Data Structures and Algorithms

Sorting
Outline

• Merge Sort
• Quick Sort
• Sorting Lower Bound
• 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
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

Algorithm $mergeSort(S, C)$

Input sequence $S$ with $n$ elements, comparator $C$

Output sequence $S$ sorted according to $C$

if $S.size() > 1$

$(S_1, S_2) \leftarrow partition(S, n/2)$

$mergeSort(S_1, C)$

$mergeSort(S_2, C)$

$S \leftarrow merge(S_1, S_2)$
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.

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 $A.first().element() < B.first().element()$
        $S.insertLast(A.remove(A.first()))$
    else
        $S.insertLast(B.remove(B.first()))$

while $\neg A.isEmpty()$

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

while $\neg B.isEmpty()$

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

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

```
[7 2 | 9 4] → [2 4 7 9]
```

```
[7 | 2] → [2 7]
[9 | 4] → [4 9]
```

```
[7] → [7]
[2] → [2]
[9] → [9]
[4] → [4]
```
Execution Example

• Partition

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

7  2  9  4
→  2  4  7  9

3  8  6  1
→  1  3  8  6

7  2  9  4
→  2  7

9  4  4  9
→  4  9

3  8  6  1
→  3  8

6  1  1  6
→  1  6

7  2  9  4
→  7

3  8  6  1
→  3

6  1  1  6
→  1

Phạm Bảo Sơn DSA
Execution Example (cont.)

• Recursive call, partition
Execution Example (cont.)

• Recursive call, partition
Execution Example (cont.)

- Recursive call, base case

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

```
7 2 9 4
---------
2 4 7 9
```

```
7 2 9 4
---------
4 9
3 8
```

```
7 2 9 4
---------
7
```

```
7 2 9 4
---------
2
9
4
```

```
7 2 9 4
---------
2
9
4
---------
3 8
3 8
6 1
1 6
---------
1
6
1
---------
2
2
9
9
4
4
---------
3
3
8
8
6
6
---------
7
7
2
2
---------
7
7
2
2
```
Execution Example (cont.)

- Recursive call, base case

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

```
7 2 | 9 4
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
7 2 9 4 | 3 8 6 1
```
Execution Example (cont.)

- Merge

7 2 9 4 | 3 8 6 1

7 2 | 9 4

7 2 → 2 7

9 4 4 9

9 6 4

3 3 8

6 6 1

3 8 3 8 1 6

1 6

7 2 9 4 | 3 8 6 1

1 2 3 4 6 7 8 9

Phạm Bảo Sơn DSA
Execution Example (cont.)

- Recursive call, …, base case, merge

Phạm Bảo Sơn DSA
Execution Example (cont.)

- Merge

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

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

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

Phạm Bảo Sơn DSA
Execution Example (cont.)

- Recursive call, ..., merge, merge

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

```
[7, 2, 9, 4] → [2, 4, 7, 9] → [1, 2, 3, 4, 6, 7, 8, 9]
```

```
7 | 2 → 2 7
9 4 → 4 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
```
Execution Example (cont.)

- Merge

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

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

```
7 2 | 9 4 2 4 7 9
```

```
3 8 6 1  1 3 6 8
```

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

```
7 2 | 9 4 2 4 7 9
```

```
3 8 6 1  1 3 6 8
```

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

```
7 2 | 9 4 2 4 7 9
```

```
3 8 6 1  1 3 6 8
```

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

```
7 2 | 9 4 2 4 7 9
```

```
3 8 6 1  1 3 6 8
```

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

```
7 2 | 9 4 2 4 7 9
```

```
3 8 6 1  1 3 6 8
```

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

```
7 2 | 9 4 2 4 7 9
```

```
3 8 6 1  1 3 6 8
```
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 $O(n \log n)$

<table>
<thead>
<tr>
<th>depth</th>
<th>#seqs</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$n$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$n/2$</td>
</tr>
<tr>
<td>$i$</td>
<td>$2^i$</td>
<td>$n/2^i$</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Phạm Bảo Sơn DSA
### 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>slow, in-place, for small data sets (&lt; 1K)</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>$O(n^2)$</td>
<td>slow, in-place, for small data sets (&lt; 1K)</td>
</tr>
<tr>
<td>heap-sort</td>
<td>$O(n \log n)$</td>
<td>fast, in-place, for large data sets (1K — 1M)</td>
</tr>
<tr>
<td>merge-sort</td>
<td>$O(n \log n)$</td>
<td>fast, sequential data access, for huge data sets (&gt; 1M)</td>
</tr>
</tbody>
</table>
Nonrecursive Merge-Sort

public static void mergeSort(Object[] orig, Comparator c) {
    // nonrecursive
    Object[] in = new Object[orig.length]; // make a new temporary array
    System.arraycopy(orig,0,in,0,in.length); // copy the input
    Object[] out = new Object[in.length]; // output array
    Object[] temp; // temp array reference used for swapping
    int n = in.length;
    for (int i=1; i < n; i*=2) { // each iteration sorts all length-2*i runs
        for (int j=0; j < n; j+=2*i) // each iteration merges two length-i pairs
            merge(in,out,c,j,i); // merge from in to out two length-i runs at j
        temp = in; in = out; out = temp; // swap arrays for next iteration
    }
    // the "in" array contains the sorted array, so re-copy it
    System.arraycopy(in,0,orig,0,in.length);
}
public static void mergeSort(Object[] orig, Comparator c) { // nonrecursive
    Object[] in = new Object[orig.length]; // make a new temporary array
    System.arraycopy(orig,0,in,0,in.length); // copy the input
    Object[] out = new Object[in.length]; // output array
    Object[] temp; // temp array reference used for swapping
    int n = in.length;
    for (int i=1; i < n; i*=2) { // each iteration sorts all length-2^i runs
        for (int j=0; j < n; j+=2*i)  // each iteration merges two length-i pairs
            merge(in,out,c,j,i); // merge from in to out two length-i runs at j
        temp = in; in = out; out = temp; // swap arrays for next iteration
    }
    // the "in" array contains the sorted array, so re-copy it
    System.arraycopy(in,0,orig,0,in.length);
}

protected static void merge(Object[] in, Object[] out, Comparator c, int start,
    int inc) { // merge in[start..start+inc-1] and in[start+inc..start+2*inc-1]
    int x = start; // index into run #1
    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 y = start+inc; // index into run #2 (could be beyond array boundary)
    int z = start; // index into the out array
    while ((x < end1) && (y < end2))
        if (c.compare(in[x],in[y]) <= 0) out[z++] = in[x++];
        else out[z++] = in[y++];
    if (x < end1) // first run didn't finish
        System.arraycopy(in, x, out, z, end1 - x);
    else if (y < end2) // second run didn't finish
        System.arraycopy(in, y, out, z, end2 - y);
}
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

- 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 $G$
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 \textit{partition}(S, p)

\textbf{Input} sequence \( S \), position \( p \) of pivot

\textbf{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

\( x \leftarrow S\.remove(p) \)

\textbf{while} \( \neg S\.isEmpty() \)

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

\textbf{if} \( y < x \)

\( L\.insertLast(y) \)

\textbf{else if} \( y = x \)

\( E\.insertLast(y) \)

\textbf{else} \( \{ y > x \} \)

\( G\.insertLast(y) \)

\textbf{return} \( L, E, G \)
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 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

```
7  4  9  6  2 → 2  4  6  7  9
```

```
4  2 → 2  4
```

```
7  9 → 7  9
```

```
2 → 2
```

```
9 → 9
```

Phạm Bảo Sơn DSA
Execution Example

- Pivot selection

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

```
7  2  9  4
2  4  7  9
```

```
3  8  6  1
1  3  8  6
```

```
2  2
9  4  4  9
```

```
9  9  4  4
```

```
3  3  8  8
```

Phạm Bảo Sơn DSA
Execution Example (cont.)

- Partition, recursive call, pivot selection

```
7 2 9 4 3 7 6 1 1 2 3 4 6 7 8 9
2 4 3 1 2 4 7 9 3 8 6 1 1 3 8 6
2 2
9 4 4 9
3 3
9 6 4
8 8
```
Execution Example (cont.)

- Partition, recursive call, base case

```
2 4 3 1 2 4 7 3 8 6 1 1 3 8 6
1 → 1
9 4 4 9 3 3
9 9 4 8 8
```
Execution Example (cont.)

• Recursive call, ..., base case, join

```
3  8  6  1
3  → 3
8  → 8
```

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

```
2  4  3  1  →  1  2  3  4

1  →  1
4  3  →  3  4
```

```
3  8  6  1
1  3  8  6
```

```
9  9
4  →  4
```

Phạm Bảo Sơn DSA
Execution Example (cont.)

- Recursive call, pivot selection
Execution Example (cont.)

• Partition, …, recursive call, base case

Phạm Bảo Sơn DSA
Execution Example (cont.)

• Join, join

1 2 3 4 5 6 7 8

7 2 9 4 3 7 6 1 → 1 2 3 4 6 7 7 9

7 9 7 → 7 7 9

2 4 3 1 → 1 2 3 4

1 → 1

4 3 → 3 4

8 → 8

4 → 4

9 → 9

Phạm Bảo Sơn DSA
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 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:

\[
\begin{array}{c}
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 \\
\end{array}
\]

Phạm Bảo Sơn DSA
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 with input size $s$, the input sizes of its children are each at most $s^{3/4}$ or $s/(4/3)$.

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 or work done at the nodes of the same depth is $O(n)$

Thus, the expected running time of quick-sort is $O(n \log n)$
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 \( \text{inPlaceQuickSort}(S, l, r) \)

Input sequence \( S \), ranks \( l \) and \( r \)

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

if \( l \geq r \)
    return

\( i \leftarrow \) a random integer between \( l \) and \( r \)
\( x \leftarrow S.\text{elemAtRank}(i) \)
\( (h, k) \leftarrow \text{inPlacePartition}(x) \)
\( \text{inPlaceQuickSort}(S, l, h - 1) \)
\( \text{inPlaceQuickSort}(S, k + 1, r) \)
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).
  
  \[
  \begin{array}{cc}
  j & k \\
  \end{array}
  \]

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

  (pivot = 6)

- 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

  \[
  \begin{array}{cccccccccccccccc}
  3 & 2 & 5 & 1 & 0 & 7 & 3 & 5 & 9 & 2 & 7 & 9 & 8 & 9 & 7 & 6 & 9 \\
  \end{array}
  \]
## 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>$O(n \log n)$ expected</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>
public static void quickSort (Object[] S, Comparator c) {
    if (S.length < 2) return; // the array is already sorted in this case
    quickSortStep(S, c, 0, S.length-1); // recursive sort method
}

private static void quickSortStep (Object[] S, Comparator c,
    int leftBound, int rightBound ) {
    if (leftBound >= rightBound) return; // the indices have crossed
    Object temp; // temp object used for swapping
    Object pivot = S[rightBound];
    int leftIndex = leftBound;     // will scan rightward
    int rightIndex = rightBound-1; // will scan leftward
    while (leftIndex <= rightIndex) { // scan right until larger than the pivot
        while ( (leftIndex <= rightIndex) && (c.compare(S[leftIndex], pivot)<=0) )
            leftIndex++;
        // scan leftward to find an element smaller than the pivot
        while ( (rightIndex >= leftIndex) && (c.compare(S[rightIndex], pivot)>=0))
            rightIndex--;
        if (leftIndex < rightIndex) { // both elements were found
            temp = S[rightIndex];
            S[rightIndex] = S[leftIndex]; // swap these elements
            S[leftIndex] = temp;
        }
    } // the loop continues until the indices cross
    temp = S[rightBound]; // swap pivot with the element at leftIndex
    S[rightBound] = S[leftIndex];
    S[leftIndex] = temp; // the pivot is now at leftIndex, so recurse
    quickSortStep(S, c, leftBound, leftIndex-1);
    quickSortStep(S, c, leftIndex+1, rightBound);

Sorting Lower Bound
Comparison-Based Sorting

• Many sorting algorithms are comparison based.
  – They sort by making comparisons between pairs of objects
  – Examples: bubble-sort, selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, ...

• Let us therefore derive a lower bound on the running time of any algorithm that uses comparisons to sort \( n \) elements, \( x_1, x_2, \ldots, x_n \).

\[
\text{Is } x_i < x_j? \\
\text{yes} \quad \text{no}
\]
Counting Comparisons

• Let us just count comparisons then.
• Each possible run of the algorithm corresponds to a root-to-leaf path in a decision tree.

```
\begin{tikzpicture}[level distance=1.5cm,
                      level 1/.style={sibling distance=3cm},
                      level 2/.style={sibling distance=1.5cm},
                      level 3/.style={sibling distance=0.75cm}]

  \node {$x_i < x_j$}
    child {node {$x_a < x_b$} 
      child {node {$x_e < x_f$}}
      child {node {$x_k < x_l$}}}
    child {node {$x_c < x_d$} 
      child {node {$x_m < x_o$}}
      child {node {$x_p < x_q$}}};
\end{tikzpicture}
```
Decision Tree Height

- The height of this decision tree is a lower bound on the running time.
- Every possible input permutation must lead to a separate leaf output.
  - If not, some input ...4...5... would have same output ordering as ...
    5...4..., which would be wrong.
- Since there are n!=1*2*...*n leaves, the height is at least log \( n! \)
The Lower Bound

- Any comparison-based sorting algorithms takes at least \( \log(n!) \) time.
- Therefore, any such algorithm takes time at least \( \Omega(n \log n) \).

\[
\log(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = (n/2) \log(n/2).
\]

- That is, any comparison-based sorting algorithm must run in \( \Omega(n \log n) \) time.
Bucket-Sort and Radix-Sort
Bucket-Sort

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

Algorithm $bucketSort(S, N)$

Input sequence $S$ of (key, element) items with keys in the range $[0, N - 1]$
Output sequence $S$ sorted by increasing keys
$B \leftarrow$ array of $N$ empty sequences
while $\neg S.isEmpty()$
  $f \leftarrow S.first()$
  $(k, o) \leftarrow S.remove(f)$
  $B[k].insertLast((k, o))$
for $i \leftarrow 0$ to $N - 1$
  while $\neg B[i].isEmpty()$
    $f \leftarrow B[i].first()$
    $(k, o) \leftarrow B[i].remove(f)$
    $S.insertLast((k, o))$
Example

- **Key range** $[0, 9]$

  $7, d \rightarrow 1, c \rightarrow 3, a \rightarrow 7, g \rightarrow 3, b \rightarrow 7, e$

  **Phase 1**

  $1, c \rightarrow 3, a \rightarrow 3, b \rightarrow 7, d \rightarrow 7, g \rightarrow 7, e$

  **Phase 2**

  $1, c \rightarrow 3, a \rightarrow 3, b \rightarrow 7, d \rightarrow 7, g \rightarrow 7, e$

Phạm Bảo Sơn DSA
Properties and Extension

- **Key-type Property**
  - The keys are used as indices into an array and cannot be arbitrary objects
  - No external comparator

- **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 entry \((k, o)\) 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 entry \((k, o)\) into bucket \(B[r(k)]\)
Lexicographic Order

- A $d$-tuple is a sequence of $d$ keys $(k_1, k_2, \ldots, k_d)$, 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:

$$(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d) \iff x_1 < y_1 \lor x_1 = y_1 \land (x_2, \ldots, x_d) < (y_2, \ldots, y_d)$$

I.e., the tuples are compared by the first dimension, then by the second dimension, etc.
Lexicographic-Sort

- Let $C_i$ be the comparator 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 a sequence 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 $stableSort$.

Algorithm $lexicographicSort(S)$

Input sequence $S$ of $d$-tuples
Output sequence $S$ sorted in lexicographic order

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

Example:

$(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)$
$(2, 1, 4) (3, 2, 4) (5,1,5) (7,4,6) (2,4,6)$
$(2, 1, 4) (5,1,5) (3, 2, 4) (7,4,6) (2,4,6)$
$(2, 1, 4) (2,4,6) (3, 2, 4) (5,1,5) (7,4,6)$
Radix-Sort

- Radix-sort is a specialization of lexicographic-sort that uses bucket-sort as the stable sorting algorithm in each dimension.
- Radix-sort is applicable to tuples where the keys in each dimension \( i \) are integers in the range \([0, N - 1]\).
- Radix-sort runs in time \( O(d(n + N)) \).

Algorithm \( \text{radixSort}(S, N) \)

Input sequence \( S \) of \( d \)-tuples such that \((0, ..., 0) \leq (x_1, ..., x_d) \) and \((x_1, ..., x_d) \leq (N - 1, ..., N - 1)\) for each tuple \((x_1, ..., x_d)\) in \( S \).

Output sequence \( S \) sorted in lexicographic order.

for \( i \leftarrow d \) downto 1

bucketSort\( (S, N) \)
Radix-Sort for Binary Numbers

• Consider a sequence of \( n \) \( b \)-bit integers
  \[ x = x_{b-1} \ldots x_1 x_0 \]
• We represent each element as a \( b \)-tuple of integers in the range \([0, 1]\) and apply radix-sort with \( N = 2 \)
• This application of the radix-sort algorithm runs in \( O(bn) \) time
• For example, we can sort a sequence of 32-bit integers in linear time

Algorithm \( \text{binaryRadixSort}(S) \)

Input sequence \( S \) of \( b \)-bit integers
Output sequence \( S \) sorted
replace each element \( x \) of \( S \) with the item \((0, x)\)
for \( i \leftarrow 0 \) to \( b - 1 \)
  replace the key \( k \) of each item \((k, x)\) of \( S \) with bit \( x_i \) of \( x \)

\( \text{bucketSort}(S, 2) \)
Example

• Sorting a sequence of 4-bit integers

1001  0010  1001  1001  0001
0010  1110  1101  0001  0010
1101  0001  1101  1110  1001
0001  1110  0010  1110  1101