CSCE629 Analysis of Algorithms

Homework 5

Texas A&M U, Fall 2019 — *09/27/19*

Lecturer: Fang Song — *Due: 10am, 10/04/19*

Instructions.

- Typeset your submission by LaTeX, and submit in PDF format. Your solutions will be graded on *correctness* and *clarity*. You should only submit work that you believe to be correct, and you will get significantly more partial credit if you clearly identify the gap(s) in your solution. You may opt for the "I'll take 15%" option (details in Syllabus).
- You may collaborate with others on this problem set. However, you must **write up your own solutions** and **list your collaborators and any external sources** for each problem. Be ready to explain your solutions orally to a course staff if asked.
- For problems that require you to provide an algorithm, you must give a precise description of the algorithm, together with a proof of correctness and an analysis of its running time. You may use algorithms from class as subroutines. You may also use any facts that we proved in class or from the book.

This assignment contains 3 questions, 6 pages for the total of 40 points and 10 bonus points. A random subset of the problems will be graded.

1. (20 points) (Longest forward-backward contiguous substring) Describe and analyze an efficient algorithm to find the length of the longest *contiguous* substring that appears both *forward* and *backward* in an input string $T[1,\dots,n]$. The forward and backward substrings must NOT overlap. Here are several examples.
   - Given the input string `ALGORITHM`, your algorithm should return 0.
   - Given the input string $\underline{\texttt{R}}\texttt{ECU}\underline{\texttt{R}}\texttt{SION}$, your algorithm should return 1, for the substring `R`.
   - Given the input string $\texttt{R}\underline{\texttt{EDI}}\texttt{V}\underline{\texttt{IDE}}$, your algorithm should return 3, for the substring `EDI`. (The forward and backward substrings must not overlap!)
   - Given the input string $\texttt{D}\underline{\texttt{YNAM}}\texttt{ICPROGRAMMING}\underline{\texttt{MANY}}\texttt{TIMES}$, your algorithm should return 4, for the substring `YNAM`. (It should not return 6, for the subsequence `YNAMIR`, because it's not contiguous.).

Solution: <u>**Definition:**</u> A ***substring*** is a contiguous sequence of characters within a string. For instance, "the best of" is a substring of "It was the best of times".

<u>Note:</u> Prefix and suffix are special cases of substring. A prefix of a string S is a substring of S that occurs at the beginning of S. A suffix of a string S is a substring that occurs at the end of S.

<u>Definition:</u> A ***subsequence*** is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements. For example, "It was times" is a subsequence of "It was the best of times", but not a substring.

Let $T[1,\ldots,n]$ be given input string, let $c[i,j]$ be the length of longest forward-backward contiguous substring for a string starting at i^{th} index and ending at j^{th} i.e.,

$$c[i,j] = |LFBCS(T[i,\ldots,j])|.$$

To develop dynamic programming algorithm for this problem we consider four cases as follows

- **<u>Case 1:</u>** If the length of the string $T[i,\ldots,j]$ is 2 i.e., there are only two elements in the string then
  - If $T[i] = T[j]$ then $c[i,j] = 1$.
  - If $T[i] \neq T[j]$ then $c[i,j] = 0$.
- **<u>Case 2:</u>** If $i = j$ then $c[i,i] = 0$, because we don't want to consider cases where the strings overlap so we make the effect induced by them in the length equal to zero. Notice when overlap occurs for example at index $i, i+1, i+2$ then, if $T[i] = T[i+2]$, we don't want the count the for $c[i,i+2]$ to be 2 but we want it to be 1 so we make $c[i,i] = 0$.
- **<u>Case 3:</u>** If $T[i] = T[j]$, then $c[i,j] = c[i+1,j-1] + 1$. Because if we have a string going from $T[i,\ldots,j]$ has the head and tail matching then, we want to add the 1 to the length of longest forward-backward contiguous substring for a string starting at $(i+1)^{th}$ index and ending at $(j-1)^{th}$. Which is nothing but the inner subproblem for $T[i+1,\ldots,j-1]$, that must be solved.
- **<u>Case 4:</u>** If $T[i] \neq T[j]$, then $c[i,j] = max\{c[i+1,j-1], c[i+1,j], c[i,j-1]\}$. Because if the string $T[i,\ldots,j]$ has different head and tail then we want check which inner subproblem of this string will lead to maximum length of longest forward-backward contiguous substring. So we consider for the cases for all possible substrings, namely $T[i+1,\ldots,j-1]$, $T[i+1,\ldots,j]$ and $T[i,\ldots,j-1]$.

By looking at the above cases we can conclude that we will need a matrix of size in the order of $O(n^2)$, where the lower triangle of the matrix will be zero because it doesn't make sense to compare the strings in reverse order. And we will fill the matrix parallel to the direction of the principal diagonal in the upper triangular matrix. If the word doesn't many repeating letters then this matrix will be mostly sparse and filled with zeros mostly. Finally we are looking for the solution for the problem $c[1, n]$.

```
Algorithm 1: LFBCS(T)
//c[i, j] memoize subproblems
for $i$ from 1 to $n$
  $c[i,i] = 0$ //principal diagonal values are zero
for $l$ form 1 to $n-1$ //diagonals
  for $i$ from 1 to $n-l$ //rows
    $j \leftarrow i+l$ //column of row $i$ on $l^{th}$ diagonal
    if $len(T[i,\ldots,j]) == 2$
      if $T[i] == T[j]$
        $c[i,j] = 1$
      else
        $c[i,j] = 0$
    if $T[i] == T[j]$
      $c[i,j] = c[i+1,j-1]+1$
    else
      $c[i,j] = max\{c[i+1,j-1], c[i+1,j], c[i,j-1]\}$
```

As we can see from the above pseudo code that this algorithm has two for loops so it should take $O(n^2)$ running time at worst case, where n is the total number of letters (characters) in the given word (string). To extract the string we can keep track where the maximum encountered then back track to position where one was encountered.

2. (20 points) (Word segmentation) If English were written without spaces, then we need to infer likely boundaries between consecutive words in the text. This is called word segmentation. For example, given `meetateight`, you can probably decide that the best segmentation is `meet␣at␣eight`. (and not `me␣et␣at␣eight`, or `meet␣ate␣ight`. This is all the more relevant in languages like Chinese and Japanese, which are written without spaces between the words.

   How could we automate this process? A simple approach that is at least reasonably effective is to find a segmentation that maximizes the cumulative "quality" of its individual constituent words. Thus, suppose you are given a black box that, for any string of letters $x = x_1x_2\ldots x_n$, will return a number $quality(x)$. This number can be either positive or negative; larger numbers correspond to more plausible English words. (So $quality($`me`$)$ would be positive, while $quality($`ght`$)$ would be negative.)

   Given a long string of letters $y = y_1y_2\ldots y_n$, a segmentation of y is a partition of its letters into contiguous blocks of letters; each block corresponds to a word in the segmentation. The total quality of a segmentation is determined by adding up the qualities of each of its blocks. (So we'd get the right answer above provided that $quality(\texttt{meet}) + quality(\texttt{at}) + quality(\texttt{eight})$ was greater than the total quality of any other segmentation of the string.)

Describe and analyze an efficient algorithm that takes a string y and computes a segmentation of maximum total quality. (You can treat a single call to the black box computing $quality()$ as a single computational step.)

Solution: We will develop an algorithm based on the assumption that we have no idea on how the *quality* function computes the cumulative quality of the words. We will define a function $OPT[i, j]$ which equals to maximum value of the *quality* for letters starting from i and going to j. Let $qual[y_i, y_j]$ be the *quality* of the letters from starting from i and going to j. Then if $i = j$,

$$OPT[i, i] = qual[y_i]$$

and if $i \neq j$ then,

$$OPT[i, j] = max \left\{ qual[y_i, y_j], max_{i \leq k < j} \left\{ OPT[i, k] + OPT[k + 1, j] \right\} \right\}.$$

Whenever if $i = j$ then the optimum value $OPT[i, i]$ will be equal to the *quality* of the letter y_i i.e., $OPT[i, i] = qual[y_i]$. If $i \neq j$ then there are two cases. One, the entire letter sequence from y_i to y_j is a possible English word then optimum value will be $OPT[i, j] = qual[y_i, y_j]$. Second, we find an optimum split k in between the letter sequence such that we get maximum *quality* i.e., $max_{i \leq k < j} \{OPT[i, k] + OPT[k + 1, j]\}$. Since we don't know if the first case or second case will lead to optimum solution we take the max of both. This analysis is quite similar to the dynamic programming analysis of matrix-chain multiplication.

The following simplification is based on the *rod-cut-problem* from the textbook. Till now we assumed that we have no idea how the *quality* function behaves. If we assume that the $qual[y_i, y_j]$ will be (large) positive number if the string of letters from y_i to y_j is an English word. And if the string of letters from y_i to y_j is not proper word it will give (large) negative values. Then in above optimization loop we can replace either $OPT[i, k]$ with $qual[y_i, y_k]$ or $OPT[k + 1, j]$ with $qual[y_{k+1}, y_j]$. We can do this because $qual[y_{k+1}, y_j]$ (or $qual[y_i, y_k]$) will take maximum value only when $y_{k+1} \dots y_j$ (or $y_i \dots y_k$) string of letters are an English word. So we don't need to do recursion on it, since we segmented one of the words from the actual sentence. Segmented word will be from front if $OPT[i, k] = qual[y_i, y_k]$ and will be from behind if $OPT[k + 1, j] = qual[y_{k+1}, y_j]$. The general formula written will work even for this case but doing this simplification will save memory and computation time. Based on this assumption we can write last case as for all $i \neq j$,

$$OPT[i,j] = max\left\{qual[y_i, y_j], max_{i \leq k < j}\left\{OPT[i,k] + qual[y_{k+1}, y_j]\right\}\right\}.$$

In both cases we are searching for, the solution for $OPT[1,n]$.

Now we have the recursion, we can build memoization data structure to solve the problem. If we were to solve the general case then we have $O(n^2)$ subproblems to solve and if we use a matrix data structure then space required will be $O(n^2)$. And the each subproblem will depend on everything on the left of the same row and everything below in the same column. We will fill up the matrix in the direction parallel to the principal diagonal in upper triangular matrix. Note that all the values in lower triangular matrix are zero because it doesn't make any sense in solving for strings of letter in reverse order. Everything on the diagonal has $OPT[i,i] = qual[y_i]$. Using this information we can fill entire matrix.

If we were to solve the simplified case then we just need an array for size $O(n)$ with $O(n)$ subproblems to solve. Basically dependency from the column is gone (if $OPT[i,k] = qual[y_i, y_k]$ will be row i.e., just transpose of the above explanation). Each subproblem will depend on all the subproblems to the left in the same row. So, we need only the first row to be solved to solve the original problem.

```
**Algorithm 2:** *WordSegmentation(y)*
for $i$ from 1 to $n$
  $OPT[i,i] = qual[y_i]$
for $l$ from 1 to $n-1$ #diagonals
  for $i$ from 1 to $n-l$ #rows
    $j \leftarrow i+l$
    $OPT[i,j] \leftarrow qual[y_i, y_j]$
      for $k$ from $i$ to $j-1$
        $OPT[i,j] = max\{OPT[i,j], OPT[i,k] + OPT[k+1,j]\}$
```

Running time for the general case would be $O(n^3)$.

```
**Algorithm 3:** *WordSegmentation(y)*
$OPT[1,1] = qual[y_1]$
for $j$ from 2 to $n$ #columns
  $OPT[1,j] \leftarrow qual[y_1, y_j]$
  for $k$ from 1 to $j-1$
    $OPT[1,j] = max\{OPT[1,j], OPT[1,k] + OPT[k+1,j]\}$
```

Running time for the simplified case would be $O(n^2)$. Because we are only solving for the first row of the OPT matrix. We can bactrack the OPT and observe rise in the optimum value between adjacent indices to find the where the word has been split.

3. (10 points (bonus)) (Greedy matrix-chain multiplication?) As suggested in class, one approach to choosing the matrix A_k at which to split the subproduct $A_i A_{i+1} \dots A_j$ is by selecting k to minimize the quantity $d_{i-1} d_k d_j$. Does this always give an optimal solution. Prove it or give a counterexample (i.e., an instance of the matrix-chain multiplication problem for which this greedy approach yields a suboptimal solution.)

Solution: We are given a sequence of matrix subproduct $A_i A_{i+1} \dots A_j$. Let each matrix A_i be of the size (d_{i-1}, d_i). Let A_k be the matrix at which we split the subproduct. Then we get two matrices $A_i A_{i+1} \dots A_k$ of size (d_i, d_k) and $A_{k+1} A_{k+2} \dots A_j$ of size (d_k, d_j). The cost of multiplying these two matrices will be $d_i d_k d_j$. The split can be anywhere i.e., $k \in \{i, \dots, j\}$ and the cost of multiplying the two matrices will always be $d_i d_k d_j$, that is d_i and d_j will always be present in the product.

In greedy algorithm we want to minimize the quantity $d_i d_k d_j$. Our greedy selection here will be choosing the product with fewest possible operations. Since greedy method is a recursive method (we use top-down approach) i.e., we starting solving the problem from the smallest subproblem and start building the solution until the final problem. It is like if we were to write a code that is similar to a DP but at each step we take the values that minimize locally, i.e., the value of k that minimizes the product $d_{i-1} d_k d_j$. Keeping this fact in mind now we will show that using greedy approach it is ***not*** possible to always find a optimal solution for matrix-chain multiplication problems.

Counterexample: Let $A = (2, 1)$, $B = (1, 2)$ and $C = (2, 5)$. If we go by greedy solution then we end up multiplying in the following way $(AB)C$, which will lead to $(2 * 1 * 2) + (2 * 2 * 4) = 24$ computations. But optimal one would be $A(BC)$ with $(2 * 1 * 5) + (1 * 2 * 5) = 20$.

Counterexample: Let $A = (3, 2)$, $B = (2, 3)$ and $C = (3, 4)$. If we go by greedy solution then we end up multiplying in the following way $(AB)C$, which will lead to $(3 * 2 * 3) + (3 * 3 * 4) = 54$ computations. But optimal one would be $A(BC)$ with $(3 * 2 * 4) + (2 * 3 * 4) = 48$.

Counterexample: Let $A = (101, 11)$, $B = (11, 9)$, $C = (9, 100)$, and $D = (100, 99)$. If we go by greedy solution then we end up multiplying in the following way $A((BC)D)$, which will lead to $(11 * 9 * 100) + (11 * 100 * 99) + (101 * 11 * 99) = 228789$ computations. But optimal one would be $(AB)(CD)$ with $(101 * 11 * 9) + (9 * 100 * 99) + (101 * 9 * 99) = 189090$.