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)
It is possible to sort data by counting for each data value the number of items that precede that data value.

Complete the following table:

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>62</td>
<td>31</td>
<td>84</td>
<td>96</td>
<td>19</td>
<td>47</td>
</tr>
<tr>
<td>count</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How can we claim this has sorted the data?
Comparison Counting Sort

The algorithm

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

Complexity Analysis

\[
C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n − 1) − (i + 1) + 1] = \sum_{i=0}^{n-2} (n − 1 − i) = \frac{n(n − 1)}{2}.
\]
Example of a Counting Sort

Here is a trace of the algorithm on the original set of data we looked at

<table>
<thead>
<tr>
<th>Array A[0..5]</th>
<th>Count []</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially</td>
<td>Count []</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>After pass i = 0</td>
<td>Count []</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>After pass i = 1</td>
<td>Count []</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>After pass i = 2</td>
<td>Count []</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>After pass i = 3</td>
<td>Count []</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>After pass i = 4</td>
<td>Count []</td>
<td>19</td>
<td>31</td>
<td>47</td>
<td>62</td>
<td>84</td>
<td>96</td>
</tr>
</tbody>
</table>

Array S[0..5]

Can you think of a situation that you would want to sort a large data set without actually moving the data itself?
A Sort Based on Distribution Counting

- The comparison counting sort was not very efficient, it is $O(n^2)$; its main advantage is you don’t have to move the data.
- If the data set only has a small set of data values, it is possible to develop a sort based on counting that does not use comparisons and can run in linear time.
- Consider the following small set of data:

  ![Data Set]

  We now count the frequencies and use it to find the distribution values:

<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>

Why can we claim this sorts the data?
ALGORITHM  DistributionCounting\((A[0..n - 1], l, u)\)

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

for \(j \leftarrow 0\) to \(u - l\) do \(D[j] \leftarrow 0\)  //initialize frequencies

for \(i \leftarrow 0\) to \(n - 1\) do \(D[A[i] - l] \leftarrow D[A[i] - l] + 1\)  //compute frequencies

for \(j \leftarrow 1\) to \(u - l\) do \(D[j] \leftarrow D[j - 1] + D[j]\)  //reuse for distribution

for \(i \leftarrow n - 1\) downto 0 do

\(j \leftarrow A[i] - l\)

\(S[D[j] - 1] \leftarrow A[i]\)

\(D[j] \leftarrow D[j] - 1\)

return \(S\)

What is the complexity of this algorithm?
**Example of Distribution Counting**

Here is a trace of the algorithm on the original set of data we looked at

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>A[1] = 11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A[0] = 13</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

Can you think of data sets that may be appropriate for distribution counting?
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
  
  \[ \ldots c \ldots \]  
  \( (c \text{ not in pattern}) \)  
  \( \text{BAOBAB} \)

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

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

- The rightmost characters do match
  
  \[ \ldots B \ldots \]  
  \( \text{BAOBAB} \)
Shift table

- Shift sizes can be precomputed by the formula:
  
  \[ t(c) = \begin{cases} 
  \text{distance from } c's \text{ rightmost occurrence in pattern among its first } m-1 \text{ characters to its right end} & \text{if } \text{pattern's length } m, \text{ otherwise} \\
  \text{by scanning pattern before search begins and stored in a table called shift table} & 
  \end{cases} \]

- Shift table is indexed by text and pattern alphabet.
  
  Eg, 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 | 3 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
### Example of Horspool’s alg. application

| 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 | 6 | 6 | 6 | 6 | 6 | 3 | 6 | 6 | 6 | 6 | 6 | 6 |

**BARD LOVED BANANAS**

**BAOBAB**

**BAOBAB**

**BAOBAB**

**BAOBAB** (unsuccessful search)
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.

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

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

- $d_2(k)$ is 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
Boyer-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:

- Compare the corresponding characters right to left.
- If no characters match, retrieve entry $t_1(c)$ from the bad-symbol table for the text’s character $c$ causing the mismatch and shift the pattern to the right by $t_1(c)$.
- If $0 < k < m$ characters are matched, retrieve entry $t_1(c)$ from the bad-symbol table for the text’s character $c$ causing the mismatch and entry $d_2(k)$ from the good-suffix table and shift the pattern to the right by $d = \max \{d_1, d_2\}$

where $d_1 = \max \{t_1(c) - k, 1\}$. 
Example of Boyer-Moore alg. application

**BE S S _ K E W _ A B O U T _ B A O B A B S**

**BA O B A B**

\[ d_1 = t_1(K) = 6 \]

**BA O B A B**

\[ d_1 = t_1(\_)-2 = 4 \]

\[ d_2(2) = 5 \]

**BA O B A B**

\[ d_1 = t_1(\_)-1 = 5 \]

\[ d_2(1) = 2 \]

**BA O B A B (success)**
Hashing

- 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)
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:

$$h(K) = K \mod m \text{ where } m \text{ is some integer (typically, prime)}$$

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 (\textit{birthday paradox})
- Two principal hashing schemes handle collisions differently:
  - \textit{Open hashing}
    - each cell is a header of linked list of all keys hashed to it
  - \textit{Closed hashing}
    - one key per cell
    - in case of collision, finds another cell by
      - \textit{linear probing}: use next free bucket
      - \textit{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

A  AND  MONEY  FOOL  HIS  ARE  PARTED  SOON
Open hashing (cont.)

- If hash function distributes keys uniformly, average length of linked list will be \( \alpha = \frac{n}{m} \). This ratio is called load factor.

- Average number of probes in successful, \( S \), and unsuccessful searches, \( U \):

  \[
  S \approx 1 + \frac{\alpha}{2}, \quad U = \alpha
  \]

- 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>

```

0 1 2 3 4 5 6 7 8 9 10 11 12

A   
A   
A   FOOL
A   AND FOOL
A   AND FOOL HIS
A   AND MONEY FOOL HIS
A   AND MONEY FOOL HIS ARE
A   AND MONEY FOOL HIS ARE SOON
PARTED A   AND MONEY FOOL HIS ARE SOON
```
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:
  \[ S = \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right) \quad \text{and} \quad U = \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right) \]
- 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>
B Trees

- B Trees are multiway, balanced sort trees
- Each key in a node has a pointer on the left and a pointer on the right; there is one more pointer than key value
- For example, given $K_i$ below, all items in $T_{i-1}$ through $T_0$ are less than $K_i$ and all items in $T_i$ through $T_{n-1}$ are greater
- All nodes, except possibly the root, are at least half full
- All leaf nodes are at the same level (so the tree is balanced)
Any node contains $n[x]$ keys and $n[x]+1$ children
Branching factors are typically between 50 - 2000
Height is very shallow minimizing disk accesses
The node size matches the sector size on the disk
Delays in Disk Access

There are three delays associated with reading or writing data to a disk. What are these delay and what are typical values for drives you would purchase for a PC?
Assume a branching factor of 1000
- Only the root node is kept in memory
- With one disk access over 1,000,000 keys are accessed; with two disk accesses over one billion keys can be accessed
An Example B Tree

- Here is an example B tree; notice that all data is stored at the leaves; the keys in the upper levels are used to steer the search (Note: many alternative algorithms only store the data once in the tree, including the middle levels).
- The maximum number of keys to a node is 3.
Searching for a Key in B-Trees

Searches are based on $h$ branching decisions if the tree is $h$ high

B-Tree-Search(key)
while not at last key and key $\geq k_i$
    increment i // search left to right
if not at leaf then
    descend to next level using pointer
else // at a leaf
    if (key == $k_i$) then
        return information associated with $k_i$
    else return nil // not found

- Each link to a child node requires a disk read
- Nil is returned if the key is not found

- The CPU time is $O(t \log_t n)$ where $t$ is half the maximum capacity per node
Inserting in B-Trees

- We first search the tree down to the leaf level to see if the key is already present, if we find the key we change the associated data as appropriate.
- If the key is not found, we insert the key and associated data in the leaf node last visited.
  - If there is room for the new key, we are done.
  - If there is no room, this causes an overflow; we split the values in the leaf node and pass the middle key back up to the parent node.
  - The pointer to the left of this new key value points to the first of the split nodes; the pointer to the right of the new key points to the second of the split nodes.
  - If the parent overflows, its middle value is passed up.
  - This process can continue until the root is reached.
  - If the root is split a new root node is created with a single key value and the tree has grown in height.
Given the tree

Assume the value 65 is inserted