2) Write *any* algorithm that runs in time $\Theta(n^2 \log^2 n)$ in worse case. Explain why this is its running time. I don’t care what it does. I only care about its running time…

```plaintext
Whatever(int m)

FOR i=1 TO m
    FOR j=1 TO m
        x=m; WHILE x>1 DO {
            x=x/2;
            y=m;
            WHILE y>1 DO y=y/2
        }
```

$n = |m| \sim \log m$. Therefore running time is $\Theta(m^2 \log^2 m) = \Theta(2^{2n} n^2)$
2) Write any algorithm that runs in time $\Theta(n^2 \log^2 n)$ in worse case. Explain why this is its running time. I don’t care what it does. I only care about its running time...

```java
Whatever(int[] A)

n = A.length;
FOR i=1 TO n
    FOR j=1 TO n
        x=n; WHILE x>1 DO { x=x/2; y=n; WHILE y>1 DO y=y/2 }
```
Mercury Course Evaluations

Course evaluations matter. Evaluate your courses and instructors!

Default period:
March 21 - May 1

Condensed period:
March 21 - April 17

Click HERE to complete your course evaluations.
Strings and Pattern Matching

- Brute Force, Rabin-Karp, Knuth-Morris-Pratt
- Regular Expressions

What's up?
I'm looking for some string.
That's quite a trick considering that you have no eyes.
Oh yeah? Have you seen your writing? It looks like an EKG!
The Knuth-Morris-Pratt Algorithm

- The Knuth-Morris-Pratt (KMP) string searching algorithm differs from the brute-force algorithm by keeping track of information gained from previous comparisons.

- A failure function \( f \) is computed that indicates how much of the last comparison can be reused if it fails.

- Specifically, \( f \) is defined to be the longest prefix of the pattern \( P[0,..,j] \) that is also a suffix of \( P[1,..,j] \)
  - Note: **not** a suffix of \( P[0,..,j] \)

The Knuth-Morris-Pratt Algorithm

- Specifically, $f$ is defined to be the longest prefix of the pattern $P[0,..,j]$ that is also a suffix of $P[1,..,j]$
  - **Note:** not a suffix of $P[0,..,j]$

- Example:
  - value of the KMP failure function:

<table>
<thead>
<tr>
<th>$j$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[j]$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>$f(j)$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

- This shows how much of the beginning of the string matches up to the portion immediately preceding a failed comparison.
  - if the comparison fails at (4), we know the $a,b$ in positions 2,3 is identical to positions 0,1
The KMP Algorithm (contd.)

- the KMP string matching algorithm: Pseudo-Code

Algorithm KMPMatch(T,P)
Input: Strings T (text) with \( n \) characters and P (pattern) with \( m \) characters.
Output: Starting index of the first substring of T matching P, or an indication that P is not a substring of T.

\[ f \leftarrow \text{KMPFailureFunction}(P) \] \{build failure function\}
\[ i \leftarrow 0 \]
\[ j \leftarrow 0 \]
while \( i < n \) do
  if \( P[j] = T[i] \) then
    if \( j = m - 1 \) then
      return \( i - m - 1 \) \{a match\}
    \[ i \leftarrow i + 1 \]
    \[ j \leftarrow j + 1 \]
  else if \( j > 0 \) then \{no match, but we have advanced\}
    \[ j \leftarrow f(j-1) \] \{j indexes just after matching prefix in P\}
  else
    \[ i \leftarrow i + 1 \]
return “There is no substring of T matching P”
The KMP Algorithm (contd.)

• The KMP failure function: Pseudo-Code

Algorithm KMPFailureFunction(P);

Input: String P (pattern) with \( m \) characters

Output: The failure function \( f \) for \( P \), which maps \( j \) to the length of the longest prefix of \( P \) that is a suffix of \( P[1,..,j] \)

\[
i \leftarrow 1 \\
j \leftarrow 0 \\
\text{while } i \leq m-1 \text{ do} \\
\quad \text{if } P[j] = P[i] \text{ then} \\
\quad \quad \{ \text{we have matched } j + 1 \text{ characters} \} \\
\quad \quad f(i) \leftarrow j + 1 \\
\quad \quad i \leftarrow i + 1 \\
\quad \quad j \leftarrow j + 1 \\
\quad \text{else if } j > 0 \text{ then} \\
\quad \quad \{ \text{j indexes just after a prefix of } P \text{ that matches} \} \\
\quad \quad j \leftarrow f(j-1) \\
\text{else} \\
\quad \{ \text{there is no match} \} \\
\quad f(i) \leftarrow 0 \\
\quad i \leftarrow i + 1 \]
The KMP Algorithm (contd.)

- A graphical representation of the KMP string searching algorithm

```
  a b a c a a b a c c a b a c a b a a
  1 2 3 4 5 6
    a b a c a b
    7
      a b a c a b
      8 9 10 11 12
        a b a c a b
        13
          a b a c a b
          14 15 16 17 18 19
            a b a c a b
```
The KMP Algorithm (contd.)

- Time Complexity Analysis

- define \( k = i - j \)

- In every iteration through the while loop, one of three things happens.
  - 1) if \( T[i] = P[j] \), then \( i \) increases by 1, as does \( j \) and \( k \) remains the same.
  - 2) if \( T[i] \neq P[j] \) and \( j > 0 \), then \( i \) does not change and \( k \) increases by at least 1, since \( k \) changes from \( i - j \) to \( i - f(j-1) \)
  - 3) if \( T[i] \neq P[j] \) and \( j = 0 \), then \( i \) increases by 1 and \( k \) increases by 1 since \( j \) remains the same.

- Thus, each time through the loop, either \( i \) or \( k \) increases by at least 1, so the greatest possible number of loops is \( 2n \)

- This of course assumes that \( f \) has already been computed.

- However, \( f \) is computed in much the same manner as KMPMatch so the time complexity argument is analogous. KMPFailureFunction is \( O(m) \)

- Total Time Complexity: \( O(n + m) \)
The KMP Algorithm (contd.)

- Thus, each time through the loop, either $i$ or $k$ increases by at least 1, so the greatest possible number of loops is $2n$
- This of course assumes that $f$ has already been computed.
- However, $f$ is computed in much the same manner as KMPMatch so the time complexity argument is analogous. KMPFailureFunction is $O(m)$
- Total Time Complexity: $O(n + m)$
Regular Expressions

- notation for describing a set of strings, possibly of infinite size
- $\varepsilon$ denotes the empty string
- $ab + c$ denotes the set \{ab, c\}
- $a^*$ denotes the set \{\varepsilon, a, aa, aaa, {...}\}

Examples
- $(a+b)^*$ all the strings from the alphabet \{a,b\}
- $b^*(ab*a)^*b^*$ strings with an even number of a’s
- $(a+b)^*sun(a+b)^*$ strings containing the pattern “sun”
- $(a+b)(a+b)(a+b)a$ 4-letter strings ending in a
Finite State Automaton

- “machine” for processing strings

Composition of FSA’s