Space-time tradeoffs, Dynamic programming

Reading: Chapter 7, Chapter 8 (skim over sections not covered)

Exercises:
7.2: 1-5
7.3: 1, 2, 3, 8
8.2: 1, 2, 3, 7, 9

Next time: Greedy algorithms (Chapter 9)

**Space-time tradeoffs**

For many problems some extra space really pays off:
- extra space in tables (breathing room?)
  - hashing
  - non comparison-based sorting
- input enhancement
  - indexing schemes (e.g., B-trees)
  - auxiliary tables (shift tables for pattern matching)
- tables of information that do all the work
  - dynamic programming

**Hashing**

A very efficient method for implementing a dictionary, i.e., a set with the operations:
- insert
- find
- delete

Applications:
- databases
- symbol tables
Hash tables and hash functions

- **Hash table**: an array with indices that correspond to buckets.
- **Hash function**: determines the bucket for each record.

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

\[ h(k) = k \mod m \]

- if \( m = 1000 \), where is record with SSN=315-37-4251 stored?

**Hash function must**:
- be easy to compute
- distribute keys evenly throughout the table

Collisions

- If \( h(k_1) = h(k_2) \) then there is a collision.
- Good hash functions result in fewer collisions.
- Collisions can never be completely eliminated.

Two types handle collisions differently:
- **Open hashing** – bucket points to linked list of all keys hashing to it.
- **Closed hashing** – one key per bucket
  - in case of collision, find another bucket for one of the keys (need collision resolution strategy)
    - linear probing: use next bucket
    - double hashing: use second hash function to compute increment

Open hashing

- If hash function distributes keys uniformly, average length of linked list will be \( n/m \)
- Average number of probes = \( 1+\alpha/2 \)
- Worst-case is still linear!
- Open hashing still works if \( n > m \)
Closed hashing

- Does not work if \( n > m \).
- Avoids pointers.
- Deletions are not straightforward.
- Number of probes to insert/find/delete a key depends on load factor \( \alpha = n/m \) (hash table density).
- Successful search: \( \left( 1 - \frac{1}{m} \right) \left( 1 + \frac{1}{1 - \alpha} \right) \)
- Unsuccessful search: \( \left( 1 + \frac{1}{1 - \alpha^2} \right) \)
- As the table gets filled (\( \alpha \) approaches 1), number of probes increases dramatically:

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>( \left(1 + \frac{1}{1 - \alpha} \right) )</th>
<th>( \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>

String matching

- **pattern**: a string of \( m \) characters to search for
- **text**: a (long) 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.

String searching - History

- 1970: Cook shows (using finite-state machines) that problem can be solved in time proportional to \( m \) \( n \)
- 1976: Knuth and Pratt find algorithm based on Cook's idea; Morris independently discovers same algorithm in attempt to avoid "backing up" over text
- At about the same time Boyer and Moore find an algorithm that examines only a fraction of the text in most cases (by comparing characters in pattern and text from right to left, instead of left to right)
- 1980: Another algorithm proposed by Rabin and Karp virtually always runs in time proportional to \( \alpha \) \( n \) and has the advantage of extending easily to two-dimensional pattern matching and being almost as simple as the brute-force method.
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 that determines how much to shift the pattern when a mismatch occurs (input enhancement)

How far to shift?

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

- The character is not in the pattern
  - not in pattern)

- The character is in the pattern (but not at rightmost position)
  - (occurs once in pattern)

- The rightmost characters produced a match
  - (occurs twice in pattern)

Shift Table: Stores number of characters to shift by depending on first character compared.

Shift table

Constructed by scanning pattern before search begins
Indexed by text and pattern alphabet
All entries are initialized to length of pattern. E.g., BACRAB:

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:
Example

| 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 |
| B A R D | L O V E D | B A N A N A S |
| B A O B A B |

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 when a mismatch occurs (input enhancement)
- Uses additional shift table with same idea applied to the number of matched characters

Dynamic Programming

- **Dynamic Programming** is a general algorithm design technique
- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems
- “Programming” here means “planning”
- **Main idea**:
  - solve several smaller (overlapping) subproblems
  - record solutions in a table so that each subproblem is only solved once
  - final state of the table will be (or contain) solution
Example: Fibonacci numbers

• Recall definition of Fibonacci numbers:

\[
\begin{align*}
 f(0) &= 0 \\
 f(1) &= 1 \\
 f(n) &= f(n-1) + f(n-2)
\end{align*}
\]

• Computing the \( n \)th Fibonacci number recursively (top-down):

\[
\begin{align*}
 f(n) &= f(n-1) + f(n-2) \\
 f(n-2) &= f(n-3) + f(n-4)
\end{align*}
\]

Computing the \( n \)th fibonacci number using bottom-up iteration:

• \( f(0) = 0 \)
• \( f(1) = 1 \)
• \( f(2) = f(1) + f(0) = 1 + 0 = 1 \)
• \( f(3) = f(2) + f(1) = 1 + 1 = 2 \)
• \( f(4) = f(3) + f(2) = 2 + 1 = 3 \)
• \( f(5) = f(4) + f(3) = 3 + 2 = 5 \)
• \( f(n) = f(n-1) + f(n-2) \)

Examples of Dynamic Programming Algorithms

• Computing binomial coefficients
• Optimal chain matrix multiplication
• Constructing an optimal binary search tree
• Warshall's algorithm for transitive closure
• Floyd's algorithms for all-pairs shortest paths
• Some instances of difficult discrete optimization problems:
  - travelling salesman
  - knapsack
Warshall's Algorithm: Transitive Closure

- Computes the transitive closure of a relation
- (Alternatively: all paths in a directed graph)
- Example of transitive closure:

\[
\begin{array}{cccc}
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 \\
\end{array}
\]

Warshall's Algorithm

- Main idea: a path exists between two vertices \( i, j \) iff
  - there is an edge from \( i \) to \( j \); or
  - there is a path from \( i \) to \( j \) going through vertex \( 1 \); or
  - there is a path from \( i \) to \( j \) going through vertices \( 1 \) and \( 2 \); or
  - there is a path from \( i \) to \( j \) going through any of the other vertices

\[
R_0 = \begin{array}{cccc}
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]

\[
R_1 = \begin{array}{cccc}
0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]

\[
R_2 = \begin{array}{cccc}
0 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]

\[
R_3 = \begin{array}{cccc}
0 & 1 & 1 & 0 \\
0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]

\[
R_4 = \begin{array}{cccc}
0 & 1 & 1 & 0 \\
0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]

- In the \( k^{th} \) stage determine if a path exists between two vertices \( i, j \) using just vertices among \( 1, \ldots, k \)

\[
R_{k-1}[i,j] = \begin{cases} 
R_{k-1}[i,j] & \text{(path using just } 1, \ldots, k-1) \\
R_{k-1}[i,k] \text{ and } R_{k-1}[k,j] & \text{(path from } i \text{ to } k \text{ and from } k \text{ to } j \text{ using just } 1, \ldots, k-1) 
\end{cases}
\]
Floyd's Algorithm: All pairs shortest paths

- In a weighted graph, find shortest paths between every pair of vertices.
- Same idea: construct solution through series of matrices $D(0)$, $D(1)$, ... using an initial subset of the vertices as intermediaries.
- Example: