Data Structures and Algorithms

Chapter 12

Sorting
Merge-Sort

- A divide-and-conquer algorithm

- Divide:
  - If input size is smaller than a certain threshold, solve it using a straightforward method.
  - Otherwise, divide the input into two or more subproblems.
- Conquer: Solve the subproblems recursively.
- Combine: Merge solutions to subproblems to generate a solution to the original problem.

Sorting
Merge-Sort

- Outline of the algorithm:
  1. Divide: If S has zero or one element, return S (because it is already sorted). Otherwise, divide S into two separate arrays, S_1 and S_2, of approximately equal size. S_1 contains the first $\lfloor n/2 \rfloor$ elements of S and S_2 contains the remaining $\lceil n/2 \rceil$ elements.
  2. Conquer: Sort S_1 and S_2 recursively.
  3. Combine: Put the elements back to S by merging the sorted sequences S_1 and S_2 into a sorted sequence.

Sorting
Merge-Sort

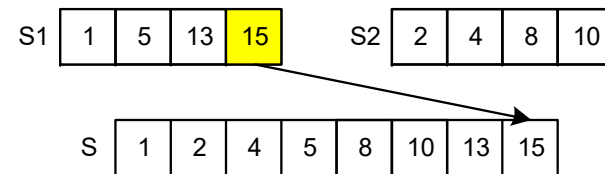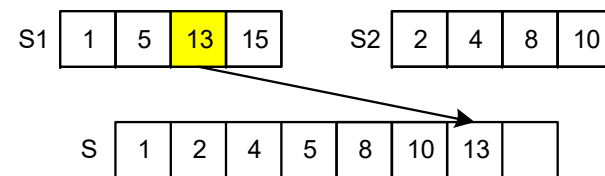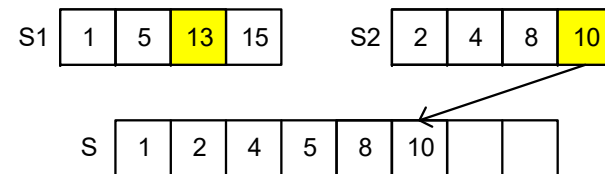- Illustration
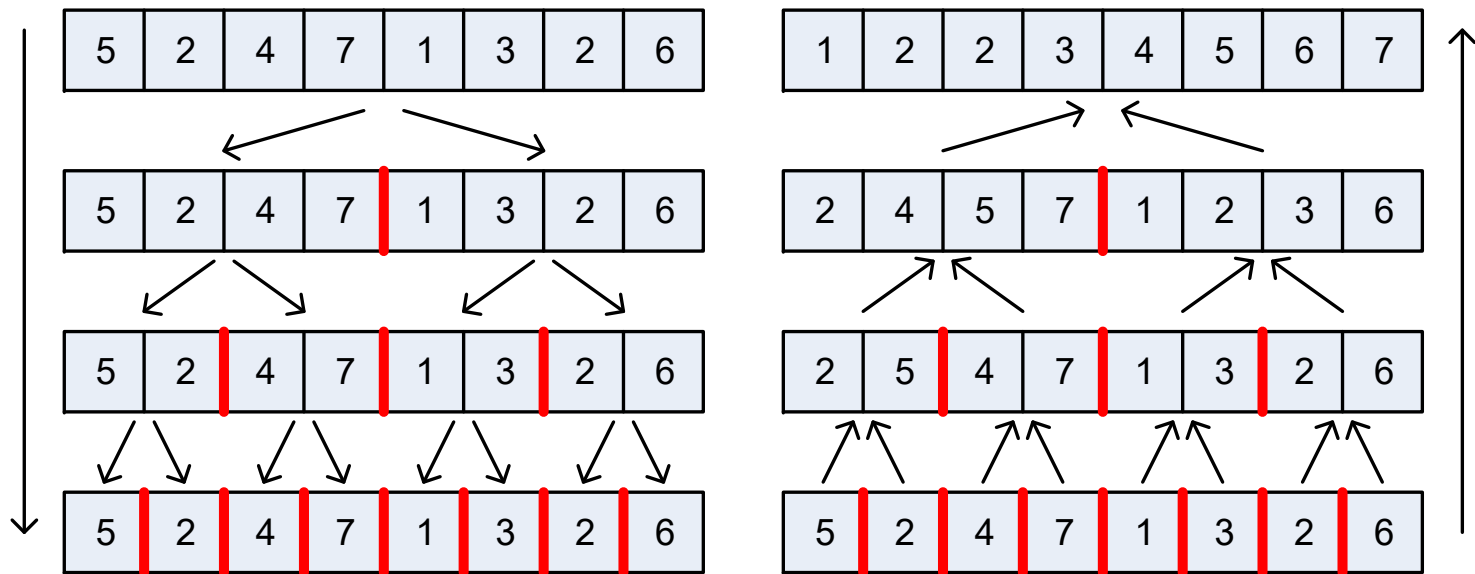
Sorting
Merge-Sort

- Array-based implementation

```
1  public static <K> void merge(K[] S1, K[] S2, K[] S, Comparator<K> comp) {
2    int i = 0, j = 0;
3    while (i + j < S.length) {
4      if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))
5        S[i+j] = S1[i++];       // copy ith element of S1 and increment i
6      else
7        S[i+j] = S2[j++];       // copy jth element of S2 and increment j
8    }
9  }
```

- Running time: $O(n)$

Sorting
Merge-Sort

- Merge
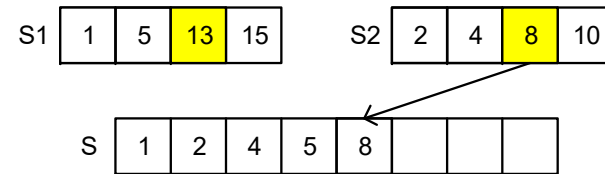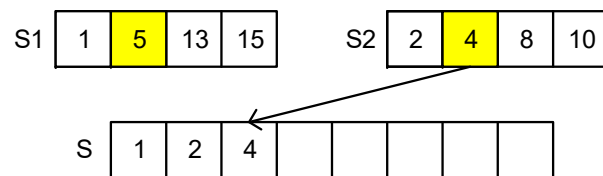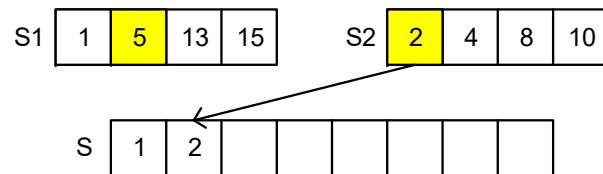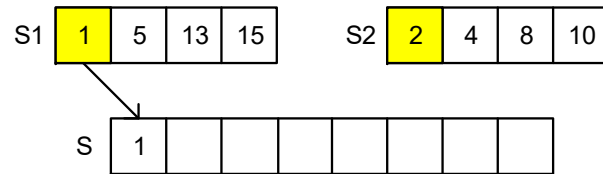
Sorting
Merge-Sort

- Java implementation
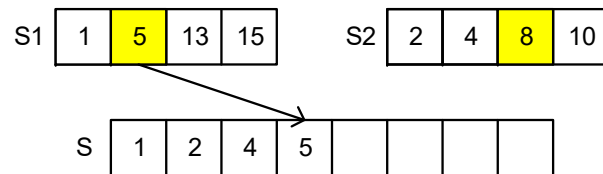
```
1 public static <K> void mergeSort(K[ ] S, Comparator<K> comp) {
2    int n = S.length;
3    if (n < 2) return;    // array is trivially sorted
4    int mid = n/2;
5    K[ ] S1 = Arrays.copyOfRange(S, 0, mid);  // copy of first half
6    K[ ] S2 = Arrays.copyOfRange(S, mid, n);  // copy of second half
7    mergeSort(S1, comp);                 // sort copy of first half
8    mergeSort(S2, comp);                 // sort copy of second half
9    merge(S1, S2, S, comp);  // merge sorted halves back into original
10 }
```

Sorting
Merge-Sort

- Running time analysis
  - Recursive calls are made in lines 7 and 8.
  - Excluding the recursive calls, the program takes $O(n)$.
  - Each recursive call is made on a subarray with $n/2$ elements.
  - The running time of the *mergeSort* on an subarray with n/2 elements is $O(n/2)$.
  - As the successive recursive calls are made, the size of subarray becomes $n/2$, $n/4$, $n/8$, …, and so on, and eventually it becomes 1.
  - This can be represented as a recursion tree.

Sorting
Merge-Sort

- Running time analysis

- Each level takes $O(n)$
- There are $(\log n + 1)$ levels
- Total running time =
    $O(n) (\log n + 1) =$
    $O(n)(\log n) + O(n) =$
    $O(n \log n)$

Height

Tiem per level

Level $i = 0$

$i = 1$

$i = 2$

$\log n$

n $\cdots$ $O(n)$

n/2 $\quad$ n/2 $\quad$ $O(n)$

n/4 $\quad$ n/4 $\quad$ n/4 $\quad$ n/4 $\quad$ $O(n)$

$\cdots$

1 $\quad$ 1 $\quad$ $\cdots$ $\quad$ 1 $\quad$ $O(n)$

n 1's

Total time: O(n log n)
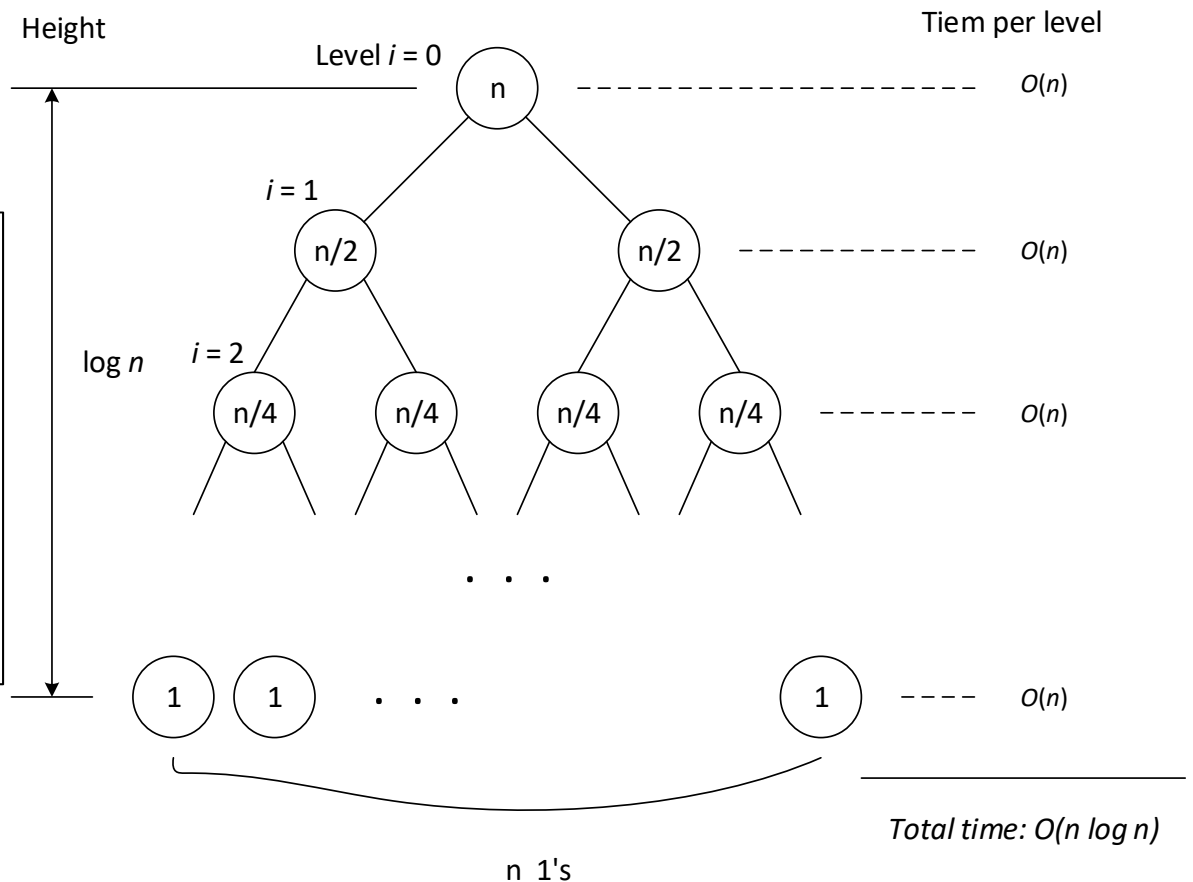
Sorting
Quick-Sort

- Outline

  - Divide: If *S* has only one element, return. Otherwise, remove all elements from *S* and put them into three sequences:
    - *L*: This sequence contains the elements that are less than *x*.
    - *E*: This sequence contains the elements that are equal to *x*.
    - *G*: This sequence contains the elements that are greater than *x*.
  - If the elements in *S* are distinct, then *E* has only one element, which is *x*.
  - Conquer: Recursively sort *L* and *G*.
  - Combine: Put back the elements from the three parts into *S* in order.
- The element *x* is called *pivot*.

Sorting
Quick-Sort

- Outline
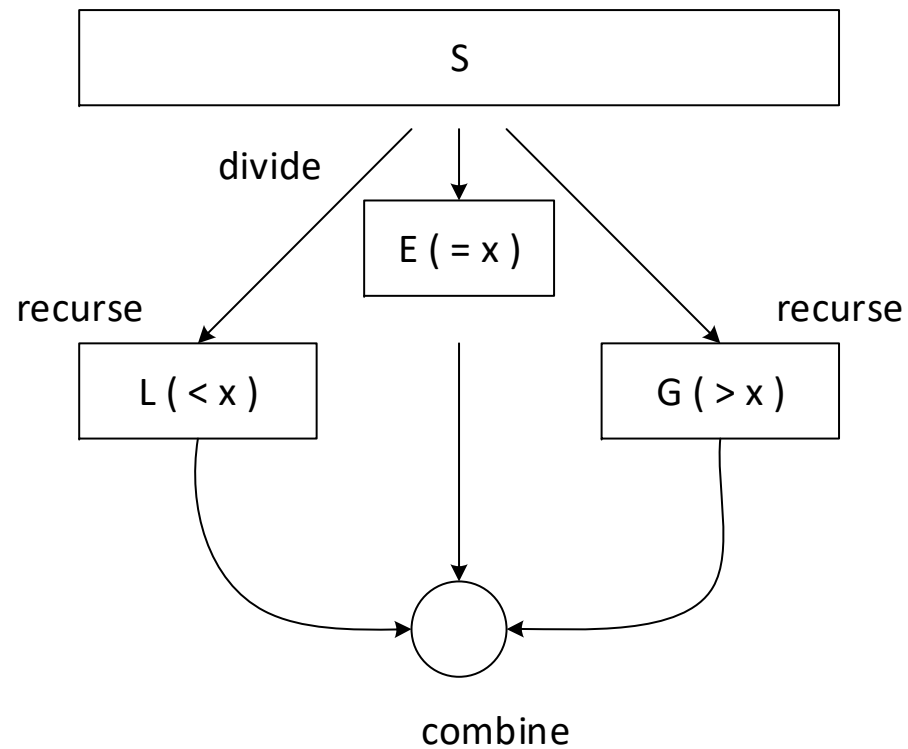
Sorting
Quick-Sort

- Illustration

Sorting
Quick-Sort

- Illustration (continued)
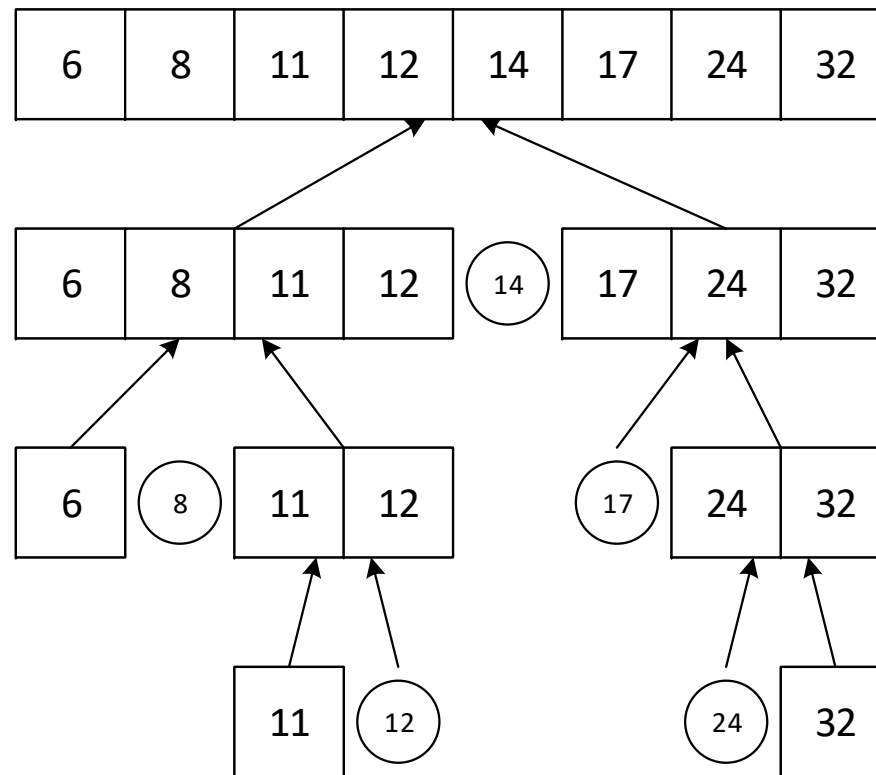
Sorting
Quick-Sort

- The quick-sort algorithm as discussed in the previous slides is not an in-place sorting algorithm.

- An array-based, in-place quicksort algorithm is described in Section 12.2.2 (page 553).
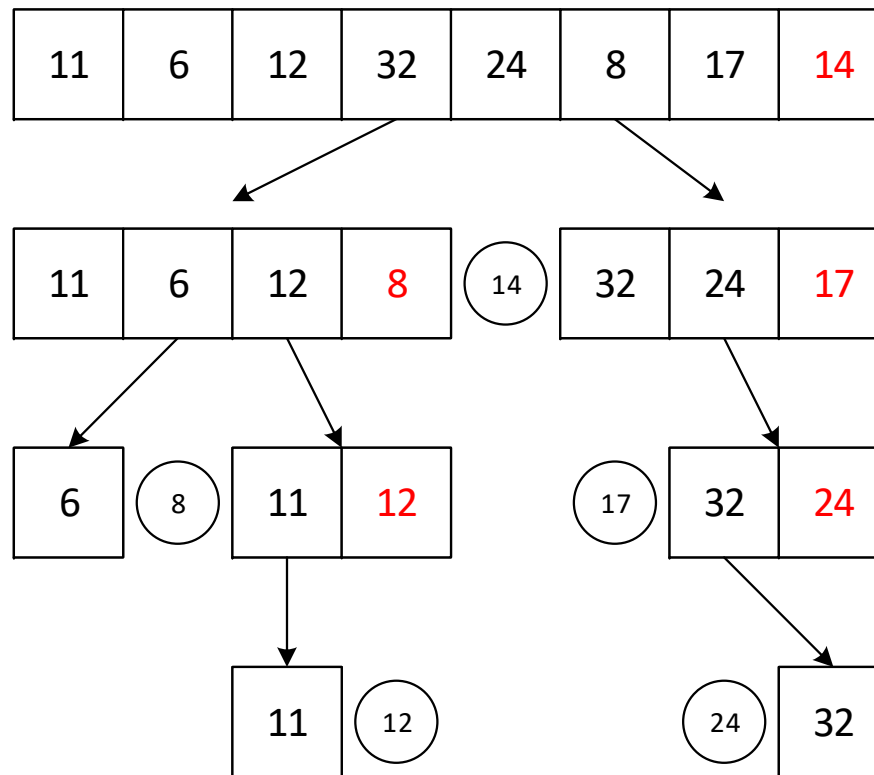
- The "partition" example in the next slides illustrates in-place sorting.

Sorting
Quick-Sort

- The "divide" step is usually called *partition*.
- Partitioning array *S* with *n* elements.
  - S[n – 1] is used as the pivot
  - Keeps two pointers, *left* and *right*
  - *Left* begins at S[0] and moves right until it meets the first element that is equal to or larger than the pivot, *right marker*.
  - *Right* begins at S[n – 2] and moves left until it meets the first element that is equal to or smaller than the pivot, *left marker*.
  - Left marker and right marker are swapped.
  - Repeat this until left and right cross each other
  - Left marker is swapped with pivot.

Sorting
Quick-Sort

- Partitioning illustration

85	24	63	45	17	31	96	50

l   *r*

85	24	63	45	17	31	96	50

l ⟵──────────⟶ *r*

swap

31	24	63	45	17	85	96	50

l ⟷ *r*

swap

31	24	17	45	63	85	96	50

r < *l* ⟷

l and r crossed; *stop; swap S[l] with pivot*

31	24	17	45	50	85	96	63

Sorting
Quick-Sort

- Running time analysis
  - Can use the same method we used for merge-sort (i.e., use a recursion tree).
  - In merge-sort, we always have a balanced divide.
  - In quick-sort, depending on the pivot value, there may be a very unbalanced partitioning
  - In the best case:
    - Always balanced partitioning is created.
    - Running time is $O(n \log n)$
    - Even when partitions are not completely balanced (for example 1 : 9), the running time is still $O(n \log n)$

Sorting
Quick-Sort

- Running time analysis (continued)
  - In the worst case:
    - We always have an extremely unbalanced partitioning, i.e., no element on one side and $n-1$ elements on the other side.
    - This occurs if an array is already sorted and the last element is chosen as a pivot.
    - Running time is $O(n^2)$.

Sorting
Quick-Sort

- Improvement
  - Randomized quick-sort: pivot is chosen randomly
  - *median-of-three* method: the median of the first element, the middle element, and the last element is used as a pivot.
  - When the input size becomes smaller than a certain threshold, we stop the recursion and sort that subarray using insertion-sort. There is no known one threshold value that is considered best. Our textbook suggests 50 and some experiments showed that a value around 15 is a reasonably good choice.

Sorting
Lower Bound for sorting

- The running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$ in the worst case.

- Linear-time sorting: counting-sort, bucket-sort, radix-sort.
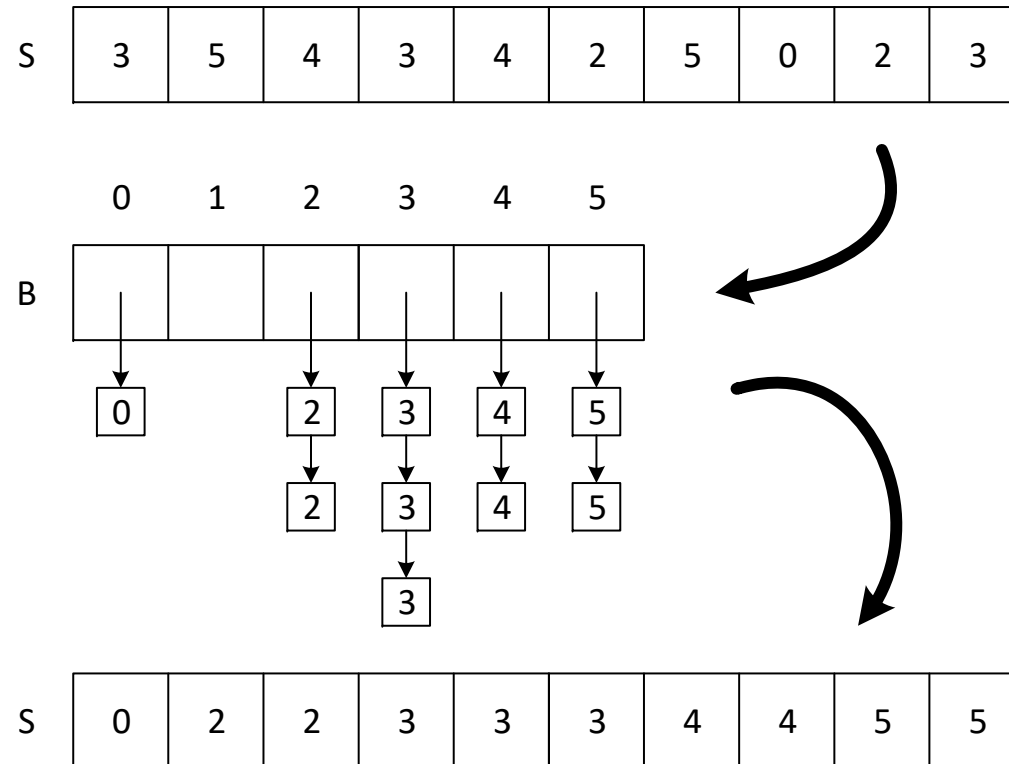- Will discuss bucket-sort and radix-sort.

Sorting
Bucket-Sort

- Sorts a sequence of elements in a linear time with a constraint.

- Constraint:
  - The elements are integers in the range [0, N – 1], for some integer $N \geq 2$.
  - If the elements to be sorted are objects, then the objects must have integer keys with total ordering.

Sorting
Bucket-Sort

- Illustration (N = 6)

Sorting
Bucket-Sort

- ## Pseudocode

  Algoritm bucketSort(S)

  Input: Sequence S of entries with integer keys in range [0, $N - 1$]

  Output: Sequence S sorted in nondecreasing order of keys

  create an empty array B of size N

  for each entry e in S do

    let k be the key of e

    remove e from S and add it to the end of bucket $B[k]$, which is a

       sequence

  for $i = 0$ to $N - 1$ do

    for each entry in sequence $B[i]$ do

      remove e from $B[i]$ and insert it at the end of S

Sorting
Stable Sorting

- Let $S = ((k_0, v_0), (k_1, v_1), \ldots, (k_{n-1}, v_{n-1}))$.

- Assume there are two entries (k_i, v_i) and (k_j, v_j) with an identical key, i.e, $k_i = k_j$, $i \neq j$

- We say a sorting algorithm is *stable* if (k_i, v_i) precedes (k_j, v_j) in S before sorting, then (k_i, v_i) also precedes (k_j, v_j) in S after sorting.

- Example:
  - $S = ((9, W), (4, F), (7, H), (4, A), (2, P))$ before sorting
  - $S = ((2, P), (4, F), (4, A), (7, H), (9, W))$ after sorting

- The bucket-sort described earlier is stable if S and B behave as queues.

Sorting
Radix-Sort

- Illustration:
  - Sorting three digit numbers
  - Each column is sorted using a stable sorting algorithm

456	932	912	148
723	912	723	239
148	723	932	456
239	745	239	648
932	456	745	723
912	148	148	745
648	648	648	912
745	239	456	932

Sorting
Comparison

- Running times

Running Time (average)	Sorting Algorithms
O(n)	bucket-sort, radix-sort
O(n log n)	heap-sort, quick-sort, merge-sort
O(n²)	insertion-sort

Sorting
Comparison

- ## Insertion-Sort

  - When the number of elements is small (typically less than 50), insertion-sort is very efficient.

  - Insertion-sort is very efficient for an "almost" sorted sequence.

  - In general, due to its quadratic running time, insertion-sort is not a good choice except for the situations listed above.

Sorting
Comparison

- Heap-Sort

  - Heap-sort runs in $O(n \log n)$ in the worst case.
  - It works well on small- and medium-sized sequences.
  - It can be made an in-place sorting algorithm.
  - Its performance is poorer than that of quick-sort and merge-sort on large sequences.
  - Heap-sort is not a stable sorting algorithm.

Sorting
Comparison

- Quick-Sort
  - Worst-case running time is $O(n^2)$.
  - Experimental studies showed quick-sort outperformed heap-sort and merge-sort.
  - Quick-sort has been a default algorithm as a general-purpose, in-memory sorting algorithm.
  - It was used in C libraries.
  - Java uses it as the standard sorting algorithm for sorting arrays of primitive types.

Sorting
Comparison

- Merge-Sort
  - Worst-case running time is $O(n \log n)$.
  - It is difficult to make merge-sort an in-place sorting algorithm. So, it is less attractive than heap-sort or quick-sort.
  - Merge-sort is an excellent algorithm for sorting data that resides on the disk (or storage outside the main memory).

Sorting
Comparison

- Tim-Sort

  – Tim-sort is a hybrid algorithm which uses a bottom-up merge-sort and insertion-sort.

  – Tim-sort has been the standard sorting algorithm in Python since 2003.

  – Java uses Tim-sort for sorting arrays of objects.

Sorting
Comparison

- Bucket-Sort and Radix-Sort

  – Excellent for sorting entries with small integer keys, character strings, or d-tuple keys from a small range.

Selection Problem

- Selection problem: Given a set S of n comparable elements and an integer k, $1 \leq k \leq n$, find the element $e \in S$ that is larger than exactly $k - 1$ elements of S.

- The kth smallest element is also referred to as the kth *order statistic*.

- We assume S is a sequence.

- Will discuss *randomized quick-select*, which runs in $O(n)$ expected time.

- Similar to the randomized quick-sort algorithm.

Selection Problem

- Pseudocode

  Algorithm quickSelect (S, k) // find the kth order statistic

  if $n == 1$ // n is the size of S

    return the (first) element

  pick a random pivot element x of S and divide S into three subsequences:

  L, storing the elements in S less than x

  E, storing the elements in S equal to x

  G, storing the elements in S greater than x

  if $k \le |L|$ then   // case 1

    return quickSelect(L, k)

  else if $k \le |L| + |E|$ // case 2

    return x

  else   // case 3

    return quickSelect(G, $k - |L| - |E|$)

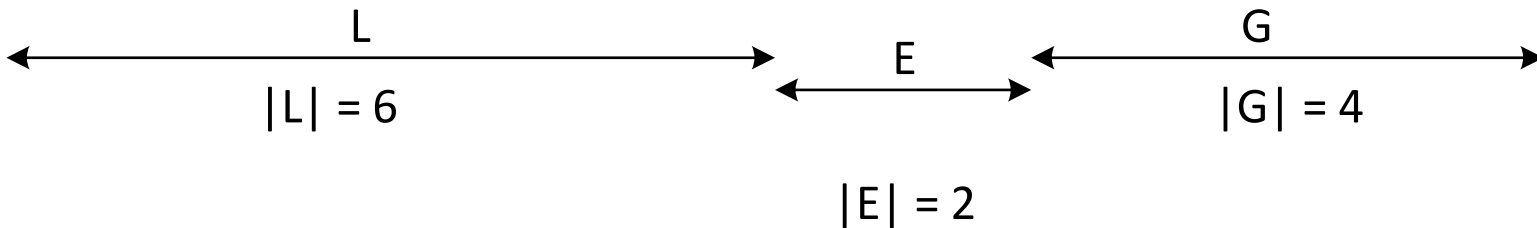# Selection Problem

- Illustration (Case 1: if $k \leq |L|$)

  Find 5th order statistic.

  pivot = 9

- After partition:

  k = 5 < |L|, recurse on L with k = 5

7	3	5	1	6	2	9	9	13	15	17	10

$\longleftrightarrow$ L

|L| = 6

E

|E| = 2

$\longleftrightarrow$ G

|G| = 4

Selection Problem

- Illustration (Case 2: else if $k \leq |L| + |E|$)

  Find 7th order statistic.

  pivot = 9

- After partition:

k = 7 ≤ |L| + |E|, return 9

7	3	5	1	6	2	9	9	13	15	17	10

L
|L| = 6

E
|E| = 2

G
|G| = 4
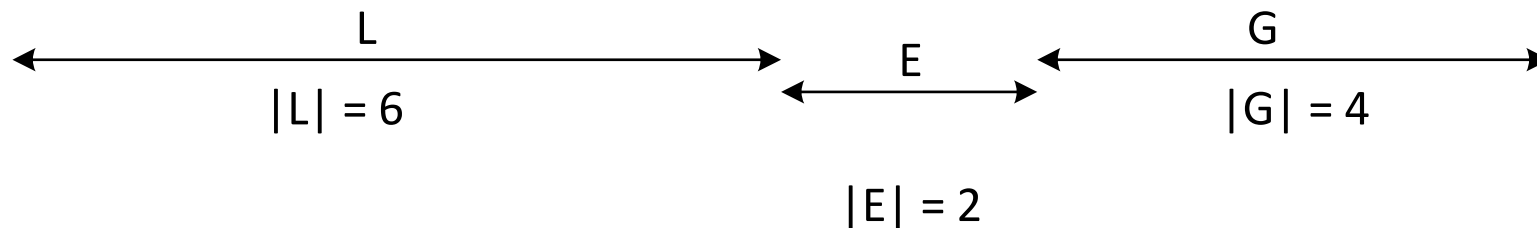
Selection Problem

- Illustration (Case 3: else if $k > |L| + |E|$)

   Find 10th order statistic.

   pivot = 9

- After partition:

k = 10 > |L| + |E|, recurse on G with k = 2

7	3	5	1	6	2	9	9	13	15	17	10

L   E   G

|L| = 6 |G| = 4

|E| = 2
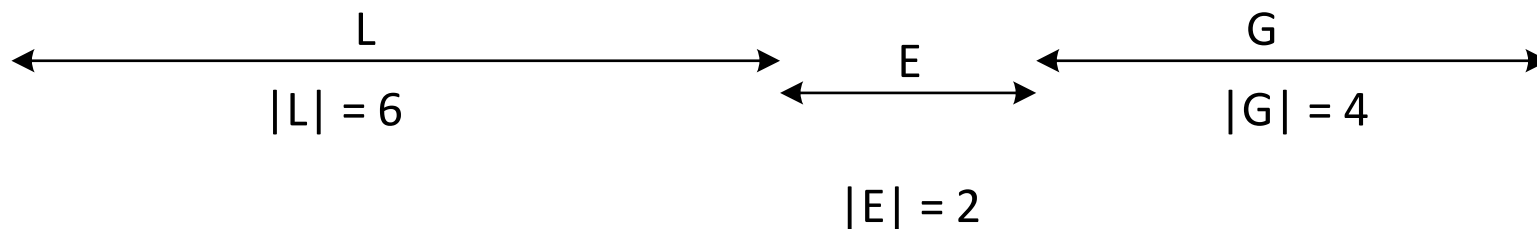
References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, "Data Structures and Algorithms in Java," Sixth Edition, Wiley, 2014.