CS/ECE 374 Spring 2023 William Cheng (shihuac2@illinois.edu)

Homework 7 Problem 1

Solution: (a) Let *SPSLength*(i, j) be the length of the shortest palindromic supersequence of the substring $A[i...j]$ of string A. Consider the following two cases:

- $A[i] \neq A[j]$. Then the shortest palindromic supersequence (SPS) of $A[i...j]$ is either obtained by adding two $A[i]$'s to the start and end of the SPS of $A[i+1...j]$, or by adding two $A[j]$'s to the start and end of the SPS of $A[i...j-1]$.
- $A[i] = A[j]$. Then the SPS of $A[i...j]$ can be obtained by adding two $A[i]$'s to the start and end of the SPS of $A[i+1...j-1]$.

For the base case, when $i = j$, *SPSLength*$(i, j) = 1$ since the SPS of a single letter is just itself. If $i > j$, then the string is empty, so *SPSLength*$(i, j) = 0$. Following are the recurrences.

$$\mathit{SPSLength}(i,j) = \begin{cases} 0 & i > j \\ 1 & i = j \\ \mathit{SPSLength}(i+1, j-1) + 2 & i < j, A[i] = A[j] \\ \min\{\mathit{SPSLength}(i, j-1), \mathit{SPSLength}(i+1, j)\} + 2 & i < j, A[i] \neq A[j] \end{cases}$$

According to the recurrences, the exploration order should be decreasing i and increasing j. The final result should be *SPSLength*$(1, n)$. Since the i^{th} row can be calculated using data from the $(i+1)^{th}$ row and the i^{th} row itself, memoizing two rows is enough, thus $O(n)$ space is needed. Following is a space-efficient algorithm.

```
CalculateSPSLength($A[1...n]$):
  integer $SPSLength[2][n]$
  for $i \leftarrow n$ to $1$
      copy $SPSLength[2]$ to $SPSLength[1]$
      for $j \leftarrow i$ to $n$
          if $i = j$
              $SPSLength[2][j] \leftarrow 1$
          else if $i > j$
              $SPSLength[2][j] \leftarrow 0$
          else if $A[i] = A[j]$
              $SPSLength[2][j] \leftarrow SPSLength[1][j-1] + 2$
          else
              $SPSLength[2][j] \leftarrow \min\{SPSLength[2][j-1], SPSLength[1][j]\} + 2$
  return $SPSLength[2][n]$
```

The time complexity of this algorithm is $O(n^2)$.

(b) Let *SCPS*(i, j, k, l) be the length of the shortest common palindromic supersequence of $X[i...j]$ and $Y[k...l]$. Let *SPS_X*(i, j) and *SPS_Y*(k, l) be the length of the shortest palindromic supersequence of $X[i...j]$ and $Y[k...l]$ respectively. The calculation of *SPS* requires $O(m^2+n^2)$

time using the algorithm in part (a). We can get the following recurrences.

$$SCPS(i,j,k,l) = \begin{cases} 0 & k > l, i > j \\ SPS\_X(i,j) & k > l, i \le j \\ SPS\_Y(k,l) & i > j, k \le l \\ 2 + SCPS(i+1, j-1, k+1, l-1) & X[i] = X[j] = Y[k] = Y[l] \\ 2 + SCPS(i+1, j-1, k+1, l) & X[i] = X[j] = Y[k] \\ 2 + SCPS(i+1, j-1, k, l-1) & X[i] = X[j] = Y[l] \\ 2 + SCPS(i+1, j, k+1, l-1) & X[i] = Y[l] = Y[k] \\ 2 + SCPS(i, j-1, k+1, l-1) & X[j] = Y[l] = Y[k] \\ 2 + \min\{SCPS(i,j,k+1,l-1), SCPS(i+1,j-1,k,l)\} & X[i] = X[j], Y[k] = Y[l] \\ 2 + \min\{SCPS(i+1,j,k+1,l), SCPS(i,j-1,k,l-1)\} & X[i] = Y[k], X[j] = Y[l] \\ 2 + \min\{SCPS(i+1,j,k,l-1), SCPS(i,j-1,k+1,l)\} & X[i] = Y[l], X[j] = Y[k] \\ 2 + SCPS(i+1, j-1, k, l) & X[i] = X[j] \\ 2 + SCPS(i+1, j, k+1, l) & X[i] = X[k] \\ 2 + SCPS(i+1, j, k, l-1) & X[i] = X[l] \\ 2 + SCPS(i, j-1, k+1, l) & X[j] = Y[k] \\ 2 + SCPS(i, j-1, k, l-1) & X[j] = Y[l] \\ 2 + SCPS(i, j, k+1, l-1) & Y[k] = Y[l] \\ 2 + \min \begin{cases} SCPS(i+1,j,k,l) \\ SCPS(i,j-1,k,l) \\ SCPS(i,j,k+1,l) \\ SCPS(i,j,k,l-1) \end{cases} & \text{otherwise} \end{cases}$$

These recurrences are obtained from the following cases regarding $X[i], X[j], Y[k], Y[l]$:

- All four are equal. Then we just add two $X[i]$'s to the start and end of the *SCPS* of $X[i+1...j-1]$ and $Y[k+1...l-1]$.
- Exactly three are equal. This is similar to the first case, except we add the unequal letter to $X[i+1...j-1]$ or $Y[k+1...l-1]$.
- There are two equal pairs. Then we either add two letters from the first pair or add two letters from the second pair. We take the minimum of the two options.
- Exactly two are equal. Then we add two these letters to the *SCPS* of the strings without these two letters.
- All four are not equal to each other. Then we take the minimum of all four possible derivations of the current *SCPS*.

When $i > j$ and $k > l$, the two strings are empty, so the *SCPS* is the empty string, and its length is 0. When $i > j$ or $k > l$, then one of the strings is empty, and the length of the *SCPS* is just the length of the *SPS* of the other string. We should explore in the following order:

```
integer SCPS[$m$][$m$][$n$][$n$]
for $i \leftarrow m$ to $1$
  for $j \leftarrow i$ to $m$
    for $k \leftarrow n$ to $1$
      for $l \leftarrow k$ to $n$
        evaluate SCPS[$i$][$j$][$k$][$l$] using recurrences above
output SCPS[$1$][$m$][$1$][$n$]
```

The final answer is *SCPS*$(1, m, 1, n)$. The time complexity of this algorithm is $O(m^2n^2)$.

■

CS/ECE 374 Spring 2023 William Cheng (shihuac2@illinois.edu)
Homework 7 Problem 2

Solution: (a) Let *NumOfMatchings*(T) be the number of distinct matchings in T. Consider all subtrees with children of T as the root. Suppose we know *NumOfMatchings*(S) for each subtree S of T. Then there are two cases in forming a new matching:

- The new matching does not contain any of the edges from root of T to its children. There are $\prod_{S=\text{subtree of } T} \textit{NumOfMatchings}(S)$ such matchings.
- The new matching contains exactly one edge e from root of T to its children. Then for each child s of root of T, the number of matchings is $n(s) = \prod_{\text{subtree } S \neq T(s) \text{ of } T} \textit{NumOfMatchings}(S) \times \prod_{\text{subtree } P \text{ of } T(s)} \textit{NumOfMatchings}(P)$. There are $\sum_{\text{child } s \text{ of root of } T} \{n(s)\}$ such matchings.

The result should be the sum of the two. The number of matchings for a single node is just 1. We can use a dictionary to memoize the value for each subtree for an efficient algorithm. Following is the pseudocode for the algorithm.

```
dictionary $d$

NumOfMatchings($T$):
  if $T$ is a leaf node
      return 1
  if $T$ is a key in $d$
      return $d[T]$
  matchingsWithRoot $\leftarrow 0$
  matchingsNoRoot $\leftarrow 1$
  for each subtree $S$ of $T$
      matchingsNoRoot $\leftarrow$ matchingsNoRoot $\times$ NumOfMatchings($S$)
  for each subtree $S$ of $T$
      matchingsWithS $\leftarrow 1$
      for each subtree $P$ of $T$ where $P \neq S$
          matchingsWithS $\leftarrow$ matchingsWithS $\times$ NumOfMatchings($P$)
      for each subtree $Q$ of $S$
          matchingsWithS $\leftarrow$ matchingsWithS $\times$ NumOfMatchings($Q$)
      matchingsWithRoot $\leftarrow$ matchingsWithRoot + matchingsWithS
  $d[T] \leftarrow$ matchingsWithRoot + matchingsNoRoot
  return $d[T]$
```

The time complexity is $O(|V|^2)$.

(b) Let *NumOfMatchings*(n) be the number of matchings in a path on n nodes.

$$\textit{NumOfMatchings}(n) = \begin{cases} 1 & n \leq 1 \\ \textit{NumOfMatchings}(n-1) + \textit{NumOfMatchings}(n-2) & n > 1 \end{cases}$$

The answer for $n = 500$ will not fit in a 64-bit integer word because $2^{64} \approx 1.85 \times 10^{19}$, but the answer will be larger than 10^{100} according to the Fibonacci sequence formula, which will overflow.

(c) We can construct a data structure that is essentially a dynamic array of 64-bit unsigned integers, each integer in the array representing a 64-bit chunk of the data. For addition, we add the 64-bit chunks from right to left. Let b be the ceiling of the number of bits of the data divided by 64. Then addition can be completed in $O(b)$ time. We can use Karatsuba's algorithm for multiplication in $O(b \log b)$ time. If the leftmost chunk overflows, then a new chunk is dynamically allocated. We use this data structure to replace integer in the algorithm in (a). Then the running time of the algorithm becomes $O(b \log b |V|^2)$.

(d) Let *maxW*$(v, i, 1)$ be the weight of a maximum weight matching in $T(v)$ with at most i edges that does not contain v, and *maxW*$(v, i, 2)$ be the weight of a maximum weight matching in $T(v)$ with at most i edges that contains v. Suppose T is a binary tree and let l, r denote the left child and the right child of v. Then we can write the following recurrences:

$$\mathit{maxW}(v,i,1) = \begin{cases} 0 & i = 0 \text{ or } v \text{ is leaf node} \\ -\infty & v \text{ is null} \\ \max\limits_{j=0}^{i} \left\{ \begin{matrix} \max\{\mathit{maxW}(l,j,1), \mathit{maxW}(l,j,2)\} \; + \\ \max\{\mathit{maxW}(r,i-j,1), \mathit{maxW}(r,i-j,2)\} \end{matrix} \right\} & \text{otherwise} \end{cases}$$

$$\mathit{maxW}(v,i,2) = \begin{cases} 0 & i = 0 \text{ or } v \text{ is leaf node} \\ -\infty & v \text{ is null} \\ \max \begin{cases} w(v,l) + \max\limits_{j=0}^{i-1} \Big\{ \mathit{maxW}(l,j,1) + \\ \qquad \max\{\mathit{maxW}(r,i-j-1,1), \mathit{maxW}(r,i-j-1,2)\} \Big\} \\ w(v,r) + \max\limits_{j=0}^{i-1} \Big\{ \mathit{maxW}(r,j,1) + \\ \qquad \max\{\mathit{maxW}(l,i-j-1,1), \mathit{maxW}(l,i-j-1,2)\} \Big\} \end{cases} & \text{otherwise} \end{cases}$$

These recurrences are obtained from two cases: if the matching does not contain v, then we distribute the i edges in all possible ways to the left and right subtree and find the maximum; if the matching contains v and v is matched with s, then we distribute the remaining $i - 1$ edges in all possible ways to the two subtrees such that the matching in $T(s)$ does not contain s, and find the maximum. We should explore from $i \leftarrow 0$ to k and in post-order. The final answer should be $\max\{\mathit{maxW}(r, k, 1), \mathit{maxW}(r, k, 2)\}$, where r is the root of T. Following is the pseudocode for the algorithm.

```
MaxWeightMatching($T, k$) :
  $n \leftarrow$ number of nodes in $T$
  integer $maxW[n][k+1][2]$    //$maxW[:][0][:] = 0$
  for $i \leftarrow 0$ to $k$
      for each node $v$ in $T$ in post-order
          if $v$ is leaf node or $i = 0$
              $maxW[v][i][1] \leftarrow 0$
              $maxW[v][i][2] \leftarrow 0$
          $l, r \leftarrow v.left, v.right$
          $maxWithoutV \leftarrow -\infty$
          for $j \leftarrow 0$ to $i$
              $currMax \leftarrow \max\{maxW[l][j][1], maxW[l][j][2]\} + \max\{maxW[r][i-j][1], maxW[r][i-j][2]\}$
              if $currMax > maxWithoutV$
                  $maxWithoutV \leftarrow currMax$
          $maxW[v][i][1] \leftarrow maxWithoutV$
          $leftMax, rightMax \leftarrow -\infty, -\infty$
          for $j \leftarrow 0$ to $i-1$
              $currLeftMax \leftarrow weight(v, l) + maxW[l][j][1] + \max\{maxW[r][i-j-1][1], maxW[r][i-j-1][2]\}$
              $currRightMax \leftarrow weight(v, r) + maxW[r][j][1] + \max\{maxW[l][i-j-1][1], maxW[l][i-j-1][2]\}$
              if $currLeftMax > leftMax$
                  $leftMax \leftarrow currLeftMax$
              if $currRightMax > rightMax$
                  $rightMax \leftarrow currRightMax$
          $maxW[v][i][2] \leftarrow \max\{leftMax, rightMax\}$
  $r \leftarrow$ root of $T$
  $maxWeight \leftarrow \max\{maxW[r][k][1], maxW[r][k][2]\}$
  return $maxWeight$
```

The time complexity of the algorithm is $O(k^2|V|)$. The space complexity is $O(k|V|)$. ■