To begin, let us assume that all we really want to know is the minimum cost, or minimum number of arithmetic operations, needed to multiply out the matrices. In general we can use recursive algorithm to find the minimum cost:

* Take the sequence of the matrices and separate it into two subsequences.
* Find the minimum cost of multiplying out each subsequence.
* Add these cost together, and add in the cost of multiplying the two result matrices.
* DO this for each possible position at which the sequence of matrices can be split, and take the minimum over all of them.

For example, if we have 4 matrices ABCD, we compute the cost required to find each of (A)(BCD), (AB)(CD), and (ABC)(D),making recursive calls to find the minimum cost to compute ABC, AB, CD, and BCD. We then choose the best one. Better still, this yields not only the minimum cost, but also demonstrates the best way of doing the multiplication: group it the way that yields the lowest total cost, and do the same for each factor.

However if we implement this algorithm we discover that it is just as slow as the naïve way of trying all permutations! What went wrong? The answer is is that we’re doing a lot of redundant work. For example, above we made a recursive call to find the best cost for computing both ABC and AB. But finding the best cost for computing ABC also requires finding the best cost for AB. As the recursion grows deeper, more and more of this type of unnecessary repetition occurs.

One simple solution is called memorization: each time we compute the minimum cost needed to multiply out a specific subsequence, we save it. If we are ever asked to compute it again, we simply give the saved answer, and do not recompute it. Since there are about n^2/2 different subsequences, where n is the number of matrices, the space required to do this is reasonable. **It can be shown that this simple trick brings the runtime down to O(n^3) from O(2^n), which is more efficient enough for real applications.**

classdef CalculateRecursionDp < handle

properties

d = []

s = []

p = []

alphabet = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'}

j = 0

res

q = 0

ans = []

sz = 0

len = 0

Min = 0

str = ''

end

methods

function str = get (this, l, r)

% fprintf ('%d %d\n', l, r);

if l == r

str = this.alphabet (l); %if the sequence consist of only one element returning letter at position l or r in our alphabet array

return;

end

k = this.s(l, r); %splitting our sequence (l, r) into two subsequences by already found position in array p

A = this.get (l, k - 1); %recursively getting answer for the first sequence

B = this.get (k, r); %recursively getting anwer for the 2nd sequence

%concateneting 2 subsequences

word = strcat ('(', A);

word = strcat (word, B);

word = strcat (word, ')');

str = word;%returning answer

end

function out = dynamic(this)

%for i = 1:this.sz

% disp (p(i))

%end

for i = 1:this.sz

this.d(i, i) = 0; %basic steps for our dynamic array

end

for L = 2:this.sz %going through all the sizes

for i = 1:this.sz-L+1 % current sequence start from i

this.j = i + L - 1;% ends at this.j of length L

this.d(i, this.j) = 111111; %filling current state of dynamic (i, j) with infinity

for k = i:this.j - 1 %going through all posible splittings from (i, j)

cost = this.d(i, k) + this.d(k + 1, this.j) + this.p(i)\*this.p(k + 1)\*this.p(this.j + 1); %if we split at k what we will get is stored at cost

if (cost < this.d(i, this.j)) %if current cost is less than our current minimum of (i, j)

this.d(i, this.j) = cost;%then we update it with new values

this.s(i, this.j) = k + 1;

end

end

end

end

out = this.d;%returning dynamic array

end

function out = rec (this, l, r)

% fprintf ('%d %d\n', l, r)

if r - l > 0

Min = 1111111; %making Min to be infinity at the beginnning

for k = l : r - 1 %choosing a position where we can divide our rec

cost = this.rec (l, k) + this.rec (k + 1, r) + this.p(l) \* this.p (k + 1) \* this.p (r + 1); %trying to splt at position k for current sequence(l, r)

%and storing result in cost

if cost < Min %if cost less than our current minimum result for sequence (l, r)

Min = cost; %then we update our minimum result

end

end

out = Min;%storing our minimum to return it

return;

end

if l == r %if our sequence is of only one element then cost of this is zero

out = 0;

return;

end

end

function this = printFunc(this)

dynamic\_array = this.dynamic; %our dynamic array

slovo = this.get (1, this.sz); %this is our sequence which is same for both recursive and dynamic

slovo = strcat (slovo, ',');

firstanswer = strcat (strcat ('Recursion based sequence, mult. count : ', slovo), num2str(this.rec(1,this.sz))); %our first answer for recursive based sequence

secondanswer = strcat (strcat ('Dynamic Programming based sequence, mult. count : ', slovo), num2str(this.d(1, this.sz)));%our 2nd answer for dynamic based sequence

disp (firstanswer);

disp (secondanswer);

disp ('Dynamic Programming Memoization Table : ');

disp (dynamic\_array);% our dynamic table

end

function this = CalculateRecursionDp (arg) %passing arguments

this.p = arg;%storing arguments in array p

this.sz = length (arg) - 1;%this.sz is length of this array

this.printFunc;%this function prints our result

end

end

end

**The output capture:**

