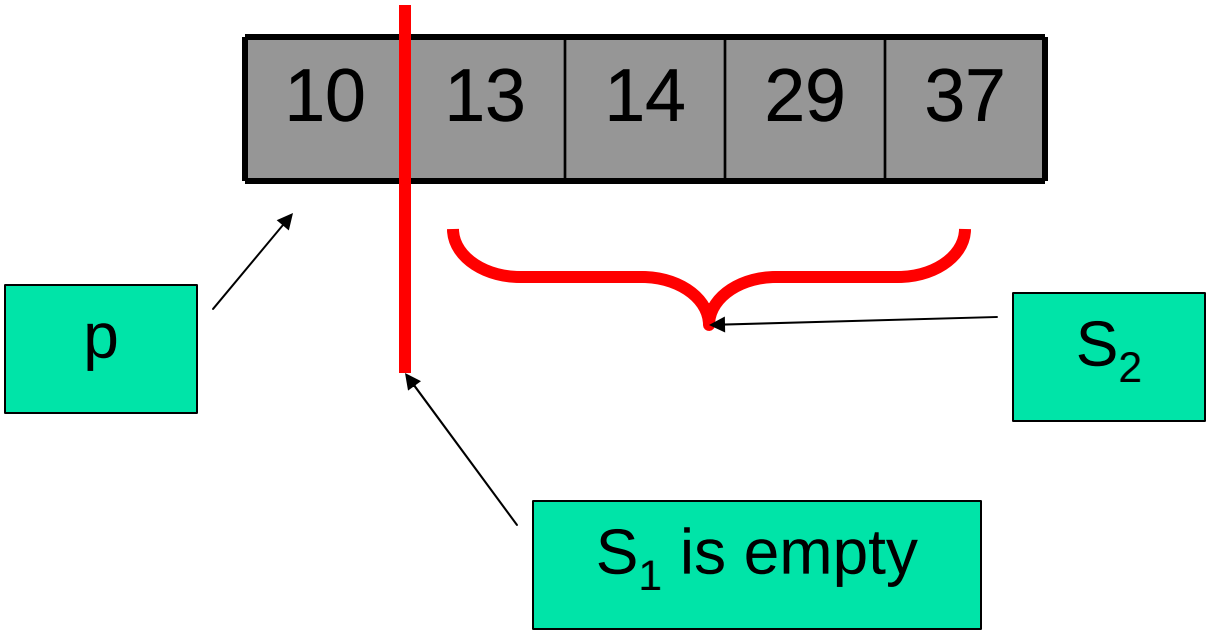
Partition algorithm by example





Worst Case for Quick sort

When a[0..n-1] is in increasing order:



What is the index returned by partition()?

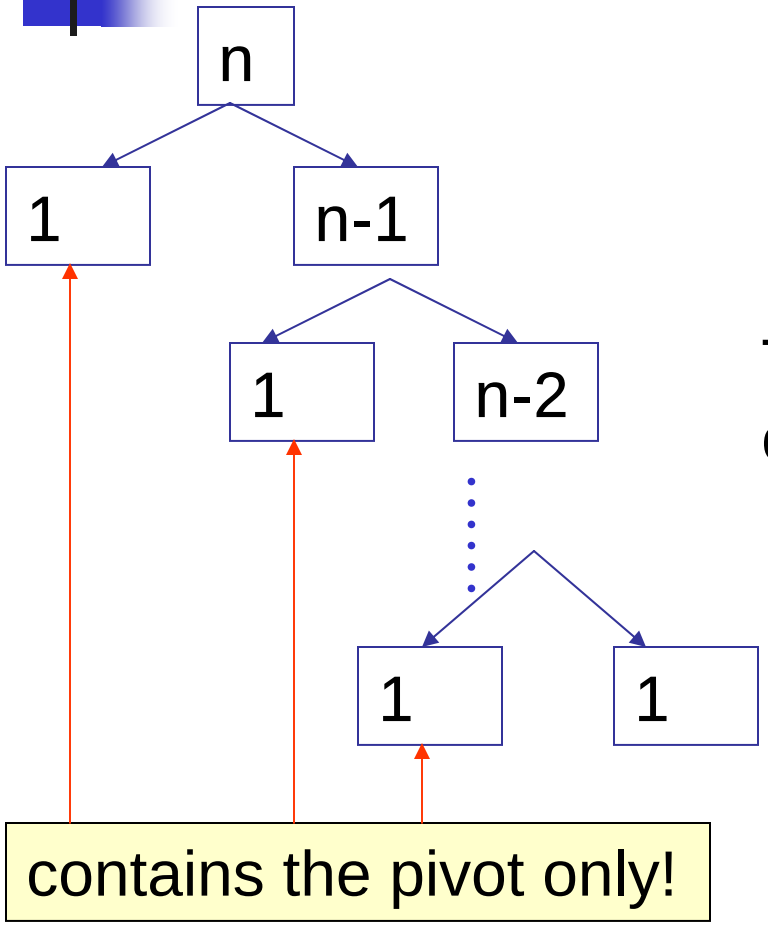
swap(a,i,m) will swap the pivot with itself!

The left partition (S1) is empty and

the right partition (S2) is the rest excluding the pivot

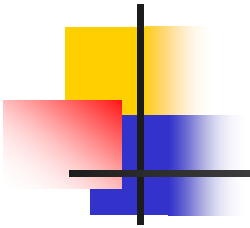
Q: What if the sequence is in descending order?

Worst Case for Quick sort (cont.)



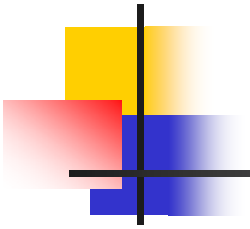
Total no. of levels = n

As partition takes O(n) time, the algorithm in its worst case takes time n+(n-1)+...+1 = $O(n^2)$



Best/average case for Quick sort

- Best case occurs when partition always splits the array into two equal halves.
  - Depth of recursion is log n.
  - Time complexity is O(n log n)
- In practice, worst case is rare, and on the average we get some good splits and some bad ones.
  - Average time is O(n log n).



Algo #6: **Radix Sort**

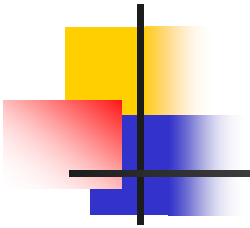
- Treats each data to be sorted as a character string.
- It is not using comparison, i.e., **no** comparison among the data is needed.
- In each iteration, organize the data into groups according to the next character in each data.

Radix Sort of Eight Integers

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150	Original integers
(156**0**, 215**0**) (106**1**) (022**2**) (012**3**, 028**3**) (215**4**, 000**4**)	Grouped by fourth digit
1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004	Combined
(00**0**4) (02**2**2, 01**2**3) (21**5**0, 21**5**4) (15**6**0, 10**6**1) (02**8**3)	Grouped by third digit
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Combined
(**0**004, 1**0**61) (**1**123, 2**1**50, 2**1**54) (**2**22, 0**2**83) (**1**560)	Grouped by second digit
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Combined
(**0**004, **0**123, **0**222, **0**283) (**1**061, **1**560) (**2**150, **2**154)	Grouped by first digit
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Combined (sorted)

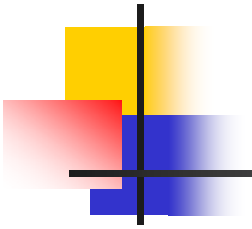
Pseudocode of Radix sort

```
radixSort (theArray, n, d) {
   // Sorts n d-digit numeric strings in the array theArray.
for (j = d down to 1) {    // for digits in last position to 1st position
   initialize 10 groups (queues) to empty    // Q: why 10?
   for (i=0 through n-1) {
      k = jth digit of theArray[i]
      place theArray[i] at the end of group k
   }
Replace theArray with all items in group 0, followed by all
items in group 1, and so on.
}
```



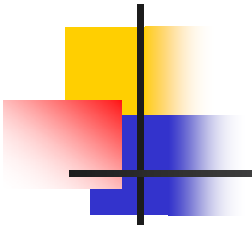
Complexity of Radix Sort

Complexity is O(n), or O(d*n) where d is the maximum number of digits in the numeric string. Since d is fixed or bounded, so the complexity is O(n).



In-place Sort

- A sort algorithm is said to be an "in-place" sort if it requires only a constant amount (ie., O(1)) of extra space during the sorting process.
  - Mergesort is not in-place, why?



Stable Sort

A sorting algorithm is "stable" if the relative order of elements with the same key value is preserved by the algorithm.

Example 1:

Names have been sorted into alphabetical order. Now if this list is sorted again according to tutorial group number, a stable sort algorithm will make all students within the same tutorial group to appear together in alphabetical order of their names.

Quick sort and Selection sort are not stable, why?

Non-Stable Sort

Example:2

Quick sort:

1285 5 150 4746 602 5 8356 // pivot in bold

1285 (5 150 602 5) (4746 8356)

5 5 150 602 **1285** 4746 8356 //pivot swapped with the last one in S1

// the **2 5's** are in different order of the initial list

Selection sort: select the largest one and swap with the last one

1285 5 4746 602 5 (8356)

1285 5 5 602 (4746 8356)

602 5 5 (1285 4746 8356)

5 5 (602 1285 4746 8356) // the **2 5's** are in different order of the initial list

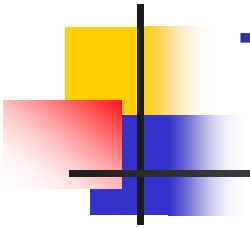
Summary of Sorting Algorithms

	Worst Case	Best Case	In-place?	Stable?
Selection Sort	$O(n^2)$	$O(n^2)$	Yes	No
Insertion Sort	$O(n^2)$	$O(n)$	Yes	Yes
Bubble Sort	$O(n^2)$	$O(n^2)$	Yes	Yes
Bubble Sort 2 (improved with flag)	$O(n^2)$	$O(n)$	Yes	Yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	No	Yes
Radix Sort (non-comparison based)	**$O(n)$**	$O(n)$	No	yes
Quick Sort	$O(n^2)$	$O(n \log n)$	Yes	No

Notes: 1. **O(n)** for Radix Sort is due to non-comparision based sorting.
2. O(n log n) is the best possible for comparison based sorting.

Java sort methods (in Arrays class)

static void **sort**(byte[] a)
static void **sort**(byte[] a, int fromIndex, int toIndex)
static void **sort**(char[] a)
static void **sort**(char[] a, int fromIndex, int toIndex)
static void **sort**(double[] a)
static void **sort**(double[] a, int fromIndex, int toIndex)
static void **sort**(float[] a)
static void **sort**(float[] a, int fromIndex, int toIndex)
static void **sort**(int[] a)
static void **sort**(int[] a, int fromIndex, int toIndex)
static void **sort**(long[] a)
static void **sort**(long[] a, int fromIndex, int toIndex)
static void **sort**(Object[] a)
static void **sort**(Object[] a, int fromIndex, int toIndex)
static void **sort**(short[] a)
static void **sort**(short[] a, int fromIndex, int toIndex)
static <T> void **sort**(T[] a, Comparator<? super T> c)
static <T> void **sort**(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)



To use the sort() in Arrays

- The entities to be sorted must be stored in an array first.
- If they are stored in a list, then we have to use Collections.sort()
- If the data to be sorted are not primitive, then Comparator must be defined and used

Note: Collections is a Java public class and Comparator is a public interface. Comparators can be passed to a sort method (such as Collections.sort) to allow precise control over the sort order.

Simple program using Collections.sort()

```
import java.util.*;
public class Sort {
        public static void main(String args[]) {
                List<String> l = Arrays.asList(args);
                Collections.sort(l);
                System.out.println(l);
        }
}
```

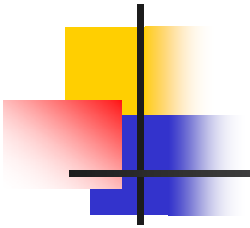
run the program:

java Sort i walk the line

The following output is produced:

[i, line, the, walk]

Note: Arrays is a Java public class and asList is a method of Arrays which returns a fixed-size list backed by the specified array.



class Person

```
class Person {
  private String name;
  private int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  } // end constructor
  public String getName() { return name;}
  public int getAge() { return age; }
  public String toString() { return name + " - " + age;}
} //end Person
```

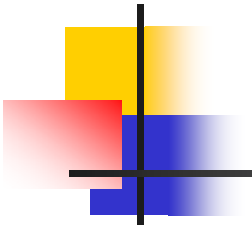
Comparator: AgeComparator

```
import java.util.Comparator;
 class AgeComparator implements Comparator<Person> {
 public int compare(Person o1, Person o2) {
    / / Returns the difference:
    / / if positive, age of o1 person is greater than o2 person
    / / if zero, the ages are equal
    / / if negative, age of o1 person is less than o2 person
    return o1.getAge() - o2.getAge();
  } // end compare

  public boolean equals(Object obj) {
    / / Simply checks to see if we have the same object
    return this==obj;
  } // end equals
} // end AgeComparator
```

Note: compare and equals are two methods of the interface Comparator. Need to implement them.

Comparator: NameComparator

```
import java.util.Comparator;

class NameComparator implements Comparator<Person> {

public int compare(Person o1, Person o2) {
    / / Compares its two arguments for order by name.
    return o1.getName().compareTo(o2.getName());
  } // end compare

  public boolean equals(Object obj) {
    / / Simply checks to see if we have the same object
    return this==obj;
  } // end equals
} // end NameComparator
```

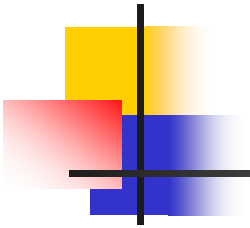


TestComparator

```
import java.util.*;

class TestComparator {
  public static void main(String args[]) {

    NameComparator nameComp = new NameComparator();
    AgeComparator ageComp = new AgeComparator();
    Person[] p = new Person[5];

    p[0] = new Person("Michael", 15);
    p[1] = new Person("Mimi", 9);
    p[2] = new Person("Sarah", 12);
    p[3] = new Person("Andrew", 15);
    p[4] = new Person("Mark", 12);
    List<Person> l = Arrays.asList(p);
```

TestComparator (cont.)

```
        System.out.println("Sorting by age:");
        Collections.sort(l, ageComp);
        System.out.println(l + "\n");

        List<Person> l1 = Arrays.asList(p);
        System.out.println("Sorting by name:");
        Collections.sort(l1, nameComp);
        System.out.println(l1 + "\n");

        System.out.println("Now sort by age, then sort by name:");
        Collections.sort(l1, ageComp);    // l1 is already sorted by name
        System.out.println(l1);
    } // end main
} // end TestComparator
```



TestComparator (cont.)

java TestComparator

Sorting by age:
[Mimi - 9, Sarah - 12, Mark - 12, Michael - 15, Andrew - 15]

Sorting by name:
[Andrew - 15, Mark - 12, Michael - 15, Mimi - 9, Sarah - 12]

Now sort by age, then sort by name:
[Mimi - 9, Mark - 12, Sarah - 12, Andrew - 15, Michael - 15]