Chapter 7

Space and Time Tradeoffs
Space-for-time tradeoffs

Two varieties of space-for-time algorithms:

- **input enhancement** — preprocess the input (or its part) to store some info to be used later in solving the problem
  - counting sorts
  - string searching algorithms

- **prestructuring** — preprocess the input to make accessing its elements easier
  - hashing
  - indexing schemes (e.g., B-trees)
Lower bounds for sorting

Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i:j$ indicates a comparison between $a_i$ and $a_j$. A leaf annotated by the permutation \((\pi(1), \pi(2), \ldots, \pi(n))\) indicates the ordering \(a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}\). The shaded path indicates the decisions made when sorting the input sequence \((a_1 = 6, a_2 = 8, a_3 = 5)\); the permutation \((3, 1, 2)\) at the leaf indicates that the sorted ordering is \(a_3 = 5 \leq a_1 = 6 \leq a_2 = 8\). There are \(3! = 6\) possible permutations of the input elements, and so the decision tree must have at least 6 leaves.
Comparison-counting Sort

### Array A[0..5]

<table>
<thead>
<tr>
<th>Initially</th>
<th>Count []</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>After pass $i = 0$</td>
<td>Count []</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>After pass $i = 1$</td>
<td>Count []</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>After pass $i = 2$</td>
<td>Count []</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>After pass $i = 3$</td>
<td>Count []</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>After pass $i = 4$</td>
<td>Count []</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final state</td>
<td>Count []</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

### Array S[0..5]

| 19    | 31    | 47    | 62    | 84    | 96    |

**FIGURE 7.1** Example of sorting by comparison counting.
Comparison-counting Sort

ALGORITHM  ComparisonCountingSort(A[0..n − 1])

// Sorts an array by comparison counting
// Input: An array A[0..n − 1] of orderable elements
// Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
for i ← 0 to n − 1 do Count[i] ← 0
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] < A[j]
            Count[j] ← Count[j] + 1
        else Count[i] ← Count[i] + 1
    for i ← 0 to n − 1 do S[Count[i]] ← A[i]
return S
Analysis of Comparison-counting Sort

\[
C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n - 1) - (i + 1) + 1] = \sum_{i=0}^{n-2} (n - 1 - i) = \frac{n(n - 1)}{2}.
\]
Distribution-counting Sort

EXAMPLE  Consider sorting the array

\[
\begin{array}{cccccc}
13 & 11 & 12 & 13 & 12 & 12 \\
\end{array}
\]

whose values are known to come from the set \{11, 12, 13\} and should not be overwritten in the process of sorting. The frequency and distribution arrays are as follows:

<table>
<thead>
<tr>
<th>Array values</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequencies</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Distribution values</td>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
Distribution-counting Sort

FIGURE 7.2 Example of sorting by distribution counting. The distribution values being decremented are shown in bold.

| A[0] = 13 | 0 | 1 | 5 |
| A[1] = 11 | 1 | 1 | 5 |

D[0..2]

| S[0..5] |
| 12 |
| 12 |
| 11 |
| 13 |

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.
**Distribution-counting Sort**

**ALGORITHM**  
`DistributionCountingSort(A[0..n − 1], l, u)`

//Sorts an array of integers from a limited range by distribution counting
//Input: An array A[0..n − 1] of integers between l and u (l ≤ u)
//Output: Array S[0..n − 1] of A’s elements sorted in nondecreasing order

for `j ← 0` to `u − l` do `D[j] ← 0`  //initialize frequencies
for `i ← n − 1` downto 0 do
  `j ← A[i] − l`
  `D[j] ← D[j] − 1`

return `S`
Radix sort

Figure 8.3  The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

**Algorithm**: Radix-Sort \( A, d \)

1. for \( i = 1 \) to \( d \)
2. use a stable sort to sort array \( A \) on digit \( i \)
Review: String searching by brute force

**pattern**: a string of \( m \) characters to search for

**text**: a (long) string of \( n \) characters to search in

**Brute force algorithm**

**Step 1**  Align pattern at beginning of text

**Step 2**  Moving from left to right, compare each character of pattern to the corresponding character in text until either all characters are found to match (successful search) or a mismatch is detected

**Step 3**  While a mismatch is detected and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2
String searching by preprocessing

Several string searching algorithms are based on the input enhancement idea of preprocessing the pattern

- Knuth-Morris-Pratt (KMP) algorithm preprocesses pattern left to right to get useful information for later searching

- Boyer-Moore algorithm preprocesses pattern right to left and store information into two tables

- Horspool’s algorithm simplifies the Boyer-Moore algorithm by using just one table
Horspool’s Algorithm

A simplified version of Boyer-Moore algorithm:

- preprocesses pattern to generate a shift table that determines how much to shift the pattern when a mismatch occurs

- always makes a shift based on the text’s character \( c \) aligned with the last character in the pattern according to the shift table’s entry for \( c \)
How far to shift?

Look at first (rightmost) character in text that was compared:

- The character is not in the pattern
  
  \[
  \ldots \ldots C \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots (c \text{ not in pattern})
  \]
  
  BAOBAB

- The character is in the pattern (but not the rightmost)
  
  \[
  \ldots \ldots O \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots (O \text{ occurs once in pattern})
  \]
  
  BAOBAB

  \[
  \ldots \ldots A \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots (A \text{ occurs twice in pattern})
  \]
  
  BAOBAB

- The rightmost characters do match
  
  \[
  \ldots \ldots B \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots
  \]
  
  BAOBAB
Shift table

- Shift sizes can be precomputed by the formula:
  
  $t(c) = \begin{cases} 
  \text{pattern’s length } m, & \text{otherwise} \\
  \text{distance from } c \text{’s rightmost occurrence in pattern} \\
  \text{among its first } m-1 \text{ characters to its right end} 
  \end{cases}$

- Shift table is indexed by text and pattern alphabet.
  
  Example, for BAOBAB:

  | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
  | 1 | 2 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 3 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
Example of Horspool’s alg. application

BARD  LOVED  BANANAS

BAOBAB

BAOBAB

BAOBAB

BAOBAB  (unsuccessful search)
Example of Horspool’s alg. application

EXAMPLE As an example of a complete application of Horspool’s algorithm, consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores). The shift table, as we mentioned, is filled as follows:

<table>
<thead>
<tr>
<th>character ( c )</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>...</th>
<th>R</th>
<th>...</th>
<th>Z</th>
<th>_</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift ( t(c) )</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

The actual search in a particular text proceeds as follows:

J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R
B A R B E R
B A R B E R
B A R B E R

Boyer-Moore algorithm

Based on same two ideas:

- comparing pattern characters to text from right to left
- precomputing shift sizes in two tables
  - *bad-symbol table* indicates how much to shift based on text’s character causing a mismatch
  - *good-suffix table* indicates how much to shift based on matched part (suffix) of the pattern
Bad-symbol shift in Boyer-Moore algorithm

- If the rightmost character of the pattern doesn’t match, BM algorithm acts as Horspool’s.
- If the rightmost character of the pattern does match, BM compares preceding characters right to left until either all pattern’s characters match or a mismatch on text’s character $c$ is encountered after $k > 0$ matches.

**bad-symbol shift** $d_1 = \max\{t_1(c) - k, 1\}$
Good-suffix shift in Boyer-Moore algorithm

- Good-suffix shift $d_2$ is applied after $0 < k < m$ last characters were matched.

- $d_2(k) =$ the distance between matched suffix of size $k$ and its rightmost occurrence in the pattern that is not preceded by the same character as the suffix.

Example: CABABA $d_2(1) = 4$

- If there is no such occurrence, match the longest part of the $k$-character suffix with corresponding prefix; if there are no such suffix-prefix matches, $d_2(k) = m$.

Example: WOWWOW $d_2(2) = 5$, $d_2(3) = 3$, $d_2(4) = 3$, $d_2(5) = 3$
Boyer-Moore Algorithm

After matching successfully $0 < k < m$ characters, the algorithm shifts the pattern right by

$$d = \max \{d_1, d_2\}$$

where $d_1 = \max \{t_1(c) - k, 1\}$ is bad-symbol shift

$d_2(k)$ is good-suffix shift

Example: Find pattern AT_THAT in

WHICH_FINALLY_HALTS . __ AT_THAT
Boyero-Moore Algorithm (cont.)

Step 1  Fill in the bad-symbol shift table
Step 2  Fill in the good-suffix shift table
Step 3  Align the pattern against the beginning of the text
Step 4  Repeat until a matching substring is found or text ends:

\[
\begin{align*}
&\text{Compare the corresponding characters right to left.} \\
&\text{If no characters match, retrieve entry } t_1(c) \text{ from the} \\
&\text{bad-symbol table for the text’s character } c \text{ causing the} \\
&\text{mismatch and shift the pattern to the right by } t_1(c). \\
&\text{If } 0 < k < m \text{ characters are matched, retrieve entry } t_1(c) \\
&\text{from the bad-symbol table for the text’s character } c \\
&\text{causing the mismatch and entry } d_2(k) \text{ from the good-} \\
&\text{suffix table and shift the pattern to the right by} \\
&d = \max \{d_1, d_2\} \\
&\text{where } d_1 = \max\{t_1(c) - k, 1\}.
\end{align*}
\]
Example of Boyer-Moore alg. application

B E S S _ K N E W _ A B O U T _ B A O B A B S
B A O B A B

d_1 = t_1(K) = 6  
B A O B A B

d_1 = t_1(\_)-2 = 4

d_2(2) = 5

<table>
<thead>
<tr>
<th>k</th>
<th>pattern</th>
<th>d_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BAOBAB</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>BAOBAB</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>BAOBAB</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>BAOBAB</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>BAOBAB</td>
<td>5</td>
</tr>
</tbody>
</table>

B A O B A B (success)
A very efficient method for implementing a *dictionary*, i.e., a set with the operations:
- find
- insert
- delete

Based on representation-change and space-for-time tradeoff ideas

Important applications:
- symbol tables
- databases (*extendible hashing*)
Hash tables and hash functions

The idea of hashing is to map keys of a given file of size \( n \) into a table of size \( m \), called the hash table, by using a predefined function, called the hash function,

\[ h: K \rightarrow \text{location (cell) in the hash table} \]

Example: student records, key = SSN. Hash function:

\[ h(K) = K \mod m \] where \( m \) is some integer (typically, prime)

If \( m = 1000 \), where is record with SSN = 314159265 stored?

Generally, a hash function should:

- be easy to compute
- distribute keys about evenly throughout the hash table
Collisions

If $h(K_1) = h(K_2)$, there is a collision

- Good hash functions result in fewer collisions but some collisions should be expected (birthday paradox)

- Two principal hashing schemes handle collisions differently:
  - Open hashing
    - each cell is a header of linked list of all keys hashed to it
  - Closed hashing
    - one key per cell
    - in case of collision, finds another cell by
      - linear probing: use next free bucket
      - double hashing: use second hash function to compute increment
Open hashing (Separate chaining)

Keys are stored in linked lists outside a hash table whose elements serve as the lists’ headers.

Example: A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

\[ h(K) = \text{sum of } K \text{ ‘s letters’ positions in the alphabet MOD 13} \]

<table>
<thead>
<tr>
<th>Key</th>
<th>A</th>
<th>FOOL</th>
<th>AND</th>
<th>HIS</th>
<th>MONEY</th>
<th>ARE</th>
<th>SOON</th>
<th>PARTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h(K) )</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Search for KID
Open hashing (cont.)

- If hash function distributes keys uniformly, average length of linked list will be $\alpha = n/m$. This ratio is called load factor.

- Average number of probes in successful, $S$, and unsuccessful searches, $U$:

  $$ S \approx 1 + \alpha/2, \quad U = \alpha $$

- Load $\alpha$ is typically kept small (ideally, about 1)

- Open hashing still works if $n > m$
Closed hashing (Open addressing)

Keys are stored **inside** a hash table.

<table>
<thead>
<tr>
<th>Key</th>
<th>A</th>
<th>FOOL</th>
<th>AND</th>
<th>HIS</th>
<th>MONEY</th>
<th>ARE</th>
<th>SOON</th>
<th>PARTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h(K)$</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></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>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARTED</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.
Closed hashing (cont.)

- Does not work if \( n > m \)
- Avoids pointers
- Deletions are not straightforward
- Number of probes to find/insert/delete a key depends on load factor \( \alpha = \frac{n}{m} \) (hash table density) and collision resolution strategy. For linear probing:

\[
S = \left(\frac{1}{2}\right) \left(1 + \frac{1}{1-\alpha}\right) \quad \text{and} \quad U = \left(\frac{1}{2}\right) \left(1 + \frac{1}{(1-\alpha)^2}\right)
\]

- As the table gets filled (\( \alpha \) approaches 1), number of probes in linear probing increases dramatically:

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>( \frac{1}{2} \left(1 + \frac{1}{1-\alpha}\right) )</th>
<th>( \frac{1}{2} \left(1 + \frac{1}{(1-\alpha)^2}\right) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>50%</td>
<td>1.5</td>
<td>2.5</td>
</tr>
<tr>
<td>75%</td>
<td>2.5</td>
<td>8.5</td>
</tr>
<tr>
<td>90%</td>
<td>5.5</td>
<td>50.5</td>
</tr>
</tbody>
</table>