Lecture 3, Review of Algorithms

Young-Rae Cho
Associate Professor
Department of Computer Science
Baylor University

Overview

1. Exhaustive Search
2. Divide-and-Conquer Algorithm
3. Dynamic Programming
4. Greedy Algorithm
5. Randomized Algorithm
Exhaustive Search

- **Process**
  - Examine all possible cases to find a solution
  - Also, called brute force search

- **Features**
  - Simple
  - Sometimes, very inefficient because of combinatorial explosion

- **Example**
  - Selection sort

- **Alternatives**
  - Random Sampling
  - Branch and bound algorithm
  - Anti-monotonic property

---

Selection Sort

- **Algorithm**
  - Iteratively search the smallest one

  
  ```
  SelectionSort(a, n)
  for i ← 1 to n - 1
  a_j ← smallest one between a_i and a_n
  swap a_i and a_j
  return a
  ```

- **Runtime ?**
Anti-monotonic Property

**Definition**
- If a case satisfies a condition, then more general cases always satisfy it
- If a case violates a condition, then more specific cases always violate it

**Example**
- Find maximal sized sets of genes that occur together at least twice

<table>
<thead>
<tr>
<th>Function ID</th>
<th>Genes</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>A, B, C</td>
</tr>
<tr>
<td>20</td>
<td>C, D, F</td>
</tr>
<tr>
<td>30</td>
<td>A, C, E</td>
</tr>
<tr>
<td>40</td>
<td>A, B, C, E</td>
</tr>
</tbody>
</table>

Divide-and-Conquer Algorithm

**Process**
1. Recursively splitting the problem into smaller sub-problems
2. Solve the smallest sub-problem independently
3. Recursively merging the solutions of sub-problems until having a solution of the original problem

**Purpose**
- Improve efficiency

**Examples**
- Merge sort
- Quick sort
Merge Sort

- **Algorithm**
  1. Recursively divide the array
  2. Recursively combine two arrays in a sorted order

  ```
  MergeSort(A[1..n])
  if (n > 1)
    m ← ⌊n/2⌋
    MergeSort(A[1..m])
    MergeSort(A[m+1..n])
    Merge(A[1..n], m)
  ```

- **Runtime ?**

Quick Sort

- **Algorithm**
  1. Recursively divide the array based on the pivot
  2. Recursively combine two arrays

  ```
  Partition(A[1..n], p)
  i ← 0; j ← n
  while(i < j)
    repeat i ← i + 1
      until i = j or A[i] ≥ A[n]
    repeat j ← j - 1
      until i = j or A[j] ≤ A[n]
    if(i < j)
      swap A[i] and A[j]
    if (i ≠ n) swap A[i] and A[n]
  return i
  ```

  ```
  QuickSort(A[1..n])
  if (n > 1)
    k ← Partition(A, p)
    QuickSort(A[1..k - 1])
    QuickSort(A[k + 1..n])
  ```

- **Runtime ?**
Dynamic Programming

➢ Process
  (1) Formulate the problem recursively by breaking it down into sub-problems
  (2) Build solutions in a linear fashion
     (Repeatedly use the result of a sub-problem to solve the next sub-problem)

➢ Features
  ▪ Optimization (finding an optimal solution)
  ▪ Memoization (storing results of intermediate sub-problems)

➢ Examples
  ▪ Sequence alignment
  ▪ Binary search tree

'Rocks' Game (1)

➢ Rule
  ▪ 2 piles of rocks
  ▪ A player may take either
     1 rock (from either pile)
     or 2 rocks (1 from each)
  ▪ The player who takes
     the last rock wins

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>2</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
</tr>
<tr>
<td>3</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>4</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
</tr>
<tr>
<td>5</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>6</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
</tr>
<tr>
<td>7</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>8</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
</tr>
<tr>
<td>9</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>10</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
</tr>
</tbody>
</table>
Recursive Formula

To solve a problem by dynamic programming, find the recursive formula at first!

\[
\begin{cases}
R_{1,0} = W \\
R_{0,1} = W \\
R_{1,1} = W \\
R_{i,j} = L & \text{if } R_{i-1,j} = W \ (\text{where } i \geq 1) \ \text{and} \\
& R_{i,j-1} = W \ (\text{where } j \geq 1) \ \text{and} \\
& R_{i-1,j-1} = W \ (\text{where } i \geq 1 \ \text{and } j \geq 1) \\
R_{i,j} = W & \text{Otherwise}
\end{cases}
\]

Search Time

- Data search time in array?
- Data search time in binary search tree?

Formula

\[
S(T) = \sum_{i=1}^{n} (\text{depth}(T, i) + 1) \cdot f[i]
\]

\[
\text{depth}(T, i) = \begin{cases}
\text{depth(left}(T), i) + 1 & \text{if } i < r \\
0 & \text{if } i = r \\
\text{depth(right}(T), i) + 1 & \text{if } i > r
\end{cases}
\]

\[
S(T) = \begin{cases}
& \text{...}
\end{cases}
\]
Greedy Algorithm

- **Process**
  1. Determine the optimal structure of a problem
  2. Find the local optimal solution at each step

- **Features**
  - Local optimization

- **Examples**
  - Huffman codes
  - Minimum spanning tree

Money Counting

- **Problem**
  - Count a certain amount of money using the fewest bills and coins

- **Local Optimum Solution**
  - Take the largest bill or coin at each step

- **Examples**
  - $14.27

To solve a problem by a greedy algorithm, find the *local optimum solution* at first!
Scheduling

- **Problem**
  - Assign $m$ jobs into $n$ processors to finish all the jobs as early as possible
  - Suppose $m > n$

- **Local Optimum Solution?**

- **Examples**
  - 9 jobs on 3 processors
    - (runtimes of 9 jobs are 3, 5, 6, 10, 11, 14, 15, 18, and 20 min.)

---

**Huffman Coding (1)**

- **Binary Codes**
  - Encoding method
  - Use binary representation (0 and 1)

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixed-length binary codes</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>variable-length binary codes</td>
<td>0</td>
<td>10</td>
<td>11</td>
<td>100</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>

- **Advantages**
  - Fixed-length codes: straightforward
  - Variable-length codes: efficient (encode with lower bits)
  - e.g., ace?
- How to make correct variable-length codes? **Make prefix codes!**
Prefix Codes
- No codeword is a prefix of any other codeword
- How to make? **Build a binary tree!**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixed-length</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>binary codes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>variable-length</td>
<td>0</td>
<td>100</td>
<td>101</td>
<td>111</td>
<td>1100</td>
<td>1101</td>
</tr>
</tbody>
</table>

Huffman Coding (3)

**Problem**
- Find optimal prefix codes for the most efficient compression
- Suppose each letter has different frequency

**Local Optimum Solution?**

**Examples**
- a: 45
- b: 13
- c: 12
- d: 16
- e: 9
- f: 5
Randomized Algorithm

- **Process**
  - Examine random samples to find a solution

- **Features**
  - Simple
  - Probabilistic
  - Sometimes, non-deterministic

- **Deterministic algorithm:**
  always produce the same solution given a particular input

- **Non-deterministic algorithm:**
  allows multiple solutions based on an input or random choices

Bolts and Nuts

- **Problem**
  - Among \( n \) nuts, find the nut that matches a given bolt

- **Expected Number of Comparison ?**
  - \( T(n) \): number of comparison to find a match for a single bolt out of \( n \) nuts

  \[
  E[T(n)] = \sum_{k=1}^{n-1} k \cdot Pr[T(n) = k]
  \]

  \[
  Pr[T(n) = k] = \begin{cases} 
  1/n & \text{if } k < n - 1 \\
  2/n & \text{if } k = n - 1
  \end{cases}
  \]

  \[
  E[T(n)] = \frac{n + 1}{2} - \frac{1}{n}
  \]
Questions?

- Lecture Slides are found on the Course Website,
  web.ecs.baylor.edu/faculty/cho/3360