**Chapter 9 B**

Algorithm

Efficiency and

Sorting

---

**Bubble Sort**

- **Bubble sort**
  - **Strategy**
    - Compare adjacent elements and exchange them if they are out of order
    - Comparing the first two elements, the second and third elements, and so on, will move the largest (or smallest) elements to the end of the array
    - Repeating this process will eventually sort the array into ascending (or descending) order

---

**Figure 9.5**

The first two passes of a bubble sort of an array of five integers: a) pass 1; b) pass 2

---

**Bubble Sort**

- **Analysis**
  - Worst case: $O(n^2)$
  - Best case: $O(n)$

---

**Insertion Sort**

- **Insertion sort**
  - **Strategy**
    - Partition the array into two regions: sorted and unsorted
    - Take each item from the unsorted region and insert it into its correct order in the sorted region

---

**Figure 9.6**

An insertion sort partitions the array into two regions

---

**Insertion Sort**

Initial array:

<table>
<thead>
<tr>
<th>29</th>
<th>10</th>
<th>14</th>
<th>37</th>
<th>13</th>
</tr>
</thead>
</table>

Copy 10

$\rightarrow$

<table>
<thead>
<tr>
<th>29</th>
<th>10</th>
<th>14</th>
<th>37</th>
<th>13</th>
</tr>
</thead>
</table>

Shift 29

<table>
<thead>
<tr>
<th>10</th>
<th>29</th>
<th>14</th>
<th>37</th>
<th>13</th>
</tr>
</thead>
</table>

Insert 10; copy 14

<table>
<thead>
<tr>
<th>10</th>
<th>29</th>
<th>14</th>
<th>37</th>
<th>13</th>
</tr>
</thead>
</table>

Shift 29

<table>
<thead>
<tr>
<th>10</th>
<th>29</th>
<th>14</th>
<th>37</th>
<th>13</th>
</tr>
</thead>
</table>

Insert 14; copy 37; insert 37 on top of itself

<table>
<thead>
<tr>
<th>10</th>
<th>29</th>
<th>14</th>
<th>37</th>
<th>13</th>
</tr>
</thead>
</table>

Copy 13

<table>
<thead>
<tr>
<th>10</th>
<th>29</th>
<th>14</th>
<th>37</th>
<th>13</th>
</tr>
</thead>
</table>

Shift 37, 29, 14

<table>
<thead>
<tr>
<th>10</th>
<th>14</th>
<th>29</th>
<th>37</th>
<th>13</th>
</tr>
</thead>
</table>

Insert 13

Sorted array:

<table>
<thead>
<tr>
<th>10</th>
<th>14</th>
<th>29</th>
<th>37</th>
<th>13</th>
</tr>
</thead>
</table>

---

**Figure 9.7**

An insertion sort of an array of five integers.
Insertion Sort

- Analysis
  - Worst case: O(n^2)
  - For small arrays
    - Insertion sort is appropriate due to its simplicity
  - For large arrays
    - Insertion sort is prohibitively inefficient

Mergesort

- Important divide-and-conquer sorting algorithms
  - Mergesort
  - Quicksort
  - Mergesort
    - A recursive sorting algorithm
    - Gives the same performance, regardless of the initial order of the array items
    - Strategy
      - Divide an array into halves
      - Sort each half
      - Merge the sorted halves into one sorted array

Mergesort

- Analysis
  - Worst case: O(n * log_2 n)
  - Average case: O(n * log_2 n)
  - Advantage
    - It is an extremely efficient algorithm with respect to time
  - Drawback
    - It requires a second array as large as the original array

Quicksort

- Quicksort
  - A divide-and-conquer algorithm
  - Strategy
    - Partition an array into items that are less than the pivot and those that are greater than or equal to the pivot
    - Sort the left section
    - Sort the right section

Figure 9.8
A mergesort with an auxiliary temporary array

Figure 9.9
A mergesort of an array of six integers

Figure 9.12
A partition about a pivot
Quicksort

- Using an invariant to develop a partition algorithm
  - Invariant for the partition algorithm
    The items in region $S_1$ are all less than the pivot, and those in $S_2$ are all greater than or equal to the pivot.

  ![Partition Algorithm Invariant](image)

- Analysis
  - Worst case
    - quicksort is $O(n^2)$ when the array is already sorted and the smallest item is chosen as the pivot.

  ![Worst-case Partitioning](image)

- Average case
  - quicksort is $O(n \log n)$ when $S_1$ and $S_2$ contain the same or nearly the same number of items arranged at random.

  ![Average-case Partitioning](image)

- Analysis
  - quicksort is usually extremely fast in practice.
  - Even if the worst case occurs, quicksort's performance is acceptable for moderately large arrays.

Radix Sort

- Radix sort
  - Treats each data element as a character string.
  - Strategy
    - Repeatedly organize the data into groups according to the $i^{th}$ character in each element.

- Analysis
  - Radix sort is $O(n)$.
Summary

- Order-of-magnitude analysis and Big O notation measure an algorithm’s time requirement as a function of the problem size by using a growth-rate function
- To compare the inherent efficiency of algorithms
  – Examine their growth-rate functions when the problems are large
  – Consider only significant differences in growth-rate functions

Summary

- Worst-case and average-case analyses
  – Worst-case analysis considers the maximum amount of work an algorithm will require on a problem of a given size
  – Average-case analysis considers the expected amount of work that an algorithm will require on a problem of a given size
- Order-of-magnitude analysis can be used to choose an implementation for an abstract data type
- Selection sort, bubble sort, and insertion sort are all $O(n^2)$ algorithms
- Quicksort and mergesort are two very efficient sorting algorithms

A Comparison of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst case</th>
<th>Average case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$n \cdot \log n$</td>
<td>$n \cdot \log n$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$n^2$</td>
<td>$n \cdot \log n$</td>
</tr>
<tr>
<td>Radix sort</td>
<td>$n$</td>
<td>$n \cdot \log n$</td>
</tr>
<tr>
<td>Treesort</td>
<td>$n^2$</td>
<td>$n \cdot \log n$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$n \cdot \log n$</td>
<td>$n \cdot \log n$</td>
</tr>
</tbody>
</table>

Figure 9.22
Approximate growth rates of time required for eight sorting algorithms