Sorting Algorithms

Sorting

- ***Sorting*** is a process that organizes a collection of data into either ascending or descending order.

- **Sorting Problem**:

Input: A sequence of n values $< a_1, a_2, \ldots, a_n >$

Output: A reordering $< a'_1, a'_2, \ldots, a'_n >$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

- In practice, we usually sort **records** (e.g., student records).
- Each record contains a **key** which is the value to be sorted.

Sorting

- An ***internal sort*** requires that the collection of data fit entirely in the computer's main memory.
- We can use an ***external sort*** when the collection of data cannot fit in the computer's main memory all at once but must reside in secondary storage such as on a disk.
- We will analyze only internal sorting algorithms.
- Any significant amount of computer output is generally arranged in some sorted order so that it can be interpreted.
- Sorting also has indirect uses. An initial sort of the data can significantly enhance the performance of an algorithm.
- Majority of programming projects use a sort somewhere, and in many cases, the sorting cost determines the running time.

Sorting Algorithms

- There are many sorting algorithms, such as:
  - Insertion Sort
  - Selection Sort
  - Bubble Sort
  - Merge Sort
  - Heap Sort
  - Quick Sort

- The first three are the foundations for faster and more efficient algorithms.

Sorting Algorithms

Sorting Algorithm	Applicable Data Structure	Suitable Storage Medium
Insertion Sort	Array, Linked Lists	Internal Sorting
Selection Sort	Array, Linked Lists	Internal, External Sorting
Bubble Sort	Array, Linked Lists	Internal Sorting
Merge Sort	Array, Linked Lists	Internal, External Sorting
Heap Sort	Array, Tree	Internal Sorting
Quick Sort	Array, Tree	Internal Sorting

Insertion Sort

- Insertion sort is a simple sorting algorithm that is appropriate for small inputs.
  - Most common sorting technique used by card players.
- The list is divided into two parts: sorted and unsorted.
- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and inserted at the appropriate place.
- A list of *n* elements will take at most *n-1* passes to sort the data.

Insertion Sort: Example

7	3	5	8	2	9	4	15	6	Original List

Sorted **Unsorted**

7	**3**	5	8	2	9	4	15	6	Pass 1
3	7	**5**	8	2	9	4	15	6	Pass 2
3	5	7	**8**	2	9	4	15	6	Pass 3
3	5	7	8	**2**	9	4	15	6	Pass 4
2	3	5	7	8	**9**	4	15	6	Pass 5

Insertion Sort: Example (cont.)

Sorted **Unsorted**

2	3	5	7	8	9	**4**	15	6	Pass 6
2	3	4	5	7	8	9	**15**	6	Pass 7
2	3	4	5	7	8	9	15	**6**	Pass 8
2	3	4	5	6	7	8	9	15	Pass 9

Insertion Sort Algorithm

```
void insertionSort(int D[], int n)
{
   int i, k, key;
   for (i = 1; i < n; i++)
   {
      key = D[i];

      for (k = i-1; k>=0 && key <= D[k]; k--)
         D[k+1] = D[k];   // shift operation
      D[k+1] = key;  // insert key
   }
}
```

Analysis of Insertion Sort Algorithm

Code	# of opers.	Times	total
`void insertionSort(int D[], int n)`	-	-	-
`{`	-	-	-
`int i, k, key;`	-	-	-
`for (i = 1; i < n; i++)`	1, 1, 1	1, n, n-1	2n
`{`	-	-	-
`key = D[i];`	1	n-1	n-1
`for (k = i-1;`	2	$\sum_{i=1}^{n-1} 1$	2(n-1)
`k >= 0 && key <= D[k];`	3	$\sum_{i=1}^{n-1} (i+1)$	3(n-1)n/2 +3(n-1)
`k--){`	1	$\sum_{i=1}^{n-1} i$	(n-1)n/2
`D[k+1] = D[k];`	2	$\sum_{i=1}^{n-1} i$	(n-1)n
`}`			
`D[k+1] = key;`	2	n-1	2(n-1)
`}`			
`}`			

Analysis of Insertion Sort Algorithm

- Running time depends on not only the size of the array but also the contents of the array.
- ***Best-case:*** ➔ **O(n)**
  - Array is already sorted in ascending order.
  - Inner loop will not be executed.
  - The number of moves: 0 ➔ O(1)
  - The number of key comparisons: (n-1) ➔ O(n)
- ***Worst-case:*** ➔ **$O(n^2)$**
  - Array is in reverse order:
  - Inner loop is executed i-1 times, for i = 1,2,3, …, n-1
  - The number of moves: (1+2+...+n-1)= n*(n-1)/2 ➔ $O(n^2)$
  - The number of key comparisons: (1+2+...+n-1)= n*(n-1)/2 ➔ $O(n^2)$
- ***Average-case:*** ➔ **$O(n^2)$**
  - We have to look at all possible initial data organizations.
- **So, Insertion Sort is $O(n^2)$**

Analysis of Insertion Sort Algorithm

- Which running time will be used to characterize this algorithm?
  - Best, worst or average?
- Worst:
  - Longest running time (this is the upper limit for the algorithm)
  - It is guaranteed that the algorithm will not be worse than this.
- Sometimes we are interested in average case. But there are some problems with the average case.
  - It is difficult to figure out the average case. i.e. what is average input?
  - Are we going to assume all possible inputs are equally likely?
  - In fact for most algorithms average case is same as the worst case.

Comments on Insertion Sort

- **Advantage of insertion sort:**

It is suitable to insert new elements into sorted arrays without destroying the "sorted" property of the array.

- **Disadvantage of insertion sort:**

To insert an element into the sorted part of the array, too many elements must be shifted.

→ Not suitable for external sort!

Selection Sort

- The array to be sorted is divided into two sublists, *sorted* and *unsorted*, which are divided by an imaginary wall.
- Take the first element in the array, then find the minimum value in the array.
- If the minimum value is not the first element, exchange these two values. So, the sorted part of the array has 1 element, and the unsorted part has n-1 elements.
- Take the second element in the array, and find the minimum value in the unsorted part.
- If the minimum value is not the second element, exchange these two values. So, the sorted part of the array has 2 elements, and the unsorted part has n-2 elements.
- Continue the above process until the array becomes sorted .
- A list of *n* elements requires *at most n-1* passes to completely sort the data.

Selection Sort: Example

7	3	5	8	2	9	4	15	6	Original List

Unsorted

7	3	5	8	2	9	4	15	6	Pass 1

Sorted **Unsorted**

2	3	5	8	**7**	9	4	15	6	After Pass 1
2	**3**	5	8	**7**	9	4	15	6	Pass 2

Selection Sort: Example (cont.)

Sorted **Unsorted**

2	**3**	**5**	8	7	9	**4**	15	6	Pass 3
2	**3**	**4**	**8**	7	9	**5**	15	6	Pass 4
2	**3**	**4**	**5**	**7**	9	8	15	**6**	Pass 5
2	**3**	**4**	**5**	**6**	**9**	8	15	**7**	Pass 6
2	**3**	**4**	**5**	**6**	**7**	**8**	15	9	Pass 7
2	**3**	**4**	**5**	**6**	**7**	**8**	**15**	**9**	Pass 8
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**15**	

Selection Sort Algorithm

```
void selectionSort(int D[], int n) {
   int i, index, j, min;
   for (i = 0; i < (n-1); i++) {
      min = D[n-1];
      index = n-1;
      for (j = i; j < (n-1); j++){
         if (D[j] < min){
            min = D[j];
            index = j;
         }
      }
      if (i != index){
         D[index] = D[i];
         D[i] = min;
      }
   }
}
```

Analysis of Selection Sort

- In general, we compare keys and exchange (or move) items in a sorting algorithm.

  ➔ **So, to analyze a sorting algorithm we should count the number of key comparisons and the number of exchanges or moves.**

  - Ignoring other operations does not affect our final result.

- In selectionSort function, the outer for loop executes n-1 times.
- We make exchange operation once at each iteration.

  ➔ Total # of exchanges: n-1

  ➔ Total # of Moves: 3*(n-1)

  (Each exchange has three moves)

Analysis of Selection Sort (cont.)

- The inner for loop executes the size of the unsorted part minus 1 (from 0 to n-2), and in each iteration we make one key comparison.

  ➔ # of key comparisons = 1+2+...+n-1 = n*(n-1)/2

  ➔ **So, Selection sort is O(n^2)**
- The best case, the worst case, and the average case of the selection sort algorithm are same. ➔ all of them are **O(n^2)**
  – This means that the behavior of the selection sort algorithm does not depend on the initial organization of data.
  – Since O(n^2) grows so rapidly, the selection sort algorithm is appropriate only for small n.
  – Although the selection sort algorithm requires O(n^2) key comparisons, it only requires O(n) exchanges (moves).
  – A selection sort could be a good choice if data moves are costly but key comparisons are not costly (short keys, long records).
  – If an element is in its right position, no exchange is made. So, the algorithm is suitable for nearly sorted arrays.

Comparison of *N*, *logN* and *N²*

N	O(LogN)	O(N^2)
16	4	256
64	6	4K
256	8	64K
1,024	10	1M
16,384	14	256M
131,072	17	16G
262,144	18	6.87E+10
524,288	19	2.74E+11
1,048,576	20	1.09E+12
1,073,741,824	30	1.15E+18

Bubble Sort

- It resembles the movement of waves at the sea side. At each iteration of the algorithm,
  - small values move towards the left, and
  - large values move towards the right of the array.

- It starts from the 1st element in the array. The 1st and the 2nd elements are compared, if the 1st value is greater, then these two values are exchanged.
- Then, 2nd and 3rd elements are compared. If 2nd element is greater then, these two values are exchanged.
- The above process continues until the array becomes sorted.
- Given a list of n elements, bubble sort requires up to n-1 passes to sort the data.

Bubble Sort: Example

7	3	5	8	2	9	4	15	6	Original List

In the 1st pass:

7	3	5	8	2	9	4	15	6

3	7	5	8	2	9	4	15	6

3	5	7	8	2	9	4	15	6

3	5	7	8	2	9	4	15	6

3	5	7	2	8	9	4	15	6

3	5	7	2	8	9	4	15	6

3	5	7	2	8	4	9	15	6

Bubble Sort: Example (cont.)

3	5	7	2	8	4	9	15	6

3	5	7	2	8	4	9	6	**15**

Largest element

In the 2nd pass:

3	5	7	2	8	4	9	6	**15**
3	5	7	2	8	4	9	6	**15**
3	5	7	2	8	4	9	6	**15**
3	5	2	7	8	4	9	6	**15**
3	5	2	7	8	4	9	6	**15**
3	5	2	7	4	8	9	6	**15**

Bubble Sort Algorithm

```
void bubleSort(int D[], int n)
{
   int temp, k, move;

   for (move = 0; move < (n-1); move++){

      for (k = 0; k < (n-1-move); k++){
         if (D[k] > D[k+1]){ //exchange the values
            temp = D[k];
            D[k] = D[k+1];
            D[k+1] = temp;
         }
      }
   }
}
```

Analysis of Bubble Sort

- ***Best-case:*** ➔ **$O(n^2)$**
  - Array is already sorted in ascending order.
  - The number of moves: 0 ➔ $O(1)$
  - The number of key comparisons: ➔ $O(n^2)$
  - It can be **O(n) algorithm** if a flag variable is employed to check that whether there is a move or not.

- ***Worst-case:*** ➔ **$O(n^2)$**
  - Array is in reverse order:
  - Outer loop is executed n-1 times,
  - The number of moves: 3*(1+2+...+n-1) = 3 * n*(n-1)/2 ➔ $O(n^2)$
  - The number of key comparisons: (1+2+...+n-1)= n*(n-1)/2 ➔ $O(n^2)$
- ***Average-case:*** ➔ **$O(n^2)$**
  - We have to look at all possible initial data organizations.
- **So, Bubble Sort is $O(n^2)$**

Comments on Bubble Sort

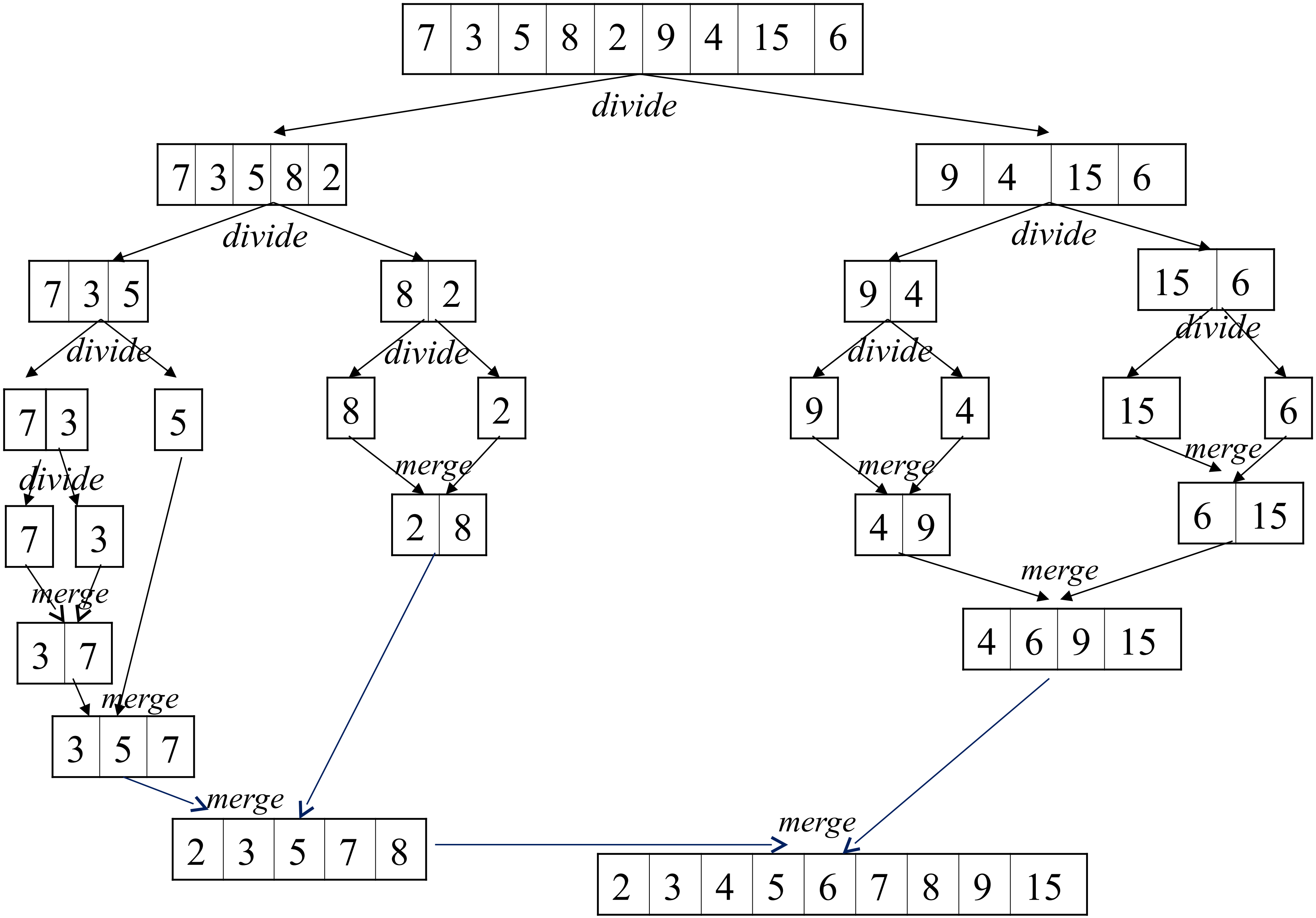
- **Advantage of bubble sort algorithm:**
  - Implementation is easy.

- **Disadvantage of bubble sort algorithm:**
  - Not efficient.
  - It can only be used for small arrays whose elements are nearly sorted.

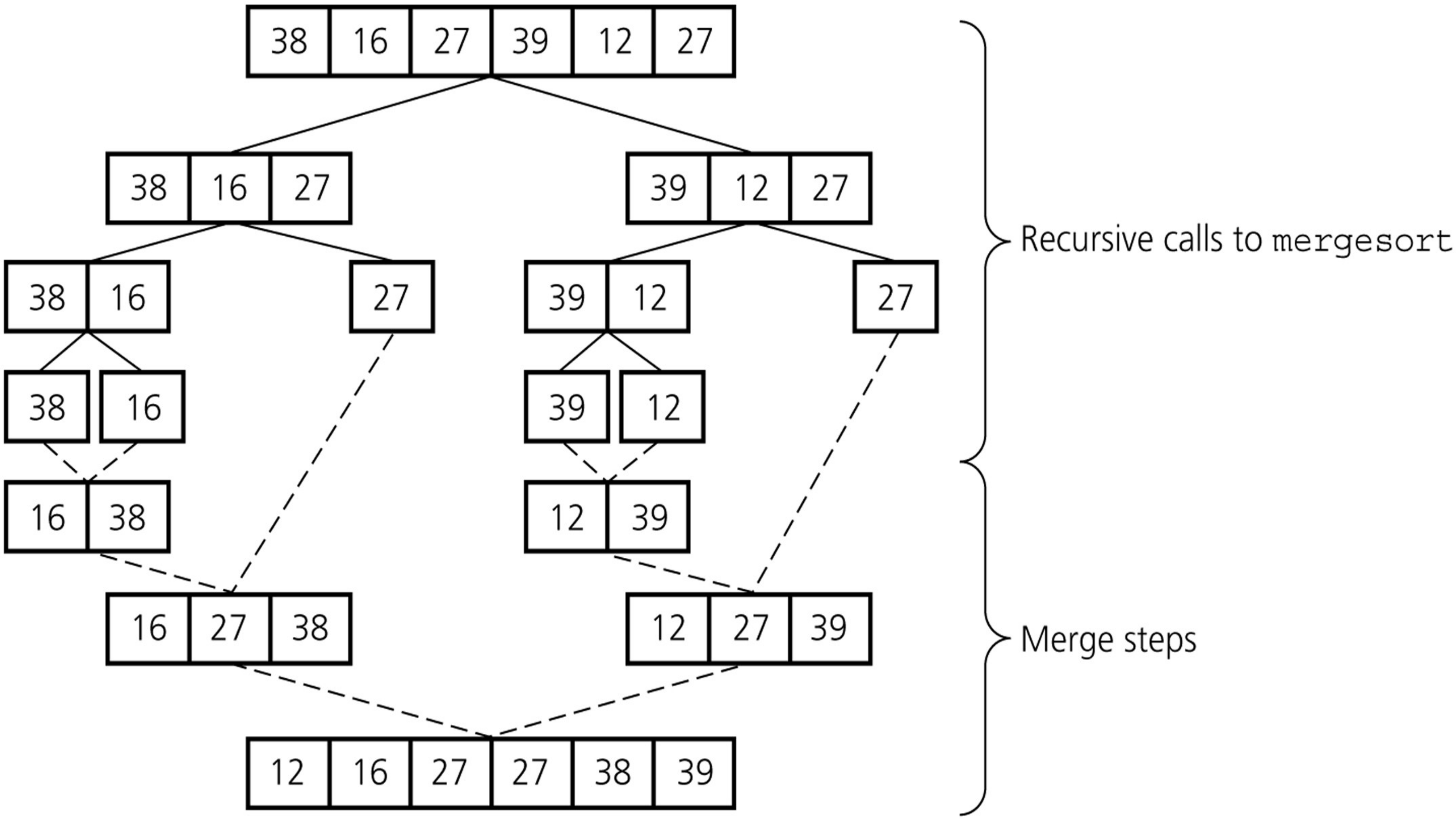
Mergesort

- Mergesort algorithm is one of two important divide-and-conquer sorting algorithms (the other one is quicksort).
- It is a recursive algorithm.
  - It divides the array into two parts,
  - Then continues to divide each part into two parts until each part has just one element.
  - After that, merges each part in sorted order until all the subparts are merged into one sorted array.

Mergesort: Example



Mergesort : Example2



Mergesort Algorithm

```
void mergesort(int D[], int left, int right) {
   int k;
   if (left < right) {
      k = (left + right)/2;  // index of midpoint
      mergesort(D, left, k);
      mergesort(D, k+1, right);

      // merge the two halves
      merge(D, left, k, right);
   }
}
```

Merge Algorithm

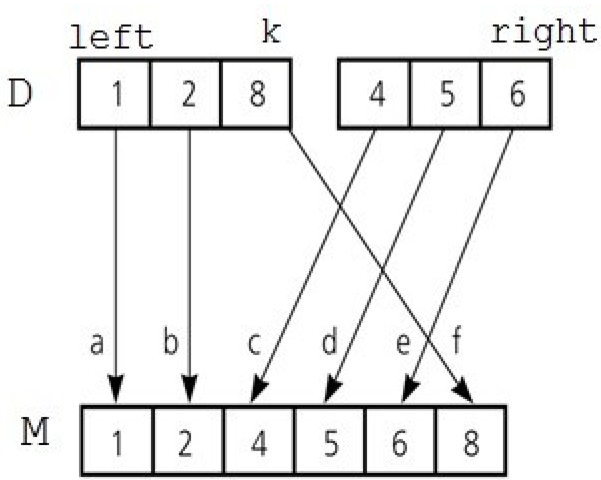
```
const int MAX_SIZE = maximum-number-of-items-in-array;

void merge(int D[], int left, int k, int right) {
   int i, j, l = 0;
   int M[MAX_SIZE];   // temporary array

   for (i=left, j=k+1; (i <= k) && (j <= right); ) {
      if (D[i] < D[j]) {
         M[l] = D[i];
         i++;
         l++;
      }
      else {
          M[l] = D[j];
          j++;
          l++
      }
   }
```

Merge Algorithm (cont.)

```
    // copy the remaining elements to M
    while (i <= k){
       M[l] = D[i];
       i++;
       l++;
    }
    while (j <= right){
       M[l] = D[j];
       j++;
       l++;
    }
    // copy M to D
    for (i = left, l = 0; i <= right; i++, l++)
       D[i] = M[l];
}
```

Analysis of Merge

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



Merge the two subarray:

a) 1 < 4, move 1 from D[left..k] to M

b) 2 < 4, move 2 from D[left..k] to M

c) 8 > 4, move 4 from D[k+1..right] to M

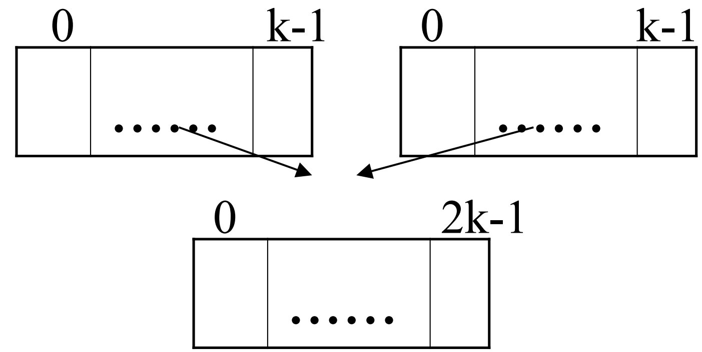
d) 8 > 5, move 5 from D[k+1..right] to M

e) 8 > 6, move 6 from D[k+1..right] to M

f) D[k+1..right] is finished, so move 8 to M

Analysis of Merge (cont.)

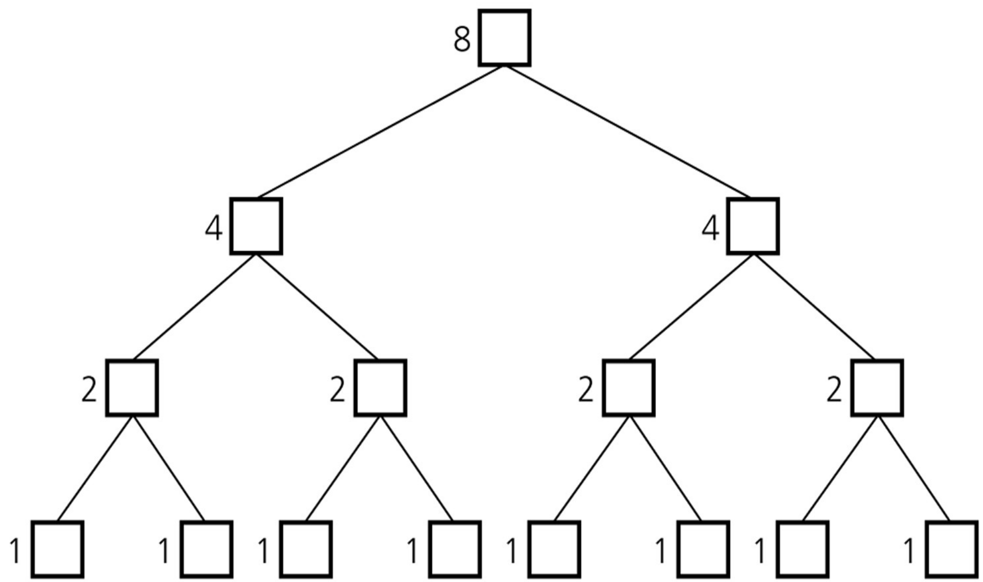
Merging two sorted arrays of size ***k***



- ***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 → O(k)
  - The number of key comparisons: k → O(k)
- ***Worst-case:***
  - The number of moves: 2k + 2k → O(k)
  - The number of key comparisons: 2k-1 → O(k)

Analysis of Mergesort

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



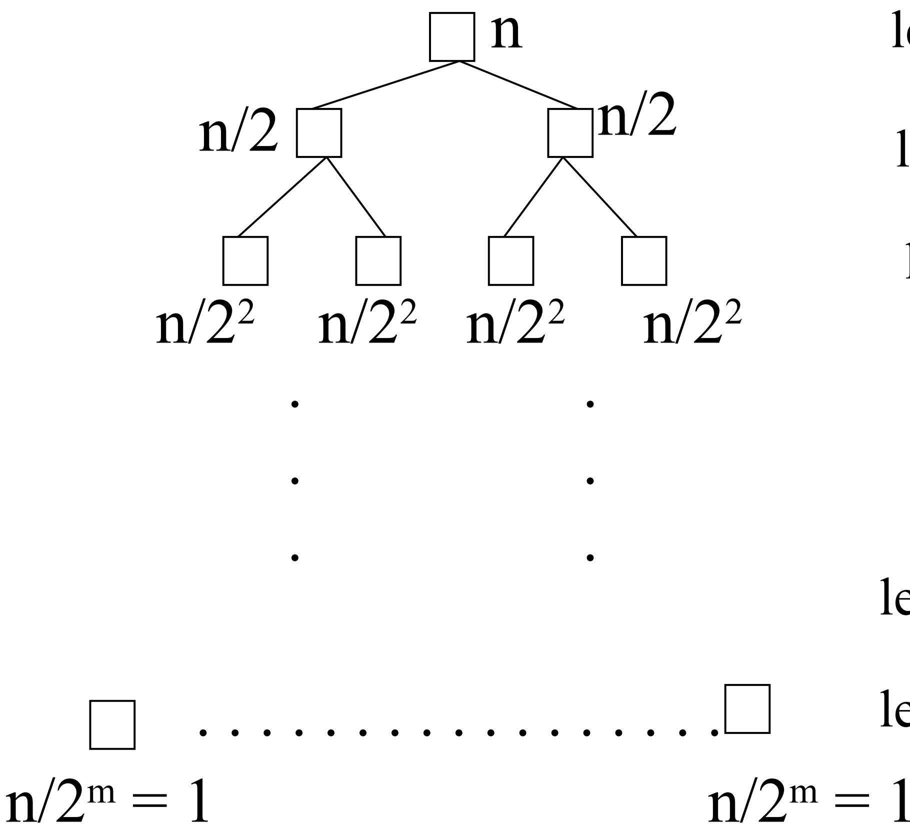
Level 0: `mergesort` 8 items

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

Analysis of Mergesort



level 0 : size n

level 1 : size $n/2 = n/2^1$

level 2 : size $n/4 = n/2^2$

level m-1 : size 2

level m: size 1

If $n/2^m = 1 \rightarrow n = 2^m \rightarrow m = \log_2 n$

Analysis of Mergesort

- ***Worst-case –***

  If $n/2^m = 1 \rightarrow n = 2^m \rightarrow m = \log_2 n$

Mergesort divides the array having n elements $\log_2 n$ times, and then merges each part.

Merge operation runs in O(n) time. Since to merge two subarrays having n/2 elements, merge operation reads and copies each subarray just once, then copies the temporary array having n elements to the original array.

So, the running time of the mergesort algorithm is

➔ $O(n * \log_2 n)$

Analysis of Mergesort

- 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))