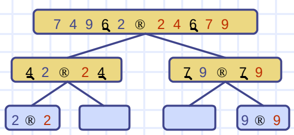
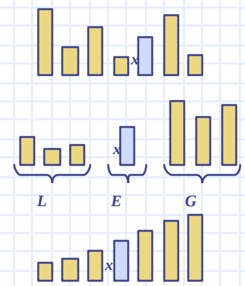
# Quick-Sort




# Quick-Sort

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element *x* (called pivot) and partition *S* into
    - *L* elements less than *x*
    - *E* elements equal *x*
    - *G* elements greater than *x*
  - Recur: sort *L* and *G*
  - Conquer: join *L*, *E* and *G*






# Quick-Sort Algorithm

```
void QuickSort(SortingArray A, int m, int n)
{  /* to sort the subarray A[m:n] of array A into ascending order */
   if (there is more than one key to sort in A[m:n]) {
       (using one of the keys in A[m:n] as a pivot key.)
       (Partition A[m:n] into a LeftPartition and a RightPartition)
       (QuickSort the LeftPartition)
       (QuickSort the RightPartition)
   }
}
```



# Quick-Sort Implementation

```
void QuickSort(SortingArray A, int m, int n) {
   int i, j;
   if (m < n) {
       i = m; j = n; /* Initially i and j point to the first and last items */
       Partition(A,&i,&j);      /* partitions A[m:n] into A[m:j] and A[i:n] */
       QuickSort(A,m,j);
       QuickSort(A,i,n);
   }
}
```



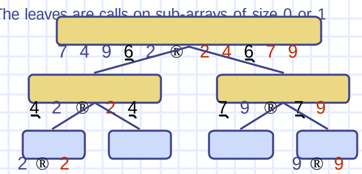
# Partition

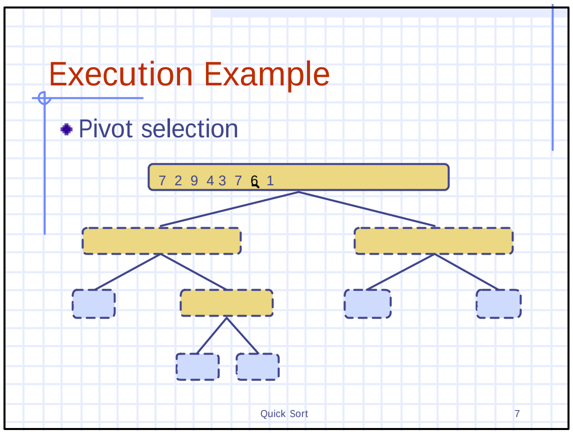
```
void Partition(SortingArray A, int *i, int *j) {
   KeyType Pivot, Temp;
   Pivot = A[ ( *i + *j ) / 2 ] ;          /* choose the middle key as the pivot */
    do {
        while (A[*i] < Pivot) (*i)++;  /* Find leftmost i such that A[i] >= Pivot.*/
        while (A[*j] > Pivot) (*j)--;  /* Find rightmost j such that A[j] <= Pivot.*/
        if (*i <= *j) {                /* if i and j didn't cross over one another */
              Temp = A[*i];            /*  swap A[i] and A[j] */
              A[*i] = A[*j];
              A[*j] = Temp;
              (*i)++;                  /* move i one space right */
              (*j)--;                  /* move j one space left */
        }
    } while (*i <= *j);       /* while the i and j pointers haven't crossed yet */
}
```

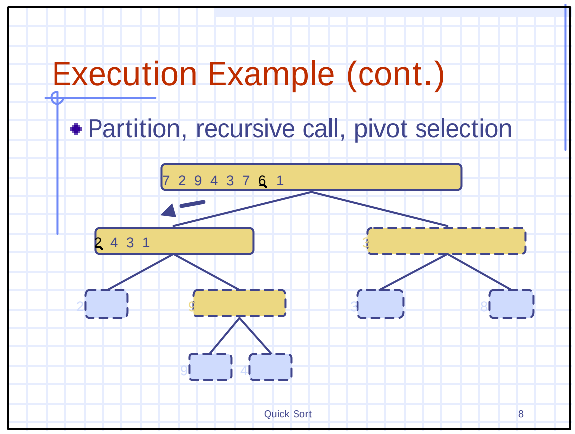


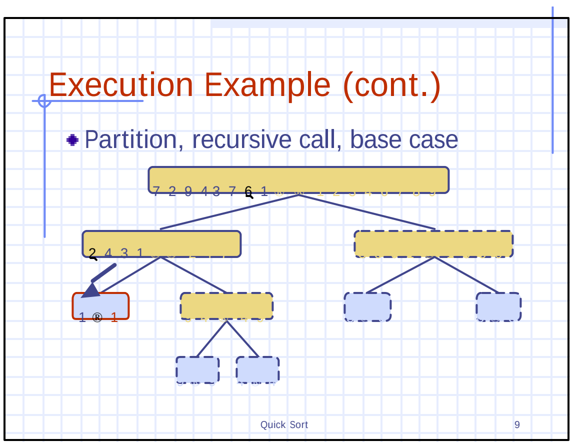
# Quick-Sort Tree

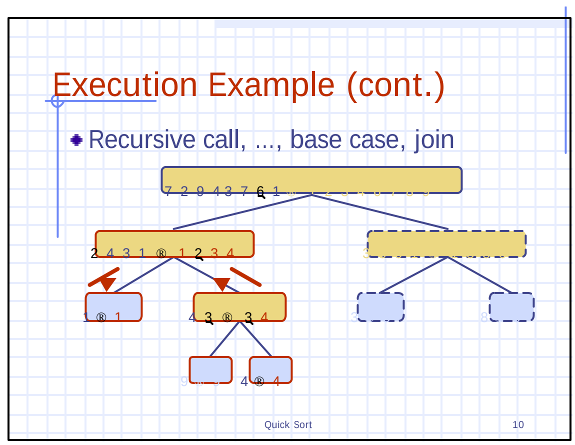
- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted array before the execution and its pivot
    - Sorted array at the end of the execution
  - The root is the initial call
  - The leaves are calls on sub-arrays of size 0 or 1

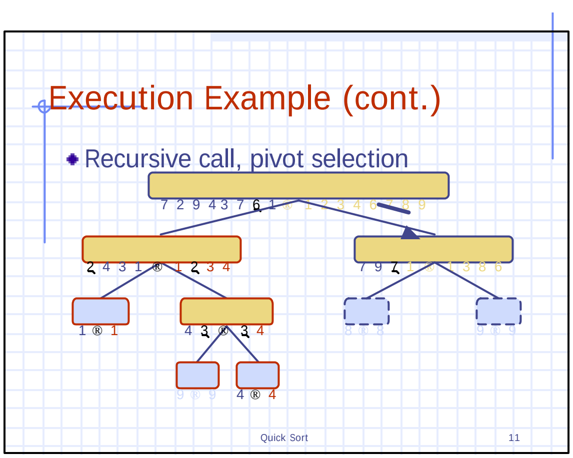


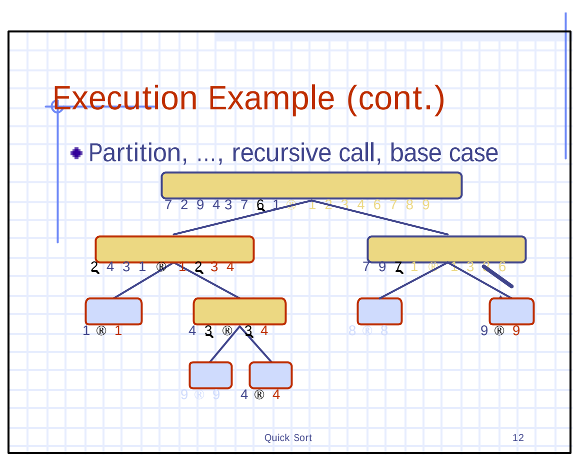

Execution Example
Pivot selection
7 2 9 4 3 7 6 1
Quick Sort
7


Execution Example (cont.)
Partition, recursive call, pivot selection
7 2 9 4 3 7 6 1
Quick Sort
8


Execution Example (cont.)
Partition, recursive call, base case
Quick Sort
9

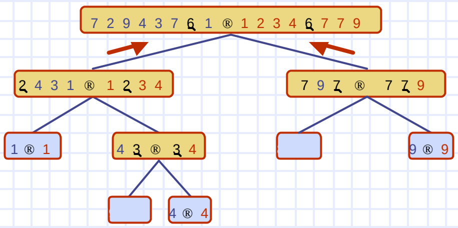

Execution Example (cont.)
Recursive call, ..., base case, join
Quick Sort
10


Execution Example (cont.)
Recursive call, pivot selection
Quick Sort
11


Execution Example (cont.)
Partition, ..., recursive call, base case
Quick Sort
12

## Execution Example (cont.)

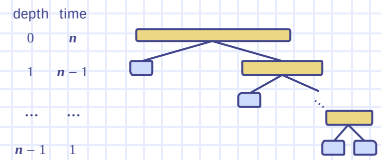
- Join, join






## Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of $L$ and $G$ has size $n - 1$ and the other has size $0$
- The running time is proportional to the sum
  $$n + (n - 1) + \ldots + 2 + 1$$
- Thus, the worst-case running time of quick-sort is $O(n^2)$






## Expected Running Time

- Consider a recursive call of quick-sort on an array of size $s$
  - Good call: the sizes of $L$ and $G$ are each less than $3s/4$
  - Bad call: one of $L$ and $G$ has size greater than $3s/4$
- A call is good with probability $1/2$
- Probabilistic Fact: The expected number of coin tosses required in order to get $k$ heads is $2k$
- Hence, for a node of depth $i$, we expect that
  - $i/2$ parent nodes are associated with good calls
  - the size of the input sequence for the current call is at most $(3/4)^{i/2} n$
- Thus, we have
  - For a node of depth $2\log_{4/3} n$, the expected size of the input sequence is one
  - The expected height of the quick-sort tree is $O(\log n)$
- The overall amount or work done at the nodes of the same depth of the quick-sort tree is $O(n)$
- Thus, the expected running time of quick-sort is $O(n \log n)$



## 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) |



## Bucket-Sort

- Let be $S$ be an array of $n$ (key, element) items with keys in the range $[0, N - 1]$
- Bucket-sort uses the keys as indices into an auxiliary array $B$ of buckets
  - Phase 1: Empty array $S$ by moving each item $(k, o)$ into its bucket $B[k]$
  - Phase 2: For $i = 0, \ldots, N - 1$, move the items of bucket $B[i]$ to array $S$
- Analysis:
  - Phase 1 takes $O(n)$ time
  - Phase 2 takes $O(n + N)$ time

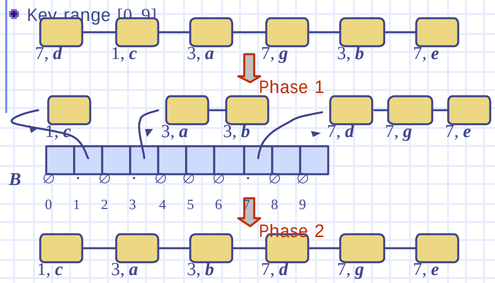
  Bucket-sort takes $O(n + N)$ time

```
function bucketSort(S, N, n)
    Input array S of n (key, element)
        items with keys in the range
        [0, N – 1]
    Output array S sorted by
        increasing keys
    B ← array of N buckets
    for (i = 0; i < n; i++){
        B[k].insertLast(S[i])
    for (i = 0 ; i < N – 1 ; i++)
        while (!B[i].isEmpty()){
            f = B[i].removefirst();
            S[i] = f;
    }
```



## Example

- Key range [0, 9]



## Properties and Extensions

- Key-type Property
  - The keys are used as indices into an array and cannot be arbitrary objects
- Stable Sort Property
  - The relative order of any two items with the same key is preserved after the execution of the algorithm

Extensions

- Integer keys in the range $[a, b]$
  - Put item $(k, o)$ into bucket $B[k - a]$
- String keys from a set $D$ of possible strings, where $D$ has constant size (e.g., names of the 50 U.S. states)
  - Sort $D$ and compute the rank $r(k)$ of each string $k$ of $D$ in the sorted sequence
  - Put item $(k, o)$ into bucket $B[r(k)]$



## Stable Sort

- Stable Sort Property
  - The relative order of any two items with the same key is preserved after the execution of the algorithm
  - Why do we need stable sort?

Example: an array of student record

Requirement:

(1) Sort the student array wrt student last name

(2) Sort the student array again wrt to final grade (for students with the same grade, must maintain the "last name" alphabet order)



## Lexicographic Order

- A $d$-tuple is a sequence of $d$ keys $(k_1, k_2, \dots, k_d)$, where key $k_i$ is said to be the $i$-th dimension of the tuple
- Example:
  - The Cartesian coordinates of a point in space are a 3-tuple
- The lexicographic order of two $d$-tuples is recursively defined as follows

$$(x_1, x_2, \dots, x_d) < (y_1, y_2, \dots, y_d)$$
$$\Leftrightarrow$$
$$x_1 < y_1 \lor x_1 = y_1 \land (x_2, \dots, x_d) < (y_2, \dots, y_d)$$

i.e., the tuples are compared by the first dimension, then by the second dimension, etc.



## Lexicographic-Sort

- Let $C_i$ be the pointer to a comparator function that compares two tuples by their $i$-th dimension
- Let *stableSort*$(S, C)$ be a stable sorting algorithm that uses comparator $C$
- Lexicographic-sort sorts an array of $d$-tuples in lexicographic order by executing $d$ times algorithm *stableSort*, one per dimension
- Lexicographic-sort runs in $O(dT(n))$ time, where $T(n)$ is the running time of *stableSort*

```
function lexicographicSort(S)
    Input array S of d-tuples
    Output array S sorted in
        lexicographic order

    for i ← d downto 1
        stableSort(S, C_i)
```

Example:

(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)

(2, 1, 4) (3, 2, 4) (5,1,5) (7,4,6) (2,4,6)

(2, 1, 4) (5,1,5) (3, 2, 4) (7,4,6) (2,4,6)

(2, 1, 4) (2,4,6) (3, 2, 4) (5,1,5) (7,4,6)