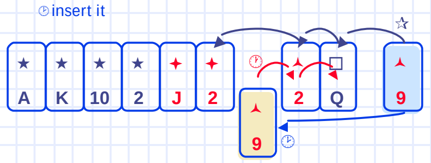
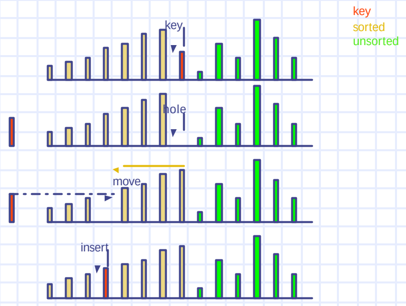
Sorting

- Card players all know how to sort …
  - First card is already sorted
  - With all the rest,
    - ☆Scan back from the end until you find the first card larger than the new one,
    - Move all the lower ones up one slot
    - insert it



Sorting - Insertion sort



Sorting - Insertion sort

- Complexity
  - For each card
    - Scan $O(n)$
    - Shift up $O(n)$
    - Insert $O(1)$
    - Total $O(n)$
  - First item requires $O(1)$, second $O(2)$, …
  - For n items $\sum_{i=1}^{n} i$ operations $\Leftarrow O(n^2)$

```
void InsertionSort(SortingArrayA) {
/* assume: typedef  enum  {false, true} Boolean; has been declared */
     int      i, j;
     KeyType  K;
     Boolean  NotFinished;
    /* For each i in the range 1:n-1, let key K be the key, A[i]. Then */
    /* insert K into the subarray A[0:i -1] in ascending order */
    for (i = 1;  i < n;  ++i)  { /* scanning */
      K = A[i];
      j = i;
      NotFinished = (A[j -1] > K);

      while (NotFinished) {
          A[j] = A[j -1];       /* move A[j -1] one space to the right */
          j--;
          if (j > 0) {
            NotFinished = (A[j -1] > K);
          } else {
             NotFinished = false;
          }
        }

       /* insert key K into hole opened up by moving previous  keys to the right */
       A[j] = K;
     }
  }
```

Sorting - Bubble

- From the first element
  - Exchange pairs if they're out of order
    - Last one must now be the largest
  - Repeat from the first to n-1
  - Stop when you have only one element to check

Bubble Sort

```
/* Bubble sort for integers */
#define SWAP(a,b)    { int t; t=a; a=b; b=t; }
void bubble( int a[], int n ) {
  int i, j;
  for(i=0;i<n;i++) { /* n passes thru the array */
     /* From start to the end of unsorted part */
     for(j=1;j<(n-i);j++) {
       /* If adjacent items out of order, swap */
          if( a[j-1]>a[j] ) SWAP(a[j-1],a[j]);
       }
     }
  }
```

$O(1)$ statement

Inner loop
n-1, n-2, n-3, … , 1 iterations

1

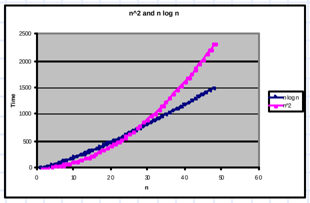
```
void BubbleSort (SortingArray A) {
  int i;
  KeyType Temp;
  Boolean NotDone;
    do {
       NotDone = false;              /* initially, assume NotDone is false */
       for (i = 0; i < n-1; ++i) {
          if (A[i] > A[i+1]) {        /* the pair (A[i], A[i+1]) is out of order */
              /* exchange A[i] and A[i + 1] to put them in sorted order */
              Temp = A[i]; A[i] = A[i + 1]; A[i + 1] =Temp;
              /* if you swapped you need another pass */
              NotDone = true;
          }
       }
    } while (NotDone);        /* NotDone == false iff no pair of keys was */
}                              /* swapped on the last pass */
```

Sorting - Simple

- Bubble sort
  - $O(n^2)$
  - Very simple code
- Insertion sort
  - Slightly better than bubble sort
    - Fewer comparisons
  - Also $O(n^2)$
- *But* HeapSort is $O(n \log n)$
- *Where would you use bubble or insertion sort?*

Simple Sorts

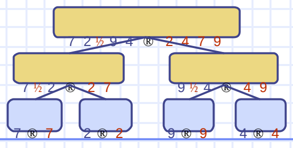
- Bubble Sort or Insertion Sort
  - Use when *n* is small
  - Simple code compensates for low efficiency!



Priority Queue Sort

```
void PriorityQueueSort(SortingArray A)
  {
      (Let Q be an initially empty output queue)
      (Let PQ be a priority queue)
       KeyType  K;
      (Organize the keys in A into a priority queue, PQ)
       while (PQ is not empty) {
          (Remove the largest key, K, from PQ)
          (Insert key, K, on the rear of output queue, Q)
       }
      (Move the keys in Q into the array A in ascending sorted order)
   }
```

Merge Sort



Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two disjoint subsets S_1 and S_2
  - Recur: solve the subproblems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S
- The base case for the recursion are subproblems of size 0 or 1
- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort
  - It uses a comparator
  - It has $O(n \log n)$ running time
- Unlike heap-sort
  - It does not use an auxiliary priority queue
  - It accesses data in a sequential manner (suitable to sort data on a disk)

Merge-Sort

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

```
function mergeSort(S, C, n)
    Input list S with n
        elements, comparator C
    Output list S sorted
        according to C
    if (n > 1){
        (S_1, S_2) = partition(S, n/2)
        mergeSort(S_1, C, n/2)
        mergeSort(S_2, C, n/2)
        S = merge(S_1, S_2)
    }
```

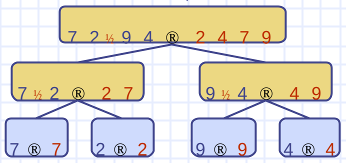
Merging Two Sorted Sequences

- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

```
function merge(A, B)
    Input list A and B with
        n/2 elements each
    Output sorted list of A ∪ B
    S = empty list
    while (!isEmpty(A) Ù !isEmpty(B))
        if (first_element(A) < first_element(B))
            insertLast(S, remove_first(A));
        else
            insertLast(S, remove_first(B));
    while (!isEmpty(A))
        insertLast(S, remove_first(A));
    while (!isEmpty(B))
        insertLast(S, remove_first(B));
    return S
```

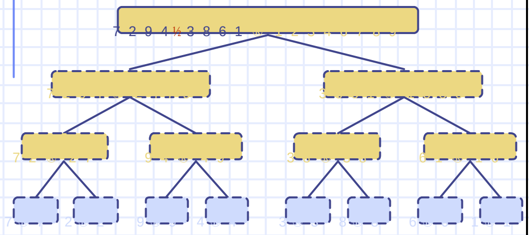
Merge-Sort Tree

- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1



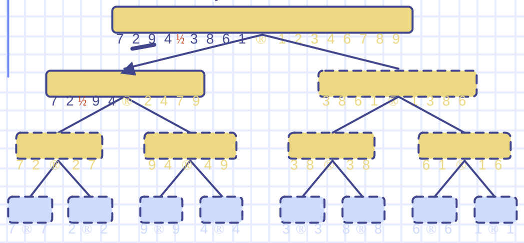
Execution Example

- Partition



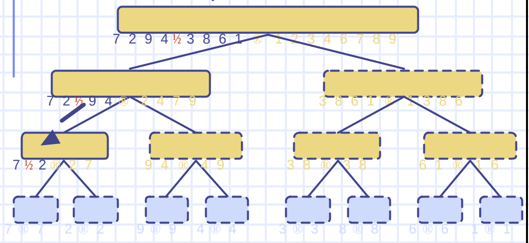
Execution Example (cont.)

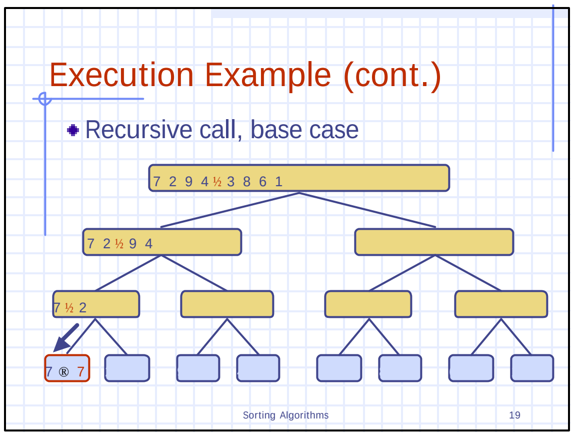
- Recursive call, partition



Execution Example (cont.)

- Recursive call, partition

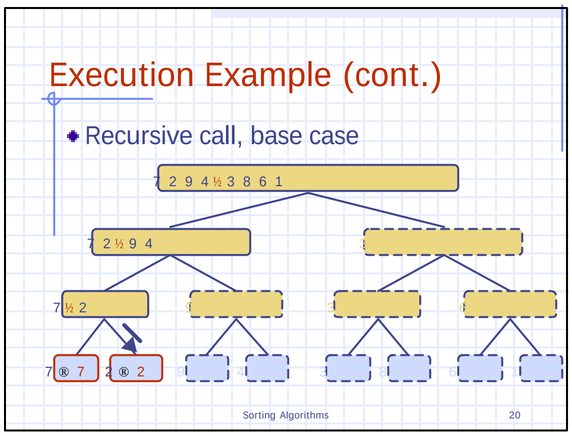




Execution Example (cont.)

- Recursive call, base case

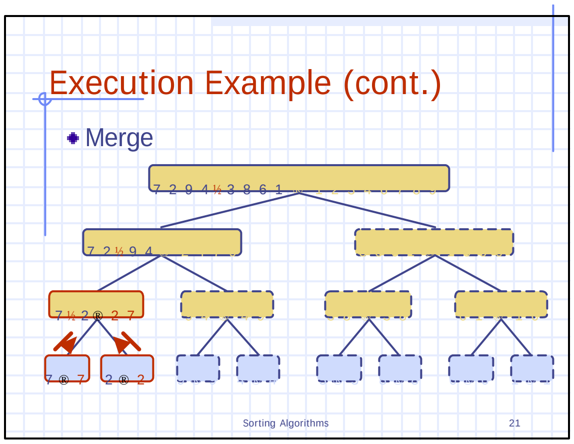
Sorting Algorithms 19



Execution Example (cont.)

- Recursive call, base case

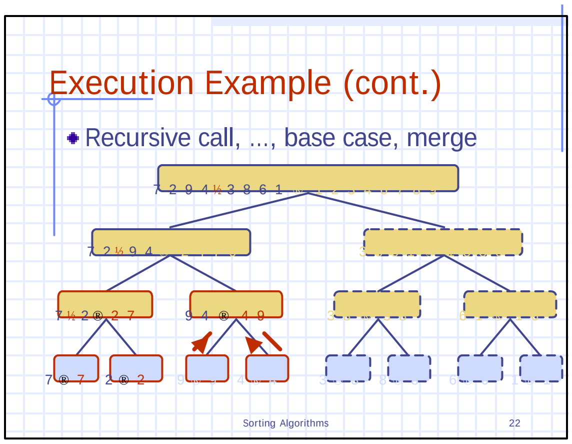
Sorting Algorithms 20



Execution Example (cont.)

- Merge

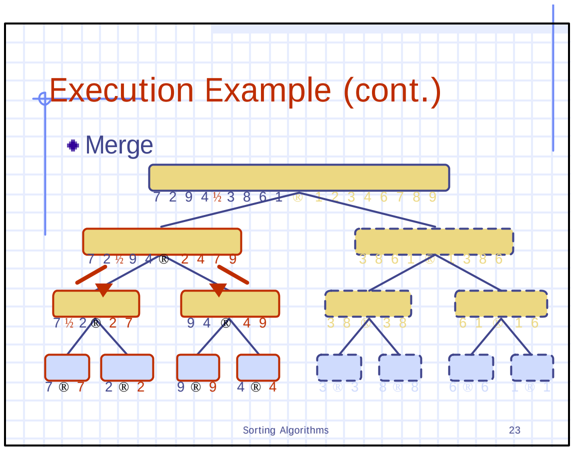
Sorting Algorithms 21



Execution Example (cont.)

- Recursive call, …, base case, merge

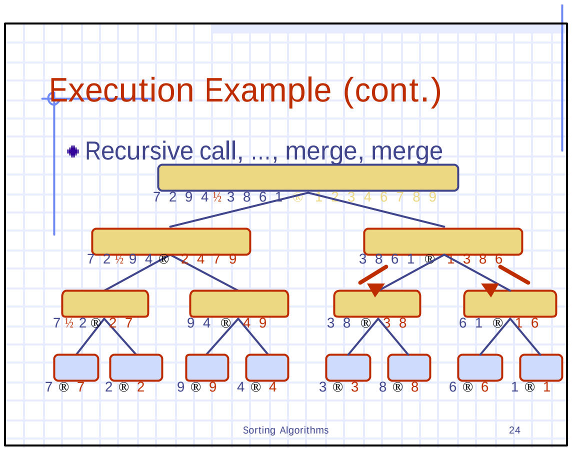
Sorting Algorithms 22



Execution Example (cont.)

- Merge

Sorting Algorithms 23



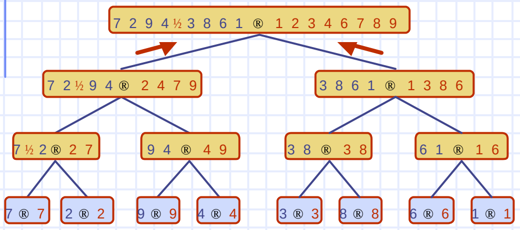
Execution Example (cont.)

- Recursive call, …, merge, merge

Sorting Algorithms 24

Execution Example (cont.)

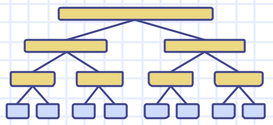
- Merge



Analysis of Merge-Sort

- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	slow in-place for small data sets (< 1K)
insertion-sort	$O(n^2)$	slow in-place for small data sets (< 1K)
heap-sort	$O(n \log n)$	fast in-place for large data sets (1K — 1M)
merge-sort	$O(n \log n)$	fast sequential data access for huge data sets (> 1M)