Sorting Algorithms

Slides adapted from Tolga Can, METU

Mergesort

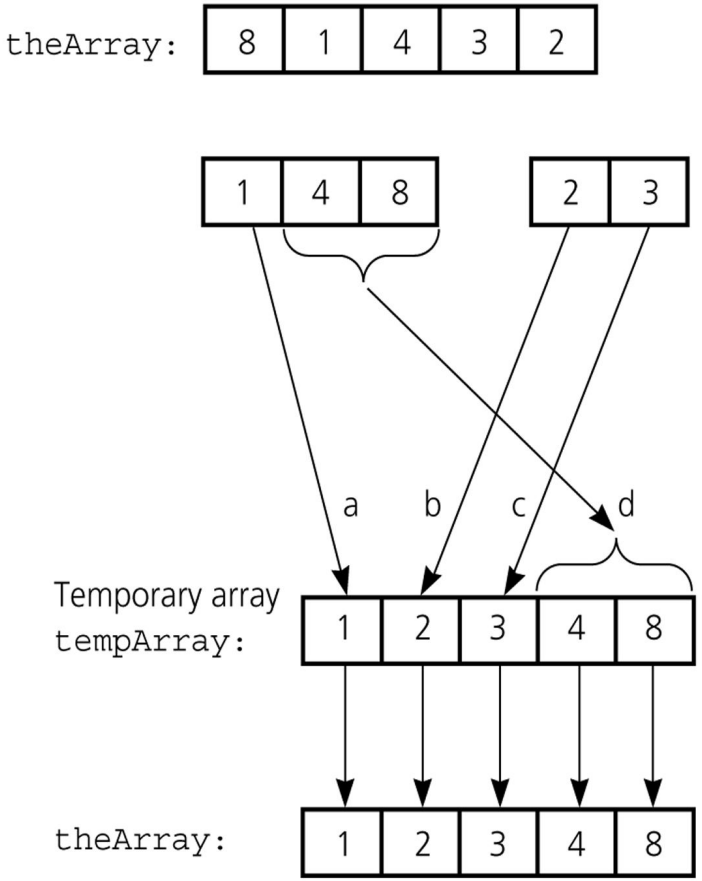
- Mergesort algorithm is one of two important divide-and-conquer sorting algorithms (the other one is quicksort).
- It is a recursive algorithm.
  - Divides the list into halves,
  - Sort each halve separately, and
  - Then merge the sorted halves into one sorted array.

```
class MergeSort
{   void sort(int arr[], int l, int r)    {
        if (l < r)        {
            // Find the middle point
            int m = (l+r)/2;
            // Sort first and second halves
            sort(arr, l, m);
            sort(arr , m+1, r);
            // Merge the sorted halves
            merge(arr, l, m, r);
        }
    }
   // Driver method
    public static void main(String args[])
{
        int arr[] = {12, 11, 13, 5, 6, 7};
        System.out.println("Given Array");
        printArray(arr);
        MergeSort ob = new MergeSort();
        ob.sort(arr, 0, arr.length-1);
        System.out.println("\nSorted array");
        printArray(arr);
    }
}
```

```
static void printArray(int arr[])    {
     int n = arr.length;
     for (int i=0; i<n; ++i)
       System.out.print(arr[i] + " ");
      System.out.println();
}
```

```
    // Merges two subarrays of arr[].
    // First subarray is arr[l..m]
    // Second subarray is arr[m+1..r]
    void merge(int arr[], int l, int m, int r)
{
        // Find sizes of two subarrays to be
merged
        int n1 = m - l + 1;
        int n2 = r - m;
        /* Create temp arrays */
        int L[] = new int [n1];
        int R[] = new int [n2];
        /*Copy data to temp arrays*/
        for (int i=0; i<n1; ++i)
            L[i] = arr[l + i];
        for (int j=0; j<n2; ++j)
            R[j] = arr[m + 1+ j];
        int i = 0, j = 0;
// Initial index of merged subarry array
        int k = l;
        while (i < n1 && j < n2)        {
            if (L[i] <= R[j])            {
                arr[k] = L[i];
                i++;
            }
            else            {
                arr[k] = R[j];
                j++;
            }
            k++;
        }
        /* Copy remaining elements of L[] if any */
        while (i < n1)        {
            arr[k] = L[i];
            i++;
            k++;
        }

        /* Copy remaining elements of R[] if any */
        while (j < n2)
        {
            arr[k] = R[j];
            j++;
            k++;
        }
    }
```

Mergesort - Example



Divide the array in half

Sort the halves

Merge the halves:

a. 1 < 2, so move 1 from left half to `tempArray`
b. 4 > 2, so move 2 from right half to `tempArray`
c. 4 > 3, so move 3 from right half to `tempArray`
d. Right half is finished, so move rest of left half to `tempArray`

Copy temporary array back into original array

Merge

```
int MAX_SIZE = maximum-number-of-items-in-array;
void merge(double[] theArray, int first, int mid, int last) {
   double[] tempArray = new double[MAX_SIZE];// temporary array
   int first1 = first;    // beginning of first subarray
   int last1 = mid;       // end of first subarray
   int first2 = mid + 1;  // beginning of second subarray
   int last2 = last;      // end of second subarray
   int index = first1; // next available location in tempArray
   for ( ; (first1 <= last1) && (first2 <= last2); ++index) {
      if (theArray[first1] < theArray[first2]) {
         tempArray[index] = theArray[first1];
         ++first1;
      }
      else {
          tempArray[index] = theArray[first2];
          ++first2;
      }
}
```

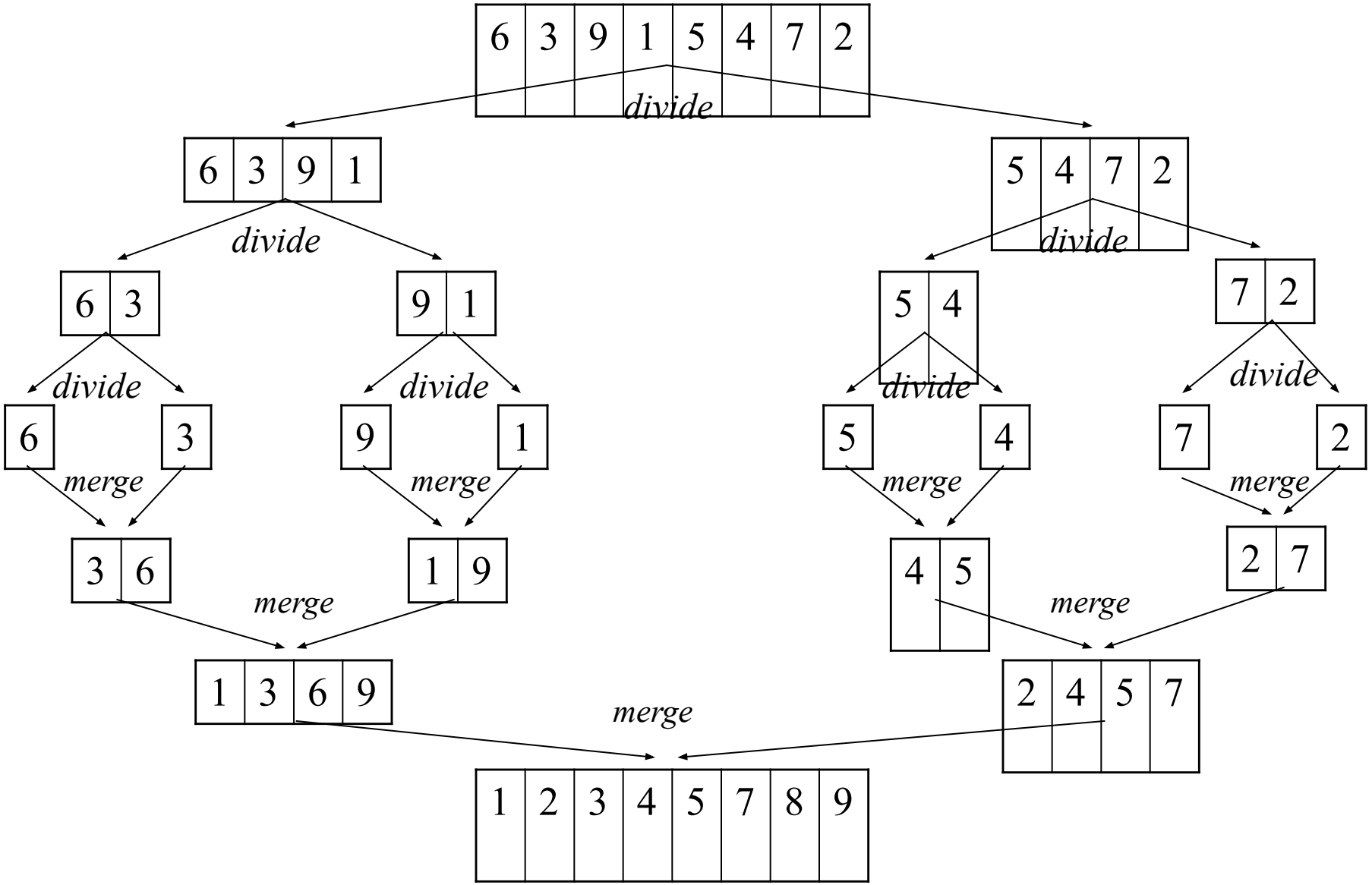
Merge (cont.)

```
 // finish off the first subarray, if necessary
    for (; first1 <= last1; ++first1, ++index)
       tempArray[index] = theArray[first1];

    // finish off the second subarray, if necessary
    for (; first2 <= last2; ++first2, ++index)
       tempArray[index] = theArray[first2];

    // copy the result back into the original array
    for (index = first; index <= last; ++index)
       theArray[index] = tempArray[index];
}   // end merge
```

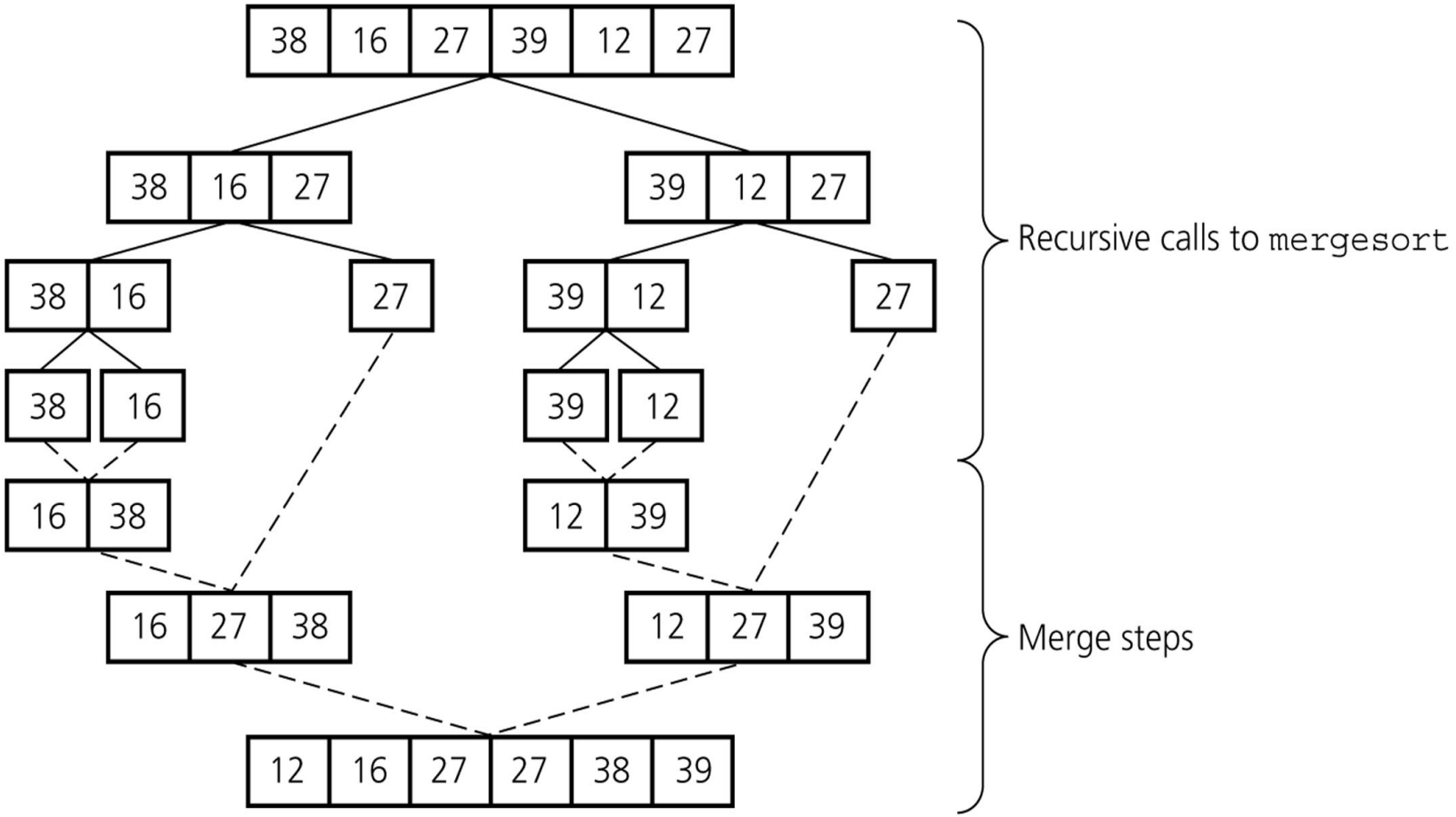
Mergesort

```
void mergesort(double[] theArray, int first, int last) {
   if (first < last) {
      int mid = (first + last)/2;   // index of midpoint
      mergesort(theArray, first, mid);
      mergesort(theArray, mid+1, last);

      // merge the two halves
      merge(theArray, first, mid, last);
   }
}  // end mergesort
```

Mergesort - Example

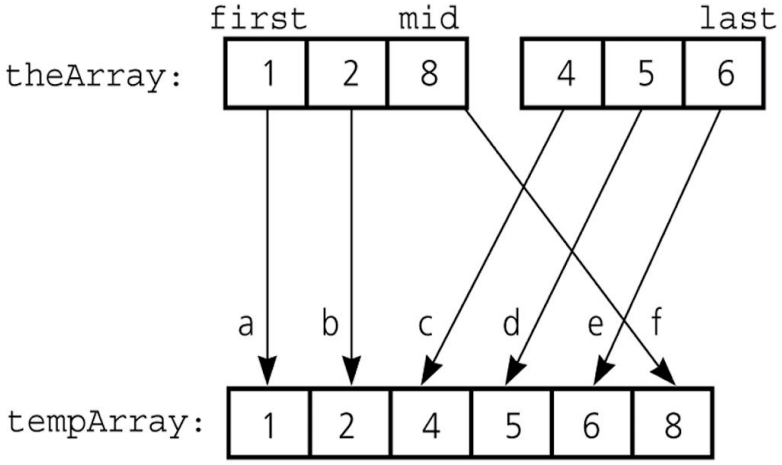


Mergesort – Example2



Mergesort – Analysis of Merge

A worst-case instance of the merge step in *mergesort*

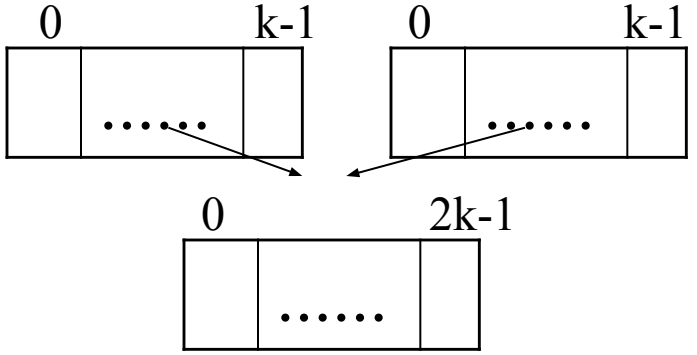


Merge the halves:

a. 1 < 4, so move 1 from `theArray[first..mid]` to `tempArray`
b. 2 < 4, so move 2 from `theArray[first..mid]` to `tempArray`
c. 8 > 4, so move 4 from `theArray[mid+1..last]` to `tempArray`
d. 8 > 5, so move 5 from `theArray[mid+1..last]` to `tempArray`
e. 8 > 6, so move 6 from `theArray[mid+1..last]` to `tempArray`
f. `theArray[mid+1..last]` is finished, so move 8 to `tempArray`

Mergesort – Analysis of Merge (cont.)

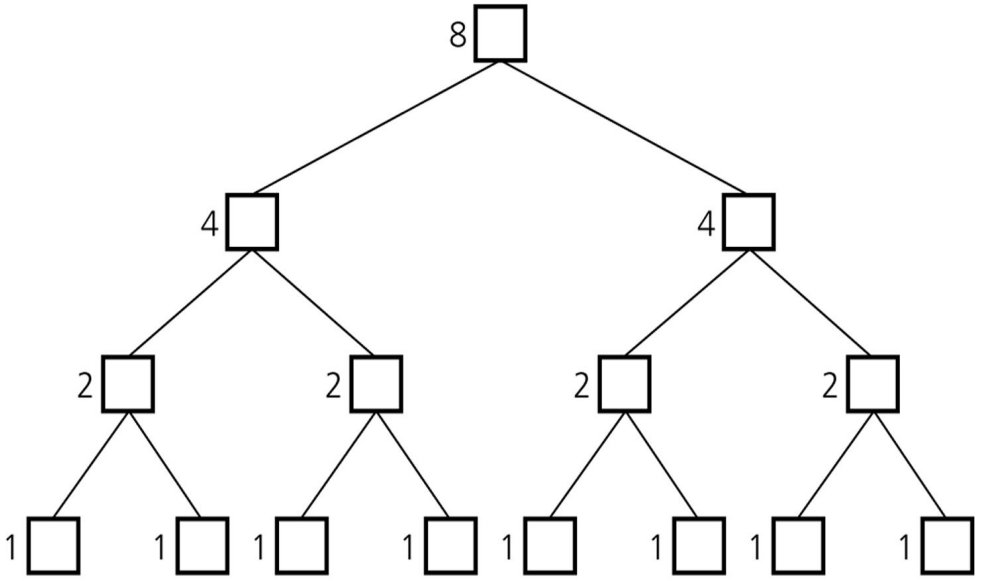
Merging two sorted arrays of size ***k***



- ***Best-case:***
  - All the elements in the first array are smaller (or larger) than all the elements in the second array.
  - The number of moves: 2k + 2k
  - The number of key comparisons: k
- ***Worst-case:***
  - The number of moves: 2k + 2k
  - The number of key comparisons: 2k-1

Mergesort - Analysis

Levels of recursive calls to *mergesort*, given an array of eight items



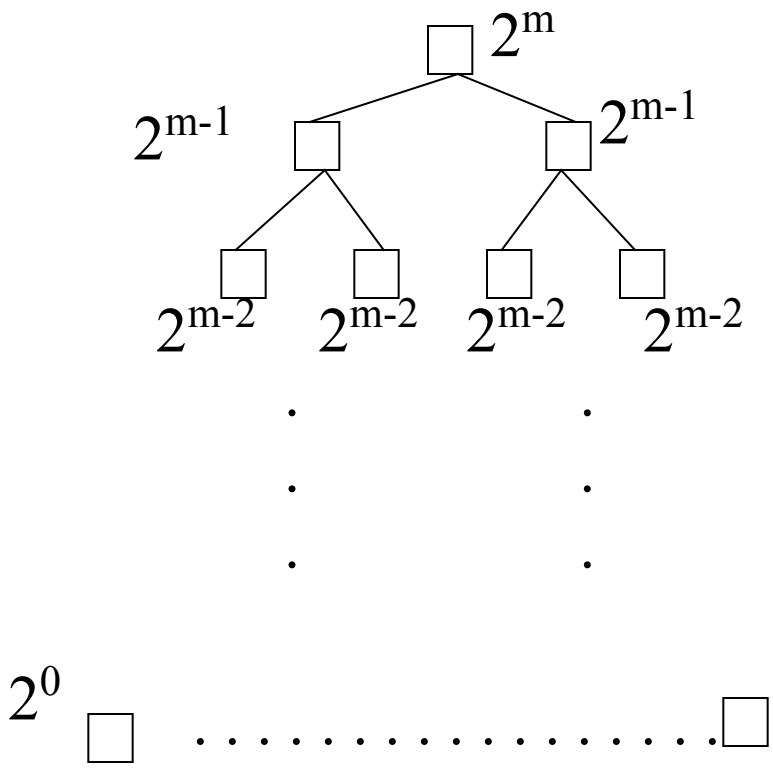
Level 0: `mergesort` 8 items

Level 1: 2 calls to `mergesort` with 4 items each

Level 2: 4 calls to `mergesort` with 2 items each

Level 3: 8 calls to `mergesort` with 1 item each

Mergesort - Analysis



level 0 : 1 merge (size 2^{m-1})

level 1 : 2 merges (size 2^{m-2})

level 2 : 4 merges (size 2^{m-3})

level m-1 : 2^{m-1} merges (size 2^0)

level m

Mergesort - Analysis

- ***Worst-case –***

The number of key comparisons:

$= 2^0*(2*2^{m-1}-1) + 2^1*(2*2^{m-2}-1) + ... + 2^{m-1}*(2*2^0-1)$

$= (2^m - 1) + (2^m - 2) + ... + (2^m – 2^{m-1})$ (m terms)

$= m*2^m - \sum_{i=0}^{m-1} 2^i$

$= m*2^m – 2^m – 1$

Using $m = \log n$

$= \mathbf{n * \log_2 n – n – 1}$

$\Rightarrow$ **O (n * $\log_2$n)**

Mergesort – Analysis

- Mergesort is extremely efficient algorithm with respect to time.
  - Both worst case and average cases are **O (n * $\log_2$n)**
- But, mergesort requires an extra array whose size equals to the size of the original array.
- If we use a linked list, we do not need an extra array
  - But, we need space for the links
  - And, it will be difficult to divide the list into half (O(n))

Analyzing recursive algorithms

recursive algorithms can be analyzed by defining a recurrence relation:

cost of searching N items using binary search =
cost of comparing middle element + cost of searching correct half (N/2 items)

more succinctly: Cost(N) = Cost(N/2) + C

Cost(N) = Cost(N/2) + C can unwind Cost(N/2)
= (Cost(N/4) + C) + C
= Cost(N/4) + 2C can unwind Cost(N/4)
= (Cost(N/8) + C) + 2C
= Cost(N/8) + 3C can continue unwinding
= …
= Cost(1) + ($\log_2 N$)*C
= C $\log_2$ N + C' where C' = Cost(1)
 O(log N)

Analyzing merge sort

cost of sorting N items using merge sort =
cost of sorting left half (N/2 items) + cost of sorting right half (N/2 items) +
cost of merging (N items)

more succinctly: $Cost(N) = 2Cost(N/2) + C_1N + C_2$

$Cost(N) = 2Cost(N/2) + C_1*N + C_2$ can unwind Cost(N/2)
$= 2(2Cost(N/4) + C_1N/2 + C_2) + C_1N + C_2$
$= 4Cost(N/4) + 2C_1N + 3C_2$ can unwind Cost(N/4)
$= 4(2Cost(N/8) + C_1N/4 + C_2) + 2C_1N + 3C_2$
$= 8Cost(N/8) + 3C_1N + 7C_2$ can continue unwinding
$= \ldots$
$= NCost(1) + (\log_2 N)C_1N + (N-1)\, C_2$
$= C_1N \log_2 N + (C'+C_2)N - C_2$ where C' = Cost(1)
 O(N log N)

Quicksort

- Like mergesort, Quicksort is also based on the *divide-and-conquer* paradigm.
- But it uses this technique in a somewhat opposite manner, as all the hard work is done *before* the recursive calls.
- It works as follows:
  1. First, it partitions an array into two parts,
  2. Then, it sorts the parts independently,
  3. Finally, it combines the sorted subsequences by a simple concatenation.

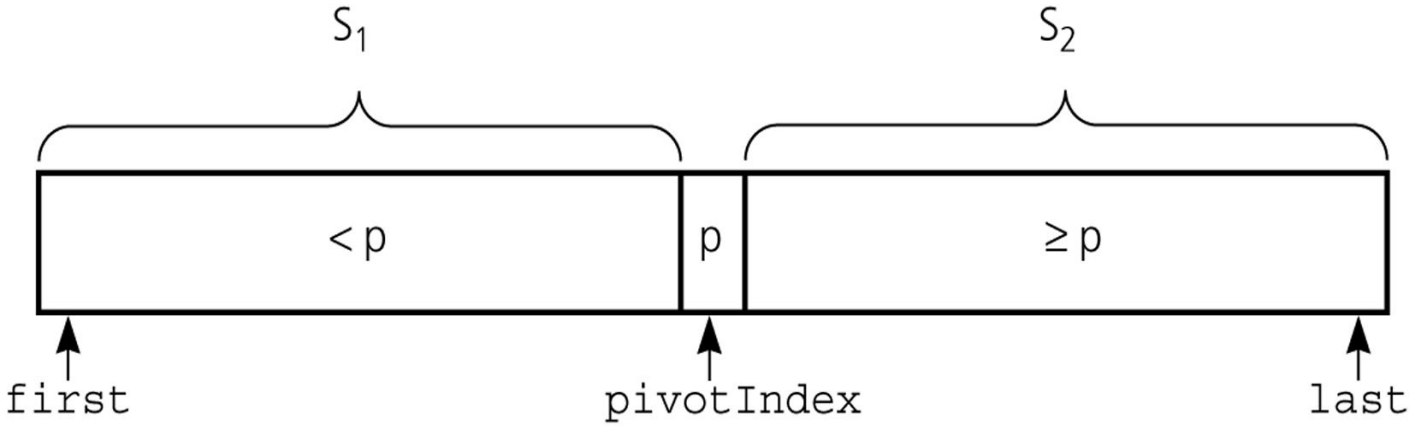
Quicksort (cont.)

The quick-sort algorithm consists of the following three steps:

1. ***Divide***: Partition the list.
   - To partition the list, we first choose some element from the list for which we hope about half the elements will come before and half after. Call this element the ***pivot***.
   - Then we partition the elements so that all those with values less than the pivot come in one sublist and all those with greater values come in another.
2. ***Recursion***: Recursively sort the sublists separately.
3. ***Conquer***: Put the sorted sublists together.

Partition

- Partitioning places the pivot in its correct place position within the array.



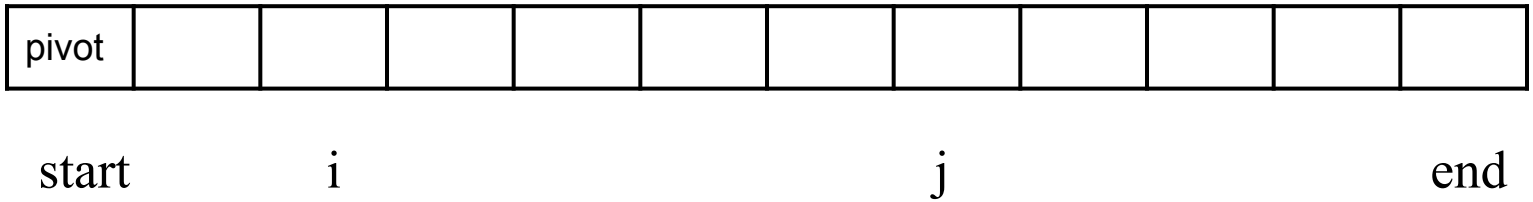
- Arranging the array elements around the pivot p generates two smaller sorting problems.
  - sort the left section of the array, and sort the right section of the array.
  - when these two smaller sorting problems are solved recursively, our bigger sorting problem is solved.

Partition – Choosing the pivot

- First, we have to select a pivot element among the elements of the given array, and we put this pivot into the first location of the array before partitioning.
- Which array item should be selected as pivot?
  - Somehow we have to select a pivot, and we hope that we will get a good partitioning.
  - If the items in the array arranged randomly, we choose a pivot randomly.
  - We can choose the first or last element as a pivot (it may not give a good partitioning).
  - We can use different techniques to select the pivot.

Partition Function (cont.)

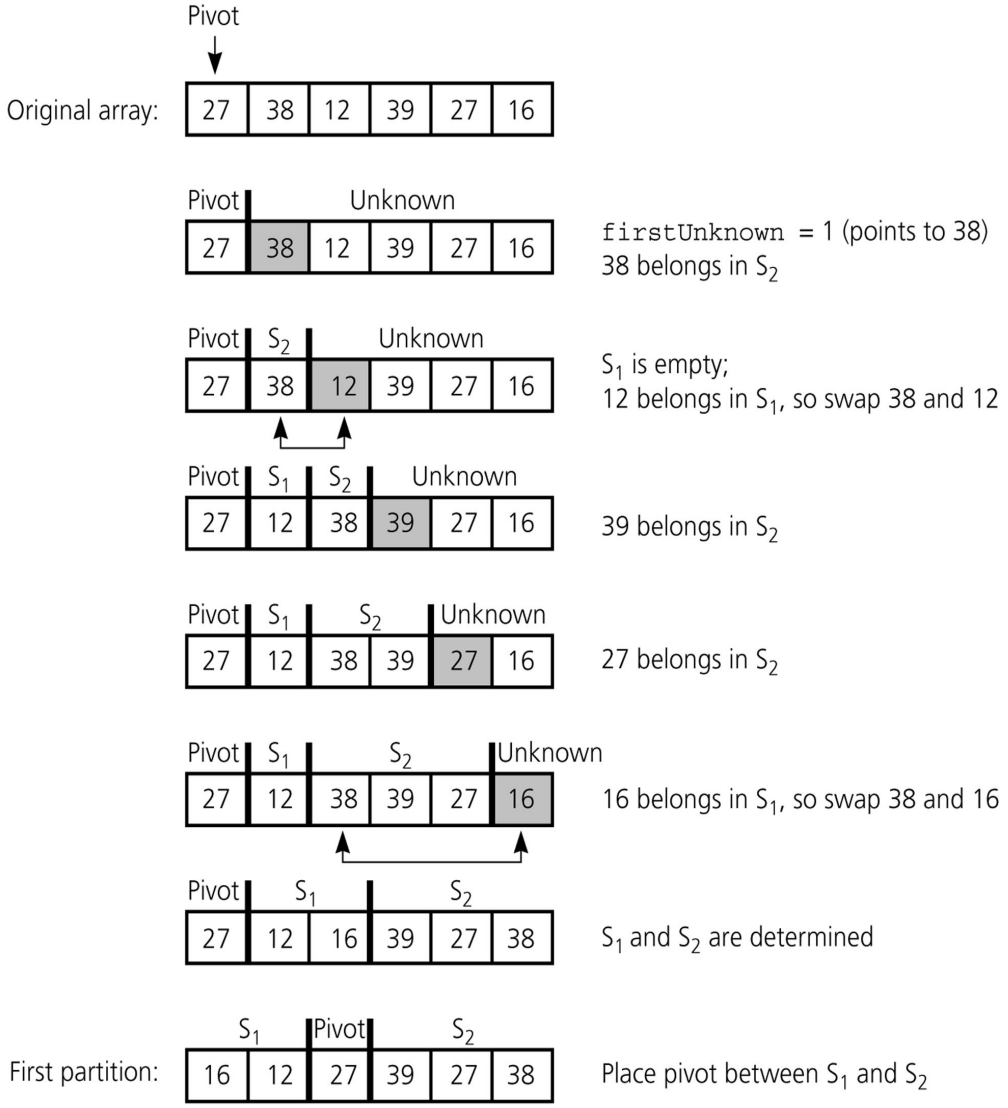
Invariant for the partition algorithm



- Rule:
  - if (arr[i+1]>p && arr[j]<p) → swap(i+1, j)
  - else
    - if (arr[i+1] <= p) → i++
    - if (arr[j] >= p) → j--
  - if (i==j) → swap(i,start), break;

Partition Function (cont.)

Developing the first partition of an array when the pivot is the first item



```
static void quick(int arr[], int start, int end){
 int p=arr[start];
 if (start>=end)
     return;
 int i=start;
 int j=end;
 while(true){
     if (i==j){
         swap(arr,i,start);
         break;
     }
     if (arr[i+1]>p && arr[j]<p){
         swap(arr,i+1,j);
     }else{
         if (arr[i+1]<=p)
             i++;
         if (arr[j]>=p)
             j--;
     }
 }
 quick(arr,start,i-1);
 quick(arr,i+1,end);
}
```

Quicksort – Analysis

Worst Case: (assume that we are selecting the first element as pivot)

- The pivot divides the list of size n into two sublists of sizes 0 and n-1.
- The number of key comparisons

  = n-1 + n-2 + ... + 1

  = $\mathbf{n^2/2 - n/2}$ ⇒ $\mathbf{O(n^2)}$
- The number of swaps =

  = n-1 + n-1 + n-2 + ... + 1

  swaps outside of the for loop swaps inside of the for loop

  = $\mathbf{n^2/2 + n/2 - 1}$ ⇒ $\mathbf{O(n^2)}$

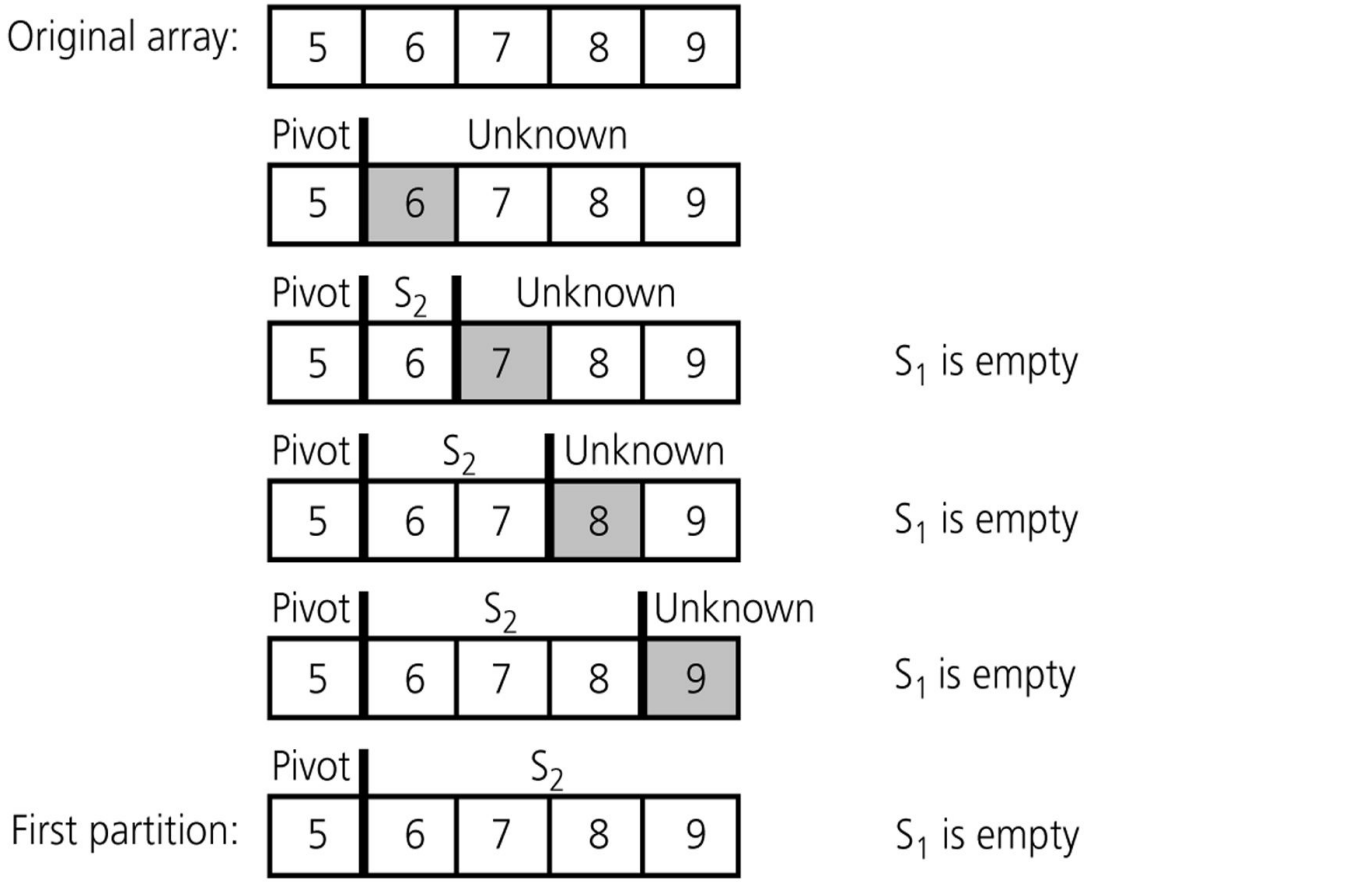
- So, Quicksort is $\mathbf{O(n^2)}$ in worst case

Quicksort – Analysis

- Quicksort is **O(n*log$_2$n)** in the best case and average case.
- Quicksort is slow when the array is sorted and we choose the first element as the pivot.
- Although the worst case behavior is not so good, its average case behavior is much better than its worst case.
  - So, Quicksort is one of best sorting algorithms using key comparisons.

Quicksort – Analysis

A worst-case partitioning with `quicksort`



Quicksort – Analysis

An average-case partitioning with `quicksort`