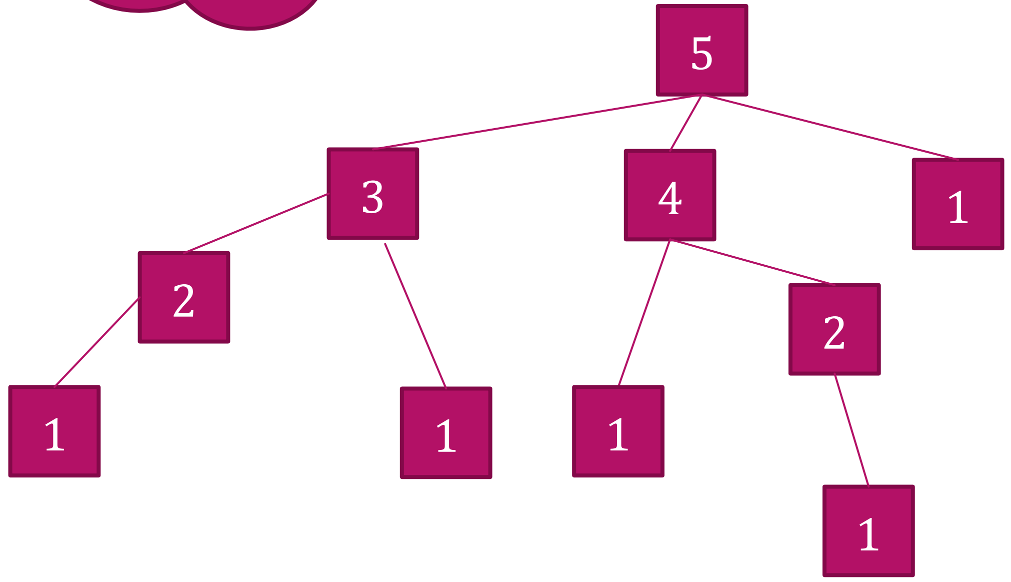
Problem analysis of Dynamic Programming(1)

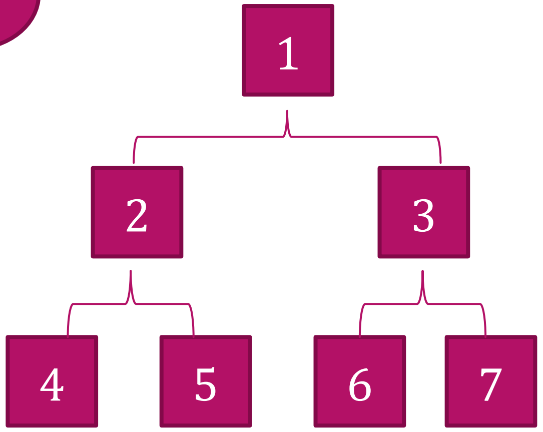
YAO ZHAO

Dynamic Programming vs Divide-and-Conquer

DP：subproblems overlap

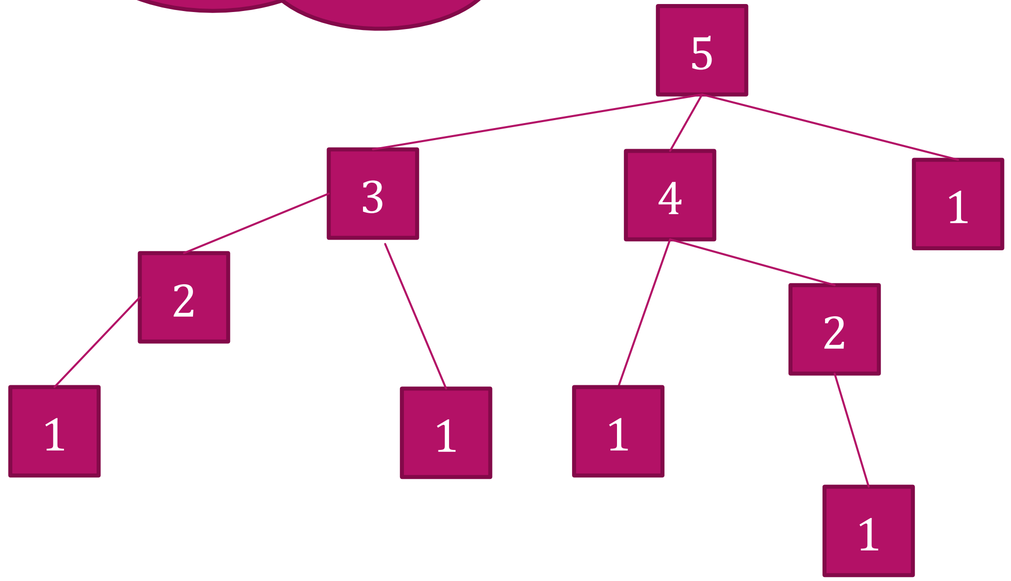


Divide-and-Conquer：Subproblems disjoint

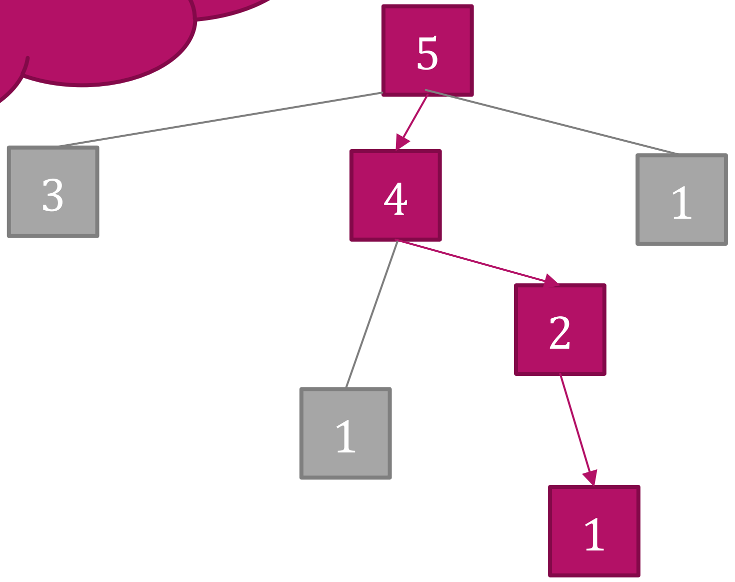


Dynamic Programming vs Greedy Algorithm

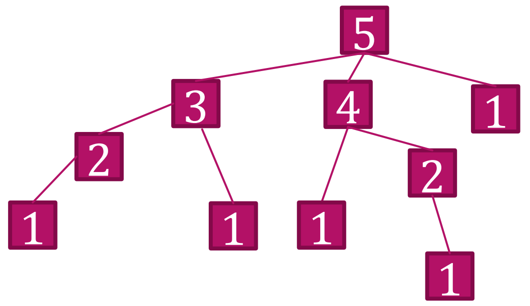
DP： go through a sequence of steps, with a set of choices at each step

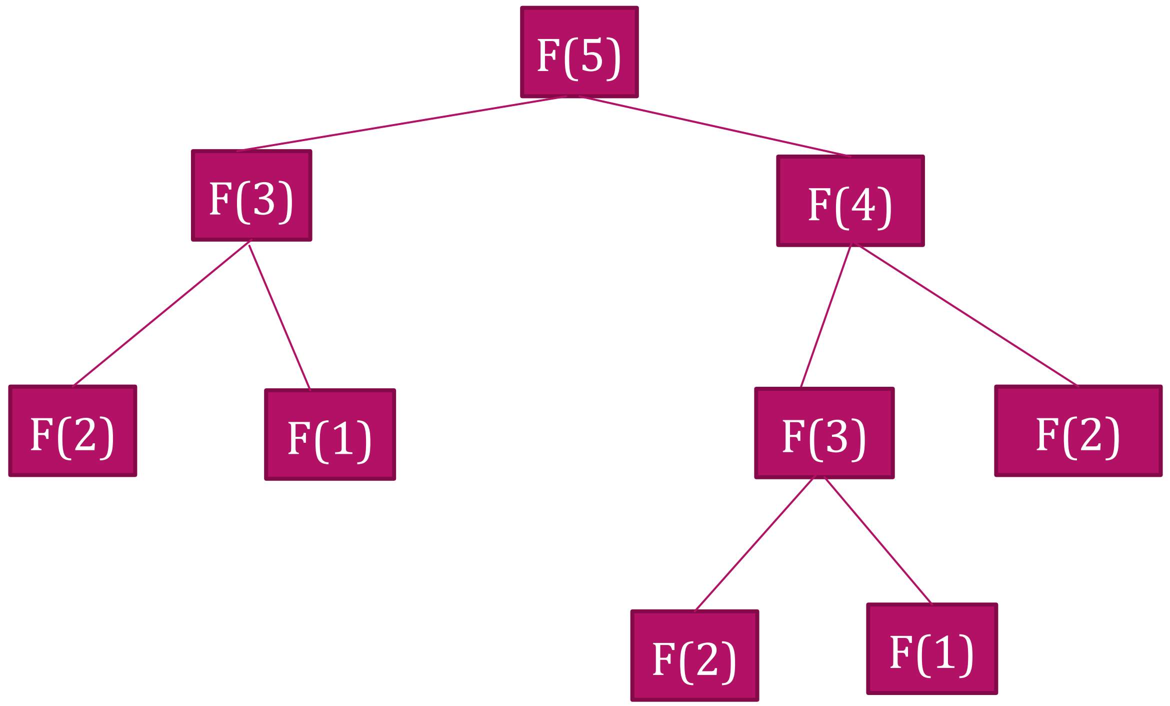


Greedy Algorithm ： always makes the choice that looks best at the moment



A Typical Dynamic Programming Problem





- **Fibonacci:**

F(1)=1，

F(2)=1,

F(n)=F(n-1)+F(n-2)（n>=3，n∈N*）

Question 1: Greedy or DP?

- Now we have some coins with face value of 1 yuan, 3 yuan, 6 yuan and 7 yuan;
- Q: how to get 18 yuan with the minimum number of coins?

Question 1: Greedy or DP?

- A:DP

Question 2: Greedy or DP?

- Now we have some coins with face value of 1 yuan, 2 yuan, 5 yuan and 10 yuan;
- Q: how to get 18 yuan with the minimum number of coins?

Question 2: Greedy or DP?

- A:Greedy

Dynamic-programming steps

- When developing a dynamic-programming algorithm, we follow a sequence of four steps:
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution, typically in a bottom-up fashion.
  4. Construct an optimal solution from computed information.

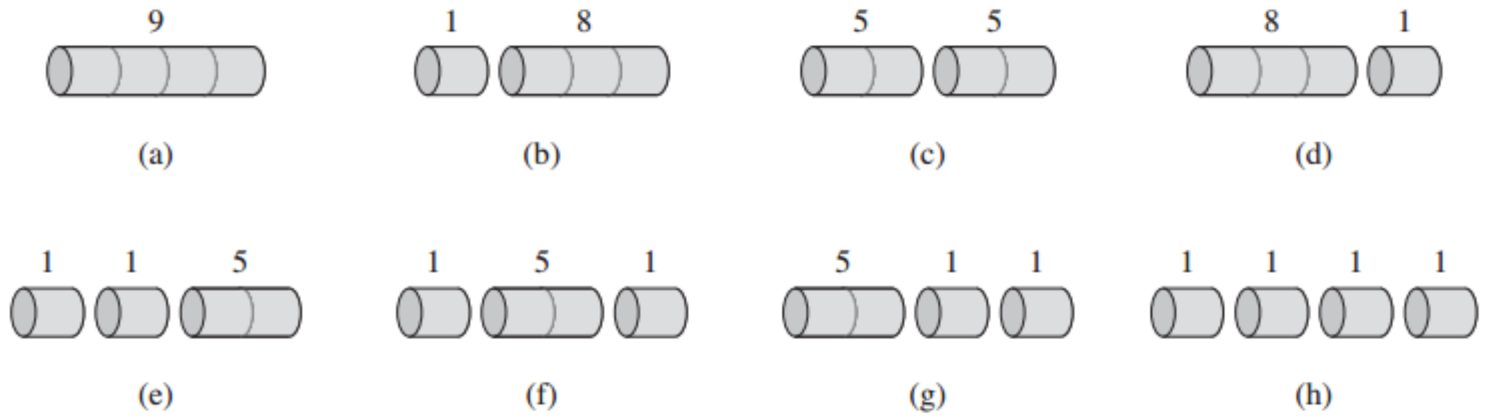
  Steps 1–3 form the basis of a dynamic-programming solution to a problem. If we need only the value of an optimal solution, and not the solution itself, then we can omit step 4.

The Rod-cutting Problem

- Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells. Each cut is free. The management of Serling Enterprises wants to know the best way to cut up the rods.
  We assume that we know, for i = 1, 2,...the price pi in dollars that Serling Enterprises charges for a rod of length i inches. Rod lengths are always an integral number of inches.
  The ***rod-cutting problem*** is the following. Given a rod of length n inches and a table of prices pi for i = 1, 2,... n, determine the maximum revenue rn obtainable by cutting up the rod and selling the pieces. Note that if the price pn for a rod of length n is large enough, an optimal solution may require no cutting at all.

A sample price table for rods. Each rod of length i inches earns the company p_i dollars of revenue.

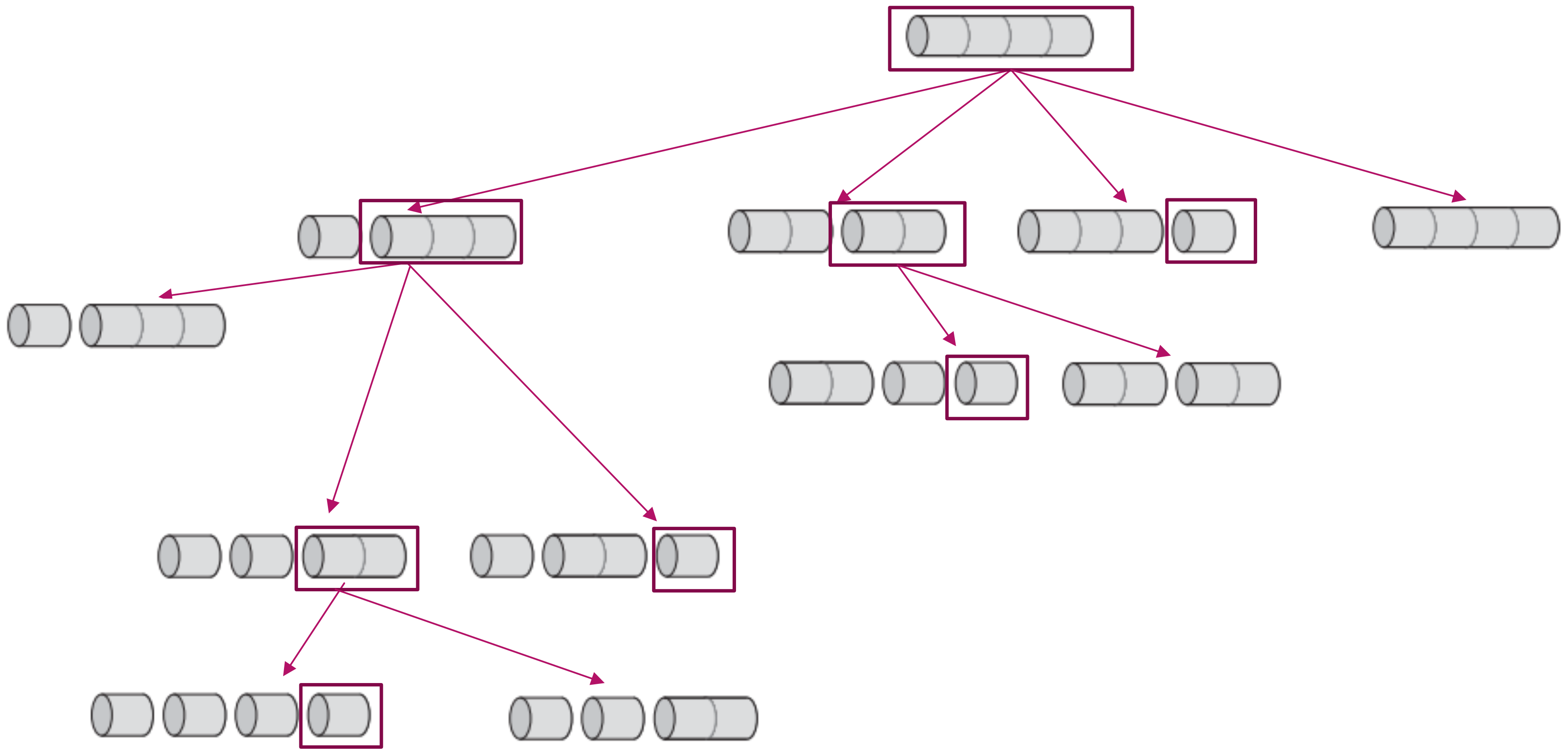
length i	1	2	3	4	5	6	7	8	9	10
price pi	1	5	8	9	10	17	17	20	24	30



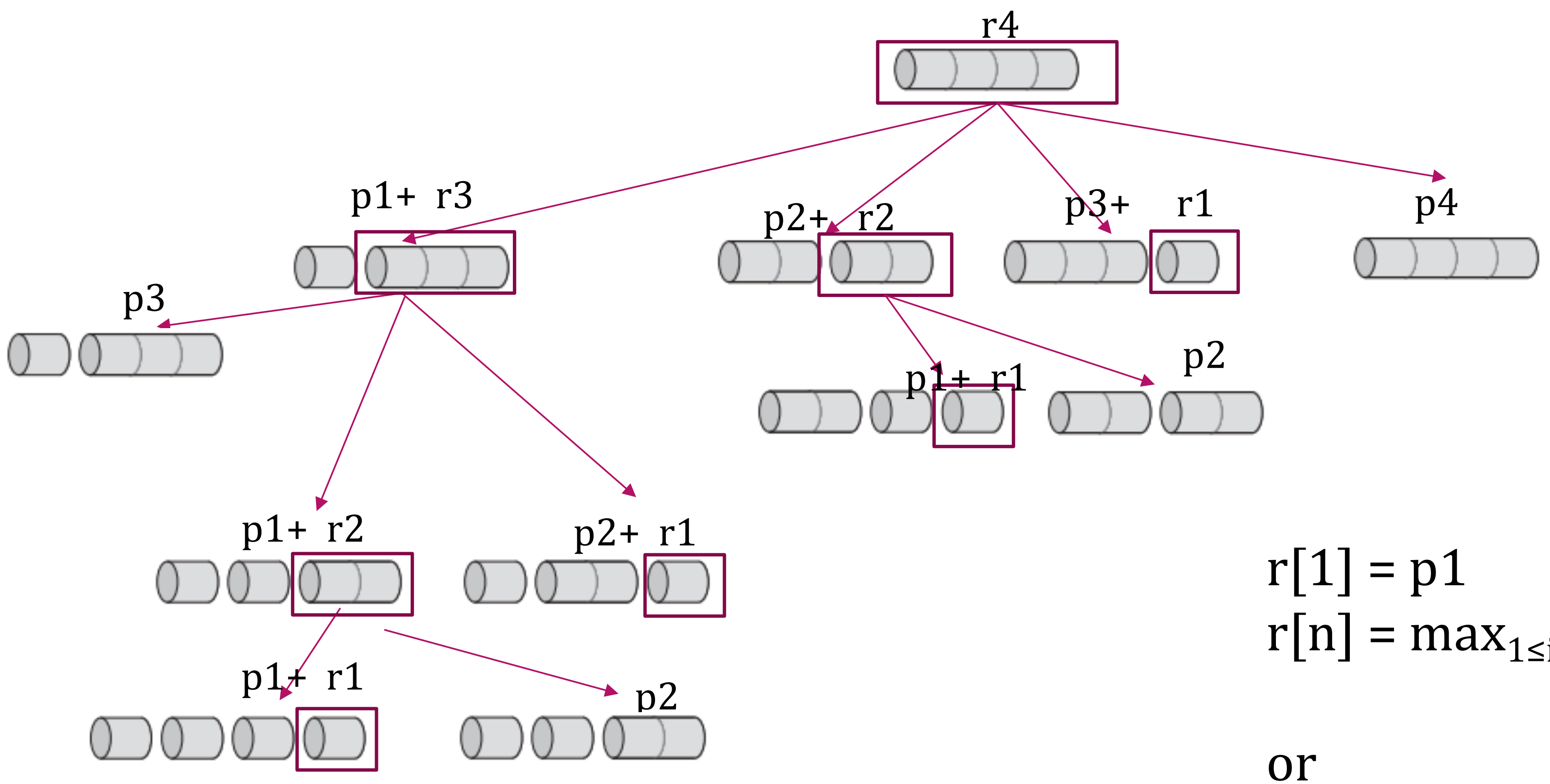
Consider the case when n=4. Above figure shows all the 8 possible ways to cut up a rod of 4 inches in length, including the way with no cuts at all.
Above each piece is the value of that piece, according to the sample price chart. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

A decomposition as consisting of first piece of length i cut off the left-hand end, and then a right-hand remainder of length n-i. Only the remainder, not the first piece, may be further divided.

1. Characterize the structure of an optimal solution.



2. Recursively define the value of an optimal solution.



$r[1] = p1$

$r[n] = \max_{1 \le i < n}\{p[i]+r[n-i], p[n]\}$

or

$r[0] = 0$

$r[n] = \max_{1 \le i \le n}\{p[i]+r[n-i]\}$

3. Compute the value of an optimal solution, typically in a bottom-up fashion.

length i	1	2	3	4	5	6	7	8	9	10
price pi	1	5	8	9	10	17	17	20	24	30

N=4

N	1	2	3	4
r	1	5	8	10

r[2] = max{p[1]+r[1], p[2]} = max{5, 2} = 5
r[3] = max{p[1]+r[2], p[2]+r[1], p[3]} = 8
r[4] = max{p[1]+r[3], p[2]+r[2], p[3]+r[1], p[4]} = 10

The rod-cutting problem is also a Knapsack problem.
Each length i is an item, a rod of length n inches is a Knapsack of capacity n.
It is an Unbounded Knapsack Problem,

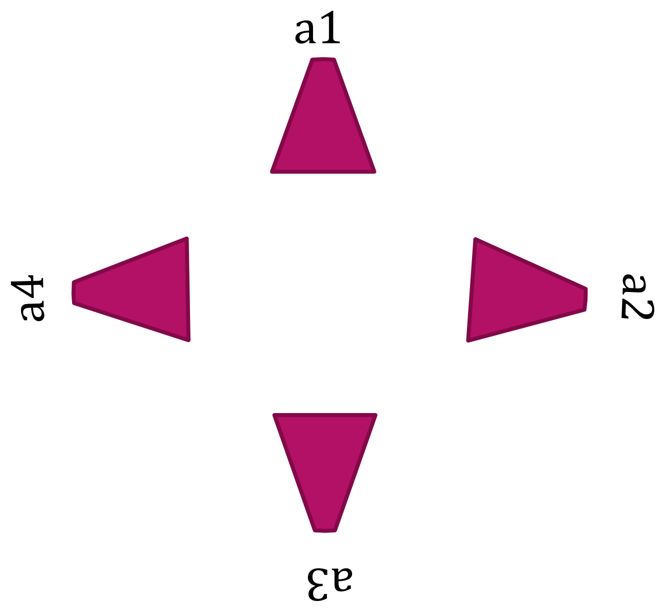
$$\text{maximize} \sum_{i=1}^{n} v_i x_i \qquad \boxed{=}$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \boxed{\leq} W \text{ and } x_i \geq 0,\ x_i \in \mathbb{Z}.$$

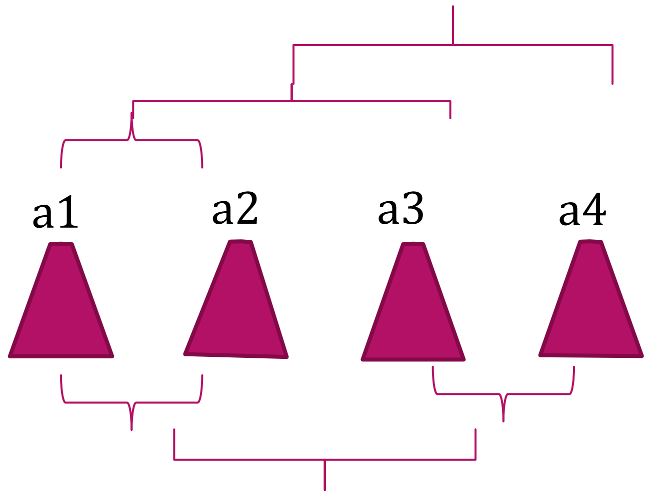
The Stones merging Problem

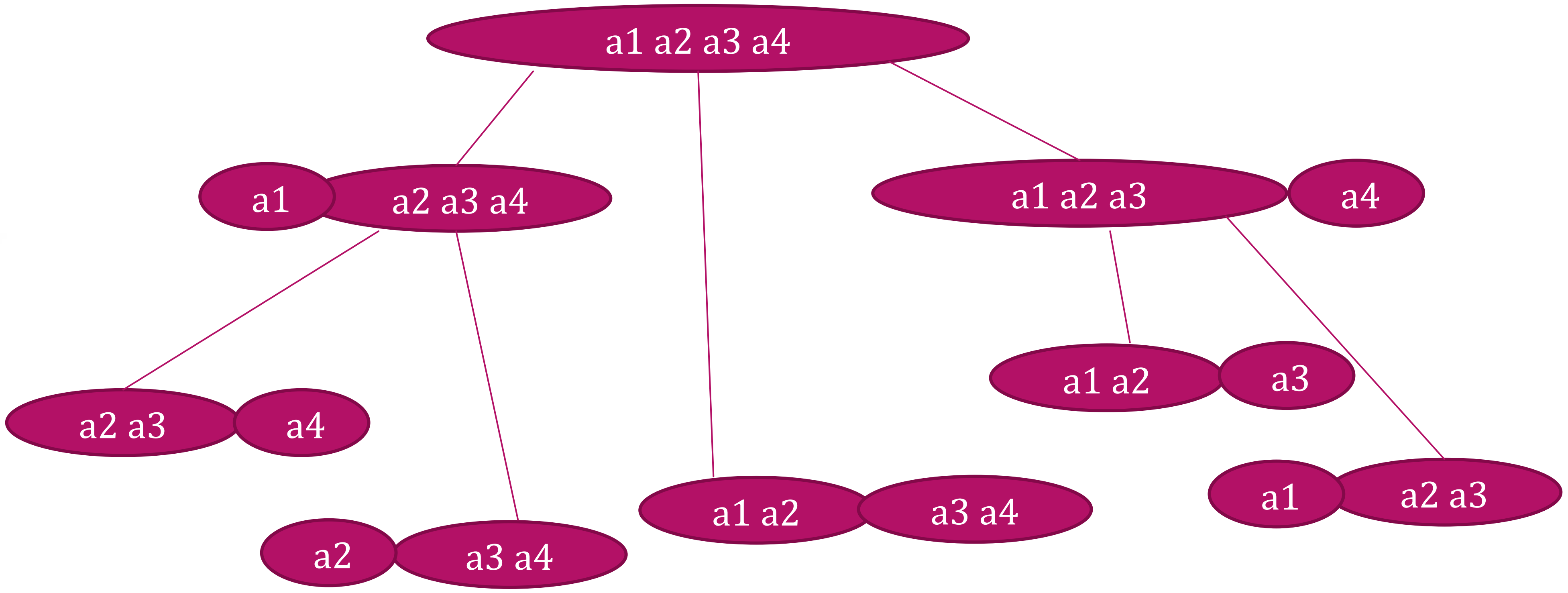
- There are n piles of stones around a round playground. The i-th pile has a[i] stones. Now we are going to merge the stones into one pile. Only two adjacent piles of stones can be merged during the merging process, and the cost of merging is the number of stones in the new pile. Try to design an algorithm to calculate the minimum total cost of merging n piles of stones into one pile.

Assume there are 4 piles of stones:



If we always merge from a1:

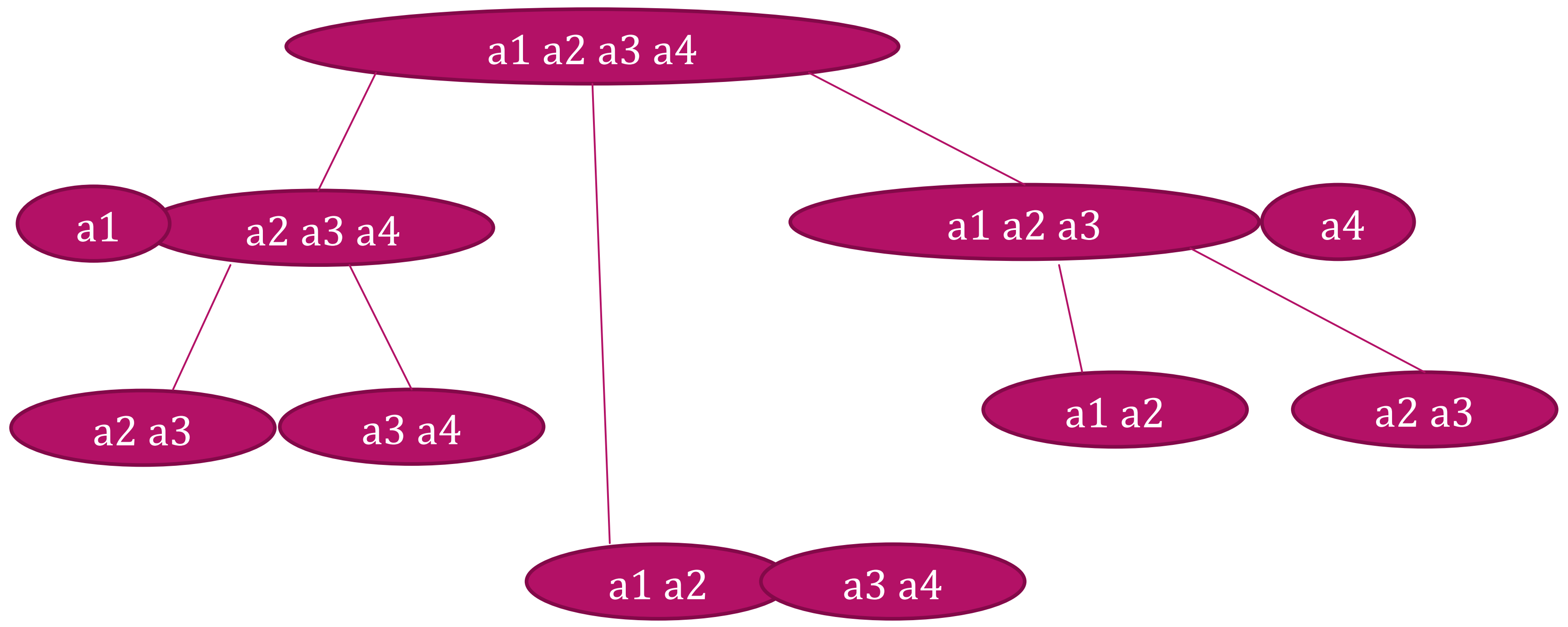



a1 a2 a3 a4
a1
a2 a3 a4
a1 a2 a3
a4
a2 a3
a4
a2
a3 a4
a1 a2
a3 a4
a1 a2
a3
a1
a2 a3

Let p(n) be the number of possible solutions:

- P(n) = 1   if n=1
- P(n) =$\sum_{k=1}^{n-1} p(k)p(n-k)$
- Solution space: p(n) = Catalan(n-1) = $\Omega(4^n/n^{\frac{3}{2}})$

1. Characterize the structure of an optimal solution.



The bottom subproblem:
a1a2
a2a3
a3a4

The higher level subproblem:
a1a2a3
a2a3a4

The top problem:
a1a2a3a4

2. Recursively define the value of an optimal solution.

Let opt[i][j] is the minimum total cost of merging a[i]a[i+1]…a[j] into one pile， then：

OPT[i][j] = $\min_{i \le k < j}$ {OPT[i][k]+ OPT[k+1][j]+$\sum_{n=i}^{j} a[n]$ }

3. Compute the value of an optimal solution, typically in a bottom-up fashion.

	1	2	3	4
1	0	50	75	110
2		0	20	45
3			0	15
4				0

Assume a1 = 35 a2 = 15 a3 = 5 a4= 10
The values on the diagonal are initialized to 0

opt[1][2]= opt[1][1] + opt[2][2] + $\sum_{n=1}^{2} a[n]$ = 50
opt[2][3]= opt[2][2] + opt[3][3] + $\sum_{n=2}^{3} a[n]$ = 20
opt[3][4]= opt[3][3] + opt[4][4] + $\sum_{n=3}^{4} a[n]$ = 15

opt[1][3]= min{opt[1][2]+ opt[3][3] + $\sum_{n=1}^{3} a[n]$,
opt[1][1] + opt[2][3]+ $\sum_{n=1}^{3} a[n]$} = 75

opt[2][4] = min{opt[2][3]+ opt[4][4] + $\sum_{n=2}^{4} a[n]$,
opt[2][2] + opt[3][4]+ $\sum_{n=2}^{4} a[n]$} = 45

opt[1][4]= min{opt[1][1] + opt[2][4] + $\sum_{n=1}^{4} a[n]$,
opt[1][2]+ opt[3][4]+ $\sum_{n=1}^{4} a[n]$,
opt[1][3]+ opt[4][4] + $\sum_{n=1}^{4} a[n]$} = 110

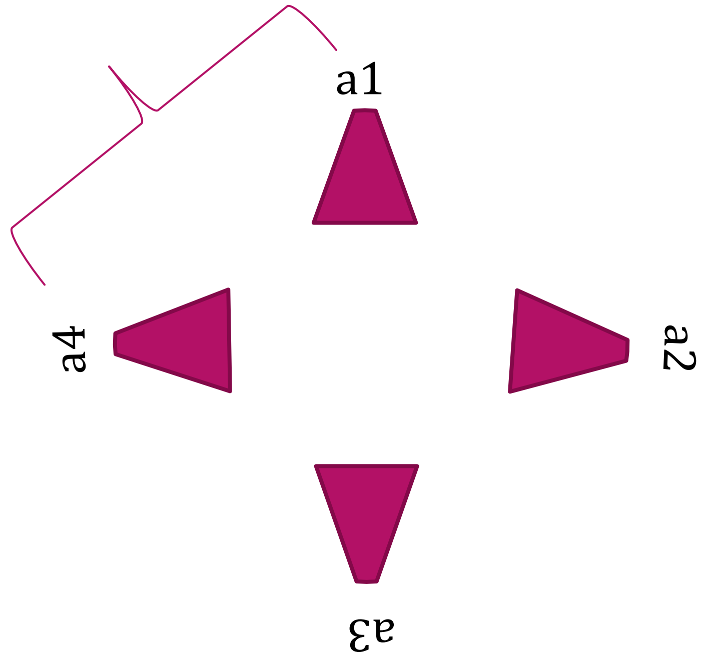
$\sum_{n=i}^{j} a[n] \longrightarrow$ Apply prefix sum

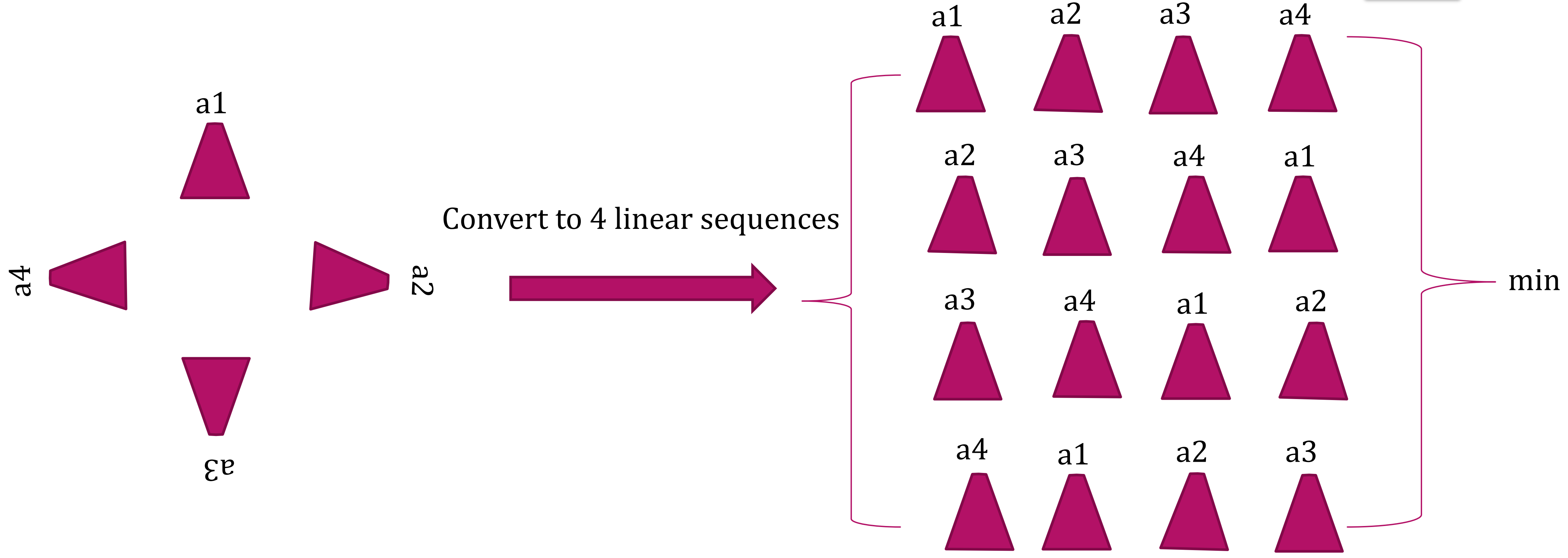
Fill the table from diagonal to top-right.

Solve the problem?

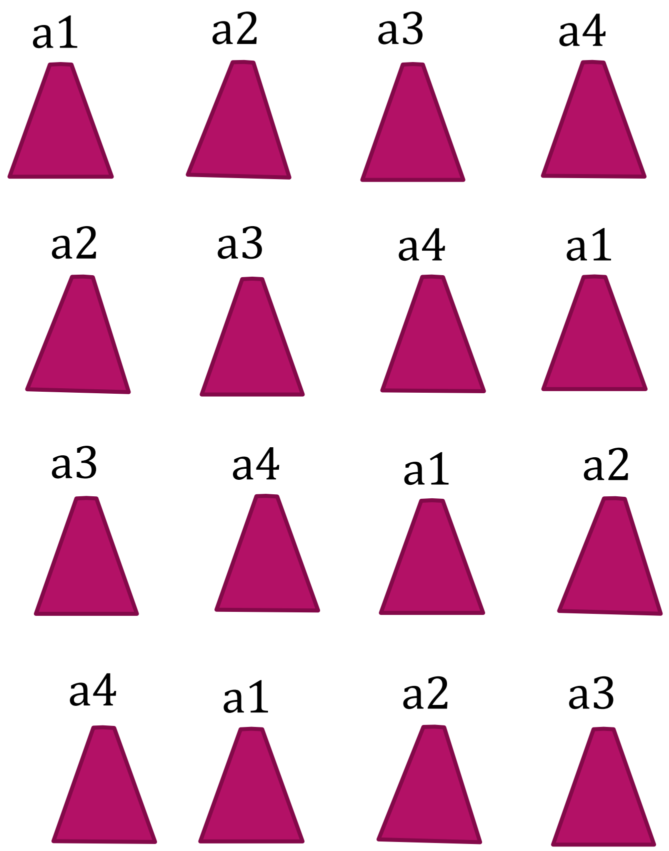
No, the original shape is a circle.

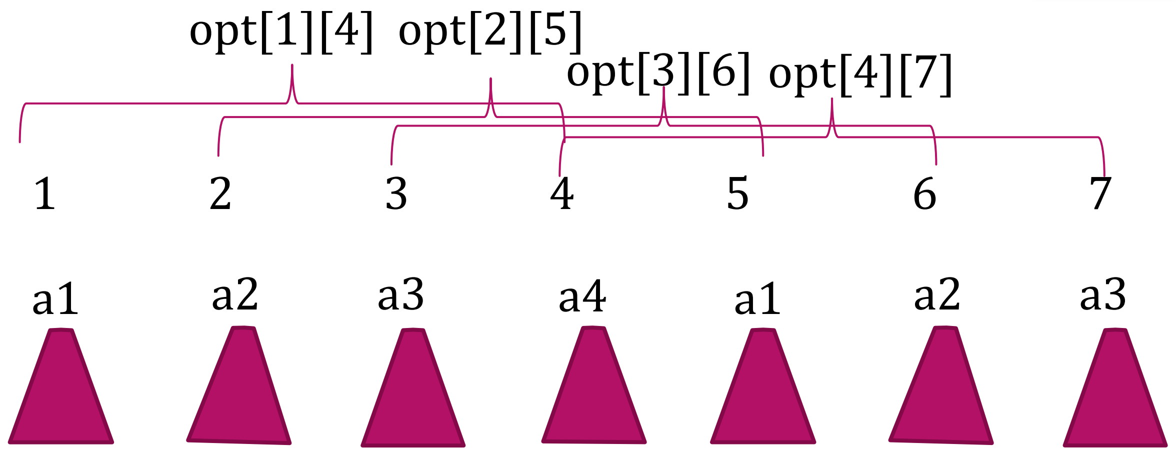
Above solution just solves one arrangement of a1a2a3...an, however, it is possible to merge an and a1.





a1
a2
a3
a4
Convert to 4 linear sequences
a1 a2 a3 a4
a2 a3 a4 a1
a3 a4 a1 a2
a4 a1 a2 a3
min





Min{opt[1][4], opt[2][5], opt[3][6], opt[4][7]}

Assume a1 = 35 a2 = 15 a3 = 5 a4= 10

	1	2	3	4	5	6	7
1	0	50	75	110			
2		0	20	45	110		
3			0	15	65	130	
4				0	45	105	130
5					0	50	75
6						0	20
7							0

Circle requires more calculated values than chain sequences

	1	2	3	4
1	0	50	75	110
2		0	20	45
3			0	15
4				0

Same type problems

- **Matrix-chain multiplication**
- Optimal binary search tree

Matrix-chain multiplication

- We can multiply two matrices A and B only if they are ***compatible***: the number of columns of A must equal the number of rows of B. If A is a p×q matrix and B is a q × r matrix, the resulting matrix C is a p × r matrix. The time to compute C is dominated by the number of scalar multiplications, which is pqr.
- Consider the problem of a chain <A1, A2, A3> of three matrices. Suppose that the dimensions of the matrices are 10 × 100, 100 × 5, and 5 × 50, respectively.
  - If we multiply according to the parenthesization ((A1*A2)*A3), we perform 7500 scalar multiplications.

    10*100*5+10*5*50=5000+2500=7500
  - If instead we multiply according to the parenthesization (A1*(A2*A3)) , we perform 75,000 scalar multiplications.

    100*5*50+10*100*50=25000+50000=75000
  - Thus, computing the product according to the first parenthesization is 10 times faster.

Matrix-chain multiplication

- Given a chain <A1, A2,... An> of n matrices, where for i = 1, 2, ...,n, matrix Ai has dimension pi-1 × pi, fully parenthesize the product A1A2 ...An in a way that minimizes the number of scalar multiplications.

Matrix-chain multiplication

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