Dynamic Programming (15.1 - 15.5

Ex.

Consider the problem of computing the Binomial coefficient $\binom{n}{k}$ using Pascal's identity

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

The obvious recursive algorithm is

BinCoef(n, k)
1.) if k = 0 or n
2.) return 1
3.) else
4.) return BinCoef(n-1, k-1) + BinCoef(n-1, k)

Observe that at the bottom level BinCoef always returns 1, so ultimately it just adds a lot of 1's. Thus BinCoef runs in time

$$\Omega\left(\binom{n}{k}\right).$$

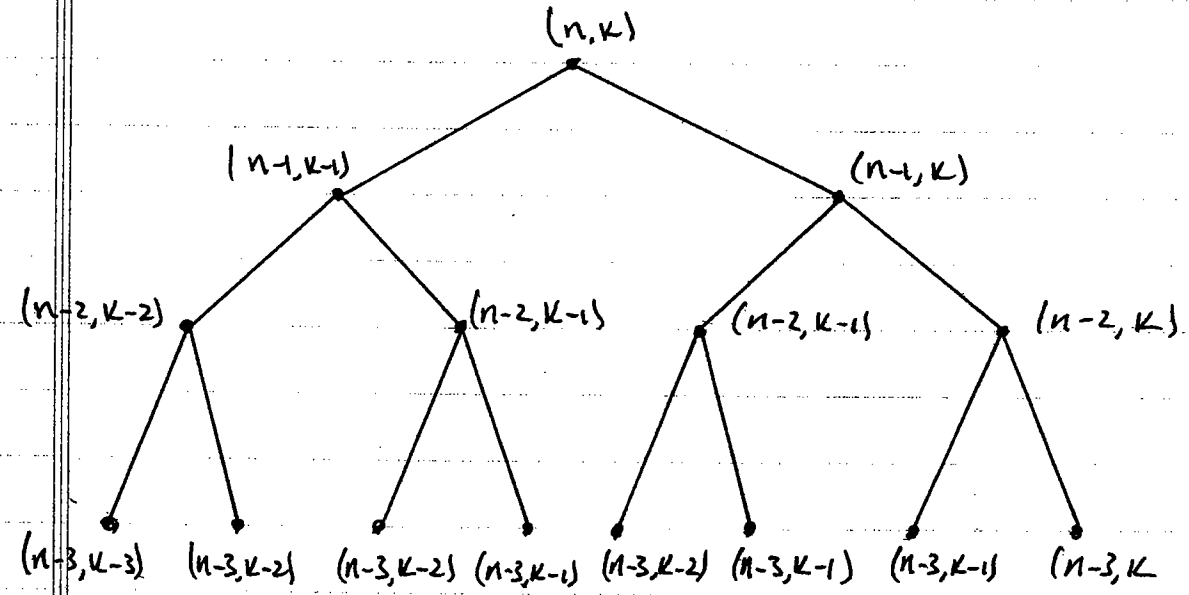
EXERCISE

SHOW THAT IF $n = 2k$, THEN

$$\binom{n}{k} = \Theta\left(\frac{2^n}{\sqrt{n}}\right) = \Theta\left(\frac{4^k}{\sqrt{k}}\right)$$

HINT: USE STIRLING'S FORMULA.

THE RECURSION TREE SHOWS THE PROBLEM



OBSERVE THAT MANY OF THE SAME VALUES ARE COMPUTED MULTIPLE TIMES.

A MORE EFFICIENT APPROACH IS TO MAINTAIN A TABLE OF INTERMEDIATE RESULTS.

	0	1	2	3	...	K-1	K
0	1	0	0	0	..	0	0
1	1	1	0	0	..	0	0
2	1	2	1	0	..	0	0
3	1	3	3	1	..	0	0
⋮						⋮	⋮
n-1	.	.	.		.	$\binom{n-1}{k-1}$	$\binom{n-1}{k}$
n	.	.	.		.		$\binom{n}{k}$

IN FACT IT'S NOT NECESSARY TO STORE THE WHOLE TABLE, JUST A SINGLE ROW AT A TIME.

DynBinCoef(n, k)

1.) C[0] ← 1
2.) for i ← 1 TO K
3.) C[i] ← 0
4.) for i ← 1 TO n
5.) for i ← K TO 1
6.) C[i] ← C[i-1] + C[i]
7.) return C[K].

NOTE: THERE IS STILL SOME EFFICIENCY TO BE GAINED BY DELETING ALL ZEROS, NOT CALCULATING ALL OF BOTTOM ROW, ETC. THIS IS LEFT AS AN EXERCISE.

THE SAME PROBLEM ARISES FREQUENTLY IN THE DIVIDE & CONQUER APPROACH. THE OBVIOUS AND NATURAL WAY OF DIVIDING A PROBLEM INTO SUBINSTANCES LEADS TO OVERLAPING (OR IDENTICAL) SUBINSTANCES WHICH ARE SOLVED MULTIPLE TIMES, LEADING TO AN INEFFICIENT ALGORITHM.

IN DYNAMIC PROGRAMING WE ARRANGE TO SOLVE EACH SUBINSTANCE ONLY ONCE, SAVING THE RESULT FOR LATER USE.

WE TYPICALLY STORE THIS INFORMATION AS A TABLE (e.g. 2-dim ARRAY) OF KNOWN RESULTS.

THE WORD "PROGRAMING" IS USED HERE IN AN ARCHAIC SENSE. "PROGRAM" IS SYNONYMOUS WITH "TABLE", AS IN LINEAR PROGRAMMING.

DYNAMIC PROGRAMING IS A TECHNIQUE FOR SOLVING OPTIMIZATION PROBLEMS. WE SEEK AN OPTIMAL SOLUTION FROM AMONG A SET OF FEASIBLE SOLUTIONS.

Coin Changing Problem

Suppose we have coins in n denominations $\{d_1, d_2, \ldots, d_n\}$, where each $d_i \geq 1$ is an integer. We wish to pay an amount N using the fewest number of coins possible.

Assumption: There is an unlimited supply of coins in each denomination.

There are two questions:

- What is the least number of coins needed to pay N units.
- Exactly which coins (i.e. which denominations) are to be disbursed.

To answer the first question we create a table $C[1 \ldots n; 0 \ldots N]$, where

$C[i,j]$ = minimum number of coins necessary to pay amount j using coins in denominations $\{d_1, \ldots, d_i\}$, $1 \leq i \leq n$, $0 \leq j \leq N$.

Thus we seek $C[n, N]$.

FIRST OBSERVE THAT $C[i,0] = 0$ FOR ALL $1 \le i \le n$.

NEXT, NOTICE THAT TO PAY THE AMOUNT j USING DENOMINATIONS $\{d_1, .., d_i\}$ WE HAVE IN GENERAL TWO CHOICES

(1) USE NO COINS VALUES d_i (EVEN THOUGH THIS IS PERMITTED). WE CAN DO THIS USING $C[i-1, j]$ COINS.

(2) USE AT LEAST 1 COIN OF VALUE d_i. AFTER HANDING OVER ONE SUCH COIN, THERE ARE $j - d_i$ UNITS LEFT TO PAY FROM DENOMINATIONS $\{d_1, .., d_i\}$. WE CAN DO THIS USING $1 + C[i, j-d_i]$ COINS.

$C[i,j]$ SHOULD BE WHICHEVER ALTERNATIVE USES THE FEWEST COINS. THUS

$$C[i,j] = \min\left(C[i-1, j],\ 1 + C[i, j-d_i]\right)$$

NOTE IF $i = 1$ OR $j < d_i$ THEN ONE OF THE VALUES ON RIGHT FALLS OUTSIDE THE TABLE.

IT IS CONVENIENT TO THINK OF SUCH VALUES AS BEING $+\infty$.

IF BOTH $i = 1$ AND $j < d_i$ THEN WE SET

$$C[i,j] = +\infty$$

INDICATING THAT IT IS IMPOSSIBLE TO PAY AMOUNT j USING ONLY COINS OF TYPE 1.

EX. $n = 4$, $N = 8$

	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 3$	0	1	2	1	2	3	2	3	4
$d_3 = 5$	0	1	2	1	2	1	2	3	2
$d_4 = 6$	0	1	2	1	2	1	1	2	2

NOTE THAT IF WE HAVE AN UNLIMITED SUPPLY OF COINS OF VALUE 1 (e.g. $d_1 = 1$) THEN IT IS POSSIBLE TO DISBURSE ANY AMOUNT. OTHERWISE IT MAY BE IMPOSSIBLE TO PAY CERTAIN AMOUNTS. WE INDICATE THIS WITH $C[i,j] = \infty$.

EX. $n = 3$, $N = 8$

	0	1	2	3	4	5	6	7	8
$d_1 = 2$	0	∞	1	∞	2	∞	3	∞	4
$d_2 = 4$	0	∞	1	∞	1	∞	2	∞	2
$d_3 = 5$	0	∞	1	∞	1	1	2	2	2

THE FOLLOWING ALGORITHM TAKES INPUTS $d[1..n]$, N, AND USES A LOCAL ARRAY $C[1..n; 0..N]$

```
CoinChange(d, N)
1.) n ← length[d]
2.) for i ← 1 TO n
3.)     C[i,0] ← 0
4.) for i ← 1 TO n
5.)     for j ← 1 TO N
6.)         if i=1 AND j<d[1]
7.)             C[1,j] ← ∞
8.)         else if i=1
9.)             C[1,j] ← 1+C[1, j-d[j]]
10.)        else if j<d[i]
11.)            C[i,j] ← C[i-1, j]
12.)        else
13.)            C[i,j] ← min(C[i-1,j], 1+C[i, j-d[i]])
14.) return C[n,N].
```

THE ALGORITHM CAN BE EASILY ALTERED TO RETURN THE ENTIRE TABLE $C[1..n, 0..N]$ OF INTERMEDIATE RESULTS.

THE RUN TIME IS OBVIOUSLY $\Theta(nN)$ SINCE EACH OF $n \cdot (N+1)$ TABLE ENTRIES MUST BE FILLED.

EXERCISE →

MODIFY THIS ALGORITHM TO DEAL WITH THE SITUATION IN WHICH THE SUPPLY OF COINS IN SOME DENOMINATIONS IS LIMITED. LET $l[1..n]$ BE ANOTHER INPUT ARRAY, AND REQUIRE AT MOST $l[i]$ COINS OF TYPE i BE USED. NOTE $0 \le l[i] \le \infty$ FOR $1 \le i \le n$.

ONCE THE TABLE $C[1..n; 0..N]$ HAS BEEN FILLED THE SECOND PROBLEM CAN BE SOLVED, i.e. EXACTLY WHICH COINS ARE TO BE DISBURSED.

IF $C[i,j] = C[i-1,j]$, THEN NO COINS OF TYPE i ARE NEEDED TO PAY j UNITS WHEN RESTRICTED TO TYPES $\{1,..,i\}$. WE MOVE UP ONE ROW TO $C[i-1,j]$ TO SEE WHAT TO DO NEXT.

IF $C[i,j] = 1 + C[i, j-d_i]$, WE PAY OUT ONE COIN OF TYPE i THEN MOVE LEFT TO $C[i, j-d_i]$ TO SEE WHAT TO DO NEXT

IF $C[i,j]$ EQUALS BOTH $C[i-1,j]$ AND $1 + C[i, j-d_i]$ THEN EITHER ACTION IS ACCEPTABLE.

EXERCISE:

WRITE A RECURSIVE ALGORITHM WHICH GIVEN THE FILLED TABLE $C[1..n, 0..M]$, PRINTS A SEQUENCE OF $C[n,M]$ COIN TYPES WHOSE VALUE ADDS TO N. IF $C[n,M] = \infty$ PRINT AN APPROPRIATE MESSAGE.

THE 0-1 KNAPSACK PROBLEM

A THIEF WISHES TO STEAL n OBJECTS INDEXED $i = 1$ TO n. LET

v_i = VALUE OF OBJECT i
w_i = WEIGHT OF OBJECT i

THE THIEF HAS A KNAPSACK WHICH CAN CARRY A MAXIMUM WEIGHT OF W. HIS GOAL IS TO FILL THE KNAPSACK IN A WAY WHICH MAXIMIZES THE TOTAL VALUE OF THE GOODS STOLEN, WHILE RESPECTING ITS CAPACITY CONSTRAINT.

LET

$$x_i = \begin{cases} 0 & \text{IF OBJECT } i \text{ IS NOT TAKEN} \\ 1 & \text{IF OBJECT } i \text{ IS TAKEN} \end{cases}$$

THUS THE PROBLEM IS TO CHOOSE $x_i \in \{0,1\}$ $(1 \le i \le n)$ TO

$$\text{MAXIMIZE} \quad \sum_{i=1}^{n} x_i v_i$$

$$\text{SUBJECT TO} \quad \sum_{i=1}^{n} x_i w_i \le W$$

WHERE $v_i > 0$, $w_i > 0$, $W > 0$.

TO SOLVE THIS PROBLEM WE CREATE A TABLE $V[1..n; 0..W]$ WHERE $V[i,j]$ IS THE MAXIMUM VALUE OF THE OBJECTS IN THE SET $\{1,..,i\}$ WHOSE TOTAL WEIGHT DOES NOT EXCEED j. $(1 \le i \le n, 0 \le j \le W)$.

TO DETERMINE $V[i,j]$ WE HAVE IN GENERAL TWO ALTERNATIVES

- DO NOT INCLUDE OBJECT i. IN THIS CASE AT MOST VALUE $V[i-1,j]$ CAN BE STOLEN.
- INCLUDE OBJECT i. THIS INCREASES THE VALUE OF THE LOAD BY v_i, AND REDUCES THE REMAINING CAPACITY BY w_i. THUS IN THIS CASE AT MOST VALUE $v_i + V[i-1, j-w_i]$ CAN BE STOLEN.

CHOOSING THE BEST ALTERNATIVE YIELDS

$$V[i,j] = \max\left(V[i-1,j],\ v_i + V[i-1, j-w_i]\right)$$

INCLUDING BOUNDARY AND OUT OF BOUNDS ENTRIES WE HAVE

$$V[i,j] = \begin{cases} 0 & i=0,\ j\geq 0 \\ \max\left(V[i-1,j],\ v_i + V[i-1, j-w_i]\right) & i>0,\ j\geq 0 \\ -\infty & j<0 \end{cases}$$

EX $n=5$, $W=10$

		0	1	2	3	4	5	6	7	8	9	10
$w_1=1$,	$v_1=1$	0	1	1	1	1	1	1	1	1	1	1
$w_2=3$,	$v_2=5$	0	1	1	5	6	6	6	6	6	6	6
$w_3=5$,	$v_3=12$	0	1	1	5	6	12	13	13	17	18	18
$w_4=6$,	$v_4=25$	0	1	1	5	6	12	25	26	26	30	31
$w_5=7$,	$v_5=30$	0	1	1	5	6	12	25	30	31	31	35

EXERCISE:

WRITE AN ALGORITHM WHICH GIVEN THE INPUT ARRAYS $v[1..n]$ AND $w[1..n]$, AND THE WEIGHT LIMIT W, FILLS IN THE TABLE $V[1..n; 0..W]$ AND RETURNS $V[n,W]$. (OR IF YOU PREFER, RETURN THE WHOLE TABLE.)

EXERCISE

WRITE AN ALGORITHM WHICH GIVEN THE FILLED TABLE $V[1 \cdots n; 0 \cdots W]$ PRINTS OUT A LIST OF EXACTLY WHICH OBJECTS TO INCLUDE.

THE PRINCIPLE OF OPTIMALITY

AN OPTIMIZATION PROBLEM SATISFIES THE PRINCIPLE OF OPTIMALITY IF THE OPTIMAL SOLUTION TO ANY (NON-TRIVIAL) INSTANCE IS A COMBINATION OF SOME OF ITS SUBINSTANCES.

i.e. AN OPTIMAL SOLUTION CONTAINS WITHIN IT OPTIMAL SOLUTIONS TO CERTAIN SUBPROBLEMS.

i.e. IN AN OPTIMAL SEQUENCE OF CHOICES, EACH SUBSEQUENCE IS ALSO OPTIMAL.

WE ALSO SAY THAT SUCH A PROBLEM EXHIBITS OPTIMAL SUBSTRUCTURE.

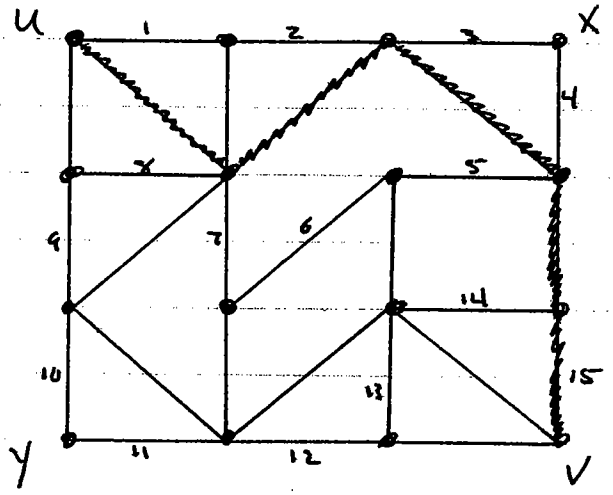
e.g. COIN CHANGE:

$$C[i,j] = \min\left(C[i-1,j],\ 1 + C[i, j-d_i]\right)$$

e.g. 0-1 KNAPSACK:

$$V[i,j] = \max\left(V[i-1,j],\ v_i + V[i-1, j-w_i]\right)$$

EX. SHORTEST PATHS IN GRAPH



$d(u,v) = 5$

$\ell(u,v) = 15$

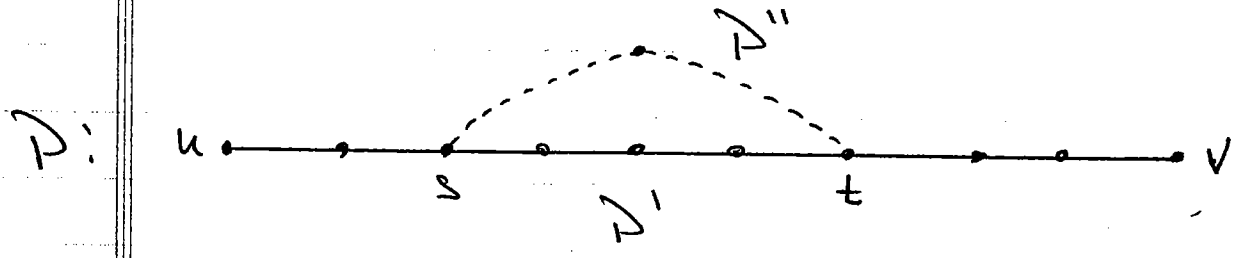
DEFN: A *u-v PATH* IS A SEQUENCE OF ALTERNATING VERTICES AND INCIDENT EDGES STARTING AT u AND ENDING AT v, IN WHICH NO VERTEX IS REPEATED (EXCEPT POSSIBLY $u=v$).

THE *LENGTH* OF A PATH IS THE NUMBER OF EDGES IN THE SEQUENCE

PROBLEM: DETERMINE A SHORTEST u-v PATH

LET $d(u,v)$ DENOTE THE LENGTH OF SUCH A PATH.

OBSERVE: ANY SEGMENT OF A SHORTEST PATH IS ALSO A SHORTEST PATH.



PROOF: LET P BE A SHORTEST u-v PATH, s AND t TWO INTERMEDIATE VERTICES ON P, AND P' THE SEGMENT OF P FROM s TO t. IF P' WERE NOT A SHORTEST s-t PATH WE COULD SPLICE P' OUT OF P AND REPLACE IT WITH A SHORTER s-t PATH P''. THE RESULTING u-v PATH WOULD THEN BE SHORTER THAN P, CONTRADICTING THAT P WAS SHORTEST.

///

THUS THE SHORTEST PATH PROBLEM SATISFIES THE PRINCIPLE OF OPTIMALITY.

ONE MIGHT EASILY COME TO BELIEVE THAT ALL OPTIMALITY PROBLEMS EXHIBIT OPTIMAL SUB-STRUCTURE, BUT THIS IS FALSE. IF A PROBLEM FAILS TO SATISFY THE OPTIMALITY PRINCIPLE, IT IS PROBABLY IMPOSSIBLE TO SOLVE IT USING DYNAMIC PROGRAMMING.

EX. LONGEST PATH IN GRAPH

PROBLEM: DETERMINE A LONGEST u-v PATH

LET $l(u,v)$ DETERMINE THE LENGTH OF SUCH A PATH.

OBSERVE THAT A SEGMENT OF A LONGEST u-v PATH MAY NOT BE A LONGEST PATH. IN THE PREVIOUS EXAMPLE ONE CHECKS THAT $l(x,y) = 8$, BUT A LONGEST u-v PATH TRAVELS FROM x TO y IN 7 STEPS.

THIS PROBLEM THEREFORE VIOLATES THE PRINCIPLE OF OPTIMALITY.

GENERAL PROCEDURE

1.) CHARACTERIZE THE STRUCTURE OF AN OPTIMAL SOLUTION. (i.e. SHOW THAT YOUR PROBLEM INVOLVES SOME CHOICE(S) WHICH LEAVE ONE OR MORE SUBPROBLEMS.)

2.) RECURSIVELY DEFINE AN OPTIMAL SOLUTION (i.e. IN TERMS OF OPTIMAL SUBPROBLEM SOLUTIONS.)

3.) COMPUTE THE VALUE OF AN OPTIMAL SOLUTION IN A BOTTOM UP FASHION (i.e. CONSTRUCT TABLE)

4.) CONSTRUCT AN OPTIMAL SOLUTION (USING THE TABLE GENERATED IN (3).)

WHEN SHOULD YOU USE DYNAMIC PROGRAMMING? LOOK FOR THE FOLLOWING.

- OPTIMAL SUBSTRUCTURE (YOU CAN USE DYNAMIC PROGRAMMING.)
- OVERLAPING SUBPROBLEMS (YOU SHOULD USE DYNAMIC PROGRAMMING.)

MATRIX CHAIN MULTIPLICATION.

CONSIDER THE PROBLEM OF MULTIPLYING TWO MATRICES A ($p \times q$) AND B ($q \times r$). THE PRODUCT AB HAS DIMENSIONS ($p \times r$). ITS ij ENTRY IS

$$\sum_{k=1}^{q} a_{ik} b_{kj} \qquad (1 \le i \le p,\ 1 \le j \le r)$$

WHICH INVOLVES q SCALAR MULTIPLICATIONS.
THUS THE TOTAL NUMBER OF SCALAR MULTIPLICATIONS PERFORMED IN COMPUTING AB IS pqr.

NOW CONSIDER THE PRODUCT OF THREE MATRICES ABC OF DIMENSIONS $(p \times q)$, $(q \times r)$, AND $(r \times s)$ RESPECTIVELY.

THERE ARE TWO POSSIBLE PARENTHESIZATIONS OF THIS PRODUCT:

(1) $A(BC)$ # SCALAR MULTIPLICATIONS $= pqs + qrs$

(2) $(AB)C$ # SCALAR MULTIPLICATIONS $= pqr + prs$

e.g. $p = 10$, $q = 100$, $r = 10$, $s = 100$. THEN (1) YIELDS 200,000 MULTIPLICATIONS AND (2) YIELDS 20,000.

THUS ONE PARENTHESIZATION MAY BE MORE EFFICIENT THAN ANOTHER.

FOR FOUR MATRICES THERE ARE FIVE PARENTHESIZATIONS.

LET $P(n)$ DENOTE THE NUMBER OF DISTINCT PARENTHESIZATIONS OF A PRODUCT OF n MATRICES.

EXERCISE: SHOW $P(n)$ SATISFIES THE RECURRENCE

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) \qquad (n \geq 1).$$

THE SEQUENCE $P(n)$ IS CALLED THE CATALAN NUMBERS. ONE CAN SHOW THAT

$$P(n) = \frac{1}{n}\binom{2n-2}{n-1}$$

EXERCISE

USE STIRLING'S FORMULA TO SHOW

$$P(n) = \Theta\left(\frac{4^n}{n^{3/2}}\right)$$

PROBLEM

GIVEN A PRODUCT OF n MATRICES $A_1 A_2 \cdots A_n$ WHERE A_i HAS DIMENSIONS $P_{i-1} \times P_i$, DETERMINE A PARENTHESIZATION WHICH MINIMIZES THE TOTAL NUMBER OF SCALAR MULTIPLICATIONS PERFORMED.

AS USUAL THERE ARE REALLY TWO PROBLEMS

- FIND THE MINIMUM NUMBER OF MULTIPLICATIONS
- FIND THE PARENTHESIZATION WHICH ACHIEVES IT.

WE CAN SEE THAT THE BRUTE FORCE APPROACH IS INEFFICIENT SINCE THE NUMBER OF PARENTHESIZATIONS IS EXPONENTIAL.