CHAPTER 12
SORTING AND SELECTION

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)
DIVIDE AND CONQUER ALGORITHMS
• **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 subproblems recursively
  • **Conquer**: combine the solutions for $S_1, S_2, ..., S_k$ into a solution for $S$

• The base case for the recursion are subproblems of constant size

• Analysis can be done using **recurrence equations** (relations)
DIVIDE AND CONQUER ALGORITHMS
ANALYSIS WITH RECURRENCE EQUATIONS

• When the size of all subproblems 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 \( k \) subproblems \( S_1, S_2, ..., S_k \)
  • There are \( k \) subproblems, each of size \( \frac{n}{c} \) that will be solved recursively
  • \( C(n) \) is the cost of combining the subproblem solutions to get the solution for \( S \)
EXERCISE
RECURRENCE EQUATION SETUP

• 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

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

Algorithm: multiply\((I,J)\)
Input: \( n \)-bit integers \( I,J \)
Output: \( I \times J \)

1. \textbf{if} \( n > 1 \) \textbf{then}
2. Split \( I \) and \( J \) into high and low order halves:
   \( I_h, I_l, J_h, J_l \)
3. \( x_1 \leftarrow I_h \times J_h; \ x_2 \leftarrow I_h \times J_l; \)
4. \( x_3 \leftarrow I_l \times J_h; \ x_4 \leftarrow I_l \times J_l; \)
5. \( Z \leftarrow x_1 \times 2^n + x_2 \times 2^{\frac{n}{2}} + x_3 \times 2^{\frac{n}{2}} + x_4 \)
6. \textbf{else}
7. \( Z \leftarrow I \times J \)
8. return \( Z \)
**EXERCISE**
**RECURRENT EQUATION SETUP**

- Algorithm – transform multiplication of two \( n \)-bit integers \( I \) and \( J \) into multiplication of \( (\frac{n}{2}) \)-bit integers and some additions/shifts.

3. 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 \( n > 1 \) then
2. Split \( I \) and \( J \) into high and low order halves:
   \( I_h, I_l, J_h, J_l \)
3. \( x_1 \leftarrow \text{multiply}(I_h, J_h); \ x_2 \leftarrow \text{multiply}(I_h, J_l) \)
4. \( x_3 \leftarrow \text{multiply}(I_l, J_h); \ x_4 \leftarrow \text{multiply}(I_l, J_l) \)
5. \( Z \leftarrow x_1 \times 2^n + x_2 \times 2^\frac{n}{2} + x_3 \times 2^\frac{n}{2} + x_4 \)
6. else
7. \( Z \leftarrow I \times J \)
8. 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 `multiply(I,J)`

**Input:** \( n \)-bit integers \( I,J \)

**Output:** \( I \times J \)

1. if \( n > 1 \) then
2. Split \( I \) and \( J \) into high and low order halves:
   \( I_h, I_l, J_h, J_l \)
3. \( x_1 \leftarrow \text{multiply}(I_h, J_h); \ x_2 \leftarrow \text{multiply}(I_h, J_l) \)
4. \( x_3 \leftarrow \text{multiply}(I_l, J_h); \ x_4 \leftarrow \text{multiply}(I_l, J_l) \)
5. \( Z \leftarrow x_1 \times 2^n + x_2 \times 2^n + x_3 \times 2^n + x_4 \)
6. else
7. \( Z \leftarrow I \times J \)
8. return \( Z \)
DIVIDE AND CONQUER ALGORITHMS
ANALYSIS WITH RECURRENCE EQUATIONS

• Remaining question: how do we solve recurrence relations?
  • **Iterative substitution** – continually expand a recurrence to yield a summation, then bound the summation
  • **Analyze the recursion tree** – determine work per level and number of levels in a recursion tree. This is not a proof technique, more of an intuitive sketch of a proof
  • **Master theorem (method)** – rule to go directly to solution of recurrence. This is slightly beyond scope of course, but we will see it anyway
In the iterative substitution, or “plug-and-chug,” technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern. Example:

- \( T(n) = 2T\left(\frac{n}{2}\right) + bn \)
  - \( = 2\left(2T\left(\frac{n}{2^2}\right) + b\left(\frac{n}{2}\right)\right) + bn = 2^2T\left(\frac{n}{2^2}\right) + 2bn \)
  - \( = 2^3T\left(\frac{n}{2^3}\right) + 3bn \)
  - \( = \ldots \)
  - \( = 2^iT\left(\frac{n}{2^i}\right) + ibn \)

Note that base, \( T(n) = b \), case occurs when \( 2^i = n \). That is, \( i = \log n \).

So,

\[
T(n) = bn + n \log n = O(n \log n)
\]
THE RECURSION TREE

• Draw the recursion tree for the recurrence relation and look for a pattern.

Example: \( T(n) = 2T \left( \frac{n}{2} \right) + bn \)

<table>
<thead>
<tr>
<th>depth</th>
<th>T’s</th>
<th>size</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>( n )</td>
<td>( bn )</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>( n/2 )</td>
<td>( bn )</td>
</tr>
<tr>
<td>( i )</td>
<td>( 2^i )</td>
<td>( n/2^i )</td>
<td>( bn )</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

• Total time: \( bn + bn \log n = \Theta(n \log n) \)
THE MASTER THEOREM (METHOD)

• Many divide-and-conquer algorithms have the form:

\[ T(n) = aT\left(\frac{n}{b}\right) + f(n) \]

• The master theorem:
  1. If \( f(n) \) is \( O\left(n^{\log_b a - \varepsilon}\right) \), then \( T(n) \) is \( \Theta\left(n^{\log_b a}\right) \)
  2. If \( f(n) \) is \( \Theta\left(n^{\log_b a \log^k n}\right) \), then \( T(n) \) is \( \Theta\left(n^{\log_b a \log^{k+1} n}\right) \)
  3. If \( f(n) \) is \( \Omega\left(n^{\log_b a + \varepsilon}\right) \), then \( T(n) \) is \( \Theta(f(n)) \), provided \( af\left(\frac{n}{b}\right) \leq \delta f(n) \) for some \( \delta < 1 \)

• Examples
  1. \( T(n) = 4T\left(\frac{n}{2}\right) + n \)
     • \( O(n^2) \)
  2. \( T(n) = T\left(\frac{n}{2}\right) + 1 \)
     • \( O(\log n) \), (binary search)
  3. \( T(n) = T\left(\frac{n}{3}\right) + n \log n \)
     • \( O(n \log n) \)
MERGE SORT

7  2 | 9  4 → 2  4  7  9

7 | 2 → 2  7

9 | 4 → 4  9

7 → 7

2 → 2

9 → 9

4 → 4
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

• What is the recurrence relation?

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

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

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

1. if $S$.size() > 1 then
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 $S$
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 \( S\.size() > 1 \) then
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 $\neg A$.isEmpty() \& \& $\neg B$.isEmpty() do
3. if $A$.first() < $B$.first() then
4. $S$.addLast($A$.removeFirst())
5. else
6. $S$.addLast($B$.removeFirst())
7. while $\neg A$.isEmpty() do
8. $S$.addLast($A$.removeFirst())
9. while $\neg B$.isEmpty() do
10. $S$.addLast($B$.removeFirst())
11. return $S$
• So, the running time of Merge Sort can be expressed by the recurrence equation:

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

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

\[ = O(n \log n) \]

---

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

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

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

1. if `S.size() > 1` then
2. \((S_1,S_2) \leftarrow \text{partition}(S, \frac{n}{2})\)
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
EXECUTION EXAMPLE

• Recursive Call, partition

7 2 9 4 | 3 8 6 1

7 2 9 4

7 2 | 9 4

1 2 3 4 6 7 8 9
EXECUTION EXAMPLE

• Recursive Call, partition

1. Initial Call

2. First Partition

3. Second Partition

4. Final Results
EXECUTION EXAMPLE

• Recursive Call, base case

7  2  9  4  |  3  8  6  1

7  2  9  4

7  2  |  9  4

7  |  2

7  →  7
EXECUTION EXAMPLE

• Recursive Call, base case
EXECUTION EXAMPLE

- Merge
EXECUTION EXAMPLE

- Recursive call, ..., base case, merge
EXECUTION EXAMPLE

- 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
```

```
```
```
EXECUTION EXAMPLE

• Recursive call, ..., merge, merge
EXECUTION EXAMPLE

• Merge
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>$n/2$</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>
<tr>
<td>Algorithm</td>
<td>Time</td>
<td>Notes</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-----------</td>
<td>--------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>Slow, in-place</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For small data sets (&lt; 1K)</td>
<td></td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$ WC, AC $O(n)$ BC</td>
<td>Slow, in-place</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For small data sets (&lt; 1K)</td>
<td></td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$O(n \log n)$</td>
<td>Fast, in-place</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For large data sets (1K – 1M)</td>
<td></td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>Fast, sequential data access</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For huge data sets (&gt;1M)</td>
<td></td>
</tr>
</tbody>
</table>
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 \( G \)
ANALYSIS OF QUICK SORT USING RECURRENCE RELATIONS

• Assumption: random pivot expected to give equal sized sublists

• The running time of Quick Sort can be expressed as:
  \[ T(n) = 2T \left( \frac{n}{2} \right) + P(n) \]

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

Algorithm quickSort(S)
Input: Sequence S
Output: Sequence S with the elements sorted

1. if S.size() ≤ 1 then
2. return S
3. \( i \leftarrow \text{rand()} \% (r - l) + l \) //random integer
4. //between \( l \) and \( r \)
5. \( x \leftarrow S.\text{at}(i) \)
6. \( (L,E,G) \leftarrow \text{partition}(x) \)
7. quickSort(L)
8. quickSort(G)
9. return splice(L, E, 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 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. \( x \leftarrow S\.\text{remove}(p) \)
3. while \( \neg S\.\text{isEmpty()} \) do
4. \( y \leftarrow S\.\text{removeFirst()} \)
5. if \( y < x \) then
6. \( L\.\text{addLast}(y) \)
7. else if \( y = x \) then
8. \( E\.\text{addLast}(y) \)
9. else \( /\!/ y > x \)
10. \( G\.\text{addLast}(y) \)
11. return \( L, E, G \)
SO, THE EXPECTED COMPLEXITY OF QUICK SORT

• Assumption: random pivot expected to give equal sized sublists

• 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)
Input: Sequence S
Output: Sequence S with the elements sorted

1. if S.size() ≤ 1 then
2. return S
3. \( i \leftarrow \text{rand()} \% (r - l) + l \) //random integer
4. //between \( l \) and \( r \)
5. \( x \leftarrow S.\text{at}(i) \)
6. \((L, E, G) \leftarrow \text{partition}(x)\)
7. quickSort(L)
8. quickSort(G)
9. return splice(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
EXECUTION EXAMPLE

- Partition, recursive call, pivot selection
EXECUTION EXAMPLE

• Partition, recursive call, base case
EXECUTION EXAMPLE

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

- Recursive call, pivot selection

```
2  4  3  1  →  1  2  3  4
```

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

```
7  9  7
```

```
1  →  1
```

```
4  3  →  3  4
```

```
4  →  4
```

```
```
EXECUTION EXAMPLE

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

• Join, join

```
7 2 9 4 3 7 6 1 -> 1 2 3 4 6 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
```
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:

\[ T(n) = T(n - 1) + O(n) = O(n^2) \]
EXPECTED RUNNING TIME
REMOVING EQUAL SPLIT ASSUMPTION

• 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 call
- Bad call

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Bad pivots Good pivots Bad pivots
**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 $\left(\frac{3}{4}\right)^{\frac{i}{2}} n$
- Therefore, we have
  - For a node of depth $\frac{2 \log_2 n}{3}$, 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 $O(n \log n)$
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 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** \( l \geq r \) **then**
2. **return** \( S \)
3. \( i \leftarrow \text{rand}()\% (r - l) + l \) //random integer
4. //between \( l \) and \( r \)
5. \( x \leftarrow S[i] \)
6. \((h, k) \leftarrow \text{inPlacePartition}(x)\)
7. inPlaceQuickSort\((S, l, h - 1)\)
8. inPlaceQuickSort\((S, k + 1, r)\)
9. **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 $G$).

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

  (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 $k$.

\[
\begin{array}{c}
\text{3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 6 9} \\
\end{array}
\]
## 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>In-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Slow, for small data sets</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$ WC, AC</td>
<td>In-place</td>
</tr>
<tr>
<td></td>
<td>$O(n)$ BC</td>
<td>Slow, for small data sets</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$O(n \log n)$</td>
<td>In-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fast, For large data sets</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>Exp. $O(n \log n)$ AC, BC</td>
<td>Randomized, in-place</td>
</tr>
<tr>
<td></td>
<td>$O(n^2)$ WC</td>
<td>Fastest, for large data sets</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>Sequential data access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fast, for huge data sets</td>
</tr>
</tbody>
</table>
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, $x_1, x_2, \ldots, x_n$. 

Is $x_i < x_j$?

no

yes
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.
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 the 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 $\log(n!)$.
THE LOWER BOUND

• Any comparison-based sorting algorithm takes at least \( \log(n!) \) time

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

• 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?

1, c  ->  3, a  ->  3, b  ->  7, d  ->  7, g  ->  7, e
B
∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅
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 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 nondecreasing order of the keys

1. \( B \leftarrow \) array of \( N \) empty sequences
2. for each entry \( e \in S \) do
3. \( k \leftarrow e \text{.key()} \)
4. remove \( e \) from \( S \)
5. insert \( e \) at the end of sequence \( B[k] \)
6. for \( i \leftarrow 0 \) to \( N - 1 \) do
7. for each entry \( e \in B[i] \) do
8. remove \( e \) from bucket \( B[i] \)
9. insert \( e \) at the end of \( S \)
EXAMPLE

- Key range [37, 46] – map to buckets [0,9]

Phase 1

Phase 2
PROPERTIES AND EXTENSIONS

• 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 \(B[i(k)]\)
LEXICOGRAPHIC ORDER

• 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.
EXERCISE
LEXICOGRAPHIC ORDER

• 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**
**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 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

1. for $i \leftarrow d$ to 1 do
2.   stableSort($S, C_i$)
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** 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.

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

• Sorting a sequence of 4-bit integers

\[ d = 4, N = 2 \text{ so } O(d(n + N)) = O(4(n + 2)) = O(n) \]

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>In-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Slow, for small data sets</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$ WC, AC</td>
<td>In-place</td>
</tr>
<tr>
<td></td>
<td>$O(n)$ BC</td>
<td>Slow, for small data sets</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$O(n \log n)$</td>
<td>In-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fast, for large data sets</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>Exp. $O(n \log n)$ AC, BC</td>
<td>Randomized, in-place</td>
</tr>
<tr>
<td></td>
<td>$O(n^2)$ WC</td>
<td>Fastest, for in-place data sets</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>Sequential data access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fast, for huge data sets</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>$O(d(n + N))$, d #digits, N range of digit values</td>
<td>Stable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fastest, only for integers</td>
</tr>
</tbody>
</table>
THE SELECTION PROBLEM

• Given an integer $k$ and $n$ elements $\{x_1, x_2, ..., x_n\}$, taken from a total order, find the $k$-th smallest element in this set.
  • Also called order statistics, the $i$th order statistic is the $i$th smallest element
  • Minimum - $k = 1$ - 1st order statistic
  • Maximum - $k = n$ - $n$th order statistic
  • Median - $k = \left\lceil \frac{n}{2} \right\rceil$
  • etc
THE SELECTION PROBLEM

• Naïve solution - SORT!

• We can sort the set in \( O(n \log n) \) time and then index the \( k \)-th element.

• Can we solve the selection problem faster?
THE MINIMUM (OR MAXIMUM)

**Algorithm** minimum(A)

**Input:** Array A

**Output:** minimum element in A

1. \( m \leftarrow A[1] \)
2. **for** \( i \leftarrow 2 \) **to** \( n \) **do**
3. \( m \leftarrow \min(m, A[i]) \)

**return** \( m \)

• Running Time
  • \( O(n) \)

• Is this the best possible?
**QUICK-SELECT**

- **Quick-select** is a randomized selection algorithm based on the prune-and-search paradigm:
  - **Prune**: pick a random element \( x \) (called pivot) and partition \( S \) into
    - \( L \) elements < \( x \)
    - \( E \) elements = \( x \)
    - \( G \) elements > \( x \)
  - **Search**: depending on \( k \), either answer is in \( E \), or we need to recur on either \( L \) or \( G \)

- Note: Partition same as Quicksort

\[
\begin{align*}
|L| &< k \leq |L| + |E| \\
\text{(done)}
\end{align*}
\]

\[
k > |L| + |E| \\
k' = k - |L| - |E|
\]
QUICK-SELECT VISUALIZATION

• An execution of quick-select can be visualized by a recursion path
  • Each node represents a recursive call of quick-select, and stores $k$ and the remaining sequence

\[
\begin{align*}
  k &= 5, S = (7, 4, 9, 3, 2, 6, 5, 1, 8) \\
  k &= 2, S = (7, 4, 9, 6, 5, 8) \\
  k &= 2, S = (7, 4, 6, 5) \\
  k &= 1, S = (7, 6, 5) \\
  &5
\end{align*}
\]
EXERCISE

• Best Case - even splits (n/2 and n/2)
• Worst Case - bad splits (1 and n-1)

• Derive and solve the recurrence relation corresponding to the best case performance of randomized quick-select.

• Derive and solve the recurrence relation corresponding to the worst case performance of randomized quick-select.
EXPECTED RUNNING TIME

• Consider a recursive call of quick-select on a sequence of size $s$
  • Good call: the size of $L$ and $G$ is at most $\frac{3s}{4}$
  • Bad call: the size of $L$ and $G$ is greater than $\frac{3s}{4}$

• A call is good with probability $1/2$
  • $1/2$ of the possible pivots cause good calls:
EXPECTED RUNNING TIME

• Probabilistic Fact #1: The expected number of coin tosses required in order to get one head is two

• Probabilistic Fact #2: Expectation is a linear function:
  • $E(X + Y) = E(X) + E(Y)$
  • $E(cX) = cE(X)$

• Let $T(n)$ denote the expected running time of quick-select.

• By Fact #2, $T(n) < T\left(\frac{3n}{4}\right) + bn \ast \text{(expected \# of calls before a good call)}$

• By Fact #1, $T(n) < T\left(\frac{3n}{4}\right) + 2bn$

• That is, $T(n)$ is a geometric series: $T(n) < 2bn + 2b \left(\frac{3}{4}\right)n + 2b \left(\frac{3}{4}\right)^2 n + 2b \left(\frac{3}{4}\right)^3 n + \cdots$

• So $T(n)$ is $O(n)$.

• We can solve the selection problem in $O(n)$ expected time.
DETERMINISTIC SELECTION

- We can do selection in $O(n)$ worst-case time.
- Main idea: recursively use the selection algorithm itself to find a good pivot for quick-select:
  - Divide $S$ into $\frac{n}{5}$ sets of 5 each
  - Find a median in each set
  - Recursively find the median of the “baby” medians.

- See Exercise C-12.56 for details of analysis.
INTERVIEW QUESTION 1

• You are given two sorted arrays, $A$ and $B$, where $A$ has a large enough buffer at the end to hold $B$. Write a method to merge $B$ into $A$ in sorted order.
INTERVIEW QUESTION 2

• Write a method to sort an array of strings so that all the anagrams are next to each other.
  • Two words are anagrams if they use the exact same letters, i.e., race and care are anagrams

INTERVIEW QUESTION 3

• Imagine you have a 2 TB file with one string per line. Explain how you would sort the file.

EXAM 4 - PROGRAMMING

• Hack sheet – you can have a single 8 ½" x 11" paper with handwritten notes on both sides with you during the exam.

• Can bring blank pieces of paper for scratch work.

• Internet only to access the API. Focus is on programming skills.

• Be sure to follow all naming conventions in the exam.

• Format – 1 multi-part question and a bonus (1 hour)
  • Q1 – Anything, but it will have unit testing, generic programming, File IO, and some use of Java library
  • Bonus – ?
EXAM 4 - WRITTEN

• Separate Hack sheet – you can have a single 8 ½" x 11" paper with handwritten notes on both sides with you during the exam.

• Can bring blank sheets of paper for scratch work.

• Focus on algorithmic skills and programming concepts.

• Format – 5 questions and a bonus (2 hours, you will have to write two algorithms)
  • Q1 – T/F and fill-in-the-blank (similar to quizzes)
  • Q2 – Fill-in-the-blank and/or tracing (similar to quizzes)
  • Q3 – Write and/or analyze algorithm (similar to homework)
  • Q4 – Write and/or analyze algorithm (similar to homework)
  • Q5 – Write and/or analyze algorithm (similar to homework)
  • Bonus – ?