Advanced Sorting

Recursive Sorting Algorithms

Problem Statement

- **Problem:**
  - Given a list of N ordinal-type elements, sort the list in ascending order.

32	35	55	...	72	89
0	1	2		N-2	N-1

- **Algorithms:**
  - MergeSort
  - QuickSort

The MergeSort Algorithm

1. Split the unsorted original list into two sub-lists of equal size.

2. Perform MergeSort on each sub-list.

   Result: Both sub-lists are now **sorted**.

3. Merge both sorted sub-lists onto the original list.

Recursive MergeSort in Java

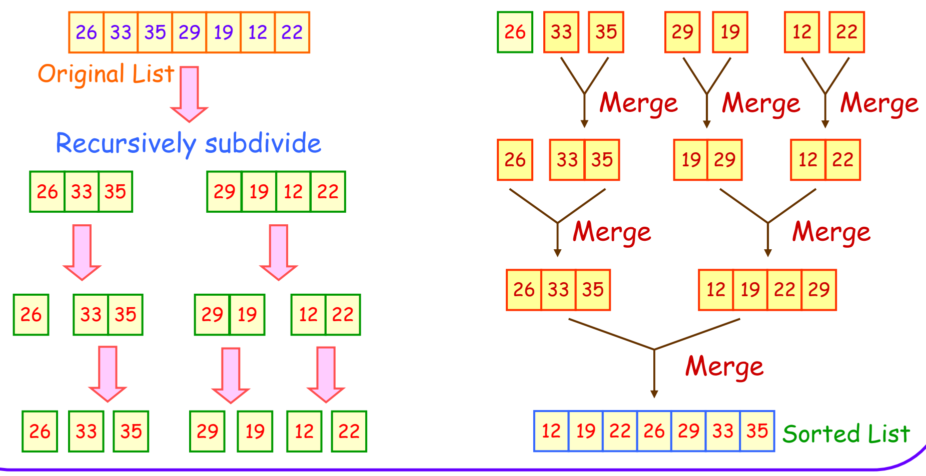
```
public static <E> void mergeSort (E[ ] data, Comparator c, int first, int n)  {
   int n1;                              // Size of the first subarray
   int n2;                              // Size of the second subarray

   if (n > 1)  {
      n1 = n / 2;                       // Compute size of first subarray
      n2 = n – n1;                      // Compute size of 2nd subarray
      mergeSort(data, c, first, n1);    // Sort n1 items from data[first]
      mergeSort(data, c, first+n1, n2); // Sort n2 items from data[first+n1]
      merge(data, c, first, n1, n2);    // Merge the two sorted halves.
   }
}
```

MergeSort
The Merging Process

- Two sub-lists are merged into one list that is twice as long as either of them as follows:
  1. Compare the first element from one sub-list with the first element of the other sub-list, the smallest is appended to the destination list, and replaced by the next element in its sub-list.

  2. Repeat step 1 until one of the sub-lists is empty, then append the remaining elements from the other sub-list to the destination list.

MergeSort
Example:



Merge in Java

```
private static <E> void merge (E[ ] data, Comparator c, int first, int n1, int n2)
{
    E[ ] temp;              // Array to hold sorted items
    int copied   = 0;       // Number of items copied from data to temp
    int copied1 = 0;        // Number copied from the first half of data
    int copied2 = 0;        // Number copied from the second half of data
    int i;                  // Array index to copy from temp back into data

    // Allocate memory for the temporary dynamic array.
    temp = (E[ ]) new Object [n1+n2];
```

Merge in Java

```
    // Merge elements, copying from two halves of data to temp array.
    while ((copied1 < n1) && (copied2 < n2))  {
       if (c.compare(data[first + copied1], data[first + n1 + copied2]) == -1)

            // Copy from first half
            temp[copied++] = data[first + (copied1++)];

       else

            // Copy from second half
            temp[copied++] = data[first + n1 + (copied2++)];
    }
```

Merge in Java

```
    // Copy any remaining entries in the left and right subarrays.
    while (copied1 < n1) temp[copied++] = data[first + (copied1++)];
    while (copied2 < n2) temp[copied++] = data[first + n1 + (copied2++)];

    // Copy temp to the data array, and release temp's memory.
    for (i = 0; i < n1+n2; i++)  data[first+i] = temp[i];
    temp = null;
}
```

MergeSort Discussion

- Suited for sorting linked lists since random access is not required.
- Also suited for external sorting for the same reason.
- For arrays, we will need a secondary array for the merging process, which is a serious disadvantage.

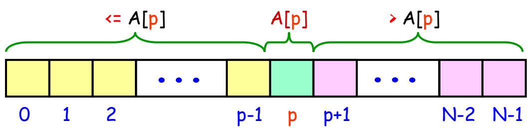
MergeSort Performance Analysis

- The list is broken in half at each stage: O(log *n*) stages
- Initially, *n* lists of size 1 each are merged: *n* steps
- Then *n*/2 lists of size 2 each are merged: *n* steps
- Then *n*/4 lists of size 4, ... : *n* steps for each stage
- The total cost of mergeSort is O(*n* log *n*)

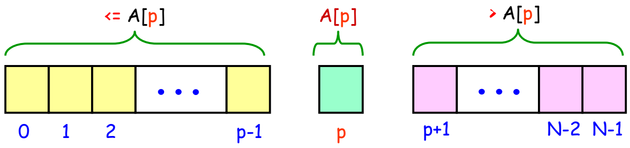
The QuickSort Algorithm

- Select one element of the array; call it the pivot element.
- Determine the final location of the pivot in the (sorted) list; call that location p.
- Shuffle the elements of the list so that all elements of the list with values less than or equal to the value of the pivot have indices less than p. Similarly all elements with values greater than the pivot's have indices greater than p.

Example:



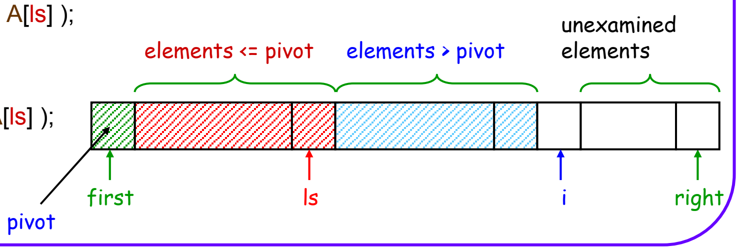
QuickSort in Java



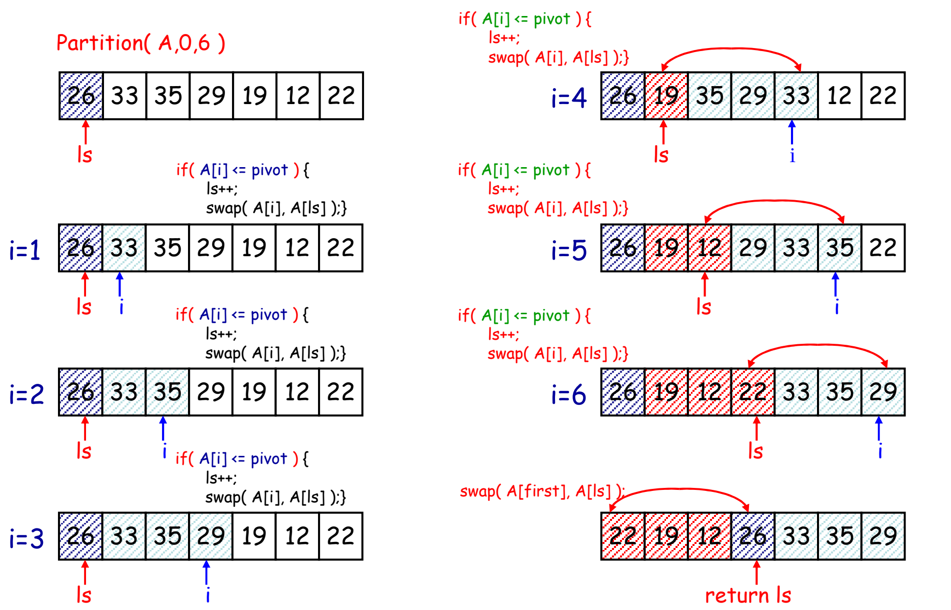

```
public static <E> void quickSort (E[ ] data, Comparator c, int first, int n)  {
    int p, n1, n2;
    if (n > 1)  {
        p = Partition( data, c, first, n );
        n1 = p – first;
        n2 = n – n1 – 1;
        quickSort( data, c, first, n1 );
        quickSort( data, c, p+1, n2 );
    }
}
```

QuickSort
The Partitioning Process

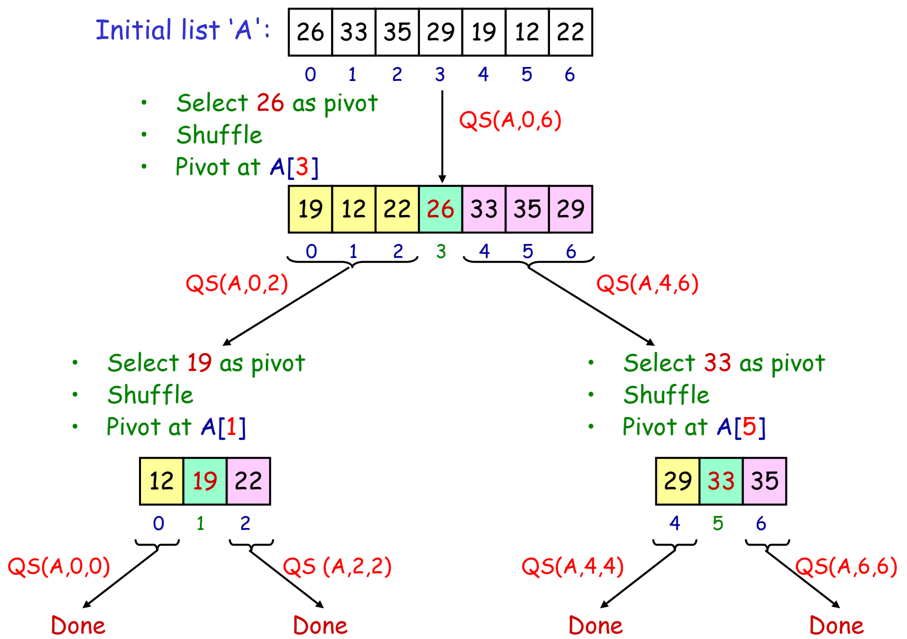
```
private static <E> int partition (E[ ] A, Comparator c, int first, int n )  {
    int right = first + n – 1;
    int ls = first;
    E pivot = A[first];

    for( int i = first+1; i <= right; i++ )  {
        if( c.compare(A[i], pivot) <= 0 )  {    // Move items smaller than pivot only,
            ls++;                                // to location that would be at left of pivot
            swap( A[i], A[ls] );
        }
    }
    swap( A[first], A[ls] );

    return ls;
}
```



QuickSort Partitioning Example



QuickSort Example



QuickSort
Performance Analysis

- Worst case: list is broken 0 elements in one partition and n -1 in the other, that is:

$$\sum_{k=1}^{n} k = O(n^2)$$

- Best case:
  - List is broken into two equal halves with $n/2$ elements each.
  - Then broken into four lists with $n/4$ elements each, etc.
  - Total work of n at each stage, with $\log n$ stages = **O (n log n)**

- Average case: assume that the pivot is equally likely to occur at any position,

$$\mathrm{T}(n) = cn + \frac{1}{n}\sum_{k=0}^{n-1}(\mathrm{T}(k) + \mathrm{T}(n-1-k)), \quad \mathrm{T}(0) = c, \quad \mathrm{T}(1) = c$$

- Solution: **O (n log n)**

QuickSort
Performance Improvements

1. Choose the pivot as the median of three: A[left], A[right], and A[(left+right)/2].

2. Keep partitioning until each sub-list has about 10 elements.
   Then after QuickSort use insertion sort.
   This gives 12% to 20% improvement.

3. Use a non-recursive implementation.

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	• in-place • slow (good for small inputs)
insertion-sort	$O(n^2)$	• in-place • slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	• in-place, randomized • fastest (good for large inputs)
heap-sort	$O(n \log n)$	• in-place • fast (good for large inputs)
merge-sort	$O(n \log n)$	• sequential data access • fast (good for huge inputs)