CHAPTER 11
SORTING

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN C++, GOODRICH, TAMASSIA AND MOUNT (WILEY 2004) AND SLIDES FROM NANCY M. AMATO AND JORY DENNY
Divide-and conquer is a general algorithm design paradigm:

- **Divide:** divide the input data $S$ into $k$ (disjoint) subsets $S_1, S_2, ..., S_k$
- **Recur:** solve the sub-problems recursively
- **Conquer:** combine the solutions for $S_1, S_2, ..., S_k$ into a solution for $S$

- The base case for the recursion are sub-problems of constant size
- Analysis can be done using **recurrence equations** (relations)
When the size of all sub-problems is the same (frequently the case) the recurrence equation representing the algorithm is:

\[ T(n) = D(n) + kT\left(\frac{n}{c}\right) + C(n) \]

Where

- \( D(n) \) is the cost of dividing \( S \) into the \( k \) sub-problems \( S_1, S_2, ..., S_k \)
- There are \( k \) sub-problems, each of size \( \frac{n}{c} \) that will be solved recursively
- \( C(n) \) is the cost of combining the sub-problem solutions to get the solution for \( S \)
Algorithm – transform multiplication of two $n$-bit integers $I$ and $J$ into multiplication of $(\frac{n}{2})$-bit integers and some additions/shifts

1. Where does recursion happen in this algorithm?
2. Rewrite the step(s) of the algorithm to show this clearly.

Algorithm $\text{multiply}(I,J)$

Input: $n$-bit integers $I, J$

Output: $I \times J$

1. if $n > 1$
2. Split $I$ and $J$ into high and low order halves: $I_h, I_l, J_h, J_l$
3. $x_1 \leftarrow I_h \times J_h; x_2 \leftarrow I_h \times J_l; x_3 \leftarrow I_l \times J_h; x_4 \leftarrow I_l \times J_l$
4. $Z \leftarrow x_1 \times 2^n + x_2 \times 2^{\frac{n}{2}} + x_3 \times 2^{\frac{n}{2}} + x_4$
5. else
6. $Z \leftarrow I \times J$
7. return $Z$
Algorithm – transform multiplication of two $n$-bit integers $I$ and $J$ into multiplication of $\left(\frac{n}{2}\right)$-bit integers and some additions/shifts

- Assuming that additions and shifts of $n$-bit numbers can be done in $O(n)$ time, describe a recurrence equation showing the running time of this multiplication algorithm

**Algorithm multiply($I, J$)**

**Input:** $n$-bit integers $I, J$

**Output:** $I \times J$

1. if $n > 1$
2. Split $I$ and $J$ into high and low order halves: $I_h, I_l, J_h, J_l$
3. $x_1 \leftarrow$ multiply($I_h, J_h$); $x_2 \leftarrow$ multiply($I_h, J_l$);
   $x_3 \leftarrow$ multiply($I_l, J_h$); $x_4 \leftarrow$ multiply($I_l, J_l$)
4. $Z \leftarrow x_1 \times 2^n + x_2 \times 2^{\frac{n}{2}} + x_3 \times 2^{\frac{n}{2}} + x_4$
5. else
6. $Z \leftarrow I \times J$
7. return $Z$
Algorithm – transform multiplication of two \( n \)-bit integers \( I \) and \( J \) into multiplication of \( \left( \frac{n}{2} \right) \)-bit integers and some additions/shifts

- The recurrence equation for this algorithm is:
  - \( T(n) = 4T \left( \frac{n}{2} \right) + O(n) \)
  - The solution is \( O(n^2) \) which is the same as naïve algorithm

Algorithm \texttt{multiply}(\( I, J \))

\textbf{Input:} \( n \)-bit integers \( I, J \)

\textbf{Output:} \( I \times J \)

1. \textbf{if} \( n > 1 \)
2. Split \( I \) and \( J \) into high and low order halves: \( I_h, I_l, J_h, J_l \)
3. \( x_1 \leftarrow \text{multiply}(I_h, J_h); x_2 \leftarrow \text{multiply}(I_h, J_l); x_3 \leftarrow \text{multiply}(I_l, J_h); x_4 \leftarrow \text{multiply}(I_l, J_l) \)
4. \( Z \leftarrow x_1 \times 2^n + x_2 \times 2^\frac{n}{2} + x_3 \times 2^\frac{n}{2} + x_4 \)
5. \textbf{else}
6. \( Z \leftarrow I \times J \)
7. \textbf{return} \( Z \)
MERGE SORT
**Merge-Sort**

- **Merge-sort** is based on the divide-and-conquer paradigm. It consists of three steps:
  - **Divide**: partition input sequence $S$ into two sequences $S_1$ and $S_2$ of about $\frac{n}{2}$ elements each
  - **Recur**: recursively sort $S_1$ and $S_2$
  - **Conquer**: merge $S_1$ and $S_2$ into a sorted sequence

---

**Algorithm** `mergeSort(S, C)`

**Input**: Sequence $S$ of $n$ elements,
  Comparator $C$

**Output**: Sequence $S$ sorted according to $C$

1. if $S$.size() > 1
2. $(S_1, S_2) \leftarrow$ partition $(S, \frac{n}{2})$
3. $S_1 \leftarrow$ mergeSort($S_1$, $C$)
4. $S_2 \leftarrow$ mergeSort($S_2$, $C$)
5. $S \leftarrow$ merge($S_1$, $S_2$)
6. return $S$
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
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 → 7
2 → 2
9 → 9
4 → 4
3 → 3
8 → 8
6 → 6
1 → 1
```
EXECUTION EXAMPLE

- Recursive Call, 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 → 2 7  
```

```
9 4 → 4 9  
```

```
3 8 → 3 8  
```

```
6 1 → 1 6  
```

```
7 → 7  
```

```
2 → 2  
```

```
9 → 9  
```

```
4 → 4  
```

```
3 → 3  
```

```
8 → 8  
```

```
6 → 6  
```

```
1 → 1  
```
EXECUTION EXAMPLE

- Recursive Call, partition

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

7  2  |  9  4  \rightarrow  2  4  7  9

7  2  |  2  \rightarrow  2  7

9  4  \rightarrow  4  9

3  8  \rightarrow  3  8

6  1  \rightarrow  1  6

7  \rightarrow  7

2  \rightarrow  2

9  \rightarrow  9

4  \rightarrow  4

3  \rightarrow  3

8  \rightarrow  8

6  \rightarrow  6

1  \rightarrow  1
```
EXECUTION EXAMPLE

- Recursive Call, base case

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

- Recursive Call, base case
EXECUTION EXAMPLE

- Merge
- Recursive call, …, base case, merge
EXECUTION EXAMPLE

- Merge

7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9

7 2 | 9 4 → 2 4 7 9

7 | 2 → 2 7
7 → 7 2 → 2

9 | 4 → 4 9
9 → 9 4 → 4

3 8 | 6 1 → 1 3 8 6

3 8 → 3 8
3 → 3 8 → 8
6 → 6 1 → 1
EXECUTION EXAMPLE

- Recursive call, …, merge, merge

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

- Merge

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

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

The running time of Merge Sort can be expressed by the recurrence equation:

\[ T(n) = 2T\left(\frac{n}{2}\right) + M(n) \]

We need to determine \( M(n) \), the time to merge two sorted sequences each of size \( \frac{n}{2} \).

**Algorithm** `mergeSort(S, C)`

**Input:** Sequence \( S \) of \( n \) elements, Comparator \( C \)

**Output:** Sequence \( S \) sorted according to \( C \)

1. if \( S.\text{size}() > 1 \)
2. \((S_1, S_2) \leftarrow \text{partition}(S, \frac{n}{2})\)
3. \( S_1 \leftarrow \text{mergeSort}(S_1, C) \)
4. \( S_2 \leftarrow \text{mergeSort}(S_2, C) \)
5. \( S \leftarrow \text{merge}(S_1, S_2) \)
6. return \( S \)
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 $\frac{n}{2}$ elements and implemented by means of a doubly linked list, takes $O(n)$ time
  - $M(n) = O(n)$

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

**Input:** Sequences $A, B$ with $\frac{n}{2}$ elements each

**Output:** Sorted sequence of $A \cup B$

1. $S \leftarrow \emptyset$
2. while $\neg A.\text{empty}()$ \& $\neg B.\text{empty}()$
3. if $A.\text{front}() < B.\text{front}()$
4. $S.\text{insertBack}(A.\text{front}()); A.\text{eraseFront}()$
5. else
6. $S.\text{insertBack}(B.\text{front}()); B.\text{eraseFront}()$
7. while $\neg A.\text{empty}()$
8. $S.\text{insertBack}(A.\text{front}()); A.\text{eraseFront}()$
9. while $\neg B.\text{empty}()$
10. $S.\text{insertBack}(B.\text{front}()); B.\text{eraseFront}()$
11. return $S$
Algorithm mergeSort($S, C$)

Input: Sequence $S$ of $n$ elements,
     Comparator $C$

Output: Sequence $S$ sorted according to $C$

1. if $S$.size() > 1
2. $(S_1, S_2) \leftarrow$ partition $(S, \frac{n}{2})$
3. $S_1 \leftarrow$ mergeSort($S_1, C$)
4. $S_2 \leftarrow$ mergeSort($S_2, C$)
5. $S \leftarrow$ merge($S_1, S_2$)
6. return $S$

So, the running time of Merge Sort can be expressed by the recurrence equation:

\[
T(n) = 2T\left(\frac{n}{2}\right) + M(n)
\]

\[
= 2T\left(\frac{n}{2}\right) + O(n)
\]

\[
= O(n \log n)
\]
ANOTHER 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 work done at each level is $O(n)$
  - At level $i$, we partition and merge $2^i$ sequences of size $\frac{n}{2^i}$
- 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>
<th>Cost for level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$\frac{n}{2}$</td>
<td>$n$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$i$</td>
<td>$2^i$</td>
<td>$\frac{n}{2^i}$</td>
<td>$n$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$\log n$</td>
<td>$2^{\log n} = n$</td>
<td>$\frac{n}{2^{\log n}} = 1$</td>
<td>$n$</td>
</tr>
</tbody>
</table>
### SUMMARY OF SORTING ALGORITHMS (SO FAR)

<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&lt;br&gt;For small data sets</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$ WC, AC&lt;br&gt;$O(n)$ BC</td>
<td>Slow, in-place&lt;br&gt;For small data sets</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$O(n \log n)$</td>
<td>Fast, in-place&lt;br&gt;For large data sets</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>Fast, sequential data access&lt;br&gt;For huge data sets</td>
</tr>
</tbody>
</table>
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 $G$
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 $x$

**Algorithm partition($S, p$)**

**Input:** Sequence $S$, position $p$ of the pivot

**Output:** Subsequences $L, E, G$ of the elements of $S$
less than, equal to, or greater than the pivot, respectively

1. $L, E, G \leftarrow \emptyset$
2. $x \leftarrow S.\text{erase}(p)$
3. **while** ¬$S.\text{empty}()$
4. $y \leftarrow S.\text{eraseFront}( )$
5. **if** $y < x$
   6. $L.\text{insertBack}(y)$
7. **else if** $y = x$
   8. $E.\text{insertBack}(y)$
8. **else** /*$y > x$*/
9. $G.\text{insertBack}(y)$
10. return $L, E, G$
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 → 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
```

```
4 → 4
```

```
9 → 9
```
EXECUTION EXAMPLE

- Partition, recursive call, pivot selection

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

```
2 4 3 1 \rightarrow 1 2 3 4
```

```
7 9 7 \rightarrow 7 7 9
```

```
1 \rightarrow 1
```

```
4 3 \rightarrow 3 4
```

```
4 \rightarrow 4
```

```
9 \rightarrow 9
```
EXECUTION EXAMPLE

- Partition, recursive call, base case
Recursive call, …, base case, join
EXECUTION EXAMPLE

- Recursive call, pivot selection

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

```
2 4 3 1 → 1 2 3 4
```

```
1 → 1
```

```
4 3 → 3 4
```

```
7 9 7 → 7 7 9
```

```
9 → 9
```

```
4 → 4
```
EXECUTION EXAMPLE

- Partition, …, recursive call, base case
EXECUTION EXAMPLE
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 indices less than $h$
- the elements equal to the pivot have indices between $h$ and $k$
- the elements greater than the pivot have indices greater than $k$

The recursive calls consider:
- elements with indices less than $h$
- elements with indices greater than $k$

**Algorithm** inPlaceQuickSort($S, l, r$)

**Input:** Array $S$, indices $l, r$

**Output:** Array $S$ with the elements between $l$ and $r$ sorted

1. if $l \geq r$
2. return $S$
3. $i \leftarrow \text{rand}() \% (r - l) + l$  
   //random integer between $l$ and $r$
4. $x \leftarrow S[i]$
5. $(h, k) \leftarrow \text{inPlacePartition}(x)$
6. inPlaceQuickSort($S, l, h - 1$)
7. inPlaceQuickSort($S, k + 1, r$)
8. return $S$
IN-PLACE PARTITIONING

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

$$\begin{array}{c}
\mathbf{j} & \mathbf{k} \\
3 & 2 & 5 & 1 & 0 & 7 & 3 & 5 & 9 & 2 & 7 & 9 & 8 & 9 & 7 & \underline{6} & 9
\end{array}$$

(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 $k$
Assumption: random pivot expected to give equal sized sub-lists

The running time of Quick Sort can be expressed as:

\[ T(n) = 2T\left(\frac{n}{2}\right) + P(n) \]

\( P(n) \) - time to run partition( ) on input of size \( n \)

Algorithm quickSort(S, l, r)

Input: Sequence S, indices l, r

Output: Sequence S with the elements between l and r sorted

1. if \( l \geq r \)
2. return S
3. \( i \leftarrow \text{rand}(\ )\% (r - l) + l \) //random integer between l and r
4. \( x \leftarrow S.\text{at}(i) \)
5. \((h, k) \leftarrow \text{partition}(x)\)
6. quickSort(S, l, h - 1)
7. quickSort(S, k + 1, r)
8. return S
ANALYSIS OF PARTITION

- 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 partition($S, p$)**

**Input:** Sequence $S$, position $p$ of the pivot

**Output:** Subsequences $L, E, G$ of the elements of $S$ less than, equal to, or greater than the pivot, respectively

1. $L, E, G \leftarrow \emptyset$
2. $x \leftarrow S.\text{erase}(p)$
3. **while** $\neg S.\text{empty}()$
4. $y \leftarrow S.\text{eraseFront}()$
5. **if** $y < x$
6. $L.\text{insertBack}(y)$
7. **else if** $y = x$
8. $E.\text{insertBack}(y)$
9. **else** // $y > x$
10. $G.\text{insertBack}(y)$
11. **return** $L, E, G$
SO, THE EXPECTED COMPLEXITY OF QUICK SORT

- Assumption: random pivot expected to give equal sized sub-lists
- The running time of Quick Sort can be expressed as:
  \[ T(n) = 2T\left(\frac{n}{2}\right) + P(n) \]
  \[ = 2T\left(\frac{n}{2}\right) + O(n) \]
  \[ = O(n \log n) \]

**Algorithm** quickSort(S, l, r)

**Input:** Sequence S, indices l, r

**Output:** Sequence S with the elements between l and r sorted

1. if \( l \geq r \)
2. return S
3. \( i \leftarrow \text{rand}(\ ) \%(r - l) + l \)
   // random integer between l and r
4. \( x \leftarrow S\.at(i) \)
5. \( (h, k) \leftarrow \text{partition}(x) \)
6. quickSort(S, l, h - 1)
7. quickSort(S, k + 1, r)
8. return S
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:
  $$n + (n - 1) + \cdots + 2 + 1 = O(n^2)$$
- Alternatively, using recurrence equations:
  $$T(n) = T(n - 1) + O(n) = O(n^2)$$

<table>
<thead>
<tr>
<th>Depth</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$n$</td>
</tr>
<tr>
<td>1</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$n - 1$</td>
<td>1</td>
</tr>
</tbody>
</table>
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 $\frac{3s}{4}$.
- **Bad call**: one of $L$ and $G$ has size greater than $\frac{3s}{4}$.

A call is good with probability $1/2$.

1/2 of the possible pivots cause good calls:

- **Good pivots**: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
- **Bad pivots**: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
**EXPECTED RUNNING TIME**

- **Probabilistic Fact:** The expected number of coin tosses required in order to get $k$ heads is $2k$ (e.g., it is expected to take 2 tosses to get heads).

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

- Therefore, we have
  - For a node of depth $2 \log_4 \frac{n}{3}$, 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 $O(n \log n)$

[Diagram showing expected height and time per level]
## SUMMARY OF SORTING ALGORITHMS (SO FAR)

<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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For small data sets</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$ WC, AC</td>
<td>Slow, in-place</td>
</tr>
<tr>
<td></td>
<td>$O(n)$ BC</td>
<td>For small data sets</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$O(n \log n)$</td>
<td>Fast, in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For large data sets</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>Exp. $O(n \log n)$ AC, BC</td>
<td>Fastest, randomized, in-place</td>
</tr>
<tr>
<td></td>
<td>$O(n^2)$ WC</td>
<td>For large data sets</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>Fast, sequential data access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For huge data sets</td>
</tr>
</tbody>
</table>
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 $n$ elements, $x_1, x_2, \ldots, x_n$. 

Is $x_i < x_j$?

no

yes
Let us just count comparisons then.

Each possible run of the algorithm corresponds to a root-to-leaf path in a decision tree.
The height of the decision tree is a lower bound on the running time.

Every 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 $n! = 1 \times 2 \times \cdots \times n$ leaves, the height is at least $\log(n!)$.
Any comparison-based sorting algorithm takes at least $\log(n!)$ time.

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

That is, any comparison-based sorting algorithm must run in $\Omega(n \log n)$ time.
BUCKET-SORT AND RADIX-SORT

CAN WE SORT IN LINEAR TIME?
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 entry into its bucket $B[k]$
- Phase 2: for $i \leftarrow 0 \ldots N - 1$, 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 bucketSort($S, N$)**

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

**Output:** Sequence $S$ sorted in non-decreasing order of the keys

1. $B \leftarrow$ array of $N$ empty sequences
2. for each entry $e \in S$ do
3.  $k \leftarrow e$.key()
4.  remove $e$ from $S$ and insert it at the end of bucket $B[k]$
5. for $i \leftarrow 0 \ldots N - 1$ do
6.  for each entry $e \in B[i]$ do
7.    remove $e$ from bucket $B[i]$ and insert it at the end of $S$
Properties

- Key-type
  - The keys are used as indices into an array and cannot be arbitrary objects
  - No external comparator
- Stable sorting
  - 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 \(e\) 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 index \(i(k)\) of each string \(k\) of \(D\) in the sorted sequence
  - Put item \(e\) into bucket \(B[i(k)]\)
Key range [37, 46] – map to buckets [0,9]
Given a list of tuples:
(7,4,6) (5,1,5) (2,4,6) (2,1,4) (5,1,6) (3,2,4)

After sorting, the list is in lexicographical order:
(2,1,4) (2,4,6) (3,2,4) (5,1,5) (5,1,6) (7,4,6)
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 is a 3-tuple $(x, y, z)$

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.
Given a list of 2-tuples, we can order the tuples lexicographically by applying a stable sorting algorithm two times: (3,3) (1,5) (2,5) (1,2) (2,3) (1,7) (3,2) (2,2)

Possible ways of doing it:
- Sort first by 1st element of tuple and then by 2nd element of tuple
- Sort first by 2nd element of tuple and then by 1st element of tuple

Show the result of sorting the list using both options
EXERCISE
LEXICOGRAPHIC ORDER

- (3,3) (1,5) (2,5) (1,2) (2,3) (1,7) (3,2) (2,2)

Using a stable sort,
- Sort first by 1st element of tuple and then by 2nd element of tuple
- Sort first by 2nd element of tuple and then by 1st element of tuple

Option 1:
- 1st sort: (1,5) (1,2) (1,7) (2,5) (2,3) (2,2) (3,3) (3,2)
- 2nd sort: (1,2) (2,2) (3,2) (2,3) (3,3) (1,5) (2,5) (1,7) - WRONG

Option 2:
- 1st sort: (1,2) (3,2) (2,2) (3,3) (2,3) (1,5) (2,5) (1,7)
- 2nd sort: (1,2) (1,5) (1,7) (2,2) (2,3) (2,5) (3,2) (3,3) - CORRECT
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** lexicographicSort$(S)$

**Input:** Sequence $S$ of $d$-tuples

**Output:** Sequence $S$ sorted in lexicographic order

1. for $i \leftarrow d \ldots 1$ do
2. $\text{stableSort}(S, C_i)$
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 $O\left(d(n + N)\right)$.

**Algorithm** \texttt{radixSort}(S, N)

**Input:** Sequence $S$ of $d$-tuples such that 
\begin{align*}
(0, \ldots, 0) &\leq (x_1, \ldots, x_d) \\
(x_1, \ldots, x_d) &\leq (N - 1, \ldots, N - 1)
\end{align*}
for each tuple $(x_1, \ldots, x_d)$ in $S$

**Output:** Sequence $S$ sorted in lexicographic order

1. \textbf{for} $i \leftarrow d \ldots 1$ \textbf{do}
2. \hspace{1em} set the key $k$ of each entry $(k, (x_1, \ldots, x_d))$ of $S$ to $i$th dimension $x_i$
3. \hspace{1em} \text{bucketSort}(S, N)
EXAMPLE
RADIUS-SORT FOR BINARY NUMBERS

- Sorting a sequence of 4-bit integers
  - \( d = 4, N = 2 \) so \( O(d(n + N)) = O(4(n + 2)) = O(n) \)

\[
\begin{align*}
1001 &\quad 0010 &\quad 1001 &\quad 1001 &\quad 0001 \\
0010 &\quad 1110 &\quad 1101 &\quad 0001 &\quad 0010 \\
1101 &\quad 1001 &\quad 0001 &\quad 1101 &\quad 1001 \\
0001 &\quad 1101 &\quad 1110 &\quad 0001 &\quad 0010 \\
1110 &\quad 0001 &\quad 1110 &\quad 1110 &\quad 1110 \\
\end{align*}
\]

Sort by \( d=4 \)  Sort by \( d=3 \)  Sort by \( d=2 \)  Sort by \( d=1 \)
<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</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$ WC, AC $O(n)$ BC</td>
<td>Slow, in-place</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$O(n \log n)$</td>
<td>Fast, in-place</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>Exp. $O(n \log n)$ AC, BC $O(n^2)$ WC</td>
<td>Fastest, randomized, in-place</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>Fast, sequential data access</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>$O(d(n + N))$, $d$ digits, $N$ range of digit values</td>
<td>Fastest, stable only for integers</td>
</tr>
</tbody>
</table>