Lab04-Dynamic Programming

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

∗ If there is any problem, please contact TA Jiahao Fan.

∗ Name:Kylin Chen   Student ID:517030910155   Email: k1017856853@icloud.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 $\mathrm{OPT}(a)$ = the least number of perfect square numbers which sum to a. Please write a recurrence for $\mathrm{OPT}(a)$.

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

Solution.

(a)

- **Notation:**
  - $\mathrm{OPT}(a)$ = the least number of perfect square numbers which sum to a.
- **Compute** OPT(a):

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

(b) For this problem, I will give two types of pseudo code implementation base on the previous section.

- **Pseudo Code 1:**

Algorithm 1: Dynamic Programming 1:

Input: A positive integer n;

$res[1] \leftarrow 1$;
for $i \leftarrow 2$ ***to*** n **do**
  $res[i] \leftarrow MAX\_const$;
for $i \leftarrow 1$ ***to*** n **do**
  $j \leftarrow 1$;
  while $i + j^2 < n$ **do**
    $res[i + j^2] \leftarrow \min\{res[i + j^2], res[i] + 1\}$;
    $j \leftarrow j + 1$;
return $res[n]$;

- **Pseudo Code 2:**

Algorithm 2: Dynamic Programming 2:

Input: A positive integer n;

$res[1] \leftarrow 1$;
for $i \leftarrow 2$ ***to*** n **do**
  $res[i] \leftarrow \min\limits_{1 \le j \le \lfloor \sqrt{i-1} \rfloor} \{res[i - j^2]\} + 1$;
return $res[n]$;

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

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

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

Solution.

(a)

- **Notation:**
  - ANS(i, j) = whether the first i $(0 \leq i \leq m)$ characters of s match the first j $(0 \leq j \leq n)$ characters of p.
  - $Match(i, j)$ = whether the i-th $(0 \leq i \leq m)$ character of s match the j-th $(0 \leq j \leq n)$ character of p. (including '*' and '?' matches)
  - $Star(j)$ = the number of characters the j-th $(0 \leq j \leq n)$ character '*' in p matches in s.
- **Compute** ANS(i,j):
  - **case 1:** $p[j] \neq' *'$, $Match(i, j)$, $ANS(i, j) = ANS(i-1, j-1)$.
  - **case 2:** $p[j] =' *'$, $Star(j) = 0$, $ANS(i, j) = ANS(i, j-1)$.
  - **case 3:** $p[j] =' *'$, $Star(j) >= 1$, $ANS(i, j) = ANS(i-1, j)$.
  - **case 4:** otherwise, $ANS(i, j) = false$;

(b) **Pseudo Code:**

Algorithm 3: Dynamic Matching Algorithm:

Input: A string s; A pattern string p;

```
ANS[0][0] ← true;
for i ← 0 to m do
    for j ← 1 to n do
        ANS[i][j] ← false;
        if p[j] ≠ * and Match(i, j) then
            ANS(i, j) = ANS(i − 1, j − 1)
        if p[j] == * and Star(j) == 0 then
            ANS(i, j) = ANS(i, j − 1)
        if p[j] == * and Star(j) ≥ 0 then
            ANS(i, j) = ANS(i − 1, j)
return ANS[n];
```

(c)

- **Time Complexity:** Assume $m = len(s)$ and $n = len(p)$. The work for calling to $ANS(i, j)$ for $i = 1, 2, \cdots, m;\ j = 1, 2, \cdots, n$ will take $O(1)$ each. Therefore, the time complexity is $O(mn)$.
- **Space Complexity:** We only use a $O(mn)$ boolean matrix to store $ANS(i, j)$ for $i = 1, 2, \cdots, m;\ j = 1, 2, \cdots, n$. Therefore, the space complexity is $O(mn)$.

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.

$$\begin{aligned} X[1..60] = &\ PSQAKADIETSJPWUOMZLNLOMOZNLTLQ \\ &\ CFQHZZRIQOQCOCFPRWOUXXCEMYSWUJ \end{aligned}$$

$$\begin{aligned} Y[1..50] = &\ SUYLVMUSDROFBXUDCOHAAEBKN \\ &\ AAPNXEVWNLMYUQRPEOCQOCIMZ \end{aligned}$$

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

Solution.

(a) The required code is attached in the *.zip* file. ($Xcode\_Code-SequenceAlignment.cpp$ is a $cin >>$ file tested in Mac OS X, Xcode or Clion. $Vscode\_Code-SequenceAlignment.cpp$ is origin file with $file >>$ code tested in Win 10,VScode).

(b) We we cin $X[1..60]$ and $Y[1..50]$ in $Xcode\_Code - SequenceAlignment.cpp$, we can get the edit distance is 439.

(c) we use code3.cpp to generate a OUT.txt, then use test.py with Tkinter to draw a graph.(all the input file and code is attached in .zip file) The graph is Fig. 1 ,and picture is attached in figures file.

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

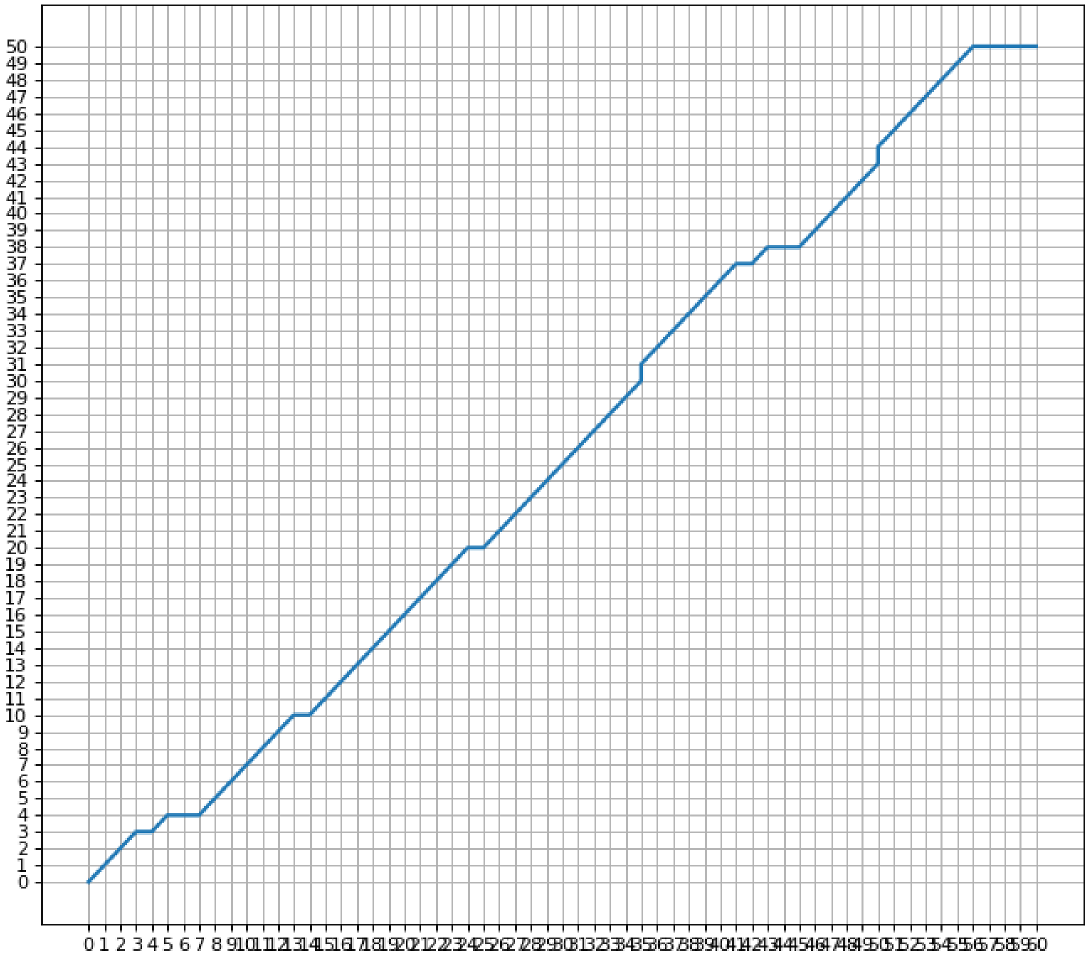

Figure 1: **Shortest Path**