Chapter 4: Sorting
What We’ll Do

- Quick Sort
- Lower bound on runtimes for comparison based sort
- Radix and Bucket sort
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 \( G \)
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 $y < x$
    $L.insertLast(y)$
  else if $y = x$
    $E.insertLast(y)$
  else { $y > x$ }
    $G.insertLast(y)$

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
Execution Example

Pivot selection

7 2 9 4 3 7 6 1

Sorting Fun
Execution Example (cont.)

Partition, recursive call, pivot selection

7 2 9 4 3 7 6 1

2 4 3 1

1 2 3 4 6 7 8 9

3 8 6 1

1 3 8 6

3 8

8
Execution Example (cont.)

Partition, recursive call, base case

7 2 9 4 3 7 6 1

2 4 3 1

1 → 1
Execution Example (cont.)

Recursive call, ..., base case, join
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
```

```
7 9 7
```

```
4 → 4
```

```
9
```

```
9
```

```
1
```

```
3
```

```
4
```

```
4
```

```
9
```

```
4
```
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

4 → 4

7 9 7

9 → 9
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
```

```
4 3 → 3 4
```

```
1 → 1
```

```
4 → 4
```

```
7 9 7 → 7 7 9
```

```
9 → 9
```
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 $\sum_{i=0}^{n-1} i + 2 + 1$.
- Thus, the worst-case running time of quick-sort is $O(n^2)$.

\[n + (n - 1) + \ldots + 2 + 1\]
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:
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 \( (3/4)^{i/2}n \)
    - Since each **good** call shrinks size to at most 3/4 of previous size

- 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 \( O(n \log n) \)

\[
\left(\frac{3}{4}\right)^i n = 1 \quad \Rightarrow \quad i = 2\log_{\frac{3}{4}} n
\]
Sorting Lower Bound
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 a set $S$ of $n$ elements, $x_1, x_2, \ldots, x_n$.

Assume that the $x_i$ are distinct, which is not a restriction

Is $x_i < x_j$?

no

yes
Counting Comparisons

Let us just count comparisons then.

First, we can map any comparison based sorting algorithm to a **decision tree** as follows:

- Let the root node of the tree correspond to the first comparison, \((x_i < x_j?)\), that occurs in the algorithm.
- The outcome of the comparison is either yes or no.
- If yes we proceed to another comparison, say \(x_a < x_b?\) We let this comparison correspond to the left child of the root.
- If no we proceed to the comparison \(x_c < x_d?\) We let this comparison correspond to the right child of the root.
- Each of those comparisons can be either yes or no...
The Decision Tree

Each possible permutation of the set $S$ will cause the sorting algorithm to execute a sequence of comparisons, effectively traversing a path in the tree from the root to some external node.
Paths Represent Permutations

Fact: Each external node $v$ in the tree can represent the sequence of comparisons for exactly one permutation of $S$

- If $P_1$ and $P_2$ are different permutations, then there is at least one pair $x_i, x_j$ with $x_i$ before $x_j$ in $P_1$ and $x_i$ after $x_j$ in $P_2$
- For both $P_1$ and $P_2$ to end up at $v$, this means every decision made along the way resulted in the exact same outcome.
  - We have a decision tree, so no cycles!
- This cannot occur if the sorting algorithm behaves correctly, because in one permutation $x_i$ started before $x_j$ and in the other their order was reversed (remember, they cannot be equal)
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 (by previous slide).

There are $n!$ permutations, so there are $n!$ leaves. Since there are $n! = 1 \times 2 \times \ldots \times n$ 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

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

- Since there are at least $n/2$ terms larger than $n/2$ in $n!$

That is, any comparison-based sorting algorithm must run no faster than $O(n \log n)$ time in the worst case.
Bucket-Sort and Radix-Sort
Bucket-Sort

Let be \( S \) be a sequence 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 sequences (buckets)

Phase 1: Empty sequence \( S \) by moving each item \((k, o)\) into its bucket \( B[k] \)

Phase 2: For \( i = 0, ..., N - 1 \), move the items 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 \textit{bucketSort}(S, N)

\textbf{Input} sequence \( S \) of (key, element) items with keys in the range \([0, N - 1]\)

\textbf{Output} sequence \( S \) sorted by increasing keys

\( B \leftarrow \) array of \( N \) empty sequences

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

\( f \leftarrow S.first() \)

\( (k, o) \leftarrow S.remove(f) \)

\( B[k].insertLast((k, o)) \)

\textbf{for} \( i \leftarrow 0 \) \textbf{to} \( N - 1 \)

\textbf{while} \( \neg B[i].isEmpty() \)

\( f \leftarrow B[i].first() \)

\( (k, o) \leftarrow B[i].remove(f) \)

\( S.insertLast((k, o)) \)
Example

Key range $[0, 9]$

Phase 1

Phase 2

<table>
<thead>
<tr>
<th>B</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>

1, c → 3, a → 3, b → 7, d → 7, g → 7, e

1, c → 3, a → 3, b → 7, d → 7, g → 7, e
Properties and Extensions

- **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 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 \(B[r(k)]\)
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:

$$(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d)$$

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

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 $\text{stableSort}(S, C)$ 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 $\text{stableSort}$, one per dimension
- Lexicographic-sort runs in $O(dT(n))$ time, where $T(n)$ is the running time of $\text{stableSort}$

**Algorithm** $\text{lexicographicSort}(S)$

**Input** sequence $S$ of $d$-tuples

**Output** sequence $S$ sorted in lexicographic order

\[
\text{for } i \leftarrow d \text{ downto 1} \\
\text{stableSort}(S, C_i)
\]

**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 \((0, \ldots, 0) \leq (x_1, \ldots, x_d)\) and \((x_1, \ldots, x_d) \leq (N - 1, \ldots, N - 1)\) for each tuple \((x_1, \ldots, x_d)\) in \(S\).
- **Output** sequence \(S\) sorted in lexicographic order.

for \(i \leftarrow d\) downto 1

\(\text{bucketSort}(S, N)\)
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 → 0001 → 0010 → 1001 → 0001 → 0010 → 1001 → 0001