Measuring the efficiency of Algorithms

Measuring an algorithms efficiency is important

- **Time efficiency**
  - Running time - the time needed to execute all the operations specified in the algorithm.
    - Grocery checkout systems
    - Automatic teller machines
    - Video games

- **Space efficiency**
  - Memory space - the amount of space used to store all data processed by the algorithm.
The execution time of algorithms

- analyze the time complexity of algorithms by determining the number of operations, such as comparisons and arithmetic operations (addition, multiplication, etc.).
  - Worst case efficiency
  - Average case efficiency

- the order of an algorithm

- Algorithm A is order \( f(n) \) – denoted \( O(f(n)) \) – if constants \( k \) and \( n_0 \) exist such that A requires no more than \( k \ast f(n) \) time units to solve a problem of size \( n \geq n_0 \)

<table>
<thead>
<tr>
<th>Function</th>
<th>10</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( \log_2 n )</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>16</td>
<td>19</td>
</tr>
<tr>
<td>( n )</td>
<td>10</td>
<td>10^2</td>
<td>10^3</td>
<td>10^4</td>
<td>10^5</td>
<td>10^6</td>
</tr>
<tr>
<td>( n \ast \log_2 n )</td>
<td>30</td>
<td>664</td>
<td>9,965</td>
<td>10^5</td>
<td>10^6</td>
<td>10^7</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>10^2</td>
<td>10^4</td>
<td>10^6</td>
<td>10^8</td>
<td>10^10</td>
<td>10^12</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>10^3</td>
<td>10^6</td>
<td>10^9</td>
<td>10^12</td>
<td>10^15</td>
<td>10^18</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>10^3</td>
<td>10^{30}</td>
<td>10^{301}</td>
<td>10^{3,010}</td>
<td>10^{30,103}</td>
<td>10^{301,030}</td>
</tr>
</tbody>
</table>
O(1) < O(log_2 n) < O(n) < O(n * log_2 n) < O(n^2) < O(n^3) < O(2^n)

**Properties of growth-rate functions**

- You can ignore low-order terms
- You can ignore a multiplicative constant in the high-order term
- $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
The Efficiency of Searching Algorithms

- **Sequential search**
  - **Strategy**: Look at each item in the data collection in turn, beginning with the first one. Stop when you find the desired item or you reach the end of the data collection.
  - **Efficiency**
    - Worst case: $O(n)$
    - Average case: $O(n)$
    - Best case: $O(1)$

- **Binary search**
  - **Strategy**: To search a sorted array for a particular item, repeatedly divide the array in half. Determine which half the item must be in, if it is indeed present, and discard the other half.
  - **Efficiency**
    - Worst case: $O(\log_2 n)$
  - For large arrays, the binary search has an enormous advantage over a sequential search.
Sorting Algorithms and Their Efficiency

- **Sorting**
  - A process that organizes a collection of data into either ascending or descending order

- **Categories of sorting algorithms**
  - **An internal sort**
    - Requires that the collection of data fit entirely in the computer’s main memory
  - **An external sort**
    - The collection of data will not fit in the computer’s main memory all at once but must reside in secondary storage
Data items to be sorted can be
- Integers
- Character strings
- Objects

Sort key
- The part of a record that determines the sorted order of the entire record within a collection of records

Selection sort
- Select the largest item and put it in its correct place
- Select the next largest item and put it in its correct place, etc.

Shaded elements are selected; boldface elements are in order.

<table>
<thead>
<tr>
<th>Initial array:</th>
<th>29</th>
<th>10</th>
<th>14</th>
<th>37</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>After 1\textsuperscript{st} swap:</td>
<td>29</td>
<td>10</td>
<td>14</td>
<td>13</td>
<td>37</td>
</tr>
<tr>
<td>After 2\textsuperscript{nd} swap:</td>
<td>13</td>
<td>10</td>
<td>14</td>
<td>29</td>
<td>37</td>
</tr>
<tr>
<td>After 3\textsuperscript{rd} swap:</td>
<td>13</td>
<td>10</td>
<td>14</td>
<td>29</td>
<td>37</td>
</tr>
<tr>
<td>After 4\textsuperscript{th} swap:</td>
<td>10</td>
<td>13</td>
<td>14</td>
<td>29</td>
<td>37</td>
</tr>
</tbody>
</table>
Bubble sort

- 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

Worst case: $O(n^2)$
Best case: $O(n)$

Insertion sort

- 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
### Insertion sort

- **Initial array:**
  - 29 10 14 37 13
  - 29 29 14 37 13
  - 10 29 14 37 13
  - 10 29 29 37 13
  - 10 14 29 37 13
  - 10 14 14 29 37
  - Sorted array: 10 13 14 29 37

- **Copy 10**
- **Shift 29**
- **Insert 10, copy 14**
- **Shift 29**
- **Insert 14, copy 37, insert 37 on top of itself**
- **Copy 13**
- **Shift 37, 29, 14**

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

---

### Mergesort

- **A recursive sorting algorithm**
  - Gives the same performance, regardless of the initial order of the array items
  - **Divide an array into halves**
  - **Sort each half**
  - **Merge the sorted halves into one sorted array**

- **Sort the halves**
- **Merge the halves:**
  - a. $1 < 2$, so move 1 from left half to temparray
  - b. $4 > 2$, so move 2 from right half to temparray
  - c. $4 > 3$, so move 3 from right half to temparray
  - d. Right half is finished, so move rest of left half to temparray
- **Copy temporary array back into original array**
Worst case: $O(n \times \log_2 n)$
Average case: $O(n \times \log_2 n)$

Quicksort
- 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

Original array: 5 6 7 8 9
Pivot: Unknown
First partition: 5 6 7 8 9

$S_1$ is empty
$S_2$ is empty
4 comparisons, 0 exchanges
Radix sort
- Treats each data element as a character string
- Repeatedly organizes the data into groups according to the \( i \)th character in each element

\[ O(n) \]

<table>
<thead>
<tr>
<th>Original integers</th>
<th>Grouped by fourth digit</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150</td>
<td>(1560, 2150)</td>
<td>(0123, 0283)</td>
</tr>
<tr>
<td>1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004</td>
<td>Combined</td>
<td></td>
</tr>
<tr>
<td>(0004)</td>
<td>(0222, 0123)</td>
<td>(2150, 2154)</td>
</tr>
<tr>
<td>0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283</td>
<td>Combined</td>
<td></td>
</tr>
<tr>
<td>(0004, 1061)</td>
<td>(0123, 2150, 2154)</td>
<td>(0222, 0283)</td>
</tr>
<tr>
<td>0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560</td>
<td>Combined</td>
<td></td>
</tr>
<tr>
<td>(0004, 0123, 0222, 0283)</td>
<td>(1061, 1560)</td>
<td>(2150, 2154)</td>
</tr>
<tr>
<td>0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154</td>
<td>Combined (sorted)</td>
<td></td>
</tr>
</tbody>
</table>

Comparison

<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 \log n )</td>
<td>( n \log n )</td>
</tr>
<tr>
<td>Quicksort</td>
<td>( n^2 )</td>
<td>( n \log n )</td>
</tr>
<tr>
<td>Radix sort</td>
<td>( n^2 )</td>
<td>( n \log n )</td>
</tr>
<tr>
<td>Treesort</td>
<td>( n^2 )</td>
<td>( n \log n )</td>
</tr>
<tr>
<td>Heapsort</td>
<td>( n \log n )</td>
<td>( n \log n )</td>
</tr>
</tbody>
</table>
Read through Chapter 10