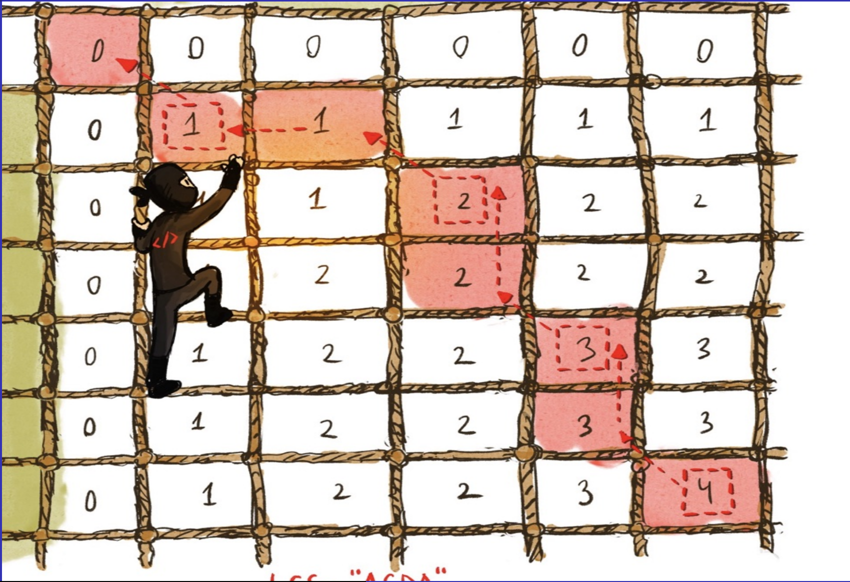
Dynamic Programming II



Right code?

Midterm exam QP 2020-2021

- We have a text coded in binary, i.e., a string with n bits and a code $D : \{a, \dots, z\}^* \to \{0, 1\}^*$.
- D transforms words to Boolean strings.
- We also have a procedure Decode(s) that, given a Boolean string s, determines in time $O(1)$ whether s is the code for some word w or not.
- We want to know if the coded binary string might have been coded with D, and if so a possible initial text.

Right code?

- Let us analyze the recursive structure of a solution.
- A coded (by D) binary string s must be a sequence of coded words, i.e., $s = s_1 \cdot s_2 \cdots s_k$ where s_i is the code of some word w_i, $D(w_i) = s_i$.
- We can think this property as, s decomposes as $s = s_1 \cdot s'$ where s_1 is a coded word and s' is a coded binary string.
- Let us define $T[k]$ to be true iff $s[k..n]$ is a coded binary string, false otherwise. Thus $T[1]$ is the answer we seek.
- From the recursive substructure, we have the recurrence:

$$T[k] = \begin{cases} \text{false} & k > n \\ \text{true} & \text{if Decode}(s[k..n]) \\ \vee_{k \leq j < n}(\text{Decode}(s[k..j]) \wedge T[j+1]) & \text{otherwise} \end{cases}$$

- The total number of subproblemas es $O(n)$, we can use PD.

Right code?

- A traversal from right to left is enough.

```
IsCoded(s)
for k = n to 1 do
  if Decode(s[k..n]) then
    T[k] = 1; P[k]=k
  else
    T[k] = 0; j = k
    while T[k] == 0 and j < n do
      if Decode(s[k..j]) ∧ T[j + 1] then
        T[k] = 1
return T[1]
```

- A call to Decode has cost $O(1)$, to compute $T[k]$ we make $O(n-k)$ calls. Therefore the algorithm has cost $O(n^2)$.

Right code?

- To recover the positions, we keep an additional array P. $P[k]$ keeps a the value j that make $T[k] = 1$, if any, 0 otherwise.

```
IsCoded(s)
for k = n to 1 do
    if Decode(s[i..n]) then
        T[k] = 1
    else
        T[k] = 0; P[k] = 0; j = k
        while T[k] == 0 and j < n do
            if Decode(s[k..j]) ∧ T[j + 1] then
                T[k] = 1; P[k] = j
return  T and P
```

- Starting from $P[1]$ we can recover the positions decomposing s in a sequence of coded words.
- Filling P irequires an additional constant time per entry. So, the overall cost is $O(n^2)$.

Matrioshka

Final exam QT 2018-2019

En Dilworth és el col·leccionista més destacat del món de matrioshkas, les nines russes nidificades, com les de la figura de sota.



En té milers de nines buides de fusta de diferents mides. Per construir un matrioshka la nina més petita es fica dintre de la segona més petita, i aquesta nina, al seu torn, es fica dintre de la la següent i així successivament.

Matrioshka

En Dilworth es pregunta si hi ha una altra manera de nidificar-les perquè acabi amb el màxim possible de nines nidificades.

Per a cada nina tenim mesures de la seva amplada i la seva altura. Una nina amb amplada w_i i altura h_i encaixa en una altra nina d'amplada w_j i alçada h_j si i només si $w_i < w_j$ i $h_i < h_j$.

Donades les mides de les nines proporcioneu un algorisme, tan eficient com pugueu, per construir la matrioshka amb el màxim nombre possible de nines nidificades.

Matrioshka: recursive substructure

An optimal matrioshka, is a doll that contains inside the largest matrioshka that fits!

- To define the subproblems, we need to find a good ordering for the dolls. For example, one that makes that the dolls that fit inside doll i appear after i
- This can be done sorting in decreasing value of h_i breaking ties according to decreasing values of w_i.

Matrioshka: recursive substructure

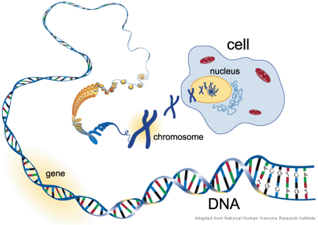
- Assume that the dolls are sorted in decreasing value of h_i breaking ties according to decreasing values of w_i.
- If doll i fits inside doll j, $j < i$. Although, there might be dolls $k > j$ that do not fit inside doll j,
- Let $N[i]$ be the number of dolls in the largest matrioshka having doll i as the visible doll.
- We have
  - $N[i] = 1$, if, for $j > i$, doll j does not fit inside doll i.
  - $N[i] = \max_{\substack{i < j \leq n \\ w_j < w_i}} \{1 + N(j)\}$
- Computing N takes time $O(n^2)$.

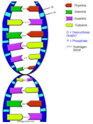
Matrioshka: recursive substructure

- Once we have precomputed N.
- The number of dolls in the largest matrioshka is $M = \max_{1 \leq i \leq n} N[i]$.
- Which can be obtained in additional $O(n)$ time.

- If we wish to construct the largest matrioshka, we have to keep track of the decision taken when computing N and M, with the usual additional pointers.
- Therefore, the problem can be solved in $O(n^2)$ time.

Matching DNA sequences



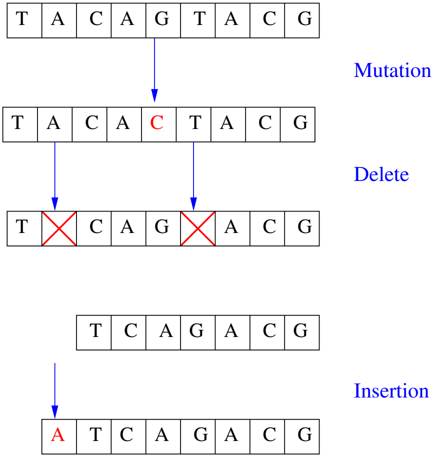


- DNA, is the hereditary material in almost all living organisms. They can reproduce by themselves.
- Its function is like a program unique to each individual organism that rules the working and evolution of the organism.
- Model as a string of 3×10^9 characters over $\{A, T, G, C\}$.

Computational genomics: Some questions

- When a new gene is discovered, one way to gain insight into its working, is to find well known genes (not necessarily in the same species) which match it closely. Biologists suggest a generalization of edit distance as a definition of approximately match.
- GenBank (`https://www.ncbi.nlm.nih.gov/genbank/`) has a collection of $> 10^{10}$ well studied genes, BLAST is a software to do fast searching for similarities between a gene an those in a DB of genes.
- Sequencing DNA: consists in the determination of the order of DNA bases, in a short sequence of 500-700 characters of DNA. To get the global picture of the whole DNA chain, we generate a large amount of DNA sequences and try to assembled them into a coherent DNA sequence. This last part is usually a difficult one, as the position of each sequence is the global DNA chain is not know before hand.

Evolution DNA



How to compare sequences?

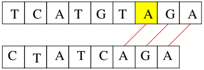
A	C	C	G	G	T	C	G	A	G	T	• • •

?

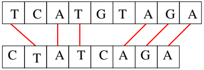
G	T	C	G	T	T	C	G	G	A	A	• • •

Three problems

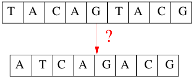
Longest common substring: Substring = consecutive characters in the string.



Longest common subsequence: Subsequence = ordered chain of characters (might have gaps).

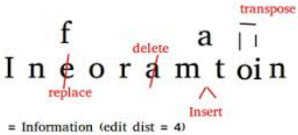


Edit distance: Convert one string into another one using a given set of operations.



The EDIT DISTANCE problem

(Section 6.3 in Dasgupta, Papadimritriou, Vazirani's book.)



The edit distance between strings $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$ is defined to be the minimum number of *edit operations* needed to transform X into Y.

All the operations are done on X

Edit distance: Applications

- Computational genomics: evolution between generations, i.e. between strings on $\{A, T, G, C, -\}$.
- Natural Language Processing: distance, between strings on the alphabet.
- Text processor, suggested corrections

Edit Distance: Levenshtein distance

In the Levenshtein distance the set of operations are

- insert(X, i, a)= $x_1 \cdots x_i a x_{i+1} \cdots x_n$.
- delete(X, i)= $x_1 \cdots x_{i-1} x_{i+1} \cdots x_n$
- modify(X, i, a)= $x_1 \cdots x_{i-1} a x_{i+1} \cdots x_n$.

the cost of modify is 2, and the cost of insert/delete is 1.

To simplify, in the following we assume that *the cost of each operation is 1.*

For other operations and costs the structure of the DP will be similar.

Exemple-1

$X = aabab$ and $Y = babb$
$aabab = X$
$X' = \text{insert}(X, 0, b)$ $baabab$
$X'' = \text{delete}(X', 2)$ $babab$
$X'' = \text{delete}(X'', 4)$ $babb$
$X = aabab \rightarrow Y = babb$

A shortest edit distance

$aabab = X$
$X' = \text{modify}(X, 1, b)$ $babab$
$Y = \text{delete}(X', 4)$ $babb$

Use dynamic programming.

The structure of an optimal solution

- In a solution O with minimum edit distance from $X = x_1 \cdots x_n$ to $Y = y_1 \cdots y_m$, we have three possible alignments for the last terms

(1)	(2)	(3)
x_n	$-$	x_n
$-$	y_m	y_m

- In (1), O performs delete x_n, and it transforms optimally, $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_m$.
- In (2), O performs insert y_m at the end of x, and it transforms optimally, $x_1 \cdots x_n$ into $y_1 \cdots y_{m-1}$.
- In (3), if $x_n \neq y_m$, O performs modify x_n by y_m, otherwise O, aligns them without cost. Furthermore O transforms optimally $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_{m-1}$.

The recurrence

Let $X[i] = x_1 \cdots x_i$, $Y[j] = y_1 \cdots y_j$.
$E[i, j]$ = edit distance from $X[i]$ to $Y[j]$ is the maximum of

- I *put y_j at the end of x*: $E[i, j-1] + 1$
- D *delete x_i*: $E[i-1, j] + 1$
- if $x_i \neq y_j$, M *change x_i into y_j*: $E[i-1, j-1] + 1$, otherwise $E[i-1, j-1]$

Edit distance: Recurrence

Adding the base cases, we have the recurrence

$$E[i,j] = \begin{cases} j & \text{if } i = 0 \text{ (converting } \lambda \to Y[j]) \\ i & \text{if } j = 0 \text{ (converting } X[i] \to \lambda) \\ \min \begin{cases} E[i-1,j]+1 & \text{if D} \\ E[i,j-1]+1, & \text{if I} \\ E[i-1,j-1]+\delta(x_i,y_j) & \text{otherwise} \end{cases} \end{cases}$$

where

$$\delta(x_i,y_j) = \begin{cases} 0 & \text{if } x_i = y_j \\ 1 & \text{otherwise} \end{cases}$$

Computing the optimal costs and pointers

Edit(X, Y)
for $i = 0$ **to** n **do**
  $E[i, 0] = i$
for $j = 0$ **to** m **do**
  $E[0, j] = j$
for $i = 1$ **to** n **do**
  for $j = 1$ **to** m **do**
    $\delta = 0$
    if $x_i \neq y_j$ **then**
      $\delta = 1$
    $E[i,j] = E[i, j-1] + 1$ $b[i,j] = \uparrow$
    if $E[i-1, j-1] + \delta < E[i,j]$ **then**
      $E[i,j] = E[i-1, j-1] + \delta,\ b[i,j] := \nwarrow$
    if $E[i-1, j] + 1 < E[i,j]$ **then**
      $E[i,j] = E[i-1, j] + 1,\ b[i,j] := \leftarrow$

Space and time complexity: $O(nm)$.

$\leftarrow$ is a I operation, $\uparrow$ is a D operation, and $\nwarrow$ is either a M or a no-operation.

Computing the optimal costs: Example

X=aabab; Y=babb. Therefore, $n = 5, m = 4$

		0	1	2	3	4
		λ	b	a	b	b
0	λ	0	$\leftarrow 1$	$\leftarrow 2$	$\leftarrow 3$	$\leftarrow 4$
1	a	$\uparrow 1$	$\nwarrow 1$	$\nwarrow 1$	$\leftarrow 2$	$\leftarrow 3$
2	a	$\uparrow 2$	$\nwarrow 2$	$\nwarrow 1$	$\leftarrow 2$	$\leftarrow 3$
3	b	$\uparrow 3$	$\nwarrow 2$	$\uparrow 2$	$\nwarrow 1$	$\nwarrow 2$
4	a	$\uparrow 4$	$\uparrow 3$	$\nwarrow 2$	$\uparrow 2$	$\nwarrow 2$
5	b	$\uparrow 5$	$\nwarrow 4$	$\uparrow 3$	$\uparrow 2$	$\nwarrow 2$

$\leftarrow$ is a I operation, $\uparrow$ is a D operation, and $\nwarrow$ is either a M or a no-operation.

Obtain Y in edit distance from X

Uses as input the arrays E and b.

The first call to the algorithm is **con-Edit** (n, m)

con-Edit(i, j)
if $i = 0$ or $j = 0$ **then**
  return
  if $b[i, j] = \nwarrow$ and $x_i = y_j$ **then**
    change$(X, i, y_j))$; **con-Edit**$(i-1, j-1)$
  if $b[i, j] = \uparrow$ **then**
    delete(X, i); **con-Edit**$(i-1, j)$
  if $b[i, j] = \leftarrow$ **then**
    insert(X, i, y_j), **con-Edit**$(i, j-1)$

This algorithm has time complexity $O(nm)$.

The Longest Common Subsequence

(Section 15.4 in CormenLRS' book.)

- $Z = z_1 \cdots z_k$ is a subsequence of X if there is a subsequence of integers $1 \leq i_1 < i_2 < \ldots < i_k \leq n$ such that $z_j = x_{i_j}$.

  TTT is a subsequence of $ATATAT$.
- If Z is a subsequence of X and Y, then Z is a common subsequence of X and Y.

LCS Given sequences $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$, compute the longest common subsequence Z.

DP approach: Characterization of optimal solution

Let $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$ and let Z be a longest common subsequence (lcs). Then,

- $Z = x_{i_1} \dots x_{i_k} = y_{j_1} \dots y_{j_k}$
- There are no i, j, with $i > i_k$ and $j > j_k$, s.t. $x_i = y_j$. Otherwise, Z will not be optimal.
- $a = x_{i_k}$ might appear after i_k in X, but not after j_k in Y, or viceversa.
- There is an optimal solution in which i_k and j_k are the last occurrence of a in X and Y respectively.

DP approach: Characterization of optimal solution

Let $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$ and let $Z = x_{i_1} \dots x_{i_k} = y_{j_1} \dots y_{j_k}$ a lcs s.t. the index of the final common symbol in Z is its last occurrence in both X and Y.

Let $X^- = x_1 \cdots x_{n-1}$ and $Y^- = y_1 \cdots y_{m-1}$

- Let us look at x_n and y_m.
- If $x_n = y_m$, $i_k = n$ and $j_k = m$ so, $x_{i_1} \dots x_{i_{k-1}}$ is a lcs of X^- and Y^-.

DP approach: Characterization of optimal solution

Let $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$ and let $Z = x_{i_1} \ldots x_{i_k} = y_{j_1} \ldots y_{j_k}$ a lcs s.t. the index of the final common symbol in Z is its last occurrence in X and Y.

Let $X^- = x_1 \cdots x_{n-1}$ and $Y^- = y_1 \cdots y_{m-1}$

- Let us look at x_n and y_m.
- If $x_n \neq y_m$,
  - If $i_k < n$ and $j_k < m$, Z is a lcs of X^- and Y^-.
  - If $i_k = n$ and $j_k < m$, Z is a lcs of X and Y^-.
  - If $i_k <$ and $j_k = m$, Z is a lcs of X^- and Y.
  - The last two include the first one!

DP approach: Supproblems

Subproblems = lcs of pairs of prefixes of the initial strings.

Notation:

- $X[i] = x_1 \dots x_i$, for $0 \leq i \leq n$
- $Y[j] = y_1 \dots y_j$, for $0 \leq j \leq m$
- $c[i,j]$ = length of the LCS of $X[i]$ and $Y[j]$.
- Want $c[n,m]$ i.e. length of the LCS for X and Y.

DP approach: Recursion

Therefore, given X and Y

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

The recursive algorithm

LCS(X, Y)
$n = X.size()$; $m = Y.size()$
if $n = 0$ or $m = 0$ **then**
  return 0
else if $x_n = y_m$ **then**
  return 1+**LCS**(X^-, Y^-)
else
  return $\max\{$**LCS**(X, Y^-), **LCS**$(X^-, Y)\}$

The algorithm makes 1 or 2 recursive calls and explores a tree of depth $O(n+m)$, therefore the time complexity is $2^{O(n+m)}$.

DP: tabulating

We need to find the correct traversal of the table holding the $c[i,j]$ values.

- Base case is $c[0,j] = 0$, for $0 \leq j \leq m$, and $c[i,0] = 0$, for $0 \leq i \leq n$.
- To compute $c[i,j]$, we have to access

$c[i-1,j-1]$	$c[i-1,j]$
$c[i,j-1]$	$c[i,j]$

A row traversal provides a correct ordering.

- To being able to recover a solution we use a table b, to indicate which one of the three options provided the value $c[i,j]$.

Tabulating

LCS(X, Y)
for $i = 0$ **to** n **do**
  $c[i, 0] = 0$
for $j = 1$ **to** m **do**
  $c[0, j] = 0$
for $i = 1$ **to** n **do**
  for $j = 1$ **to** m **do**
    if $x_i = y_j$ **then**
      $c[i, j] = c[i-1, j-1] + 1,\ b[i.j] = \nwarrow$
    else if $c[i-1, j] \geq c[i, j-1]$ **then**
      $c[i, j] = c[i-1, j],\ b[i, j] = \leftarrow$
    else
      $c[i, j] = c[i, j-1],\ b[i, j] = \uparrow.$

complexity:
$T = O(nm)$.

Example.

X=(ATCTGAT); Y=(TGCATA). Therefore, $m = 6, n = 7$

		0	1 T	2 G	3 C	4 A	5 T	6 A
0		0	0	0	0	0	0	0
1	A	0	$\uparrow 0$	$\uparrow 0$	$\uparrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\nwarrow 1$
2	T	0	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$
3	C	0	$\uparrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$	$\uparrow 2$	$\uparrow 2$
4	T	0	$\nwarrow 1$	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\leftarrow 3$
5	G	0	$\uparrow 1$	$\nwarrow 2$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$	$\uparrow 3$
6	A	0	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\uparrow 3$	$\nwarrow 4$
7	T	0	$\nwarrow 1$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$	$\nwarrow 4$	$\uparrow 4$

Following the arrows: TCTA

Construct the solution

Access the tables c and d.

The first call to the algorithm is **sol-LCS**(n, m)

sol-LCS(i, j)
if $i = 0$ or $j = 0$ **then**
  STOP.
else if $b[i, j] = \nwarrow$ **then**
  sol-LCS$(i - 1, j - 1)$
  return x_i
else if $b[i, j] = \uparrow$ **then**
  sol-LCS$(i - 1, j)$
else
  sol-LCS$(i, j - 1)$

The algorithm has time complexity $O(n + m)$.

Longest common substring

- A slightly different problem with a similar solution
- LCSt: Given two strings $X = x_1 \dots x_n$ and $Y = y_1 \dots y_m$, compute their longest common substring Z, i.e., the largest k for which there are indices i and j with $x_i x_{i+1} \dots x_{i+k} = y_j y_{j+1} \dots y_{j+k}$.
- For example:
  X : DEADBEEF
  Y : EATBEEF
  Z :

Longest common substring

- A slightly different problem with a similar solution
- LCSt Given two strings $X = x_1 \dots x_n$ and $Y = y_1 \dots y_m$, compute their longest common substring Z, i.e., corresponding to the largest k for which there are indices i and j with $x_i x_{i+1} \dots x_{i+k} = y_j y_{j+1} \dots y_{j+k}$.
- For example:
  X : DEADBBEEF
  Y : EATBEEF
  Z : BEEF pick the longest substring

Characterization of optimal solution

- Let $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$ and let Z be a longest common substring.
  - $Z = x_i \dots x_{i+k} = y_j \dots y_{j+k}$
  - Z is the longest common suffix of $X(i+k)$ and $Y(j+k)$.
- We can consider the subproblems $LCStf(i,j)$: compute the longest common suffix of $X(i)$ and $Y(j)$.
- The $LCSf(X,Y)$ is the longest of such common suffixes.

Computing the LC Suffixes

- To solve $LCSf(i, j)$ it is enough to go backward from position i in X and j in Y until we find two different characters.
- This has cost $O(n + m)$ per subproblem.
- We get a $O(nm(n + m))$ algorithm for LCSt
- Can we do it faster? Let us use DP!

A recursive solution for LC Suffixes

Notation:

- $X[i] = x_1 \dots x_i$, for $0 \leq i \leq n$
- $Y[j] = y_1 \dots y_j$, for $0 \leq j \leq m$
- $s[i,j]$ = the length of the LC Suffix of $X[i]$ and $Y[j]$.
- Want $\max_{i,j} s[i,j]$ i.e., the length of the LCSt of X, Y.

DP approach: Recursion

Therefore, given X and Y

$$s[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 0 & \text{if } x_i \neq y_j \\ s[i-1, j-1] + 1 & \text{if } x_i = y_j \end{cases}$$

Using the recurrence the cost per recursive call (or per element in the table) is constant

Tabulating

LCSf(X, Y)
for $i = 0$ **to** n **do**
  $s[i, 0] = 0$
for $j = 1$ **to** m **do**
  $s[0, j] = 0$
for $i = 1$ **to** n **do**
  for $j = 1$ **to** m **do**
    s[i,j]=0
    if $x_i = y_j$ **then**
      $s[i, j] = s[i-1, j-1] + 1$

complexity: $O(nm)$.

Which gives an algorithm with cost $O(nm)$ for LCSt

Multiplying a Sequence of Matrices

(This example is from Section 15.2 in CormenLRS' book.)

MULTIPLICATION OF n MATRICES Given as input a sequence of n matrices $(A_1 \times A_2 \times \cdots \times A_n)$. Minimize the number of operation in the computation $A_1 \times A_2 \times \cdots \times A_n$

Recall that Given matrices A_1, A_2 with $\dim(A_1) = p_0 \times p_1$ and $\dim(A_2) = p_1 \times p_2$, the basic algorithm to $A_1 \times A_2$ takes time at most $p_0 p_1 p_2$.

Example:

$$\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix} = \begin{bmatrix} 13 & 18 & 23 \\ 18 & 25 & 32 \\ 23 & 32 & 41 \end{bmatrix}$$

Multiplying a Sequence of Matrices

- Matrix multiplication is NOT commutative, so we can not permute the order of the matrices without changing the result.
- It is associative, so we can put parenthesis as we wish.
- How to multiply is equivalent to the problem of how to parenthesize.
- We want to find the way to put parenthesis so that the product requires the minimum total number of operations. And use it to compute the product.

Example Consider $A_1 \times A_2 \times A_3$, where dim $(A_1) = 10 \times 100$ dim $(A_2) = 100 \times 5$ and dim $(A_3) = 5 \times 50$.

- $((A_1 A_2) A_3)$ takes $(10 \times 100 \times 5) + (10 \times 5 \times 50) =$ *7500* operations,
- $(A_1 (A_2 A_3))$ takes $(100 \times 5 \times 50) + (10 \times 100 \times 50) =$ 75000 operations.

The order in which we make the computation of products of two matrices makes a big difference in the total computation's time.

How to parenthesize $(A_1 \times \ldots \times A_n)$?

- If $n = 1$ we do not need parenthesis.
- Otherwise, decide where to break the sequence $((A_1 \times \cdots \times A_k)(A_{k+1} \times \cdots \times A_n))$ for some k, $1 \leq k < n$.
- Then, combine any way to parenthesize $(A_1 \times \cdots \times A_k)$ with any way to parenthesize $(A_{k+1} \times \cdots \times A_n)$.

Using this structure, we can count the number of ways to parenthesize $(A_1 \times \cdots \times A_n)$ as well as to define a backtracking algorithm that goes over all those ways to parenthesize and eventually to a brute force recursive algorithm to solve the problem of computing efficiently the product.

How many ways to parenthesize $(A_1 \times \cdots \times A_n)$?

Let $P(n)$ be the number of ways to paranthesize $(A_1 \times \cdots \times A_n)$. Then,

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{si } n \geq 2 \end{cases}$$

with solution $P(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n/n^{3/2})$

The Catalan numbers.

Brute force will take too long!

Structure of an optimal solution

- We want to compute $(A_1 \times \cdots \times A_n)$ efficiently.
- In an optimal solution the last matrix product must correspond to a break at some position k, $((A_1 \times \cdots \times A_k)(A_{k+1} \times \cdots \times A_n))$ Let $A_{i-j} = (A_i A_{i+1} \cdots A_j)$.
- The parenthesization of the subchains $(A_1 \times \cdots \times A_k)$ and $(A_{k+1} \times \cdots \times A_n)$ within the optimal parenthesization must be an optimal paranthesization of $(A_1 \times \cdots \times A_k)$, $(A_{k+1} \times \cdots \times A_n)$. So,

$$\begin{aligned}\text{cost}(A_1 \ldots A_n) = &\text{cost}(A_1 \ldots A_k) \\ &+ \text{cost}(A_{k+1} \ldots A_n) + p_0 p_k p_n.\end{aligned}$$

Structure of an optimal solution

- An optimal solution decomposes in optimal solutions of the same problem on subchains.
- Subproblems: compute the product $A_i \times A_{i+1} \times \cdots \times A_j$, for $1 \leq i \leq j \leq n$
- Let us call $B_i^j = A_i \times A_{i+1} \times \cdots \times A_j$.

Cost Recurrence

- Let $m[i,j]$ be the minimum cost of computing $B_i^j = (A_i \times \ldots \times A_j)$, for $1 \leq i \leq j \leq n$.
- $m[i,j]$ is defined by the value k, $i \leq k \leq j$ that minimizes

$$m[i,k] + m[k+1,j] + \text{ cost } (B_i^k, B_{k+1}^j).$$

- That is,

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{otherwise} \end{cases}$$

Computing the cost of an optimal solution: Rec

Assume that vector P holds the values $(p_0, p_1, \dots, p_n)$.

MCR(i, j)
if $i = j$ **then**
  return 0
$m[i, j] = \infty$
for $k = i$ **to** $j - 1$ **do**
  $q = \text{MCR}(i, k) + \text{MCR}(k + 1, j) + P[i - 1] * P[k] * P[j]$
  if $q < m[i, j]$ **then**
    $m[i, j] = q$
return $(m[i, j])$

Cost: $T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n \sim \Omega(2^n)$.

Can we apply dynamic programming?

- We have an optimal recursive algorithm which takes exponential time.
- Subproblems?
  The subproblems are identified by the two inputs in the recursive call, the pair (i, j).
- How many subproblems?
  As $1 \leq i < j \leq n$, we have only $O(n^2)$ subproblems.
- We can use DP!

Dynamic programming: Memoization

MCP(P)
for all $1 \leq i < j \leq n$ **do**
  $m[i,j] = -1$
for $i = 1$ **to** n **do**
  $m[i,i] = 0$
MCR$(1,n)$
return $(m[1,n])$

MCR(i,j)
if $m[i,j]! = -1$ **then**
  return $(m[i,j])$
$m[i,j] = \infty$
for $k = i$ **to** $j-1$ **do**
  $q = \text{MCR}(i,k) + \text{MCR}(k+1,j) + P[i-1] * P[k] * P[j]$
  if $q < m[i,j]$ **then**
    $m[i,j] = q$
return $(m[i,j])$

$T(n) = \Theta(n^3)$ additional space $\Theta(n^2)$.

Dynamic programming: Tabulating

To compute the element $m[i,j]$ the base case is when $i = j$, we need to access $m[i,k]$ and $m[k+1,j]$. We can achieve that by filling the (half) table by diagonals.

MCP(P)
for $i = 1$ **to** n **do**
  $m[i,i] = 0$
for $d = 2$ **to** n **do**
  for $i = 1$ **to** $n - d + 1$ **do**
    $j = i + d - 1$
    $m[i,j] = \infty$
    for $k = i$ **to** $j - 1$ **do**
      $q =$
      $m[i,k] + m[k+1,j] + P[i-1] * P[k] * P[j]$
      if $q < m[i,j]$ **then**
        $m[i,j] = q$
return $(m[1,n])$

$T(n) = \Theta(n^3)$,
space $= \Theta(n^2)$.

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1				
2				
3				
4				

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1	0			
2		0		
3			0	
4				0

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1	0	45		
2		0	30	
3			0	24
4				0

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1	0	45	60	
2		0	30	70
3			0	24
4				0

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1	0	45	60	84
2		0	30	70
3			0	24
4				0

Recording more information about the optimal solution

We have been working with the recurrence

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{otherwise} \end{cases}$$

To keep information about the optimal solution the algorithm keep additional information about the value of k that provides the optimal cost as

$$s[i,j] = \begin{cases} i & \text{if } i = j \\ \arg\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{otherwise} \end{cases}$$

Dynamic programming: Memoization

MCP(P)
for all $1 \leq i < j \leq n$ **do**
  $m[i,j] = -1$
for $i = 1$ **to** n **do**
  $m[i,i] = 0$; $s[i,i] = i$;
MCR$(1,n)$
return m, s

MCR(i,j)
if $m[i,j]! = -1$ **then**
  return $(m[i,j])$
$m[i,j] = \infty$
for $k = i$ **to** $j-1$ **do**
  $q = \text{MCR}(i,k) + \text{MCR}(k+1,j) + P[i-1] * P[k] * P[j]$
  if $q < m[i,j]$ **then**
    $m[i,j] = q$; $s[i,j] = k$;
return $(m[i,j])$

Dynamic programming: Tabulating

MCP(P)
for $i = 1$ **to** n **do**
  $m[i,i] = 0$; $s[i,i] = 0$;
for $d = 2$ **to** n **do**
  for $i = 1$ **to** $n - d + 1$ **do**
    $j = i + d - 1$
    $m[i,j] = \infty$
    for $k = i$ **to** $j - 1$ **do**
      $q =$
      $m[i,k] + m[k+1,j] + P[i-1] * P[k] * P[j]$
      if $q < m[i,j]$ **then**
        $m[i,j] = q$; $s[i,j] = k$;
return m, s.

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P = (3, 5, 3, 2, 4)$

$i \setminus j$	1	2	3	4
1				
2				
3				
4				

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P = (3, 5, 3, 2, 4)$

$i \setminus j$	1	2	3	4
1	0 1			
2		0 2		
3			0 3	
4				0 4

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P = (3, 5, 3, 2, 4)$

$i \setminus j$	1	2	3	4
1	0 1	45 1		
2		0 2	30 2	
3			0 3	24 3
4				0 4

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P = (3, 5, 3, 2, 4)$

$i \setminus j$	1	2	3	4
1	0 1	45 1	60 1	
2		0 2	30 2	70 3
3			0 3	24 3
4				0 4

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P = (3, 5, 3, 2, 4)$

$i \setminus j$	1	2	3	4
1	0 1	45 1	60 1	84 3
2		0 2	30 2	70 3
3			0 3	24 3
4				0 4

Computing optimally the product

- $s[i,j]$ contains the value of k that decomposes optimally the product as product of two submatrices, i.e.,

$$A_i \times \cdots \times A_j = (A_i \times \cdots \times A_{s[i,j]})(A_{s[i,j]+1} \times \cdots \times A_j).$$

- Therefore,

$$A_1 \times \cdots \times A_n = (A_1 \times \cdots \times A_{s[1,n]})(A_{s[1,n]+1} \times \cdots \times A_n).$$

- We can design a recursive algorithm to perform the product in an optimal way.

The product algorithm

The input is the sequence of matrices $A = A_1, \dots, A_n$ and the table s computed before.

Product(A, s, i, j)
if $i = j$ **then**
  return (A_i)
$X =$**Product**$(A, s, i, s[i, j])$
$Y =$**Product**$(A, s, s[i, j] + 1, j)$
return $(X \times Y)$

The total number operations required to compute the product is $m[1, n]$ and the cost of the complete algorithm is $T(n) = O(n^3 + m[1, n])$

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P = (3, 5, 3, 2, 4)$

$i \setminus j$	1	2	3	4
1	0 1	45 1	60 1	84 3
2		0 2	30 2	70 3
3			0 3	24 3
4				0 4

The optimal way to minimize the number of operations is

$$(((A_1) \times (A_2 \times A_3)) \times (A_4))$$

Multiplying matrices

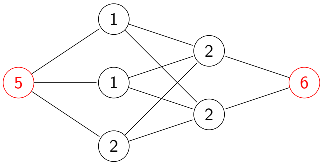
- In order to compute s, we only need the dimensions of the matrices.
- What if we use Strassen algorithm to compute a two matrices product instead of the naive algorithm?

Dynamic Programming in Trees

- Trees are nice graphs easily adapted to recursion.
- Once you root the tree each node can be seen as the root of a subtree .
- We can use Dynamic Programming to give polynomial solutions to "difficult" graph problems when the input is restricted to be a tree, or to have a treee-like structure (small treewidth).
- In this case instead of having a global table, each node in the tree keeps additional information about the associated subproblem.

The Maximum Weight Independent Set (MWIS)

Given as input $G = (V, E)$, together with a weight $w : V \to \mathbb{R}$. Find the heaviest $S \subseteq V$ such that no two vertices in S are connected in G.

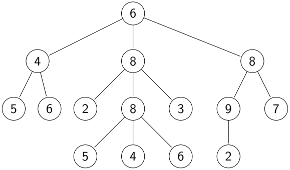


For general graphs, the problem is hard, even for the case in which all vertex have weight 1, i.e. Maximum Independent Set is NP-complete.

Maximum Weight Independent Set on Trees

Given a tree $T = (V, E)$ choose a $r \in V$ and root it from r

i.e. Given a rooted tree $T = (V, E, r)$ and weights $w : V \to \mathbb{R}$, find the independent set with maximum weight.



Notation:

- For $v \in V$, let T_v be the subtree rooted at v. $T = T_r$.
- Given $v \in V$ let $C(v)$ be the set of children of v, and $G(v)$ be the set of grandchildren of v.

Characterization of the optimal solution

Key observation: An IS can't contain vertices which are father-son.

Let S be an optimal solution.

- If $r \in S$: then $C(r) \not\subseteq S_r$. So $S - \{r\}$ contains an optimum solution for each T_v, with $v \in G(r)$.
- If $r \notin S$: S contains an optimum solution for each T_u, with $u \in C(r)$.

Recursive definition of the optimal solution

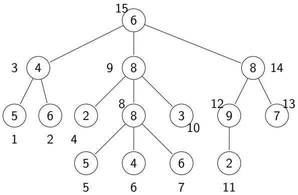
- To implement DP, tor every node v, we add one value, $v.M$: the value of the optimal solution for T_v
  Following the recursive structure of the solution we have the following recurrence

$$v.M = \begin{cases} w(v) & v \text{ a leaf,} \\ \max\{\sum_{u\in C(v)} u.M, w(v) + \sum_{u\in G(v)} u.M\} & \text{otherwise.} \end{cases}$$

- Notice that for any $v \in T$: we have to compute $\sum_{u\in C(v)} u.M$ and for this we must access to the children of its children
- To avoid this we add another value to the node $v.M'$: the sum of the values of the optimal solutions of their children, i.e., $\sum_{u\in C(v)} u.M$.

Post-order traversal of a rooted tree

To perform the computation, we can follow a DFS, post-order, traversal of the nodes in the tree, computing the additional values at each node.



DP Algorithm to compute the optimal weight

Let $v_1, \ldots, v_n = r$ be the post-order traversal of T_r

WIS T_r
Let $v_1, \ldots, v_n = r$ the post-order traversal of T_r
for $i = 1$ **to** n **do**
  if v_i is a leaf **then**
    $v_i.M = w[v_i], v_i.M' = 0$
  else
    $v_i.M' = \sum_{u \in C(v)} u.M$
    $aux = \sum_{u \in C(v)} u.M'$
    $v_i.M = \max\{aux + w[v_i], v_i.M'\}$
return r.M

Complexity: space $= O(n)$, time $= O(n)$

Top-down traversal to obtain an optimal IS

RWIS(v)
if v is a leaf **then**
  return $(\{v\})$
if $v_i.M = v_i.M' + w[v_i]$ **then**
  $S = S \cup \{v_i\}$
  for $w \in G(v)$ **do**
    $S = S \cup$ **RWIS**(w)
else
  for $w \in N(v)$ **do**
    $S = S \cup$ **RWIS**(w)
return S

RWIS(r)

provides an optimal solution in time $O(n)$

Total cost $O(n)$ and additional space $O(n)$