Algorithms: Solutions 10

```
                      X
                      X
                      X
                      X   X
                      X   X
                      X   X
                      X   X
                      X   X
number of             X   X
homeworks         X   X   X
                  X   X   X   X
                  X   X   X   X
                  X   X   X   X
                  X   X   X   X
                  X   X   X   X
          X       X   X   X   X           X
          X   X   X   X   X   X   X       X
          -----------------------------------
          2   3   4   5   6   7   8   9   10
                       grades
```

Problem 1
Assume that all characters in a pattern $P[1..m]$ are *distinct,* and you need to find all occurrences of P in a text $T[1..n]$. Write an "accelerated" version of NAIVE-STRING-MATCHER, which solves this problem in $O(n)$ time.

FAST-NAIVE-MATCHER(T, P, n, m)
initialize empty list *Shifts*
$i \leftarrow 1$
while $s \leq n - m$
    do if $i > m$
        then add s to *Shifts*
            $s \leftarrow s + m$
            $i \leftarrow 1$
        else if $P[i] = T[s + i]$
            then $i \leftarrow i + 1$
            else $s \leftarrow s + i$
                $i \leftarrow 1$
return *Shifts*

Problem 2
Write an algorithm that looks for a given $m \times m$ pattern in an $n \times n$ array of characters, based on the Rabin-Karp method.

We compute a separate Rabin-Karp numerical value for each row of the $m \times m$ pattern, and then compute numerical values for each row of the $n \times n$ array. For each possible position of the pattern in the array, we need to compare m numerical values, which correspond to the rows of the pattern. When all values match, we perform a character-by-character comparison of the pattern with the array.

Problem 3

Design an efficient algorithm for finding the longest common substring of two strings.

We denote the input strings by $X[1..m]$ and $Y[1..n]$, and define $l[i, j]$ as the length of the longest common suffix of $X[1..i]$ and $Y[1..j]$. We compute $l[i, j]$ for every $i \leq m$ and every $j \leq n$; the maximal $l[i, j]$ value is the length of the longest common substring.

The following algorithm finds the maximal $l[i, j]$ value, uses it to identify the longest common substring, and prints out this substring. The algorithm runs in $\Theta(m \cdot n)$ time and requires $\Theta(m \cdot n)$ memory.

```
COMMON-SUBSTRING(X, Y, m, n)
i_max ← 0
l_max ← 0
for i ← 0 to m
    do l[i, 0] ← 0
for j ← 1 to n
    do l[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
           do if X[i] ≠ Y[j]
                  then l[i, j] ← 0
                  else l[i, j] ← l[i − 1, j − 1] + 1
                       if l[i, j] > l_max
                          then i_max ← i
                               l_max ← l[i, j]
print X[(i_max − l_max + 1)..(i_max)]
```

We can modify this algorithm to reduce its memory usage, without affecting the running time; the modified version uses an auxiliary array $l[1..n]$, which takes only $\Theta(n)$ memory.

```
LOW-MEMORY-SUBSTRING(X, Y, m, n)
i_max ← 0
l_max ← 0
for j ← 1 to n
    do l[j] ← 0
for i ← 1 to m
    do old ← 0
       for j ← 1 to n
           do temp ← l[j]
              if X[i] ≠ Y[j]
                then l[j] ← 0
                else l[j] ← old + 1
                     if l[j] > l_max
                        then i_max ← i
                             l_max ← l[j]
              old ← temp
print X[(i_max − l_max + 1)..(i_max)]
```

Note that this algorithm for finding the longest common substring is *not* optimal. We can solve this problem in $O(m + n)$ time, using the concept of a *suffix tree.*