Theory of Algorithms: 
Space and Time Tradeoffs
Objectives

- To introduce the mind set of trading space for time
- To show a variety of tradeoff solutions:
  - Sorting by Counting
  - Input Enhancement for String Matching
- To discuss the strengths and weaknesses of space and time tradeoffs
Space-Time Tradeoffs

- For many problems some extra space really pays off
- Originally human ‘computers’ would precompute a function’s values and record the results in maths tables
- Similarly, electronic computers can preprocess inputs and store additional info to accelerate the problem
- Input Enhancement: store additional input information
  - Sorting by Counting
  - Horspool’s and Boyer-Moore’s Algorithms for String Matching
- Prestructuring: use extra space for faster data access
  - Hashing
  - B-trees
- Dynamic Programming: record solutions to overlapping subproblems
Sorting by Counting

- Assume elements to be sorted belong to a known set of small values between \( l \) and \( u \), with potential duplication.
- Constraint: we cannot overwrite the original list.
- Distribution Counting: compute the frequency of each element and later accumulate sum of frequencies (distribution).
- Algorithm:

```plaintext
for j ← 0 to u-l do D[j] ← 0 // init frequencies
for i ← n-1 downto 0 do
    j ← A[i] - l
    D[j] ← D[j] - 1
return S
```
Notes on Sorting by Counting

Example: \( A = \)

<table>
<thead>
<tr>
<th></th>
<th>13</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>12</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Array Values</strong></td>
<td>11</td>
<td>12</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Frequencies</strong></td>
<td>1</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Distribution</strong></td>
<td>1</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( D[0] )</th>
<th>( D[1] )</th>
<th>( D[2] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Efficiency: \( \Theta(n) \)

- Best so far but only for specific types of input
Reminder: String Matching

- Pattern: a string of \( m \) characters to search for
- Text: a (longer) string of \( n \) characters to search in

Brute force algorithm:
1. Align pattern at beginning of text
2. Moving from left to right, compare each character of pattern to the corresponding character in text until
   - All characters are found to match (successful search); or
   - A mismatch is detected
3. While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat step 2.
Horspool’s Algorithm

- According to Cook [1970] problem can be solved in time proportional to $n+m$
- Horspool’s Algorithm: a simplified version of Boyer-Moore algorithm that retains key insights
- Compare pattern characters to text from right to left
- Given a pattern, create a shift table:
  - Has a shift value for each possible character
  - Determines how much to shift the pattern when a mismatch occurs (*input enhancement*)
- Note: the pattern is still shifted left to right. Only comparison is right to left
How Far to Shift?

Three cases (look to last character C in mismatched text segment):

- C is not in the pattern ⇒ shift by m
  
  .......D......................... (D not in pattern)
  
  BAOBAB
  
  →   BAOBAB

- C is in the pattern (but not at last position) ⇒ align rightmost occurrence in pattern with text
  
  .......O......................... (O occurs once in pattern)
  
  BAOBAB
  
  →   BAOBAB

- C produced a match ⇒ shift by m OR align to rightmost occurrence before last
  
  .......B.........................
Shift Table

- Stores number of characters to shift by depending on first character compared
- Construct by scanning pattern before searching starts
- Indexed by the alphabet of text and pattern
- All entries are initialized to pattern length. Eg, 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 |
| 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |

- For c occurring in pattern, update table entry to distance of rightmost occurrence of c from end of pattern
- We can do this by processing pattern from L→R
Horspool’s Algorithm

1. Construct the Shift Table for a given pattern and alphabet
2. Align the pattern against the beginning of the text
3. Repeat until a match is found or the pattern reaches beyond the text:
   - Starting from the pattern end, compare corresponding characters until all $m$ are matched (success!) or a mismatch is found
   - On a mismatch, retrieve the shift table entry $t(c)$, where $c$ is the text character aligned with the end of the pattern. Shift the pattern right by $t(c)$
Example: Horspool’s Algorithm

- **Pattern**: ZIGZAG
- **Text**: A ZIG, A ZAG, AGAIN A ZIGZAG
- **Shift Table**:

|   | 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 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 3 | 6 | 4 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 2 |

- **Exercise**: Simulate the execution of Horspool’s Algorithm
Boyer-Moore Algorithm

Based on same two ideas:
- Compare pattern characters to text from right to left
- Given a pattern, create a shift table that determines how much to shift the pattern

Except:
- Uses an additional good-suffix shift table with same idea applied to the number of matched characters

Efficiency:
- Horspool’s approach is simpler and at least as efficient in the average case
Strengths and Weaknesses of Space-Time Tradeoffs

✓ Strengths:
  - Suited to amortised situations
  - Particularly effective in accelerating access to data

✘ Weaknesses:
  - Can be space expensive and this might have empirical effects on speed