Lab04-Dynamic Programming

CS214-Algorithm and Complexity, Xiaofeng Gao, Spring 2018.

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

∗ Name: Hongyi Guo   Student ID: 516030910306   Email: guohongyi@sjtu.edu.cn

1. **Coin Change**: Given currency denominations $D = \{d_1, d_2, \cdots, d_n\}$ ($0 < d_i < d_j$ for $1 \leq i < j \leq n$), devise a method based on dynamic programming to pay amount A using fewest number of coins. Return -1 if A cannot be made up by any combination of the coins.

   (a) Assume that $\mathrm{OPT}(a)$ is the fewest number of coins for the problem of paying amount a ($a \geq 0$). Please write a recurrence for $\mathrm{OPT}(a)$.

$$\mathrm{OPT}(a) = \begin{cases} 0, a = 0 \\ \min\limits_{d_i \leq a, \mathrm{OPT}(a-d_i) > -1} \{\mathrm{OPT}(a - d_i) + 1\}, a > 0 \\ -1, no - solution \end{cases}$$

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

Solution.

Algorithm 1: Coin Change

```
input  : D = {d_1, d_2, ..., d_n}, a
output: OPT(a)
for i ← 1 to n do
    F[i] ← empty;
F[0] ← 0;
OPT (a)
    if F(a) ≠ empty then
        return F(a);
    ans ← ∞;
    for i ← 0 to n do
        if d_i ≤ a then
            tmp ← OPT(a − d_i);
            if tmp ≠ −1 then
                ans ← min{ans, 1 + tmp};
        else
            break;
    if ans ≠ ∞ then
        F(a) ← ans;
    else
        F(a) ← −1;
    return F(a);
output OPT(a);
```

□

2. **Crowdsourcing** is the process of obtaining needed services, ideas, or content by soliciting contributions from a large group of people, especially an online community. Suppose you want to form a team to complete a crowdsourcing task, and there are n individuals to choose from. Each person p_i can contribute v_i ($v_i > 0$) to the team, but he/she can only work with up to c_i other people. Now it is up to you to choose a certain group of people and maximize their total contributions ($\sum_i v_i$).

   (a) Given $\text{OPT}(i, b, c)$ = maximum contributions when choosing from $\{p_1, p_2, \cdots, p_i\}$ with b persons already on board and at most c seats left before any of the existing team members gets uncomfortable. Describe the optimal substructure as we did in class and write a recurrence for $\text{OPT}(i, b, c)$.

   Solution. To compute $\text{OPT}(i, b, c)$, we consider whether we choose the i^{th} person or not. If we don't choose him, then we compute $\text{OPT}(i-1, b, c)$. If we want to choose the i^{th} person, considering there are already b persons on board, we must make sure no one will get uncomfortable, so $c_i \geq b$ and $c \geq 1$. After we choose the i^{th} person, $\text{OPT}(i, b, c) = \text{OPT}(i-1, b+1, c')$, $c' = \min\{c_i - b, c-1\}$.
   So, we get the following recurrence. The answer to the problem is $\text{OPT}(n, 0, n)$.

$$\text{OPT}(i, b, c) = \begin{cases} 0, \quad i = 0 \quad or \quad c = 0 \\ \text{OPT}(i-1, b, c), \quad c_i < b \\ \max\{\text{OPT}(i-1, b, c), v_i + \text{OPT}(i-1, b+1, \min\{c_i - b, c-1\})\}, \quad otherwise \end{cases}$$

□

   Solution. Here I have another solution, I define $\text{OPT}(i, b, c)$ = maximum contributions when we have considered all persons from $\{p_1, p_2, \cdots, p_i\}$ with b persons from them already chosen and at most c seats left before any of the existing team members gets uncomfortable. To compute $\text{OPT}(i, b, c)$, we consider whether we choose the i^{th} person or not. If so, $\text{OPT}(i, b, c) = \text{OPT}(i-1, b-1, c+1) + v_i$, but he must be able to work with at least $b-1+c$ other persons since there are already $b-1$ persons in the last $i-1$ rounds and we are waiting for at most c persons to join. If we pass the i^{th} person, then $\text{OPT}(i, b, c) = \text{OPT}(i-1, b, c)$.
   Specially, in the original situation $i = 0$ or the illegal situation $i < b$, OPT should be 0.
   So, we get the following recurrence. The answer to the problem is $\max\limits_{0<b\leq n}\{\text{OPT}(n, b, 0)\}$.

$$\text{OPT}(i, b, c) = \begin{cases} 0, \quad i = 0 \quad or \quad i < b \\ \text{OPT}(i-1, b, c), \quad c_i < b + c - 1 \\ \max\{\text{OPT}(i-1, b, c), v_i + \text{OPT}(i-1, b-1, c+1)\}, \quad otherwise \end{cases}$$

□

   (b) Design an algorithm to form your team using dynamic programming, and write it down in the form of *pseudo code*.

Solution.

Algorithm 2: Crowd Sourcing

input : n, $\{v_1, v_2, \ldots, v_n\}$, $\{c_1, c_2, \ldots, c_n\}$
output: maximized $\sum_i v_i$

for $i \leftarrow 0$ ***to*** n **do**
    for $j \leftarrow 0$ ***to*** n **do**
        for $k \leftarrow 0$ ***to*** n **do**
            $f[i, j, k] \leftarrow \mathit{empty}$;

`OPT` (i, b, c)
    if $f[i, b, c] \neq \mathit{empty}$ **then**
        return $f[i, b, c]$;
    if $i = 0$ *or* $c = 0$ **then**
        $f[i, b, c] \leftarrow 0$
    else
        $c' \leftarrow \min\{c_i - b, c - 1\}$;
        if $c' < 0$ **then**
            $f[i, b, c] \leftarrow \texttt{OPT}(i - 1, b, c)$;
        else
            $f[i, b, c] \leftarrow \max\{\texttt{OPT}(i - 1, b, c), v_i + \texttt{OPT}(i - 1, b + 1, c')\}$;
    return $f[i, b, c]$;

output `OPT` $(n, 0, n)$;

□

Solution. The algorithm of the second solution.

Algorithm 3: Crowd Sourcing

---

input : n, $\{v_1, v_2, \ldots, v_n\}$, $\{c_1, c_2, \ldots, c_n\}$
output: maximized $\sum_i v_i$

for $i \leftarrow 0$ ***to*** n **do**
  for $j \leftarrow 0$ ***to*** n **do**
    for $k \leftarrow 0$ ***to*** n **do**
      $f[i, j, k] \leftarrow$ *empty*;

`OPT` (i, b, c)
  if $f[i, b, c] \neq$ *empty* **then**
    return $f[i, b, c]$;
  if $i = 0$ *or* $i < b$ **then**
    $f[i, b, c] \leftarrow 0$
  else
    if $c_i < b + c - 1$ **then**
      $f[i, b, c] \leftarrow \texttt{OPT}(i-1, b, c)$;
    else
      $f[i, b, c] \leftarrow \max\{\texttt{OPT}(i-1, b, c), v_i + \texttt{OPT}(i-1, b-1, c+1)\}$;
  return $f[i, b, c]$;

$ans \leftarrow 0$;
for $b \leftarrow 1$ ***to*** n **do**
  $ans \leftarrow \max\{ans, \texttt{OPT}(n, b, 0)\}$;
output ans;

□

3. **Take Them Down:** Given n targets $T = \{t_1, t_2, \cdots, t_n\}$ on a straight line, your task is to shoot all of them and collect some coins. A nonnegative integer b_i is painted on the target t_i. You can only shoot one target at a time, and by destroying t_i, you will earn $b_{\text{left}} \cdot b_i \cdot b_{\text{right}}$ coins. Here t_{left} and t_{right} are adjacent targets of t_i. After the shot, t_{left} and t_{right} becomes adjacent. There are two virtual targets t_0 and t_{n+1} with $b_0 = b_{n+1} = 1$, but you cannot shoot them. Please find the maximum coins you can collect by shooting the targets wisely. (Hint: Consider which target to destroy last.)

Example:

Given 4 targets $T = \{t_1, t_2, t_3, t_4\}$ with $B = \{b_1, b_2, b_3, b_4\} = \{3, 1, 5, 8\}$, return 167.

The corresponding firing order: t_2, t_3, t_1, t_4. $(3 \times 1 \times 5 + 3 \times 5 \times 8 + 1 \times 3 \times 8 + 1 \times 8 \times 1 = 167)$

(a) Design an algorithm based on dynamic programming and implement it in C/C++ (The framework *Code-Targets.cpp* is attached on the course webpage).

Solution.

```
#include <algorithm>

int maxCoins(vector<int>& nums) {
    // Create two virtual targets
    vector<int> B(1, 1);
    for (int i = 0; i < nums.size(); i++)
        B.push_back(nums[i]);
    B.push_back(1);

    int n = B.size(); // n = N + 2
    vector<vector<int> > f(n, vector<int>(n, 0)); //f: n x n, all-zero

    for (int t = 2; t < n; t++)
        for (int i = 0; i + t < n; i++) {
            int k = i + t;
            for (int j = i + 1; j < k; j++)
                f[i][k] = max(f[i][k], f[i][j] + f[j][k] + B[i]*B[j]*B[k]);
    }
    return f[0][num - 1];
}
```

□

(b) Analysis the time complexity of your implementation.

Solution. The time complexity of constructing vector B is $\Theta(n)$. Then we analyze the dynamic programming part. The outer for loop is executed $n - 2 = \Theta(n)$ times. The middle for loop is executed $n - t$ times. The inner for loop is executed $t - 1$ times. So, the time complexity of the dynamic programming part is

$$T(n) = \sum_{t=2}^{n-1} \sum_{i=0}^{n-t-1} \sum_{j=i+1}^{k-1} = \sum_{t=2}^{n-1} \sum_{i=0}^{n-t-1} k - i - 1 = \sum_{t=2}^{n-1} \frac{(n-t)(n-t-1)}{2} = \Theta(n^3) = \Theta(N^3)$$

.

The total time complexity of my implementation is $\Theta(N^3)$;

□

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