CSC-1S003-EP Introduction to Algorithms

TD 4: Dynamic Programming and Greedy Algorithms

March 6th, 2025

Exercise 1: the grid game, dynamically and greedily

Remember the grid game from the lecture: we are given an $m \times n$ integer matrix `G`, and we must find the maximum weight of a monotonic path from position $(0,0)$ to position $(m-1, n-1)$, where:

- *monotonic* means that, from position (i,j), a path may only go to positions $(i+1,j)$ or $(i,j+1)$;
- the weight of a path is the sum of all integers at the positions crossed by the path.

We considered the following dynamic programming algorithm:

```
def localMaxs(G, i, j, H):
    if i == len(G) or j == len(G[0]):
        H[i][j] = 0
        return
    if H[i+1][j] < 0:
        localMaxs(G, i+1, j, H)
    if H[i][j+1] < 0:
        localMaxs(G, i, j+1, H)
    H[i][j] = max(H[i+1][j], H[i][j+1]) + G[i][j]

def maxWDyn(G):
    # G rectangular, non-empty
    m,n = len(G)+1, len(G[0])+1
    H = [None] * m
    for i in range(m):
        H[i] = [-1] * n
    localMaxs(G, 0, 0, H)
    return H[0][0]
```

The main function `maxWDyn` creates and initializes a table `H`, and then calls the function `localMaxs` which actually does the work, by filling in the table `H` with local maxima, *i.e.*, `H[i][j]` contains the weight of the best monotonic path from (i,j) to $(m-1, n-1)$. So, when `localMaxs` is done, `maxWDyn` simply returns `H[0][0]`.

Notice that `localMaxs` is defined recursively. In the lecture, this was done to make more apparent the link with the naive recursive algorithm, which computes the optimal cost at (i,j) as the max of the optimal costs at $(i+1,j)$ and $(i,j+1)$, plus `G[i][j]`. `localMaxs` does exactly the same, except that it stores the intermediate results in a table and does not recompute them when they are needed again.

However, in the dynamic programming approach, the table `H` (also known as the *DP matrix*) may be computed without recursion, with two nested `for` loops, observing that the recursive relation of the naive algorithm also holds for the table:

```
H[i][j] = max(H[i+1][j], H[i][j+1]) + G[i][j]
```

1. Re-implement the dynamic programming solution as a non-recursive function `maxWDynNonRec` creating the table `H` and filling it up with two nested `for` loops, using the above relation. (Be careful! The DP matrix is filled up "backwards", from `H[m-1][n-1]` to `H[0][0]`).
2. Remember the greedy algorithm for solving this problem: at position (i,j), take the path going to whichever of $(i+1,j)$ or $(i,j+1)$ holds the maximum weight. Implement such an algorithm as a function `maxWGreedy`.

3. Initialize a random matrix with one million entries between 0 and 9, for example using the code

```
import random
n = 1000
G = [0] * n
for i in range(n):
    G[i] = [0] * n
    for j in range(n):
        G[i][j] = random.randint(0,9)
```

Now test the different implementations on `G`, for example with the code

```
import time
t0 = time.time()
maxWDyn(G)
print("Recursive DP algorithm took", time.time()-t0, "seconds")
t0 = time.time()
opt = maxWDynNonRec(G)
print("Non-recursive DP algorithm took", time.time()-t0, "seconds")
t0 = time.time()
subopt = maxWGreedy(G)
print("Greedy algorithm took", time.time()-t0, "seconds")
print("Optimal weight:", opt)
print("Suboptimal weight:", subopt)
print("Ratio:", float(subopt)/float(opt))
```

What do you observe?

Exercise 2: the edit distance

When we type using a small touchscreen (such as on a mobile phone) we often make mistakes. For example, we might type "Hello workd" when we intended to write "Hello world". The software on the phone promptly recognizes non-existing words and replaces them with what it thinks are the closest existing words.[1] But what does "closest" mean?

Autocorrect algorithms are usually based on the so-called *edit distance* between words: when we type a non-existing word u, they look for an existing word v which has minimal edit distance from u. The simplest form of edit distance is defined in terms of the following basic transformations:

- substituting a character for another character: for example, "rain" $\longrightarrow$ "pain";
- inserting a character: for example, "rain" $\longrightarrow$ "train";
- deleting a character: for example, "rain" $\longrightarrow$ "ran".

Each of these basic transformations is attributed a cost:

- $w_{\text{sub}}(a,b)$ is the cost of substituting a character a for a character b. It may depend on the specific characters: for example, for autocorrect algorithms, $w_{\text{sub}}(a,b)$ typically increases as the distance of b from a in the keyboard increases. Here, we will make the simplifying assumption that the cost is constant, except when $a = b$:

$$w_{\text{sub}}(a,b) := \begin{cases} 0 & \text{if } a = b, \\ 2 & \text{if } a \neq b \end{cases}$$

(substituting a letter for itself of course is useless, but it is convenient to define the cost anyway, and set it to zero).

[1] Sometimes producing hilarious results...

- $w_{\text{ins}}(a)$ and $w_{\text{del}}(a)$ are the costs of inserting and deleting a character a, respectively. Here, we will make the simplifying assumption that every character has the same insertion and deletion cost:

$$w_{\text{ins}} := w_{\text{del}} := 1.$$

Given two words u, v, the edit distance $d(u, v)$ is defined as the cost of the lowest-costing sequence of basic transformations turning u into v. Formally, it may be defined by induction on the length of words, as follows (below, we write ε for the empty word, $|u|$ for the length of a word u, and ua for the word u followed by the character a):

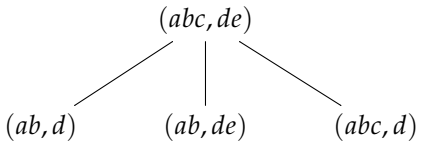
$$\begin{aligned} d(u, \varepsilon) &:= |u| w_{\text{del}} \\ d(\varepsilon, v) &:= |v| w_{\text{ins}} \\ d(ua, vb) &:= \min(\ d(u, v) + w_{\text{sub}}(a, b) \quad , \quad d(u, vb) + w_{\text{del}} \quad , \quad d(ua, v) + w_{\text{ins}}\ ) \end{aligned}$$

Intuitively, when comparing two words, we consider, for every character, every possible way in which they could be related (by substitution, deletion or insertion), and we take the minimum.

1. Write a recursive Python function `editDist` computing the edit distance of two strings (with the above costs) by directly implementing the definition. You may get the length of a string `s` with `len(s)`, access the last character of a non-empty string `s` with `s[len(s)-1]`, and obtain a string equal to `s` minus the last character with `s[0:len(s)-1]`.

2. Let $T(m, n)$ be the asymptotic time complexity of `editDist(u,v)`, where m and n are the length of u and v, respectively. Assuming that all the string operations described above have cost $\Theta(1)$, what recurrence does T verify? Prove that $T(m, n) = O(3^{m+n})$. (*Hint: prove by induction that, for large enough m and n, $T(m, n) \leq \frac{c}{2}(3^{m+n+1} - 1)$, where c is a positive constant given by the Θ notation*).

3. If we run `editDist("abc","de")`, the tree of the inputs of each recursive call starts like this:



Complete the tree. Are there any repeated nodes? What does that suggest?

Exercise 3: the Needleman-Wunsch algorithm

The edit distance is not only useful for correcting typing errors; it is also extensively used to compare gene sequences in bioinformatics. In that context, the strings on which the edit distance must be computed are much longer than any word in any existing language (they can be hundreds of characters long). Therefore, an exponential algorithm like the one implemented in the previous exercise is unusable. Luckily, a dynamic programming optimization is possible, giving what is known as the Needleman-Wunsch algorithm.

If we call D the DP matrix of the Needlman-Wunsch algorithm, its size will be $(m + 1) \times (n + 1)$, where m and n are the lengths of the two input strings, and its entries will satisfy the following equation:

$$D_{i,j} = \min \begin{cases} D_{i-1,j-1} & \text{if } x_i = y_j \text{ (match)} \\ D_{i-1,j-1} + 2 & \text{if } x_i \neq y_j \text{ (mismatch)} \\ D_{i-1,j} + 1 & \text{(insertion)} \\ D_{i,j-1} + 1 & \text{(deletion)} \end{cases}$$

The numbers 2 and 1 come from the costs of substitution, insertion and deletion, as set in the previous exercise.

On input two strings u and v, the entry $D_{i,j}$ will contain $d(u[0:i], v[0:j])$, the edit distance of the prefix of length i of u and the prefix of length j of v. Therefore, $d(u,v) = D_{|u|,|v|}$.

When applied to gene sequences, one is usually not interested so much in the value of the edit distance, but in how exactly the letters of the two strings are matched to obtain the minimum cost. For example, if $u =$ CATATA and $v =$ GGATACATA, then the edit distance is 5, and a matching with such a minimum cost is the following:

u	-	-	C	A	T	A	-	-	T	A
v	G	G	-	A	T	A	C	A	T	A
single costs:	1	1	1	0	0	0	1	1	0	0

In other words, we obtained GGATACATA from CATATA by: inserting the neucleotides GG (total cost 2), deleting the neucleotide C (cost 1), keeping the next three neucleotides (no cost), inserting CA (total cost 2) and keeping the last two neucleotides (no cost).

In order to compute such optimal matchings, the Needleman-Wunsch algorithm is completed with a *backward direction* (the computation of the matrix D being dubbed the *forward direction*):

- we start from position $(|u|, |v|)$ and backtrack through the matrix D by always following the path from where the minimal value came;
- if the path goes diagonal, thus from (i,j) to $(i-1, j-1)$, the letters (u_i, v_j) are matched (here, we denote by u_i the i-th letter of u, and similarly for v);
- if not, we have either (i,j) to $(i-1,j)$, and then a '–' (a gap) is assigned to the letter u_i, or we have (i,j) to $(i,j-1)$, and then a '–' is assigned to the letter v_j.

Optimal matchings are not unique: there may be other matchings having optimal cost. They are called *co-optimal solutions*.

1. Find another matching with cost 5 for the above example.
2. Fill in the matrix below manually starting with the cell $D_{1,1}$ based on the equation given above. What is the distance of the two words *Friday* and *Railway*, that is, the value in the cell $D_{7,6}$?

   The outer row and column are added to indicate the index of the input strings. The initialization is done by adding a row and column with '-' which shows the scores of matching each of the input words against an empty word.

idx		0	1	2	3	4	5	6	7
	inp	-	R	A	I	L	W	A	Y
0	-	0	1	2	3	4	5	6	7
1	F	1							
2	R	2							
3	I	3							
4	D	4							
5	A	5							
6	Y	6							

3. Execute the backtracking step: start from cell $D_{7,6}$ and go back through the matrix by always following the path from where the minimal value came, as described above. What does the matching of the two words look like?
4. Are there co-optimal solutions in this case?

5. Implement the forward direction of the Needleman-Wunsch algorithm, the one filling in the table. The input should be two strings and the output should be the edit distance, that is, the bottom-right value of the matrix.

   Try to run your program with different strings as input or with the same strings but a different weighting scheme (that is, changing $w_{\text{sub}}(a, b)$, w_{ins} and w_{del}).

   The initialization is the following:

   $$D_{0,j} = D_{0,j-1} + 1 \quad \text{for } 1 \leq j \leq m$$

   $$D_{i,0} = D_{i-1,0} + 1 \quad \text{for } 1 \leq i \leq n$$

   In fact, the '+1' in the initialization corresponds to the weight of the deletion and insertion rules. In case those weights are changed, the initialization has to be changed, too.

6. Instead of calculating a distance, one can also calculate a similarity score. What has to be changed in the algorithm in order to calculate similarities?

Exercise 4: matrix chain multiplication

Let A be a $m \times n$ matrix, and let B be a $n \times p$ matrix. Assuming that sum and product of individual elements of the matrices are constant-time operations, the naive algorithm for computing the product AB has complexity $\Theta(mnp)$, as is clear from the following definition:

```
def matProd(A, B):
    # we assume that A and B are non-empty matrices
    m = len(A)
    n = len(A[0])
    if len(B) != n:
        return None  # the size does not match
    p = len(B[0])
    C = [None] * m
    for i in range(m):
        C[i] = [0] * p
        for k in range(p):
            for j in range(n):
                C[i][k] += A[i][j] * B[j][k]
    return C
```

Suppose now that we have three matrices A_1, A_2 and A_3, such that the product $A_1A_2A_3$ is well defined, which means that the number of columns of A_i is equal to the number of rows of A_{i+1}, for all $i \in \{1, 2\}$. If we denote by p_i the number of columns of A_i, for $i \in \{1, 2, 3\}$, and if we denote by p_0 the number of rows of A_1, we have that each A_i is a $p_{i-1} \times p_i$ matrix, with $i \in \{1, 2, 3\}$, and that $A_1A_2A_3$ is a $p_0 \times p_3$ matrix. Our `matProd` function only takes two inputs, so in order to compute $A_1A_2A_3$ we must choose a bracketing: we may either do `matProd(A1,matProd(A2,A3))`, which corresponds to the backeting $A_1(A_2A_3)$, or `matProd(matProd(A1,A2),A3)`, which corresponds to the bracketing $(A_1A_2)A_3$. Since matrix product is associative, both computations will give the same result. However, their complexity is *not* the same:

- the bracketing $A_1(A_2A_3)$ has complexity $\Theta(p_1p_2p_3 + p_0p_1p_3) = \Theta(p_1p_3(p_2 + p_0))$;
- the bracketing $(A_1A_2)A_3$ has complexity $\Theta(p_0p_1p_2 + p_0p_2p_3) = \Theta(p_0p_2(p_1 + p_3))$.

The difference may be substantial: for instance, if A_1 and A_2 are two $n \times n$ matrices and A_3 is just a column vector, we have $p_0 = p_1 = p_2 = n$ and $p_3 = 1$, so the first bracketing has complexity $\Theta(n^2)$, whereas the second bracketing has complexity $\Theta(n^3)$.

The difference is increasingly bigger as the number of matrices to be multiplied increases: for example, if we want to compute $A_1 \cdots A_m$, where each A_i for $1 \leq i \leq m-1$ is an $n \times n$ matrix and A_m is a column vector, the bracketing which starts from the right (that is, we start with $A_{m-1}A_m$) has complexity $\Theta(n^2)$, whereas the bracketing that starts from the left (that is, we start with A_1A_2) has complexity $\Theta(n^m)$.

Therefore, given a sequence of matrices $A_1A_2 \cdots A_m$ that should be multiplied, such that p_0 is the number of rows of A_1 and, for each $i \in \{1, \ldots, m\}$, p_i is the number of columns of A_i (and, therefore, the number of rows of A_{i+1}), it is interesting to find out which bracketing gives the mimimum number of multiplications needed. Notice that, when $n > 3$, more complex bracketings than the "always right" and "always left" are possible, for example $A_1((A_2A_3)A_4)$ in case $m = 4$. These should be considered too. The number of possible bracketings for a sequence of $m > 0$ matrices is equal to the so-called $(m-1)$-th Catalan number, a quantity which grows exponentially in m.[2] Therefore, a brute-force approach is out of the question.

Luckily, we may resort to dynamic programming. Let $M(i, j)$ be the minimal number of multiplications for a product including matrices i to j, with $1 \leq i < j \leq m$. This is the DP matrix of the algorithm (not to be confused with any of the matrices A_i!), which may be filled in as follows:

- the initialization is defined as $M(i, i) := 0$;
- the recursive relation is given by:

$$M(i, j) = min_{i \leq k < j}(M(i, k) + M(k + 1, j) + p_{i-1}p_kp_j).$$

In the sequel, let us assume that we have 4 matrices A, B, C and D with the following dimensions:

- $A : 10 \times 5$,
- $B : 5 \times 3$,
- $C : 3 \times 7$,
- $D : 7 \times 2$.

Please answer the following questions:

1. There are 5 different bracketings for the product $ABCD$. List all of them.
2. Give the number of needed multiplication operations for each of the bracketings.
3. Fill in the upper right triangle of the DP matrix using the recursive relation above. As stated above, the initialization step is given by $M(i, i) := 0$. Cells with a '-' don't need to be filled.

A	B	C	D	
				A
-				B
-	-			C
-	-	-		D

4. What is the minimal number of multiplications needed to calculate the matrix product? The number should be written in the upper right cell of the matrix.
5. Just as in the Needleman-Wunsch algorithm, after computing the DP matrix, we can perform a "backward phase" in which we reconstruct the optimal bracketing. This may be done by following the steps of the algorithm backwards and always taking the direction where the minimal value came from. What is the optimal bracketing for this example?

[2] More precisely, as $\Theta(m^{-\frac{2}{3}}4^m)$.

Exercise 5: coin changing problem

The coin changing problem introduced in the lecture requires finding the smallest number of coins whose values add up to a given sum.

In most real-world cases, a greedy algorithm is enough to find the optimal answer. The greedy strategy is to start at the coin with the highest value, and add it to the solution if the given sum is not yet reached.

1. Write a greedy algorithm for the coin changing problem. The input is a sum to reach and a list of coins. Assume that we have unlimited quantities for each coin. The output should also be a list of coins, for example [1,2] for a sum of 3.
2. What is the output of the greedy algorithm given sum 6 and coins [1,3,4]?
3. What is the optimal solution of the coin changing problem given sum 6 and coins [1,3,4]?
4. As we saw in class, the coin changing problem can also be solved optimally using a dynamic programming strategy. Write the equation that the entries of the DP matrix must satisfy, based on the number of coins and the sum to reach.