Dynamic Programming Algorithms

Sample Solved Exercises

Prof. Vanessa Silva

Departamento de Ciência de Computadores (DCC)
Faculdade de Ciências da Universidade do Porto (FCUP)

Spring 2023

Exercise 1

Given a n -level pyramid of numbers, find the path with the maximum sum that starts at the top of the pyramid and ends at its base. Note that at each step you can go diagonally down and to the left or down and to the right. For example, consider the following 5-level pyramid of numbers:

```
        7
      3   8
    8   1   0
  2   7   4   4
4   5   2   6   5
```

The path $7 \rightarrow 8 \rightarrow 0 \rightarrow 4 \rightarrow 2$ results in a sum of 21 and path $7 \rightarrow 3 \rightarrow 8 \rightarrow 7 \rightarrow 5$ results in a sum of 30 which is the maximum sum.

Consider that all numbers are integers between 0 and 99 and the number of lines in the pyramid is as most 100. Write an efficient algorithm which finds the path that results in the maximum sum of a given pyramid of numbers. Starts by writing a recursive function that solves this problem and explain why that function does not solve the problem in the most efficient way.

Solution:

Starting at the top of the pyramid, we have two possible decisions: go down and left or go down and go right. In each case we must consider all the paths of the respective sub-pyramids, that is, we need to know the value (sum) of the best internal route in each sub-pyramid (smallest instance of the same problem). For the example above, the solution is 7 plus the maximum between the best path value of each of the sub-pyramids.
Then we can solve the problem recursively. Considering,

- $p[i][j]$ the *j*-th number of *i*-th line (level)
- $maximum\_sum(i, j)$ the highest sum we get from position *i*, *j*

We can write the following recursive function:

```
function maximum_sumRec(p,n,i,j) :
    if i = n-1 then
        return p[i][j]
    else
        return p[i][j] + max(maximum_sumRec(i+1,j), maximum_sumRec(i+1,j+1))
```

To solve the problem just call $maximum\_sumRec(p, n, 0,0)$.
The recursive function above has an exponential time complexity, i.e., $T(2^n)$, as it computes the same subproblems multiple times.

A more efficient way to solve the problem is to reuse the results of overlapping subproblems calculated at each step, avoiding repeated computations. For this we can use a table matrix $m[][]$ with the value computed for each subproblem, filling the table from bottom-up.
So, we can write the following function:

```
function maximum_sumDP(i,j) :
    m[n][n]
    for j = 0 to n-1 do
        m[n-1][j] = p[n-1][j]
    for i = n-2 to 0 do
        for j = 0 to i do
            m[i][j] = p[i][j] + max(m[i+1][j], m[i+1][j+1])
    return p[0][0]
```

Thus, applying the concept of bottom-up dynamic programming, we obtain a polynomial time complexity, i.e., $T(n^2)$.
To get the elements in the path that result in the maximum sum just use the calculated table.

Exercise 2

The Catalan numbers are a sequence of non-negative numbers that occur in several counting problems, often involving recursively defined items. It also can be defined as a mathematical sequence used to find the number of possibilities of various combinations.
The n -th Catalan number (C_n, $n \geq 0$) can be expressed directly in terms of binomial coefficients by

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!}$$

The first few Catalan numbers for $n = 0, 1, 2, 3, \ldots$ are:

$$1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \ldots$$

Write two algorithms, one using the recursive method and the other using the dynamic programming methods, that compute the Catalan number for a given number n. Analyze the time complexity of the algorithms.

Solution:

Catalan numbers satisfy the following recurrence relations:

- $C_0 = 1$
- $C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$

Then, we can write the following recursive algorithm:

```
function catalanRec(n) :
    if n <= 1 then
        return 1
    res = 0
    for i = 0 to n-1 do
        res = res + catalanRec(i) * catalanRec(n-i-1)
    return res
```

The above function has time complexity exponential, since n -th Catalan number is exponential, i.e., $T(n) = \sum_{i=0}^{n-1} T(i)T(n-i-1)$ for $n \geq 1$.

The recursive algorithm above repeatedly computes the same subproblem, and therefore does not compute the n -th Catalan number efficiently. So, we can use dynamic programming with bottom-up method to obtain a more efficient solution. For this we can use a binomial coefficient $C(n, k)$ to compute the n -th Catalan number. We use a bottom-up dynamic programming method to compute the binomial coefficient. The value of $C(n, k)$ can be recursively by,

$$C\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ C\binom{n-1}{k-1} + C\binom{n-1}{k} & \text{if } 0 < k < n \\ 0 & \text{otherwise} \end{cases}$$

Since the same subproblems are called again, this problem has the overlapping subproblems property. So, we use an table array $\mathrm{m}[]$ to store the overlapping subproblems, constructing $m[]$ in a bottom-up manner.
We can write the following algorithm:

```
function binomial_coefficientDP(n,k) :
    C[n+1][k+1]
    for i = 0 to n do
        for j = 0 to min(i, k) do
            if j = 0 or j = i then
                C[i][j] = 1
            else
                C[i][j] = C[i-1][j-1] + C[i-1][j]
    return C[n][k]

function catalanDP(n) :
    res = (1/(n+1)) * binomial_coefficientDP(2*n, n)
    return res
```

The above function has time complexity polynomial, $T(n^2)$, we use two *for* cycle to compute the binomial coefficient $C(2n, n)$.

Exercise 3

With the popular Harry Potter films in mind, consider that in one of these movies Harry was challenged to mix a set of colorful mixtures together. He has available n mixtures that are arranged in a row. Each mixture has one of 100 different colors (colors have numbers from 0 to 99). At each step, he can only take two mixtures that stand next to each other and mix them together. The resulting mixture is then put in the place of the others. Harry must carry out this process repeatedly until he mixes all the mixtures and gets one. Furthermore, Harry's challenge has the following implications:

- When mixing two mixtures of colors a and b, the resulting mixture will have the color $(a + b)\ mod\ 100$.
- There will be some smoke in the process. The amount of smoke generated when mixing two mixtures (at each step) of colors a and b is $a * b$.

Your task is to write an algorithm that find the minimum amount of smoke that Harry can get when mixing all the mixtures together. Let n the number of mixtures and $X[0 \dots n-1]$ an ordered array with the colors of the n mixtures.
Consider the simple illustrative example to $n = 3$ and $\mathrm{X} = \{40{,}60{,}20\}$, there are two possibilities to mix the mixtures:
- mix 40 and 60 (smoke: 2400), resulting in 0, and then mix 0 and 20 (smoke: 0). The total amount of smoke is 2400
- mix 60 and 20 (smoke: 1200), resulting in 80, and then mix 40 and 80 (smoke: 3200). The total amount of smoke is 4400

The first process option is Harry's best way to get the minimum amount of smoke.

Solution:

The presented problem is similar to matrix-matrix multiplication problem. In the matrix-matrix multiplication problem, when we multiply two matrices of the order (a, b) and (b, c), we get a matrix of the order (a, c). In the above problem, when Harry mix two adjacent colors a and b, he will get the color $(a + b)\ \%\ 100$. So, using a bottom-up dynamic programming methods (similar to the used in matrix-matrix multiplication problem) and store the result of sub-problems in a table matrix $m[\,][\,]$, we can solve the problem.
The matrix element $m[i][j]$ means the minimum amount of smoke produced while mixing the mixtures from $X[i \dots j]$ into a single mixture. At the previous step, we would have had to mix two mixtures which are resultants of ranges $X[i \dots k]$ and $X[k + 1 \dots j]$ where $i \le k < j$, and so on. Thus, we can write the following recurrence:

$$m[i,j] = \begin{cases} 0 & \text{if } i = i \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + smix(i,k) * smix(k+1,j)\} & \text{if i < j} \end{cases}$$

where $smix(a, b) = (a + b)\%100$, i.e., the color of the resulting mixture $from\ X[a \dots b]$.
The solution to the problem is in $m[1, n]$.

We can write the following dynamic programming algorithm:

```
function mixturesDP(X,n) :
    m[n+1][n+1]
    smix[n+1][n+1]
    for i = 1 to n do
        m[i][i] = 0
        smix[i][i] = X[n-1]
    for l = 2 to n do
        for i = 1 to n-l+1 do
            j = i+l-1
            m[i][j] = INT_MAX
            for k = i to j-1 do
            x = m[i][k] + m[k+1][j] + smix[i,k]*smix[k+1,j]
            if x < m[i][j] do
                m[i][j] = x
                smix[i][j] = (smix[i,k] + smix[k+1,j])%100
    return m[1][n]
```

Exercise 4

Given a sequence of positive integers $S = \{s_1, \dots, s_n\}$, it is intended to find the number of ways numbers $s_1, \dots, s_n$ can be divided into two subsequences such that the sum of the elements in both subsequences is equal.

Consider some examples:

- $S = \{1,6,11,6\}$, we can divide S in only 1 way: $\{6,6\}$ and $\{1,11\}$ obtaining a sum of 12 in both resulting subsequences.
- $S = \{6,11,6\}$ we cannot divide S into equal sum subsequences, resulting into 0 ways of dividing S.
- $S = \{1,2,3,4,5,6,7\}$, we can divide S in 4 different ways (with sum of 12 in the resulting subsequences):
  - $\{1,3,4,6\}$ and $\{2,5,7\}$
  - $\{1,2,5,6\}$ and $\{3,4,7\}$
  - $\{1,2,4,7\}$ and $\{3,5,6\}$
  - $\{1,6,7\}$ and $\{2,3,4,5\}$

Write an algorithm based on dynamic programming that count the number of ways numbers $s_1, \dots, s_n$ can be divided into two subsequences of equal sum.

Solution:

If we intend to have two subsets of equal sum, then they must add up to half of the total sum each. Therefore, if the total sum $\sum_1^n s_i$ is an odd number, the answer is 0 (there is no way to divide S into two equal-sum subsequences). However, if the total sum is an even number, there are 1 or more ways to divide S.

Knowing that we can divide S, the problem boils down to finding the number of ways to obtain two subsequences that each add up to $\sum_1^n s_i /2$, considering the possible combinations of numbers in S. Knowing that if we find one given subsequence that adds up $\sum_1^n s_i /2$, the remaining numbers form a subsequence that add up to the same, and therefore we obtain two subsequences of equal sum. We can solve this problem using dynamic programming similar to the 0/1 knapsack problem, since for each number in S, we can put it in the first subsequence or not.

We start by putting the number s_n in the second subsequence, only counting each pair of subsequences once (otherwise we count every pair of subsequences twice). And we consider a table matrix with elements $m[i][j]$ that represent the number of ways to gotten sum *j* using the first *i* numbers, i.e., $s_1, \dots, s_i$. We consider as the base case the empty sequence, we can obtain two subsequences (both empty) with equal sum 0, so $m[0][0] = 1$. For the remaining cases we can two options (in each iteration):

- do not include the s_i number in the sum, and then we see if we can get j from the previous subsequence, i.e., there are $m[i-1][j]$ possibilities
- include the s_i number in the sum (if its value is not more than j), and then we see if we can find a subsequence to get the remaining sum, i.e., there are $m[i-1][j-s[i]]$ possibilities

So, we have $m[i][j] = m[i-1][j] + m[i-1][j-s[i]]$. With this we can write the next algorithm, where $m[n][sum]$ contains the solution to the presented problem.

```
function two_subseq_sumDP(S,n) :
    sum = 0
    for i = 0 to n-1 do
        sum = sum + S[i]
    if sum%2 != 0 then
        return 0
    sum = sum/2
    m[n+1][sum+1] = {0}
    m[0][0] = 1
    for i = 1 to n do
        for j = 0 to sum do
            m[i][j] = m[i-1][j];
            l = j - S[i-1];
            if l >= 0 then
                m[i][j] = m[i][j] + m[i-1][l]
    return m[n][sum]
```

Exercise 5

Consider a sequence of n characters $X =< x_1, \dots, x_n >$, write an algorithm that takes a sequence X and returns the length of the longest palindrome subsequence in X. A subsequence is palindromic if it is the same whether read left to right or right to left.
For example, if $X = ABBABCCABBAB$ the longest palindrome subsequence in X is "ABBACCABBA" with length 10. Note that the subsequences "ABBAABBA", "ABABABA" and "BBBBBB" are also palindromic subsequences of the X, but not the longest ones.

Formulate an optimal substructure for the presented problem and write a solution algorithm that returns the length of the longest palindromic subsequence of a given sequence X.

Solution:

A brute-force solution to this problem is to generate all subsequences of the given X sequence and find the longest palindrome subsequence. However, such a solution has exponential time complexity, and we need to find an algorithm that responds to the problem more efficiently.
Considering a given sequence $X[0 \dots n-1]$ of length n. Strings of length 1 (i.e., each single character) are palindromes of length 1. If the last and first characters of X are the same it means that 2 characters count towards the length of the palindrome subsequence and the initial problem reduces to an identical subproblem for the remaining subsequence, that is, find the length of the longest palindrome subsequence in the subsequence $X[1 \dots n-2]$. If the last and first characters of X are different the problem reduces to finding the length of the longest palindrome subsequence between the subsequences $X[1 \dots n-1]$ and $X[0 \dots n-2]$. Such a solution leads to the following recurrence function,

$$m[i,j] = \begin{cases} 1 & \text{if } i = j \\ m[i+1, j-1] + 2 & \text{if } X[i] = X[j] \\ \max\left(m[i+1,j], m[i,j-1]\right) & \text{otherwise} \end{cases}$$

where $m[0, n-1]$ represents the length value of the longest palindrome subsequence of X.
A recursive algorithm that follows the above recursion function solves many subproblems that are repeated many times. This implies that we can solve the problem more efficiently using dynamic programming method, to avoid these overlapping subproblems. For this we use a table matrix $m[][]$ that stores the results of these subproblems.

We thus have the following polynomial algorithm ($T(n^2)$) based on the bottom-up method:

```
function longest_palindromeDP(X,n) :
    m[n][n]
    for i = 0 to n-1 do
        m[i][i] = 1
    for i = 2 to n do
        for j = 0 to n-i do
            k = j+i-1
            if X[j] == X[k] && i = 2 then
                m[j][k] = 2
            else if X[j] == X[k] then
                m[j][k] = m[j+1][k-1] + 2
            else
                m[j][k] = max(m[j][k-1], m[j+1][k])
    return p[0][n-1]
```

Exercise 6

Consider that you are in a store shopping, you have a selected set of n items $(1, \ldots, n)$ that you would like to buy. However, your knapsack cannot support a weight greater than W. The value v_i and the weight w_i associated with each of these items are known. Given that information and the weight restriction, choose the items to put in the knapsack to maximize the total value of the items carried in the knapsack.
Formalize the problem and write a dynamic programming algorithm that takes as arguments two integer values W and n which represent, respectively, the knapsack capacity and the number of items, and two integer arrays $w[0 \ldots n-1]$ and $v[0 \ldots n-1]$ which represent, respectively, the weights and values associated with the n items. Your solution should return the sum of item values that maximizes the total value. Note that you cannot break an item and there are no duplicate items.

Consider the simple examples:

- to $v[] = \{4,5,10,11,13\}$, $w[] = \{3,4,7,8,9\}$ and $W = 17$, the maximum total value is 24. Buying 1 item of value 4 and 2 items of value 10 or buying 1 item of value 11 and another of value 13 are two possible solutions that maximize the total value.
- to $v[] = \{1,2,3\}$, $w[] = \{4,5,6\}$ and $W = 3$, the maximum total value is 0. All items weigh more than the knapsack capacity, i.e., $w[i] > W$ for all $0 \leq i < n$.

Solution:

The above problem implies considering all possible subsets of items and calculate the respective total weight and total value, and considering only the subsets who's the total weight does not exceed W. The goal is finding the maximum value subset. Let $x_i = 1$ if i-th item is included in the knapsack and $x_i = 0$ otherwise, the objective is find the subset that maximize the total value, i.e., $\max \sum_{i=1}^{n} x_i v_i$, and that respects the weight restriction: $\sum_{i=1}^{n} x_i w_i \leq W$.

Considering the above formulation, we use a table matrix with elements $m[i][j]$ to store the maximum value that is possible to transport with a weight limit of *j*. The matrix $m[][]$ allow us to test all possible subsets of items without recalculating similar sub-problems. Such that, $m[i][j] = \max(m[i-1][j], m[i-1][j - w[i-1] + v[i-1]])$ (take the maximum value, taking into account do not consider or consider *i*-th item in the subset.) and $m[0][j] = m[i][0] = 0$ (base cases). The solution to the problem is then stored in $m[n][W]$. We can write the following algorithm:

```
function knapsack(int W, int n, int w[], int v[]) :
    m[n+1][W+1]
    for i = 0 to n do
        for j = 0 to W do
            if i = 0 or j = 0 then
                m[i][j] = 0
            else if w[i - 1] <= j then
                m[i][j] = max(m[i-1][j], v[i-1] + m[i-1][j-w[i-1]])
            else
                m[i][j] = m[i - 1][j]
    return m[n][W]
```

Exercise 7

People in Cubeland use cubic coins. Not only the unit of currency is called a cube but also the coins are shaped like cubes and their values are cubes. Coins with values of all cubic numbers up to $9261 (= 21^3)$, i.e., coins with the denominations of $1, 8, 27, \ldots$, up to 9261 cubes, are available in Cubeland.
John works in a grocery store in Cubeland and he wants to give change for a certain amount to a customer. Your task is to count the number of ways that John can make the amount n using cubic coins of Cubeland. Write an algorithm that returns this quantity. You may assume that all the amounts are positive and less than 10000.

For example, there are 3 ways to Jonh pay $n = 21$ cubes: twenty-one 1 cube coins, or one 8 cube coin and thirteen 1 cube coins, or two 8 cube coin and five 1 cube coins.

Solution:

The above problem is similar to unlimited coin change problem. Then, we can use a bottom-up dynamic programming methods to find the result. So, we start by initialize a table array $m[]$ with values equal to 0. The base case occurs when $n = 0$ and therefore we have only 1 possible way to solve it. Then, iteratively, we update the number of ways make j using the *i*-th cube coin (the overlapping subproblems are stored in m) while $j \leq n$. Note that, we need $n + 1$ rows as the table is constructed in bottom-up manner using the base case ($n = 0$). We can write the following dynamic programming algorithm:

```
function cube_coin_changeDP(n) :
    m[n+1]
    m[0] = 1
    for i = 1 to n+1 do
        m[i] = 0
    for i = 1 to 22 do
        c = i*i*i
        for j = c to n+1 do
            m[j] = m[j] + m[j-c]
    return m[n]
```

Exercise 8

Consider a context free grammar G and a string w, G is constituted by finite set of variables V, a finite set of terminal symbols (i.e., the alphabet), a finite set of rules P, and a start symbol S (distinguished element of V). Assume that V is disjoint of the alphabet and that G is used to generate the string of a language L.
G is said to be in Chomsky Normal Form if all production rules satisfy one of the following conditions:

- a non-terminal generating a terminal (e.g. $A \rightarrow a$)
- a non-terminal generating two non-terminals (e.g., $A \rightarrow BC$)
- the start symbol generating null string (i.e., $S \rightarrow \varepsilon$)

where $B, C \in V/\{S\}$.

Your task is to write a dynamic programming algorithm that checks whether a given string w of length n is in the language L of a grammar G.

For example, to a $w = baaba$ and a G with rules:

- $S \rightarrow AB \mid BC$
- $A \rightarrow BA \mid a$
- $B \rightarrow CC \mid b$
- $C \rightarrow AB \mid a$

checking the rules, we can find that w is in $L(G)$.

Solution:

We start to construct a triangular table $m[][]$ where each row corresponds to one length of substrings (bottom row corresponds to strings of length 1, second from bottom row corresponds to strings of length 2, till top row that corresponds to string w). Let $m[i][i]$ the set of variables A (non-terminal) such that $A \rightarrow w[i]$ is a production of G, first we check if $w = \varepsilon$ (empty string) and $S \rightarrow \varepsilon$ is a rule in G then we accept the string else we reject, and then we compare at most n pairs of previously computed sets: $(m[i][i], m[i+1][j]), (m[i][i+1], m[i+2][j]) \ldots (m[i][j-1], m[j][j])$.

We can obtain the following algorithm:

```
function cykDP(V,P,S,w,n) :
    m[n+1][n+1]
    if w = {} then
        if S -> {} then
            return true
        else
            return false

    for i = 1 to n do
        for each A in V do
            if A->a is in P and a=w[i] do
                m[i][i] = A
```

```
for l = 2 to n do
    for i = 1 to n-l+1 do
        j = i+l-1
        for k = i to j-1 do
            for each rule A -> BC in P do
                if B is in m[i][k] and C is in m[k+1][j] do
                    m[i][j] = A

if S is in m[1][n] do
    return true
else
    return false
```