CS 677 Homework
Assignment 06

Hai Nguyen

November 15, 2018

1. (a) Write a recursive formula of an optimal solution (i.e., define the variable that you wish to optimize and explain how a solution to computing it can be obtained from solutions to subproblems).

   i. Sort advertisements locations:

   $$x_1 < x_2 < ... < x_n$$

   ii. Define:

   $$p(j) = \text{largest index } i < j \text{ such that } x_i < x_j - 5$$

   $$\text{if p(j)} = 0 \rightarrow \text{There is no location satisfying } x_i < x_j - 5$$

   iii. Define:
   OPT(j) = value of optimal solution to the problem consisting of place the advertisements at a subset of possible sites x_1, x_2, ..., x_j.
   Case 1:
   OPT selects location x_j to put the advertisement.
   - Cannot choose locations x_i such that $x_i \geq x_j - 5$
   - Must include optimal solution to problem consisting of remaining location x_1, x_2, ..., $x_{p(j)}$.

   $$\text{OPT}(j) = r_j + \text{OPT}(p(j))$$

   Case 2: OPT does not select location x_j to put the advertisement.
   - Must include optimal solution to problem consisting of remaining location x_1, x_2, ..., x_{j-1}.

   $$\text{OPT}(j) = \text{OPT}(j-1)$$

   Combine two cases, we have the recursive formula of an optimal solution:

   $$OPT(j) = \max(r_j + \text{OPT}(p(j)), \text{OPT}(j-1))$$

   (b) Write an algorithm that computes the optimal value to this problem, based on the recurrence above. Implement your algorithm in C/C++ and run it on the following values:
   Algorithm:
   - Inputs: Locations: $x_1 < x_2 < ... < x_n$ and corresponding revenue: $r_1, r_2, ..., r_n$.
   - Compute: $p(1), p(2), ..., p(n)$
   - *Interative-Compute-Opt*
     {
       OPT[0] = 0
       for j = 1 to n
         OPT[j] = max(r_j + OPT(p(j)), OPT(j-1))
     }

   Implement in C/C++:

```
#include <iostream>

using namespace std;

int locations[] = {6, 7, 12, 14};
int revenues[] = {5, 6, 5, 1};

void printSolutionTable(int arr[], int n)
```

```
{
    cout << "Sub-solution table: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    cout << endl;
}

int main()
{
    int size = sizeof(locations) / sizeof(locations[0]);

    // Compute p
    int p[size] = {0};

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < i; j++)
        {
            if (locations[j] < locations[i] - 5)
                p[i] = j + 1;
        }
    }

    p[0] = 0;

    // Compute M
    int M[size] = {0};

    for(int j = 0; j < size; j++)
    {
        M[j] = max(revenues[j] + ((p[j] > 0) ? M[p[j] - 1] : 0), M[(j >= 1) ? j-1 :
    }


    printSolutionTable(M, size);

    cout << "Optimal value: " << M[size - 1] << endl;

    return 0;
}
```

Output:
Sub-solution table: 5 6 10 10
Optimal value: 10

(c) Algorithm to enable the construction of the optimal solution:

Store additional information: at each time step store either j or $p(j)$ value that gave the optimal solution.

Modified Algorithm:

Interative-Compute-Opt

{
  OPT[0] = 0
  for j = 1 to n
  if r_j + OPT(p(j) $\geq$ OPT(j-1):
    Store j
    OPT[j] = r_j + OPT(p(j)
  else
    Store p(j)
    OPT[j] = OPT(j-1)
}

Implement in C++

```
#include <iostream>

using namespace std;

int locations[] = {6, 7, 12, 14};
int revenues[] = {5, 6, 5, 1};

void printTable(int arr[], int n, const char name[])
{
    cout << name << endl;
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    cout << endl;
}

int main()
{
    int size = sizeof(locations) / sizeof(locations[0]);

    // Compute p
    int p[size] = {0};

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < i; j++)
        {
            if (locations[j] < locations[i] - 5)
                p[i] = j + 1;
        }
    }

    p[0] = 0;
```

```
    // Compute M
    int M[size] = {0};

    int additionalInfo[size] = {0};

    for(int j = 0; j < size; j++)
    {
        int a = revenues[j] + ((p[j] > 0) ? M[p[j] - 1] : 0);
        int b = M[(j >= 1) ? j-1 : 0];


        if (a >= b)
        {
            additionalInfo[j]  = ((p[j] > 0) ? p[j] - 1 : -1);
            M[j] = a;
        }
        else
        {
            additionalInfo[j] = ((j >= 1) ? j-1 : 0);
            M[j] = b;
        }
    }


    printTable(additionalInfo, size, "Additional table:");
    printTable(M, size, "Sub-solution table: ");

    cout << "Optimal value: " << M[size - 1] << endl;

    return 0;
}
```

Outputs:

Additional table:
-1 -1 0 2
Sub-solution table:
5 6 10 10
Optimal value: 10

Intepration of additional table A:

If starting from the location x_4 (there are four possible sites for putting advertisements):

- $A[3] = 2 = 3 - 1 \rightarrow x_4$ is not selected, the previous location x_3 is selected.
- $A[2] = 0 \rightarrow$ the next selected location x_1.
- $A[0] = -1 \rightarrow$ the selection ends here at x_1.
- Finally, x_3 and x_1 is the optimal solution.

If starting from the location x_3 (there are only three possible sites for putting advertisements):

- $A[2] = 0 \neq 2 - 1 \rightarrow x_3$ is selected, the next selected location x_1.
- $A[0] = -1 \rightarrow$ the selection ends here at x_1.

- Finally, x_3 and x_1 is the optimal solution.

(d) An algorithm that outputs the locations you choose to place the commercial advertisements in the optimal solution.

There are two possible ways to print the solutions, either using additional information or by interating through array M:

Algorithm 1:

Find-Solution-1 (A, startIndex)
if (A[startIndex] == -1)
  return
else if $A[startIndex] == startIndex - 1$
  Find-Solution-1(startIndex - 1)
else
  Find-Solution-1(p[j])
  print j

Algorithm 2:

Find-Solution-2 (j, M, p)
if (j == 0)
  output nothing
else if $(r_j + M[p(j)] > M[j-1])$
  Find-Solution-2(p(j))
  print j
else
  Find-Solution-2(j-1)

Implement in C/C++

```
#include <iostream>

using namespace std;

int locations[] = {6, 7, 12, 14};
int revenues[] = {5, 6, 5, 1};

void printTable(int arr[], int n, const char name[])
{
    cout << name << endl;
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    cout << endl;
}

void Find_Solution_1(int j, int M[], int p[])
{
    if (j == -1)
```

```
        return;

    if (M[j] == M[j - 1])
    {
        Find_Solution_1(j-1, M, p);
    }
    else
    {
        Find_Solution_1(p[j] - 1, M, p);
        cout << "x" << j + 1 << " ";
    }
}

void Find_Solution_2(int additionalInfo[], int startIdx)
{
        if (additionalInfo[startIdx] == -1)
        {
            cout << "x" << startIdx + 1 << " ";
            return;
        }

        if (additionalInfo[startIdx] == startIdx - 1)
        {
            Find_Solution_2(additionalInfo, startIdx - 1);
        }

        else
        {
            Find_Solution_2(additionalInfo, additionalInfo[startIdx]);
            cout << "x" << startIdx + 1 << " ";
        }
}

int main()
{
    int size = sizeof(locations) / sizeof(locations[0]);

    // Compute p
    int p[size] = {0};

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < i; j++)
        {
            if (locations[j] < locations[i] - 5)
                p[i] = j + 1;
        }
    }

    p[0] = 0;
```

```
    // Compute M
    int M[size] = {0};

    int additionalInfo[size] = {0};

    for(int j = 0; j < size; j++)
    {
        int a = revenues[j] + ((p[j] > 0) ? M[p[j] - 1] : 0);
        int b = M[(j >= 1) ? j-1 : 0];


        if (a >= b)
        {
            additionalInfo[j]  = ((p[j] > 0) ? p[j] - 1 : -1);
            M[j] = a;
        }
        else
        {
            additionalInfo[j] = ((j >= 1) ? j-1 : 0);
            M[j] = b;
        }
    }


    printTable(additionalInfo, size, "Additional table:");
    printTable(M, size, "Sub-solution table: ");

    cout << "Solution way 1: ";
    Find_Solution_1(size - 1, M, p);
    cout << endl;

    cout << "Solution way 2: ";
    Find_Solution_2(additionalInfo, size - 1);
    cout << endl;

    cout << "Optimal value: " << M[size - 1] << endl;

    return 0;
}
```

Output:
Additional table:
-1 -1 0 2
Sub-solution table:
5 6 10 10
Solution way 1: x1 x3
Solution way 2: x1 x3
Optimal value: 10

2. Show how the algorithm MATRIX-CHAIN-ORDER discussed in class computes the number of scalar multiplications for the product of the following three matrices (i.e., give the values in table m as computed by the algorithm): A[4x3], B[3x5] and C[5x2].

   Recurrence relationship:

$$m[i,j] = \begin{cases} 0 \quad \text{if} \quad i = j \\ \min\limits_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j) \quad \text{if} \quad i < j \end{cases}$$

   Dimensions of matrices:

$$A_1 : 4 \times 3 = p_0 \times p_1$$
$$A_2 : 3 \times 5 = p_1 \times p_2$$
$$A_3 : 5 \times 2 = p_2 \times p_3$$

   Construct the table using the recurrence formula:

$$\begin{aligned} m[1,1] &= m[2,2] = m[3,3] = 0 \\ m[1,2] &= m[1,1] + m[2,2] + p_0 \times p_1 \times p_2 \\ &= 4 \times 3 \times 5 = 60 \\ m[2,3] &= m[2,2] + m[3,3] + p_1 \times p_2 \times p_3 \\ &= 3 \times 5 \times 2 = 30 \\ m[1,3] &= \mathbf{min}(m[1,1] + m[2,3] + p_0 \times p_1 \times p_3, m[1,2] + m[3,3] + p_0 \times p_2 \times p_3) \\ &= \mathbf{min}(30 + 4 \times 3 \times 2, 60 + 4 \times 5 \times 3) \\ &= 54 \end{aligned}$$

	1	2	3
3	54	30	0
2	60	0	
1	0		

Figure 1: Table to calculate optimal solution.

   The minimum number of scalar multiplications for the product is 54.

3. Answer the following questions and justify your answers.

   (a) If X and Y are sequences that both begin with the character A, every longest common subsequence of X and Y begins with A.

True. Because if no LCS begins with A we can insert A to the front of any optimal LCS to get a longer common subsequence. But this would contradict the definition of the LCS as being longest.

(b) If X and Y are sequences that both end with the character A, some longest common subsequence of X and Y ends with A.

True. More precisely, every LCS will need to contain A. Supposing that no LCS will contain that letter. Thus, we can add it to the end of the LCS, creating a longer common subsequence. But this would contradict the definition of the LCS as being longest.