cmpt-225

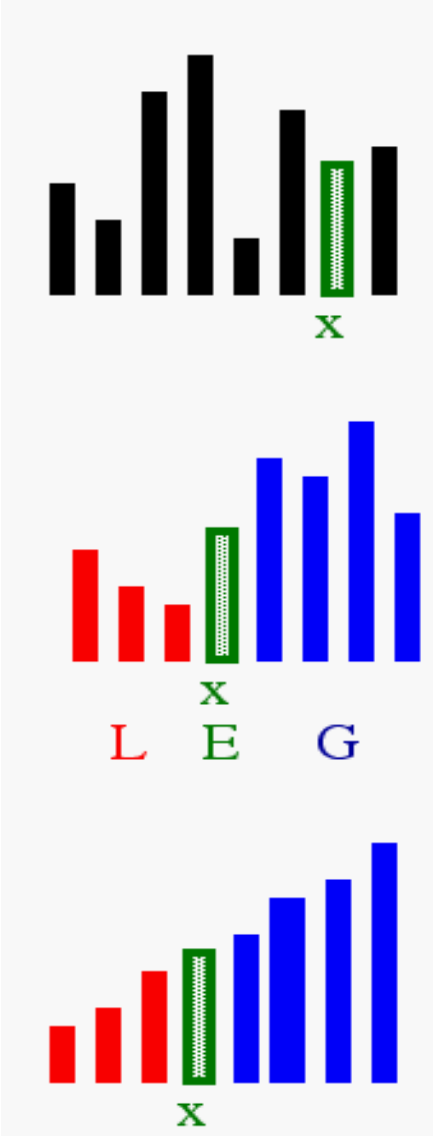
Sorting – Part two

Idea of Quick Sort

1) **Select**: pick an element

2) **Divide**: partition elements so that x goes to its final position E

3) **Conquer**: recursively sort left and right partitions

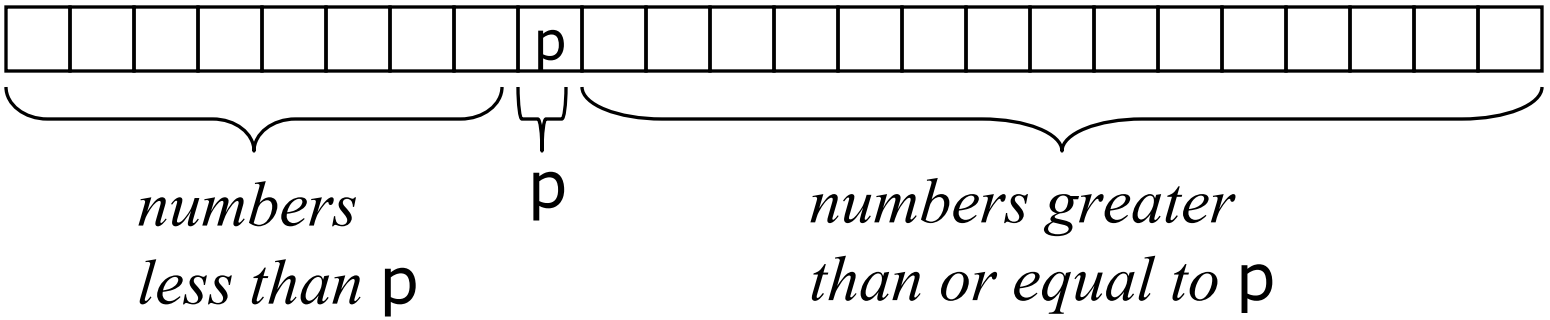


Quick Sort - algorithm

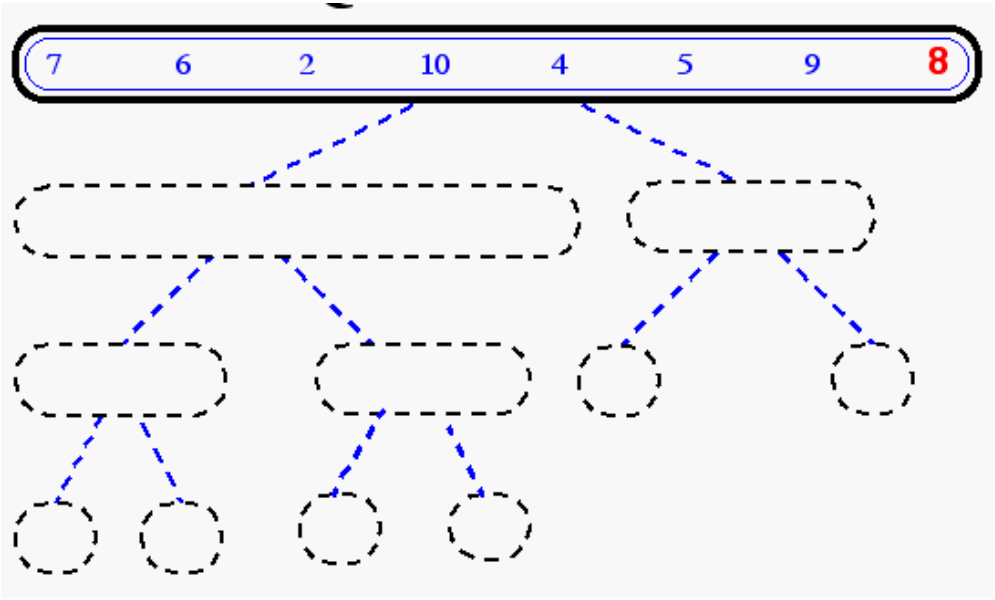
```
public void QuickSort(Comparable[] arr, int low, int
   high) {
     if (low <= high) // if size <= 1 already sorted
       return;
     else // size is 2 or larger
     {
       // partition range
       int pivotIndex = partition(arr, low, high);
       // sort left subarray
       QuickSort(arr, low, pivotIndex - 1);
       // sort right subarray
       QuickSort(arr, pivotIndex + 1, high);
     }
   }
```

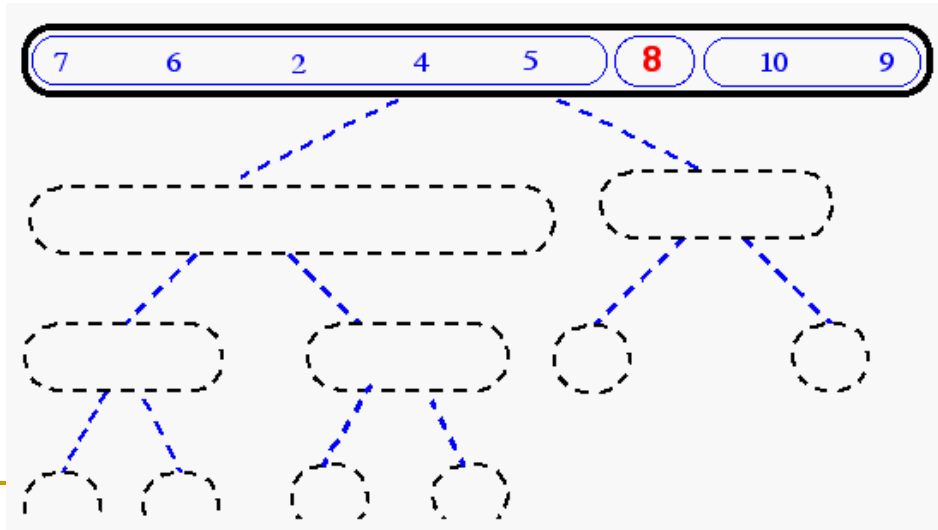
Quick Sort - Partitioning

- A key step in the Quick sort algorithm is partitioning the array
  - We choose some (any) number p in the array to use as a pivot
  - We partition the array into three parts:



Quick Sort – Partitioning

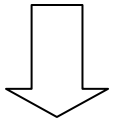




Partitioning using an extra array.

85	24	63	45	17	31	96	50

Put the smaller elements in front of the temporary array and the larger element in the back

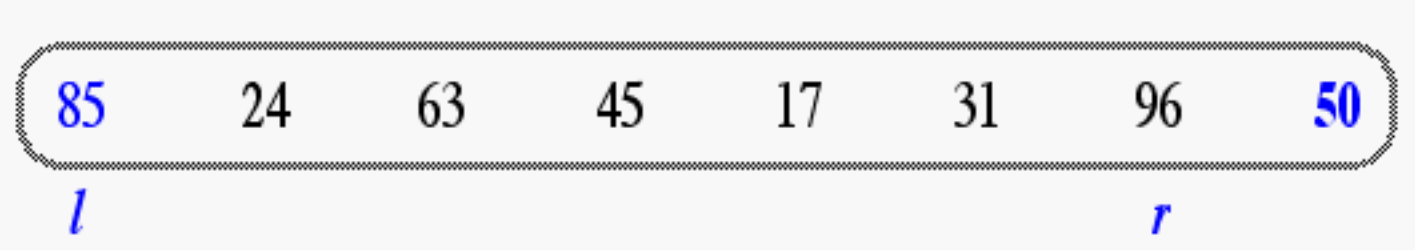


24	45	17	31	50	96	63	85

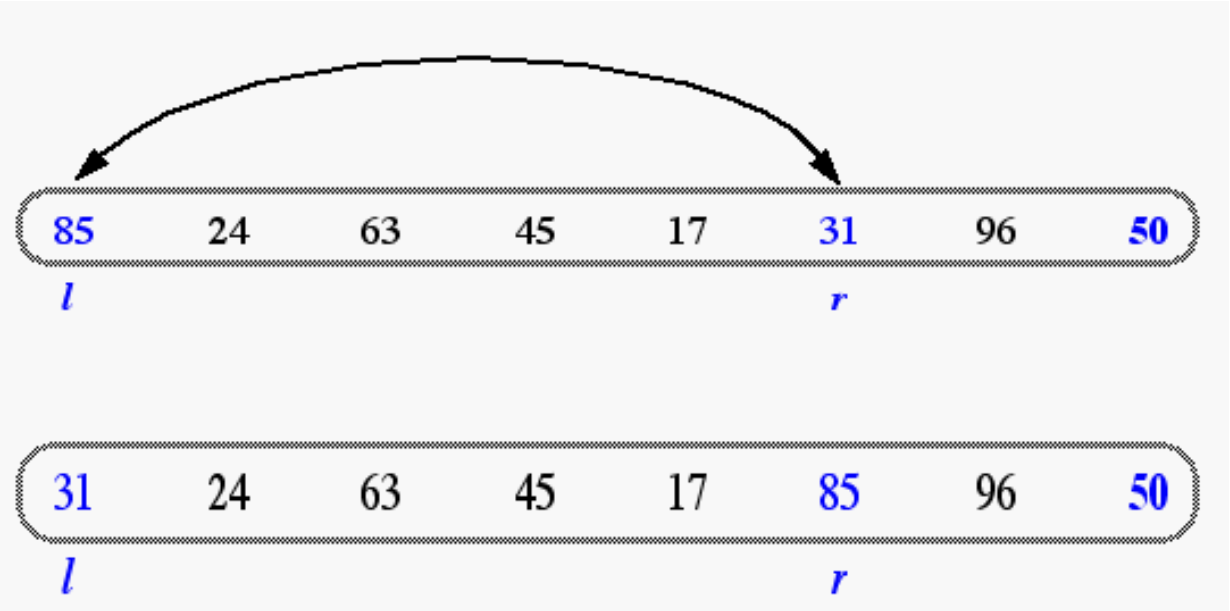
Return the temporary array

Quick Sort – Partitioning – algorithm

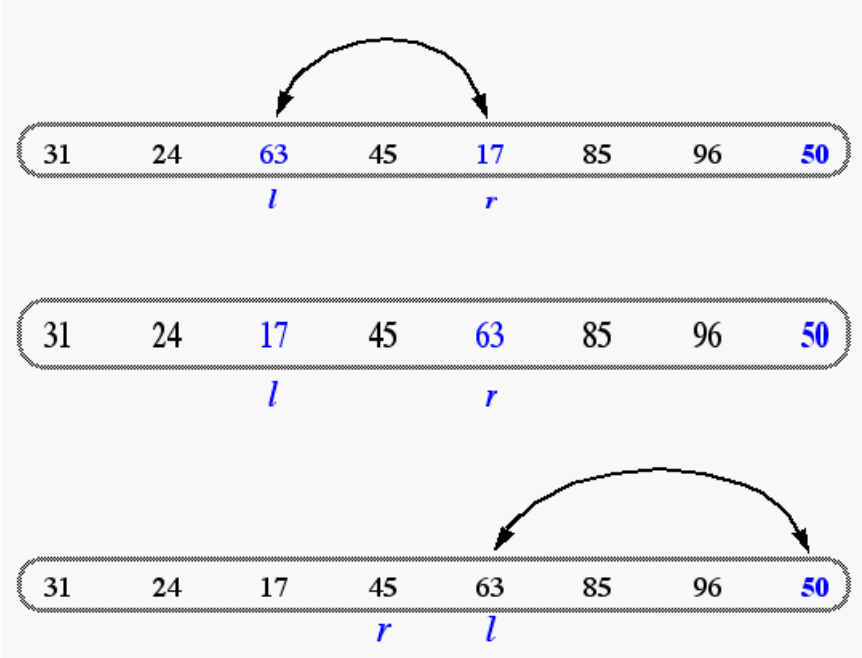
Index *l* scans the sequence from the left, and index *r* from the right.



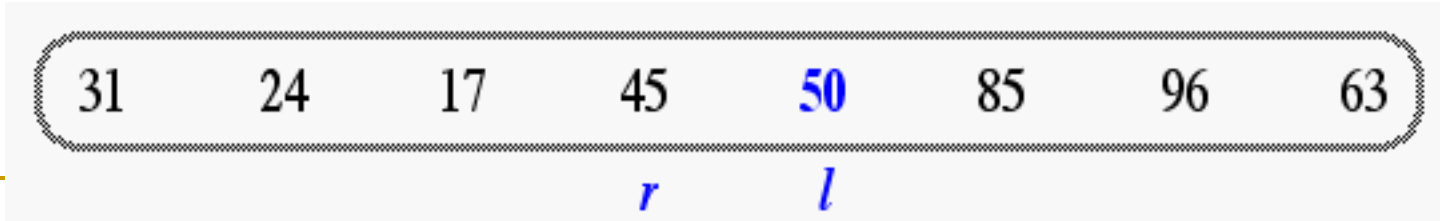
Increment *l* until *arr[l]* is larger than the pivot; and decrement *r* until *arr[r]* is smaller than the pivot.
Then swap elements indexed by *l* and *r*.
Repeat until whole array is processed.



Quick Sort – Partitioning – algorithm



A final swap with the pivot completes the partitioning.



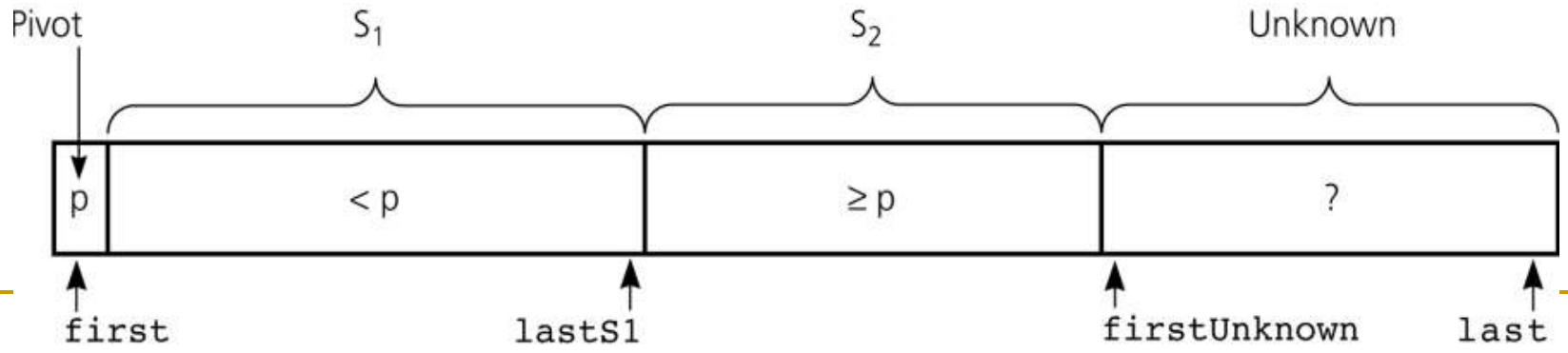
Quick Sort – Partitioning – algorithm

```
public int partition(Comparable[] arr, int low, int high) {
    Comparable pivot = arr[high]; // choose pivot
    int l = low;
    int r = high-1;
    while (l<=r) {
    // find bigger item on the left
      while (l<=r && arr[l].compareTo(pivot) <= 0)
      l++;

      // find smaller item on the right
      while (l<=r && arr[r].compareTo(pivot) >= 0)
      r--;

      if (l<r) {
        swap(arr[l], arr[r]);
        l++;
        r--;
      }
    }
    // put pivot to the correct location
    swap(arr[l], arr[high]);
    return r;
  }
```

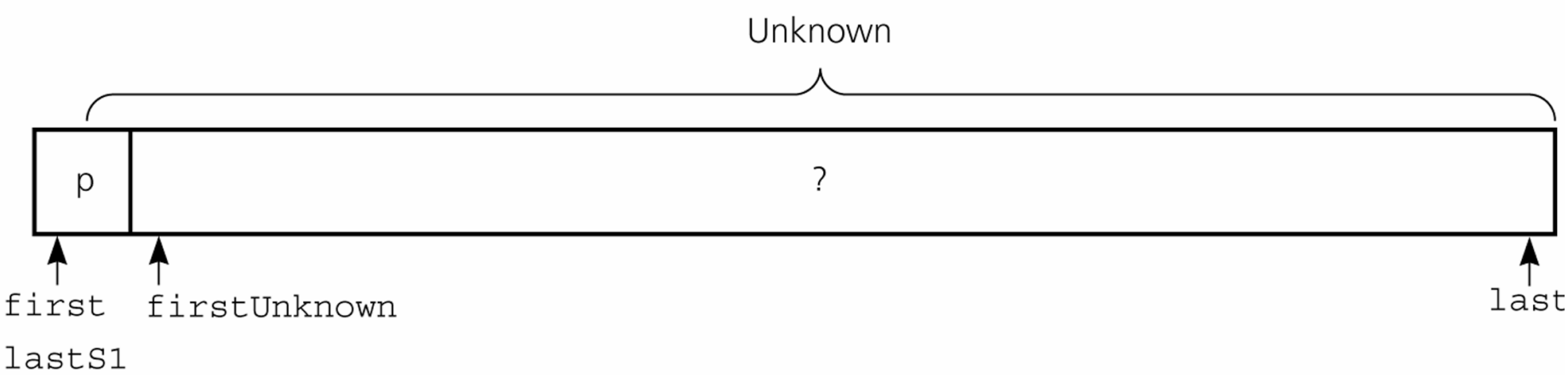
Quick Sort – Partitioning – another algorithm (textbook)

- Pivot is chosen to be the first element of the array (does not really matter)
- The array is divided to 4 parts (see bellow), initially “<p” and “>=p” parts are empty
- Invariant for the partition algorithm:

  The items in region S_1 are all less than the pivot, and those in S_2 are all greater than or equal to the pivot
- In each step the first element in “?” part is added either to “<p” or “>=p” part.



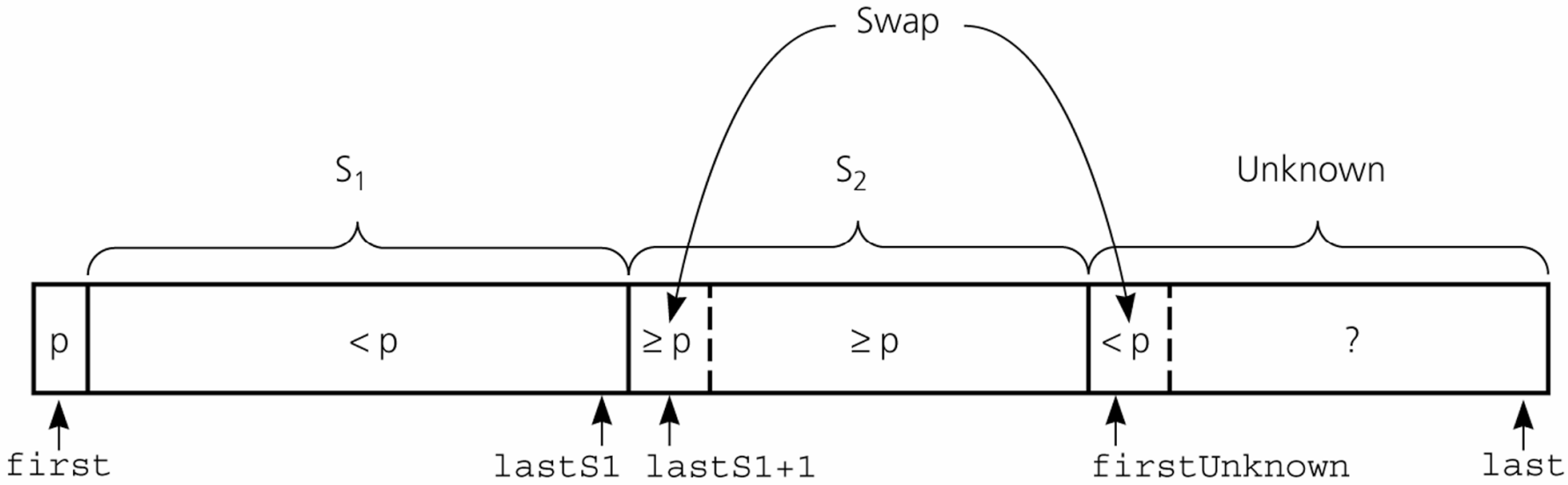
Quick Sort – Partitioning – another algorithm (textbook)



S_1: arr[first+1]..arr[lastS1]) empty
S_2: arr[lastS1+1]..arr[firstUnknown-1]) empty
?: arr[firstUnknown]..arr[last]) all elements but pivot

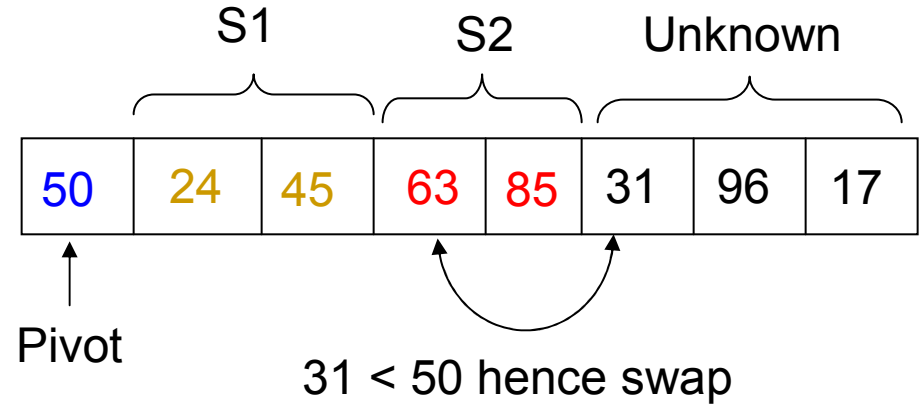
Initial state of the array

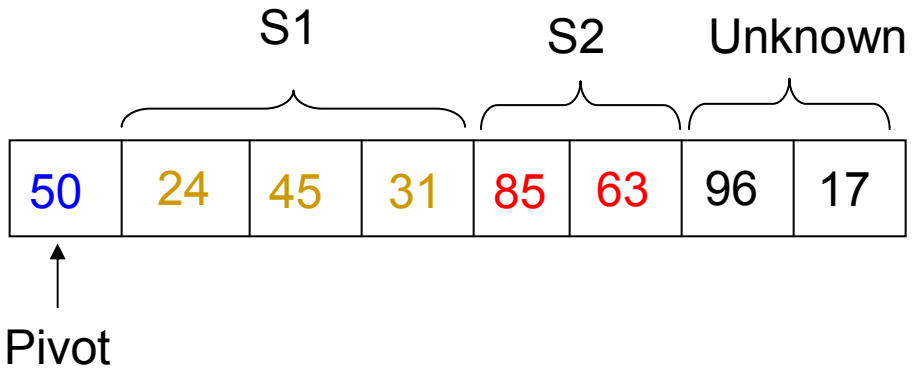
Quick Sort – Partitioning – another algorithm (textbook)

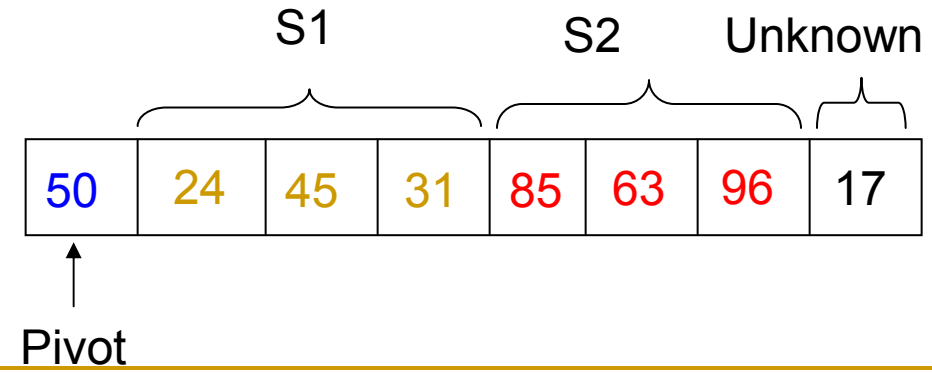


Processing arr[firstUnknown]: case "< pivot"

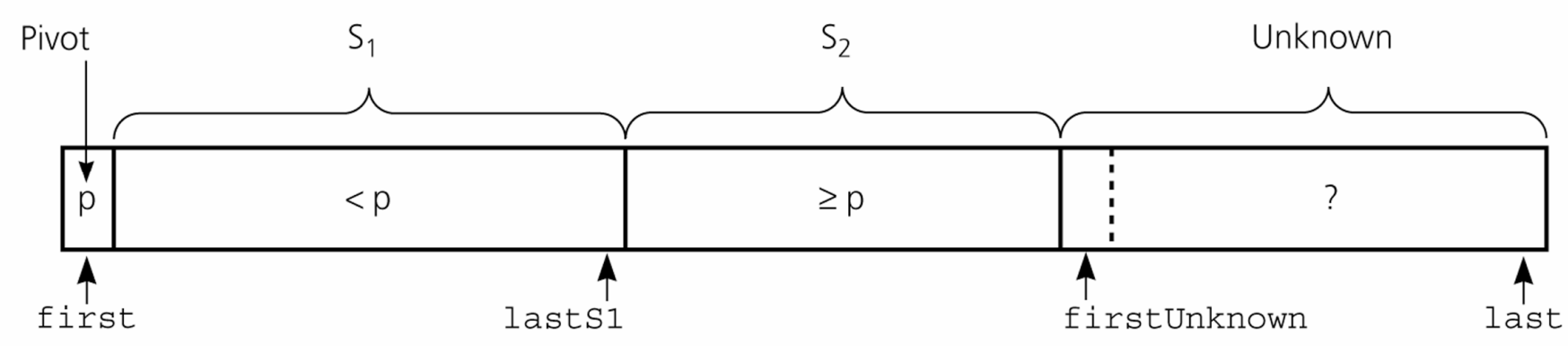
Move *arr[firstUnknown]* into S_1 by swapping it with *theArray[lastS1+1]* and by incrementing both *lastS1* and *firstUnknown.*


S1
S2
Unknown
50
24
45
63
85
31
96
17
Pivot
31 < 50 hence swap


S1
S2
Unknown
50
24
45
31
85
63
96
17
Pivot


S1
S2
Unknown
50
24
45
31
85
63
96
17
Pivot

Quick Sort – Partitioning – another algorithm (textbook)



Processing arr[firstUnknown]: case “>= pivot”

Moving *theArray[firstUnknown]* into S_2 by incrementing *firstUnknown.*

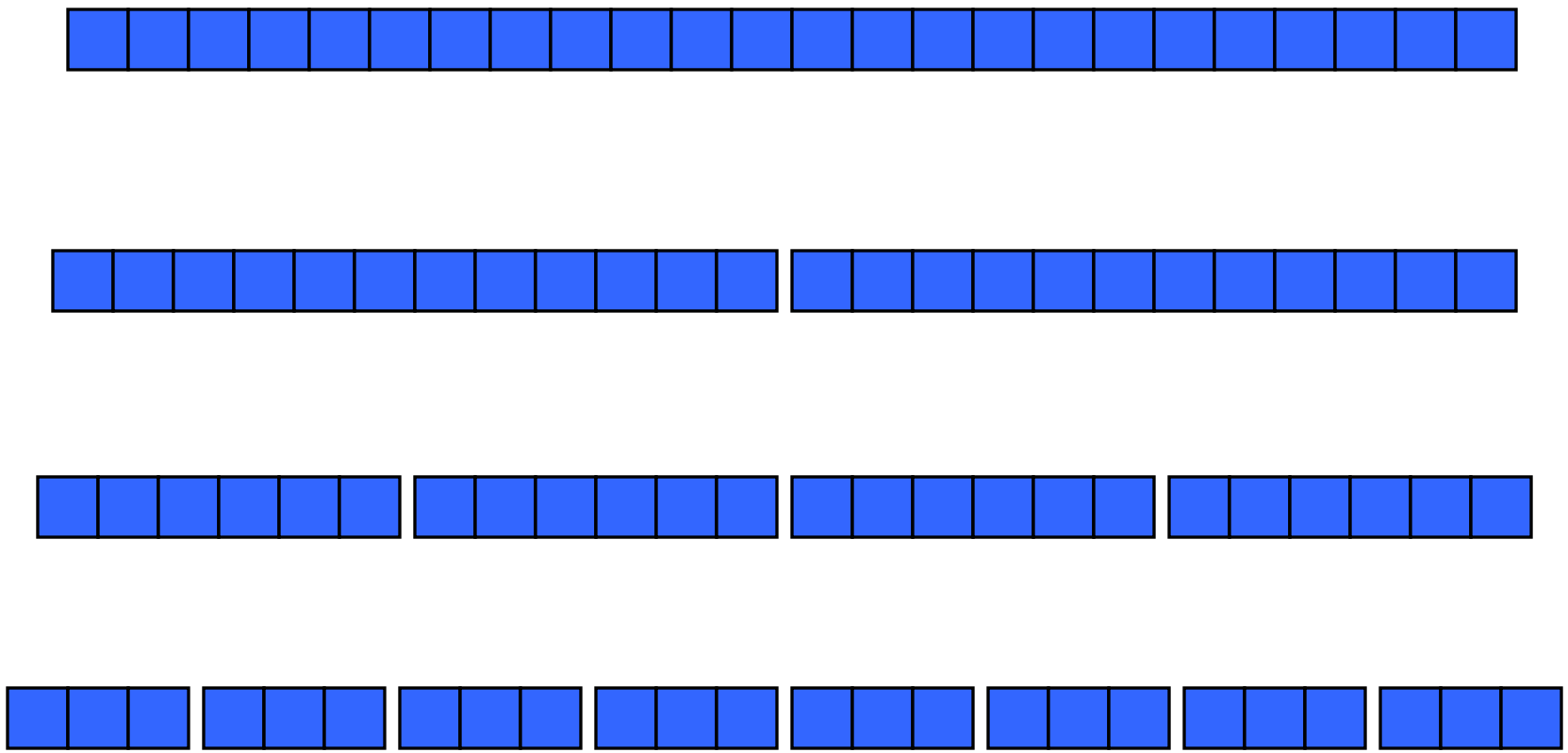
Quick Sort – Selection of pivot

- In the above algorithm we selected the pivot to be the last or the first element of subarray which we want to partition
- It turns out that the selection of pivot is crucial for performance of Quick Sort – see best and worst cases
- Other strategies used:
  - select 3 (or more elements) and pick the median
  - randomly select (especially used when the arrays might be originally sorted)
  - select an element “close to the median” in the subarray

Analysis of Quick Sort: Best Case

- How much time do we need to partition an array of size *n*?
- O(n) using any of two algorithms
- **Best case:** Suppose each partition operation divides the array almost exactly in half

Best case Partitioning at various levels



Analysis of Quick Sort: Best Case

- How much time do we need to partition an array of size *n*?
- O(n) using any of two algorithms
- **Best case:** Suppose each partition operation divides the array almost exactly in half
- When could the best case happen?
- For example, array was sorted and the pivot is selected to be the middle element of the subarray.

Analysis of Quick Sort: Best Case

- **Best case:** Suppose each partition operation divides the array almost exactly in half
- The running time (time cost) can be expressed with the following recurrence:

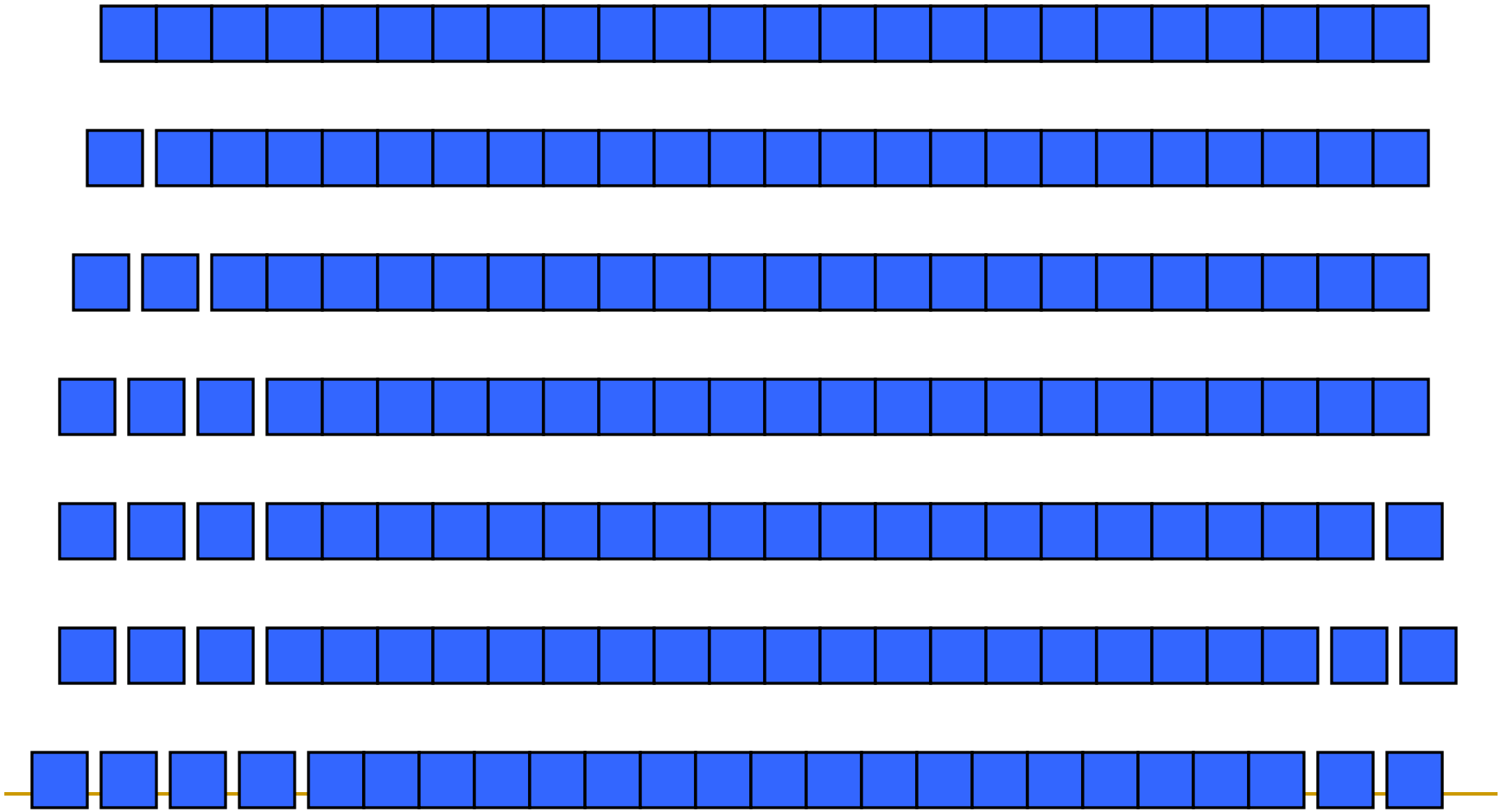
$$T(n) = 2.T(n/2) + T(\text{partitioning array of size } n)$$
$$= 2.T(n/2) + O(n)$$

- The same recurrence as for merge sort, i.e., T(n) is of order O(n.log n).

Analysis of Quick Sort: Worst Case

- In the worst case, partitioning always divides the size `n` array into these three parts:
  - A length one part, containing the pivot itself
  - A length zero part, and
  - A length `n-1` part, containing everything else

Worst case partitioning



Analysis of Quick Sort: Worst Case

- In the worst case, partitioning always divides the size `n` array into these three parts:
  - A length one part, containing the pivot itself
  - A length zero part, and
  - A length `n-1` part, containing everything else
- When could this happen?
- Example: the array is sorted and the pivot is selected to be the first or the last element.

Analysis of Quick Sort: Worst Case

- The recurrent formula for the time cost of Quick Sort in the worst case:
  $T(n) = T(0) + T(n-1) + O(n)$
  $= T(n-1) + O(n)$

- By repeated substitution (or Master's theorem) we get the running time of Quick Sort in the worst case is $O(n^2)$
- Similar, situation as for Insertion Sort. Does it mean that the performance of Quick Sort is bad on average?

Quick Sort: Average Case

- If the array is sorted to begin with, Quick sort running time is terrible: $O(n^2)$
  (Remark: could be improved by random selection of pivot.)
- It is possible to construct other bad cases
- However, Quick sort runs *usually* (on average) in time $O(n.\log_2 n)$
  -> CMPT307 for detailed analysis
- The constant in front of $n.\log_2 n$ is so good that Quick sort is generally the fastest algorithm known.
- Most real-world sorting is done by Quick sort.

Exercise Problem on Quick Sort.

1. What is the running time of QUICKSORT when

   a) All elements of array A have the same value ?

   b) The array A contains distinct elements and in sorted decreasing order ?

Answer – 1st algorithm

- Pivot is chosen to be the last element in the subarray.

  a) Whatever **pivot** you choose in each subarray it would result in WORST CASE PARTITIONING (*l*=*high*) and hence the running time is $O(n^2)$.

  b) Same is the case. Since you always pick the minimum element in the subarray as the **pivot** each partition you do would be a worst case partition and hence the running time is $O(n^2)$ again !

Answer – 2nd algorithm

- Pivot is chosen to be the first element in the subarray

  a) Whatever **pivot** you choose in each subarray it would result in WORST CASE PARTITIONING (everything will be put to S_2 part) and hence the running time is $O(n^2)$.

  b) Same is the case. Since you always pick the maximum element in the sub array as the **pivot** each partition you do would be a worst case partition and hence the running time is $O(n^2)$ again !

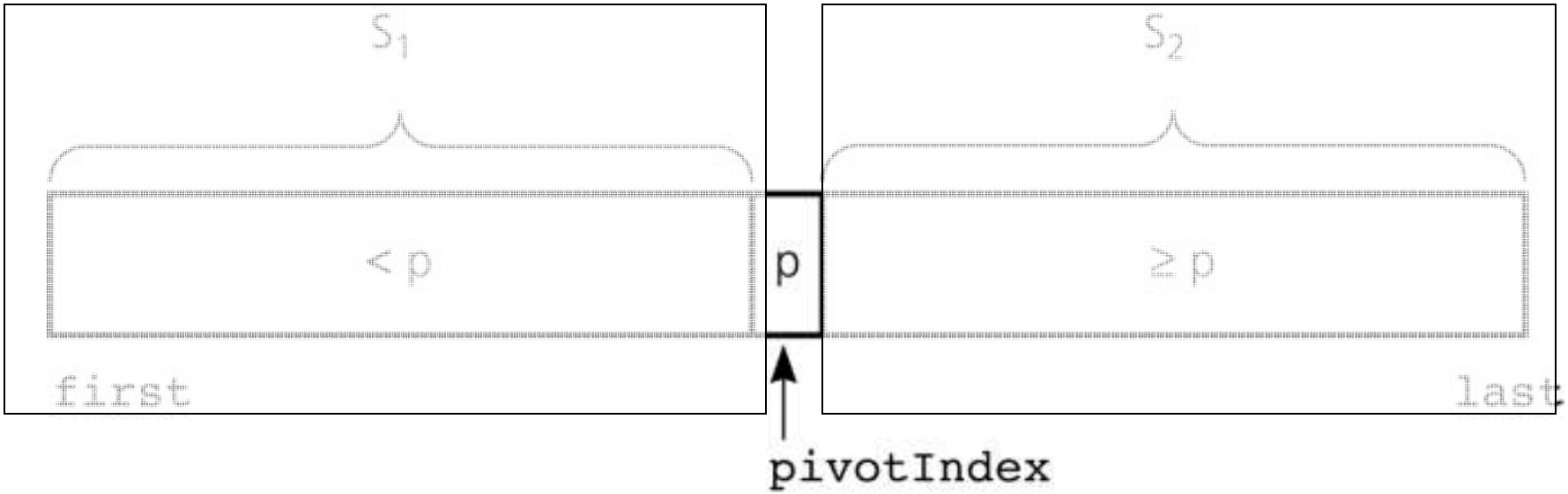
Finding the k-th Smallest Element in an Array (Selection Problem)

- One possible strategy: sort an array and just take the k-th element in the array
- This would require O(n.log n) time if use some efficient sorting algorithm

- Question: could we use partitioning idea (from Quicksort)?

Finding the k-th Smallest Element in an Array

- Assume we have partition the subarray as before.



If S1 contains k or more items -> S1 contains kth smallest item
If S1 contains k-1 items -> k-th smalles item is pivot *p*
If S1 contains fewer then k-1 items -> S2 contains kth smallest item

Finding the k-th Smallest Element in an Array

```
public Comparable select(int k, Comparable[] arr, int low, int high)
// pre: low <= high  and
//      k <= high-low+1 (number of elements in the subarray)
// return the k-th smallest element
// of the subarray arr[low..high]
{
    int pivotIndex = partition(arr, low, high);
    // Note: pivotIndex - low is the local index
    //   of pivot in the subarray
    if (k == pivotIndex - low + 1) {
          // the pivot is the k-th element of the subarray
          return arr[pivotIndex];
    } else if (k < pivotIndex - low + 1) {
          // the k-th element must be in S1 partition
          return select(k, arr, low, pivotIndex-1);
    } else { // k > pivotIndex - low +1
          // the k-th element must be in S2 partition
          // Note: there are pivotIndex-first elements in S1
          //       and one pivot, i.e., all smaller than
          //       elements in S2, so we have to recalculate
          //       index k
          return select(k - (pivotIndex-first+1), arr, pivotIndex+1, high);
    }
} // end kSmall
```

Finding the k-th Smallest Item in an Array

- The running time in the best case:
  $T(n) = T(n/2) + O(n)$
- It can be shown with repeated substitution that T(n) is of order O(n)

- The running time in the worst case:
  $T(n) = T(n-1) + O(n)$
- This gives the time $O(n^2)$

- average case: O(n)
- By selecting the pivot close to median (using a recursive linear time algorithm), we can achieve O(n) time in the worst case as well.

Radix Sort

- Treats each data item as a character string.
- The data items are all of the same length K, if not pad the shorter strings with blank characters (or 0 digits) on the right (on the left).

  AC, CBC, JKP, O, CF
  ACb, CBC, JKP, Obb, CFb //padding with blanks

  12, 345, 5, 678, 226
  012, 345, 005, 678, 026 //padding with 0's

- Idea: forming groups and combine them.

AB**C**, XY**Z**, BW**Z**, AA**C**, RL**T**, JB**X**, RD**T**, KL**T**, AEO, TL**J**

- Group based on the last character:

C: AB**C**, AA**C**
J: TL**J**
O: AE**O**
T: RT**L**, RD**T**, KL**T**
X: JB**X**
Z: XY**Z**, BW**Z**

- Combine the groups**.**

A**B**C, A**A**C, T**L**J, A**E**O, R**T**L, R**D**T, K**L**T, J**B**X, X**Y**Z, B**W**Z

- Regroup based on second character

(A**A**C) (A**B**C, J**B**X) (R**D**T) (A**E**O) (T**L**J, R**L**T, K**L**T) (B**W**Z) (X**Y**Z)

- Combine groups base on first character

AAC, **A**BC, **J**BX, **R**DT, **A**EO, **T**LJ, **R**LT, **K**LT, **B**WZ, **X**YZ

- Regroup based on first character

(**A**AC, **A**BC, **A**EO)(**B**WZ)(**J**BX)(**K**LT)(**R**DT, **R**LT)(**T**LJ)(**X**YZ)

- Combine groups

AAC, ABC, AEO, BWZ, JBX, KLT, RDT, RLT, TLJ, XYZ

Complexity of Radix sort.

- We do the grouping and combining procedure K times, where K is the length of the strings.
- Each grouping and combining pass takes O(n) time.
- The whole sort takes O(Kn) time.
- If K is constant the Radix sort is O(n).

A Comparison of Sorting Algorithms

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	n * log n	n * log n
Quicksort	n^2	n * log n
Radix sort	n	n
Treesort	n^2	n * log n
Heapsort	n * log n	n * log n

Approximate growth rates of time required for eight sorting algorithms