Lecture 20
Review

Topis Coverage
- Algorithm design paradigm: Dynamic Programming, and parallel computing
- Algorithms covered:
  - String Matching: Knuth-Morris-Pratt algorithm, Boyer-Moore algorithm
  - Approximate String Matching
  - Polynomial and Matrices: Horner’s algorithm and Strassen’s algorithm
  - Parallel algorithms: CREW PRAM, CRCW (Common-Write) PRAM
- NP Completeness
  - P can be solved in polynomial time.
  - NP: solution can be verified in polynomial time.
  - NP-complete: as hard as any problem in NP. Known NP-complete problem can be polynomially reduced to this problem.
  - NP-completeness: problem A is problem B: Show A is hard.
  - If A is NP-complete and B is in NP, a polynomial reduction of A to B proves B is NP-complete too.
  - Method: Take any input x for A. Use it to construct an input y for B, such that
    1. the size of y is a polynomial in the size of x,
    2. the correct A answer for x is true if the correct B answer for y is true
    3. the correct B answer for y is true if the correct A answer for x is true.

How to prove a problem is in NP.
How to prove a problem is NP-complete
Familiarity with some well-known NP-complete problems:
- CNF SAT, Dominant Set, Graph Coloring, Hamiltonian Cycle, TSP (Travelling Salesman Problem)
- Approximate solutions to bin pack and TSP.

Format of Exam
- The exam is closed book and notes. The exam will have a heavy emphasis on
  understanding and applying the concepts.
- There will be four problems.
Example questions:
- Given a pattern, calculate its fail indices for KMP algorithm.
- Given a pattern and a text string, demo how KMP algorithm (or Boyer-Moore algorithm) works.
- Dynamic programming
  - Concept,
  - Procedure, and
  - Applications (string alignment, e.g., hw5 q2)
- Polynomials and Matrices
  - Understand Horner’s rule and Strassen’s algorithm
- NP-complete
  - Concepts: P, NP, NP-Complete, NP-hard, polynomial reduction
  - Given a problem, prove it is NP complete
  - Given problems A and B, prove A is no harder than B.
- Parallel algorithms
  - PRAM
  - NC class
  - Design parallel algorithms that solve a given problem using specified model (or variations).

The KMP algorithm

| i=0: X           | i=1: X           |
| i=2: nanX        | i=3: X           |
| i=4: nano         | i=5: X           |
| i=6: nX           | i=7: X           |
| i=8: X           | i=9: nX          |
| i=10: X          |

1. Skip outer iteration i = 3
2. Skip first inner iteration testing "n" vs "n" at outer iteration i = 4
3. 9 comparisons are made for find the first occurrence.
The Boyer-Moore algorithm

The numbers of positions we can "jump" forward when there is a mismatch depends on the text character being read, say $T[j]$, more precisely, depends on $T[j]$'s occurrence in pattern $P$.

Only 18 comparisons are needed to find the first occurrence.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>T</th>
<th>G</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>+4</td>
<td>-4</td>
<td>-4</td>
<td>-2</td>
<td>-8</td>
</tr>
<tr>
<td>C</td>
<td>-4</td>
<td>+4</td>
<td>-2</td>
<td>-4</td>
<td>-8</td>
</tr>
<tr>
<td>T</td>
<td>-4</td>
<td>-2</td>
<td>+4</td>
<td>-4</td>
<td>-8</td>
</tr>
<tr>
<td>G</td>
<td>-2</td>
<td>-4</td>
<td>-4</td>
<td>+4</td>
<td>-8</td>
</tr>
<tr>
<td>-</td>
<td>-8</td>
<td>-8</td>
<td>-8</td>
<td>-8</td>
<td>-8</td>
</tr>
</tbody>
</table>

where "-" stands for an blank space (i.e., a deletion) in a sequence.
Horner’s algorithm

To observe that \( p(x) \) can be rewritten as follows.

\[
p(x) = (\ldots(a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0.
\]

Horner-poly(a, n, x)

1. \( p = a[n] \)
2. for \( i \leftarrow n-1 \) to 0
3. \( p \leftarrow p \* x + a[i] \)
4. return \( p \)

Running time: \( n \) multiplications and \( n \) additions.
In practice, a polynomial reduction from the optimization problem \( U \) to the decision problem \( V \) can be constructed as follows: Suppose there is an algorithm \( S \) for the decision problem, then use \( S \) as a subroutine to construct an algorithm \( W \) that solves the optimization problem. Assume that \( S \) takes constant time. If \( W \) runs in polynomial time, then the reduction is in polynomial time. Therefore, we found algorithm \( W \) that solves \( U \) in as much time as needed (module a polynomial) for solving \( V \).

Example: A dominant set of a graph \( G \) is a subset of vertices \( W \) such that every vertex of \( G \) is either in \( W \) or is adjacent to a vertex in \( W \).

**Decision Problem**
- Input: A graph \( G \) and integer \( k \)
- Question: Does \( G \) have a dominant set of size at most \( k \)?

**Optimization Problem:**
- Input: A graph \( G \)
- Output: A smallest dominant set of \( G \)

The Claim: the optimization problem for dominant set is no harder than* the decision problem.

Proof: Use Polynomial reduction.

1. **Step 1:** For \( i \) from 1 to \(|G|\), call the algorithm for the decision problem as a subroutine on input \((G_i)\). This shall give us \( M \), the size of the smallest dominant set of \( G \).

2. **Step 2:** Initialize all vertices in \( G \) as unmarked. Now choose an unmarked vertex \( v \) in \( G \), add a new vertex \( X \) attached by an edge to \( v \). Ask the decision algorithm if the modified graph has a dominant set of size \( M \). If yes, then mark \( v \) as a member of the dominant set and leave \( X \) in the graph. If no, mark \( v \) as non-member and remove the vertex \( X \). Repeat until \( M \) vertices are marked as members of the dominant set.

* Module a polynomial reduction.
Input: Keys $x[0], \ldots, x[n-1]$, initially in memory cells $M[0], \ldots, M[n-1]$, and integer $n$.
Output: The largest key will be left in $M[0]$.

Remarks: cells $M[n]$ to $M[2n-1]$ are initialized to $-\infty$.

```c
parTournamentMax(M, n)
    int incr;
    Write (some very small value) into $M[n+pid]$
    incr = 1;
    While (incr < n) {
        Key big, tmp0, tmp1;
        Read $M[pid]$ into tmp0.
        Read $M[pid + incr]$ into tmp1.
        big = max(tmp0, tmp1);
        write big into $M[pid]$.
        incr = 2 * incr;
    }
```

Performance: $O(\lg n)$ time, $n$ processors.

---

**Handling Write Conflicts**

- **CREW (Concurrent Read, Exclusive Write):**  
  only one processor write in a particular cell at any one step; It is illegal to have more than one processor write to one cell at the same time.

- **CRCW (Concurrent Read, Concurrent Write):**
  - Common-Write model
  - Arbitrary-Write model
  - Priority-Write model
Problem  Boolean \textbf{AND} on \emph{n} Bits
\begin{itemize}
  \item Input: Bits $x_0, \ldots, x_{n-1}$, as 0's and 1's, in $M[0], \ldots, M[n-1]$
  \item Output: $x_0 \land \ldots \land x_{n-1}$ in $M[0]$.
\end{itemize}

Solution 2 (CRCW)
\begin{itemize}
  \item \texttt{commonWriteOr}(M, n)
  \item $P_i$ reads $x_i$ from $M[i]$;
  \item If $x_i$ is 0; then $P_i$ writes 0 in $M[0]$.
\end{itemize}

Correctness: Since all the processors that write into $M[0]$ write the same value, the program is a legal program under Common-Write model.

Running time: It takes just two steps, regardless the input size.