Lab04-Dynamic Programming

CS214-Algorithm and Complexity, Xiaofeng Gao, Spring 2019.

* If there is any problem, please contact TA Jiahao Fan.
* Name:BowenZhang   Student ID:517021910797   Email: 372799293@qq.com

1. Given a positive integer n, find the least number of perfect square numbers (e.g., 1, 4, 9, ...) which sum to n.

   (a) Assume that OPT(a) = the least number of perfect square numbers which sum to a. Please write a recurrence for OPT(a).

   Solution.

$$OPT(a) = \begin{cases} 0 & a = 0 \\ \min\limits_{1 \le b \le \lfloor \sqrt{a} \rfloor} \{1 + OPT(a - b^2)\} & otherwise \end{cases}$$

□

   (b) Base on the recurrence, write down your algorithm in the form of *pseudo code*.

   Solution. The *pseudo code* is as follow:

Algorithm 1: Tabulation

Input: n
Output: $M[n]$

```
M[0]=0 ;
for j = 1 → n do
    M[j]=1+M[j-1] ;
    for i = 2 → √j do
        M[j] = min{M[j], 1 + M[j − i²]} ;
```

□

2. Given an input string s (could be empty, and contains only lowercase letters a-z) and a pattern p (could be empty, and contains only lowercase letters a-z and characters like '?' or '*'), please design an algorithm using dynamic programming to determine whether s matches p based on the following rules:

   - '?' matches any single character.
   - '*' matches any sequence of characters (including the empty sequence).
   - The matching should cover the entire input string (not partial).

   Assume $m = len(s)$ and $n = len(p)$. Output **true** if s matches p, or **false** otherwise.

   (a) Assume that ANS(i, j) means whether the first i ($0 \le i \le m$) characters of s match the first j ($0 \le j \le n$) characters of p. Please write a recurrence for ANS(i, j).

Solution. We denote that $s[i]$ is the i^{th} letter of string s and $p[j]$ is the j^{th} letter of string p.

$$ANS(i,j) = \begin{cases} true & i=0, j=0 \\ false & i=0, j\neq 0, p[j] \neq' *' \\ ANS[0][j-1] & i=0, j\neq 0, p[j] =' *' \\ false & i\neq 0, j=0 \\ false & s[i]\neq p[j] \quad \textbf{and} \quad p[j]\neq'?' \\ & \textbf{and} \quad p[j]\neq' *' \\ ANS(i-1,j-1) & s[i]=p[j] \quad \textbf{or} \quad p[j]='?' \\ ANS(i,j-1)\cup ANS(i-1,j) & p[j]=' *' \end{cases}$$

□

(b) Base on the recurrence, write down your algorithm in the form of *pseudo code*.

Solution. The *pseudo code* is as follow:

Algorithm 2: Tabulation

Input: s,p,$m = len(s)$,$n = len(p)$
Output: $ANS[n][n]$

```
ANS[0][0]=true ;
for i = 1 → m do
    ANS[i][0]=false ;
for i = 0 → m do
    for j = 1 → n do
        if i == 0 then
            if p[j] ==' *' then
                ANS[i][j]=ANS[i][j-1];
            else
                ANS[i][j]=false ;
        else if s[i] == p[j] or p[j] ='?' then
            ANS[i][j]=ANS[i-1][j-1] ;
        else if p[j] ==' *' then
            ANS[i][j]=max{ANS[i − 1][j], ANS[i][j − 1]}
        else
            ANS[i][j]=false;
```

Algorithm 3: ImprovedTabulation

Input: s,p,$m = len(s)$,$n = len(p)$
Output: $ANS[n]$

```
for j = 0 → n do
    for i = 0 → m do
        if j == 0 then
            if i == 0 then
                ANS[i]=true;
            else
                ANS[i]=false ;
        else if i == 0 then
            if p[j] ==' *' then
                ANS[i]=PREV[i];
            else
                ANS[i]=false ;
        else if s[i] == p[j] or p[j] ='?' then
            ANS[i]=PREV[i-1] ;
        else if p[j] ==' *' then
            ANS[i]=max{ANS[i − 1], PREV[i]}
        else
            ANS[i]=false;
    PREV=ANS;
```

(c) Analyze the time and space complexity of your algorithm.

Solution.

i. Time Complexity We need $O(1)$ time to compute each $ANS[i][j]$,and we have to compute mn items like this,so the time complexity is $O(mn)$.

ii. Space Complexity Originally,we need to store the matrix ANS with $O(mn)$ space complexity.Then,I find that if we keep updating every column of matrix,we can also solve the problem.So in the **ImprovedTabulation**,we only need $O(m)$ space complexity.

3. Recall the *String Similarity* problem in class, in which we calculate the edit distance between two strings in a sequence alignment manner.

(a) Implement the algorithm combining dynamic programming and divide-and-conquer strategy in C/C++ with time complexity $O(mn)$ and space complexity $O(m + n)$. (The template *Code-SequenceAlignment.cpp* is attached on the course webpage).

(b) Given $\alpha(x, y) = |ascii(x) - acsii(y)|$, where $ascii(c)$ is the ASCII code of character c, and $\delta = 13$. Find the edit distance between the following two strings.

$$X[1..60] = PSQAKADIETSJPWUOMZLNLOMOZNLTLQ$$
$$CFQHZZRIQOQCOCFPRWOUXXCEMYSWUJ$$

$$Y[1..50] = SUYLVMUSDROFBXUDCOHAAEBKN$$
$$AAPNXEVWNLMYUQRPEOCQOCIMZ$$

Solution. By running the code,we find that the edit distance is **439**. □

(c) (Bonus) Visualize the shortest path found in (b) on the corresponding edit distance graph using any tools you like.

Solution. I export the points on the path computed in **C++** code to the *.txt* file.Then I import the points to **python** and use **Tkinter** to visualize the path.The horizontal axis represents the string p and the vertical axis represents the string s.

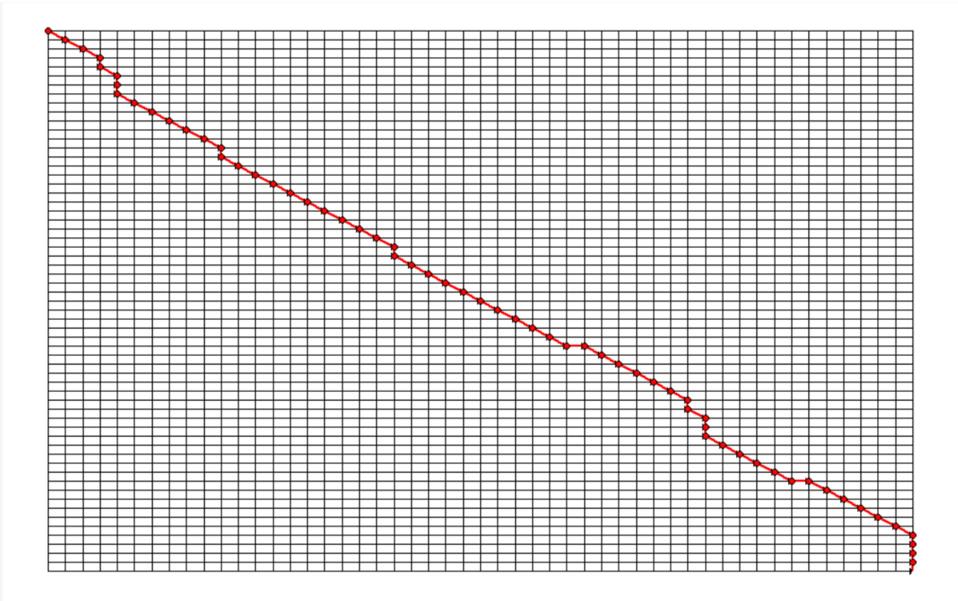

图 1: The shortest path

□

Remark: You need to include your .cpp, .pdf and .tex files in your uploaded .rar or .zip file.