SORTING WITH DIVIDE AND CONQUER SCHEME

TYPES OF SORTING

- Sorting algorithms we have seen so far:
  + insertion-sort
  + selection-sort
  + heap-sort

- Divide-and-conquer based sorting
  + merge-sort
  + quick-sort

- Linear time Sorting
  + bucket-sort
  + radix-sort
Merge Sort

7 2 | 9 4 → 2 4 7 9

7 | 2 → 2 7
7 → 7

2 → 2

9 | 4 → 4 9
9 → 9
4 → 4
Merge sort on an input sequence $S$ with $n$ elements consists of three steps:

- **Divide**: If $S$ has zero or one element, return $S$. Otherwise partition $S$ into two sequences $S_1$ and $S_2$ of about $n/2$ elements each.
- **Conquer**: recursively sort $S_1$ and $S_2$
- **Combine**: merge sorted $S_1$ and sorted $S_2$ into a unique sorted sequence

**Algorithm** `mergeSort(S)`

- **Input** sequence $S$ with $n$ elements
- **Output** sequence $S$ sorted according to $C$

  ```
  if $S$.size() > 1
  $(S_1, S_2) \leftarrow$ partition($S$, $n/2$)
  mergeSort($S_1$)
  mergeSort($S_2$)
  $S \leftarrow$ merge($S_1$, $S_2$)
  ```

© 2014 Goodrich, Tamassia, Goldwasser
Divide-and-conquer is a general algorithm design paradigm:

- **Divide**: divide the input data \( S \) in two disjoint subsets \( S_1 \) and \( S_2 \)
- **Conquer**: solve the subproblems associated with \( S_1 \) and \( S_2 \)
- **Combine**: 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 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

**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.

**Algorithm merge($A$, $B$)**

Input sequences $A$ and $B$ with $n/2$ elements each

Output sorted sequence of $A \cup B$

$S \leftarrow$ empty sequence

while $\neg A.isEmpty() \land \neg B.isEmpty()$

if $A.first().element() < B.first().element()$

$S.addLast(A.remove(A.first()))$

else

$S.addLast(B.remove(B.first()))$

while $\neg A.isEmpty()$

$S.addLast(A.remove(A.first()))$

while $\neg B.isEmpty()$

$S.addLast(B.remove(B.first()))$

return $S$
An execution of merge-sort is depicted by a binary tree \( T \), called the *merge-sort 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.
EXAMPLE MERGE-SORT TREE $T$

- input sequences processed at each node of $T$
- output sequences generated at each node of $T$. 

© 2014 Goodrich, Tamassia, Goldwasser
EXECUTION EXAMPLE

Partition

7 2 9 4 | 3 8 6 1

© 2014 Goodrich, Tamassia, Goldwasser
EXECUTION EXAMPLE (CONT.)

Recursive call, partition

7 2 9 4 | 3 8 6 1

7 2 | 9 4

© 2014 Goodrich, Tamassia, Goldwasser
**EXECUTION EXAMPLE (CONT.)**

- **Recursive call, partition**

```
7  2  9  4  |  3  8  6  1
```

```
7  2  |  9  4
```

```
7  |  2
```

© 2014 Goodrich, Tamassia, Goldwasser
EXECUTION EXAMPLE (CONT.)

- Recursive call, base case

```
7 2 9 4 | 3 8 6 1
```

```
7 2 | 9 4
```

```
7 | 2
```

```
7 → 7
```
Recursive call, base case

EXECUTION EXAMPLE (CONT.)
EXECUTION EXAMPLE (CONT.)

- **Merge**

```
7 2 9 4 | 3 8 6 1
```

```
7 2 | 9 4
```

```
7 2 | 2 7
```

```
7 -> 7  2 -> 2
```

© 2014 Goodrich, Tamassia, Goldwasser
Recursion call, ..., base case, merge
EXECUTION EXAMPLE (CONT.)

- Merge

7 2 9 4 3 8 6 1

7 2 9 4 2 4 7 9

7 2 → 2 7
9 4 → 4 9

7 → 7 2 → 2 9 → 9 4 → 4

© 2014 Goodrich, Tamassia, Goldwasser
Merge Sort

EXECUTION EXAMPLE (CONT.)

- Recursive call, …, merge, merge

```
7 2 9 4 | 3 8 6 1
```

```
7 2 | 9 4 → 2 4 7 9
```

```
3 8 6 1 → 1 3 6 8
```

© 2014 Goodrich, Tamassia, Goldwasser
EXECUTION EXAMPLE (CONT.)

- **Merge**

```
7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9
```

- **Merge**

```
7 2 9 4 → 2 4 7 9
3 8 6 1 → 1 3 6 8
```

```
7 → 7 2 → 2 9 → 9 4 → 4
3 → 3 8 → 8 6 → 6 1 → 1
```

© 2014 Goodrich, Tamassia, Goldwasser
/** Merge-sort contents of array S. */

```java
public static <K> void mergeSort(K[] S, Comparator<K> comp) {
    int n = S.length;
    if (n < 2) return; // array is trivially sorted
    // divide
    int mid = n/2;
    K[] S1 = Arrays.copyOfRange(S, 0, mid); // copy of first half
    K[] S2 = Arrays.copyOfRange(S, mid, n); // copy of second half
    // conquer (with recursion)
    mergeSort(S1, comp); // sort copy of first half
    mergeSort(S2, comp); // sort copy of second half
    // merge results
    merge(S1, S2, S, comp); // merge sorted halves back into original
}
```
A step in the merge of two sorted arrays for which $S_2[j] < S_1[i]$.  

Indices $i$ & $j$ represents the number of elements of $S_1$ & $S_2$ that have been copied to $S$. 

```java
/** Merge contents of arrays S1 and S2 into properly sized array S. */
public static <K> void merge(K[] S1, K[] S2, K[] S, Comparator<K> comp) {
    int i = 0, j = 0;
    while (i + j < S.length) {
        if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))
            S[i+j] = S1[i++]; // copy ith element of S1 and increment i
        else
            S[i+j] = S2[j++]; // copy jth element of S2 and increment j
    }
}
```
The height $h$ of the merge-sort tree is $O(\log n)$
- at each recursive call we divide in half the sequence,

The overall 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 $O(n\log n)$
/** Merge-sort contents of queue. */
public static <K> void mergeSort(Queue<K> S, Comparator<K> comp) {
    int n = S.size();
    if (n < 2) return; // queue is trivially sorted
    // divide
    Queue<K> S1 = new LinkedQueue<>(); // (or any queue implementation)
    Queue<K> S2 = new LinkedQueue<>();
    while (S1.size() < n/2)
        S1.enqueue(S.dequeue()); // move the first n/2 elements to S1
    while (!S.isEmpty())
        S2.enqueue(S.dequeue()); // move remaining elements to S2
    // conquer (with recursion)
    mergeSort(S1, comp); // sort first half
    mergeSort(S2, comp); // sort second half
    // merge results
    merge(S1, S2, S, comp); // merge sorted halves back into original
Using basic queue as its container type

```java
/** Merge contents of sorted queues S1 and S2 into empty queue S. */
public static <K> void merge(Queue<K> S1, Queue<K> S2, Queue<K> S,
                            Comparator<K> comp) {
    while (!S1.isEmpty() && !S2.isEmpty()) {
        if (comp.compare(S1.first(), S2.first()) < 0)
            S.enqueue(S1.dequeue()); // take next element from S1
        else
            S.enqueue(S2.dequeue()); // take next element from S2
    }
    while (!S1.isEmpty())
        S.enqueue(S1.dequeue()); // move any elements that remain in S1
    while (!S2.isEmpty())
        S.enqueue(S2.dequeue()); // move any elements that remain in S2
}
```
EXAMPLE MERGE IN LINKED-LIST IMPLEMENTATION

\[ S_1 \quad 24 \quad 45 \quad 63 \quad 85 \]
\[ S_2 \quad 17 \quad 31 \quad 50 \quad 96 \]
\[ S \quad 17 \quad 24 \]

\[ S_1 \quad 45 \quad 63 \quad 85 \]
\[ S_2 \quad 31 \quad 50 \quad 96 \]
\[ S \quad 17 \]

\[ S_1 \quad 45 \quad 63 \quad 85 \]
\[ S_2 \quad 50 \quad 96 \]
\[ S \quad 17 \quad 24 \quad 31 \]
EXAMPLE MERGE IN LINKED-LIST IMPLEMENTATION 2
nonrecursive version of array-based merge-sort, which runs in $O(n \log n)$

The main idea is to perform merge-sort bottom-up, performing the merges level by level going up the merge-sort tree.
A BOTTOM-UP (NONRECURSIVE) MERGE-SORT

```java
public static <K> void mergeSortBottomUp(K[] orig, Comparator<K> comp) {
    int n = orig.length;
    K[] src = orig; // alias for the original
    K[] dest = (K[]) new Object[n]; // make a new temporary array
    K[] temp; // reference used only for swapping
    for (int i=1; i < n; i *= 2) {
        for (int j=0; j < n; j += 2*i) // each iteration sorts all runs of length i
            merge(src, dest, comp, j, i);
        temp = src; src = dest; dest = temp; // reverse roles of the arrays
    }
    if (orig != src) // additional copy to get result to original
        System.arraycopy(src, 0, orig, 0, n);
}
```
/* Merges in[start..start+inc−1] and in[start+inc..start+2*inc−1] into out. */
public static <K> void merge(K[] in, K[] out, Comparator<K> comp,
int start, int inc) {
    int end1 = Math.min(start + inc, in.length);     // boundary for run 1
    int end2 = Math.min(start + 2 * inc, in.length);  // boundary for run 2
    int x=start;                                      // index into run 1
    int y=start+inc;                                  // index into run 2
    int z=start;                                      // index into output
    while (x < end1 && y < end2)
        if (comp.compare(in[x], in[y]) < 0)
            out[z++] = in[x++];                        // take next from run 1
        else
            out[z++] = in[y++];                        // take next from run 2
    if (x < end1) System.arraycopy(in, x, out, z, end1 - x);  // copy rest of run 1
    else if (y < end2) System.arraycopy(in, y, out, z, end2 - y); // copy rest of run 2
}
Quick-Sort

7 4 9 6 2 → 2 4 6 7 9

4 2 → 2 4

7 9 → 7 9

2 → 2

9 → 9
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 $x$
- **Conquer**: Recursively sort $L$ and $G$
- **Combine**: join $L$, $E$ and $G
Quick-Sort

PARTITION

- We partition an input sequence as follows:
  - We remove, in turn, each element $y$ from $S$ and
  - We insert $y$ into $L$, $E$ or $G$, depending on the result of the comparison with the pivot $x$
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

Algorithm $\text{partition}(S, p)$

Input sequence $S$, position $p$ of pivot

Output subsequences $L$, $E$, $G$ of the elements of $S$ less than, equal to, or greater than the pivot, resp.

$L$, $E$, $G \leftarrow$ empty sequences

$x \leftarrow S.remove(p)$

while $\neg S.isEmpty()$

  $y \leftarrow S.remove(S.first())$

  if $y < x$
    $L.addLast(y)$
  else if $y = x$
    $E.addLast(y)$
  else
    $G.addLast(y)$

return $L$, $E$, $G$
An execution of quick-sort is depicted by a binary tree called **quick-sort tree**.

- Each **node** represents a recursive call of quick-sort and stores
  - Unsorted sequence before the execution and its pivot
  - 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
Quick-Sort

**EXECUTION EXAMPLE**

- **Pivot selection**

```
7  2  9  4  3  7  6  1
```

© 2014 Goodrich, Tamassia, Goldwasser
Quick-Sort

EXECUTION EXAMPLE (CONT.)

- Partition, recursive call, pivot selection

2 4 3 1

7 2 9 4 3 7 6 1
Quick-Sort

EXECUTION EXAMPLE (CONT.)

- Partition, recursive call, base case

```
7 2 9 4 3 7 6 1

2 4 3 1

1 → 1

1

3 8 6

4 9

7 2 9 4 3 7 6 1

EXECUTION EXAMPLE (CONT.)

- Partition, recursive call, base case

```

© 2014 Goodrich, Tamassia, Goldwasser
**Quick-Sort**

**EXECUTION EXAMPLE (CONT.)**

- Recursive call, ..., base case, join

![Diagram of Quick-Sort execution example](image-url)
Quick-Sort

EXECUTION EXAMPLE (CONT.)

- Recursive call, pivot selection

```
7 2 9 4 3 7 6 1
2 4 3 1 → 1 2 3 4
1 → 1
4 3 → 3 4
4 → 4
7 9 7
```
Quick-Sort

EXECUTION EXAMPLE (CONT.)

- Partition, ..., recursive call, base case

```
7 2 9 4 3 7 6 1
```

```
2 4 3 1  →  1 2 3 4
```

```
1 → 1
```

```
4 3  →  3 4
```

```
9 → 9
```

© 2014 Goodrich, Tamassia, Goldwasser
Quick-Sort

EXECUTION EXAMPLE (CONT.)

Join, join

7 2 9 4 3 7 6 1 → 1 2 3 4 6 7 7 9

2 4 3 1 → 1 2 3 4

7 9 7 → 7 7 9

1 → 1

4 3 → 3 4

9 → 9

4 → 4
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 $O(n^2)$.

```
depth  time
0      n
1      n - 1
...    ...
n - 1  1
```
Consider a recursive call of quick-sort on a sequence 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 $1/2$
+ $1/2$ of the possible pivots cause good calls:
/** Quick-sort contents of a queue. */
public static <K> void quickSort(Queue<K> S, Comparator<K> comp) {
    int n = S.size();
    if (n < 2) return; // queue is trivially sorted
    // divide
    K pivot = S.first();
    Queue<K> L = new LinkedQueue<>();
    Queue<K> E = new LinkedQueue<>();
    Queue<K> G = new LinkedQueue<>();
    while (!S.isEmpty()) {
        K element = S.dequeue();
        int c = comp.compare(element, pivot);
        if (c < 0) // element is less than pivot
            L.enqueue(element);
        else if (c == 0) // element is equal to pivot
            E.enqueue(element);
        else // element is greater than pivot
            G.enqueue(element);
    }
    // conquer
    quickSort(L, comp);
    quickSort(G, comp);
    // concatenate results
    while (!L.isEmpty())
        S.enqueue(L.dequeue());
    while (!E.isEmpty())
        S.enqueue(E.dequeue());
    while (!G.isEmpty())
        S.enqueue(G.dequeue());
}
Quick-Sort

EXPECTED RUNNING TIME, PART 2

- Probabilistic Fact: The expected number of coin tosses required in order to get \( k \) heads is \( 2^k \)

- For a node of depth \( i \), we expect
  + \( i/2 \) ancestors are good calls
  + The size of the input sequence for the current call is at most \((3/4)^{i/2}n\)

- Therefore, we have
  + For a node of depth \( 2\log_{4/3}n \), the expected input size is one
  + The expected height of the quick-sort tree is \( O(\log n) \)

- The amount of work done at the nodes of the same depth is \( O(n) \)

- Thus, the expected running time of quick-sort is \( O(n \log n) \)
Quick-Sort

IN-PLACE QUICK-SORT

Quick-sort can be implemented to run in-place

In the partition step, we use replace operations to rearrange the elements of the input sequence such that

- the elements less than the pivot have rank less than \( h \)
- the elements equal to the pivot have rank between \( h \) and \( k \)
- the elements greater than the pivot have rank greater than \( k \)

The recursive calls consider

- elements with rank less than \( h \)
- elements with rank greater than \( k \)

Algorithm \textit{inPlaceQuickSort}(S, l, r)

\textbf{Input} sequence S, ranks l and r

\textbf{Output} sequence S with the elements of rank between \( l \) and \( r \) rearranged in increasing order

\begin{verbatim}
if \( l \geq r \)
    return

i ← a random integer between \( l \) and \( r \)

x ← S.elemAtRank(i)

(h, k) ← inPlacePartition(x)

inPlaceQuickSort(S, l, h - 1)
inPlaceQuickSort(S, k + 1, r)
\end{verbatim}
Quick-Sort

IN-PLACE PARTITIONING

- Perform the partition using two indices to split S into L and E U G (a similar method can split E U G into E and G).

- Repeat until j and k cross:
  - Scan j to the right until finding an element \( \geq x \).
  - Scan k to the left until finding an element \( < x \).
  - Swap elements at indices j and k

\[
\begin{array}{cccccccccccccccc}
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
3 & 2 & 5 & 1 & 0 & 7 & 3 & 5 & 9 & 2 & 7 & 9 & 8 & 9 & 7 & 6 & 9
\end{array}
\]

(pivot = 6)
/** Sort the subarray S[\ldots b] inclusive. */
private static <K> void quickSortInPlace(K[] S, Comparator<K> comp,
    int a, int b) {
    if (a >= b) return;  // subarray is trivially sorted
    int left = a;
    int right = b-1;
    K pivot = S[b];
    K temp;  // temp object used for swapping
    while (left <= right) {
        // scan until reaching value equal or larger than pivot (or right marker)
        while (left <= right && comp.compare(S[left], pivot) < 0) left++;
        // scan until reaching value equal or smaller than pivot (or left marker)
        while (left <= right && comp.compare(S[right], pivot) > 0) right--;
        if (left <= right) {  // indices did not strictly cross
            // so swap values and shrink range
            temp = S[left]; S[left] = S[right]; S[right] = temp;
            left++; right--;
        }
    }
    // put pivot into its final place (currently marked by left index)
    temp = S[left]; S[left] = S[b]; S[b] = temp;
    // make recursive calls
    quickSortInPlace(S, comp, a, left - 1);
    quickSortInPlace(S, comp, left + 1, b);
}
Divide step of in-place quick-sort, using index $l$ as shorthand for identifier left, and index $r$ as shorthand for identifier right.

- Index $l$ scans the sequence from left to right, and
- index $r$ scans the sequence from right to left.
- A swap is performed when $l$ is at an element as large as the pivot and $r$ is at an element as small as the pivot.
- A final swap with the pivot, in part (f), completes the divide step.
### Summary of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection-sort</td>
<td>$O(n^2)$</td>
<td>in-place, slow (good for small inputs)</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>$O(n^2)$</td>
<td>in-place, slow (good for small inputs)</td>
</tr>
<tr>
<td>quick-sort</td>
<td>$O(n \log n)$</td>
<td>in-place, randomized, fastest (good for large inputs)</td>
</tr>
<tr>
<td>heap-sort</td>
<td>$O(n \log n)$</td>
<td>in-place, fast (good for large inputs)</td>
</tr>
<tr>
<td>merge-sort</td>
<td>$O(n \log n)$</td>
<td>sequential data access, fast (good for huge inputs)</td>
</tr>
</tbody>
</table>