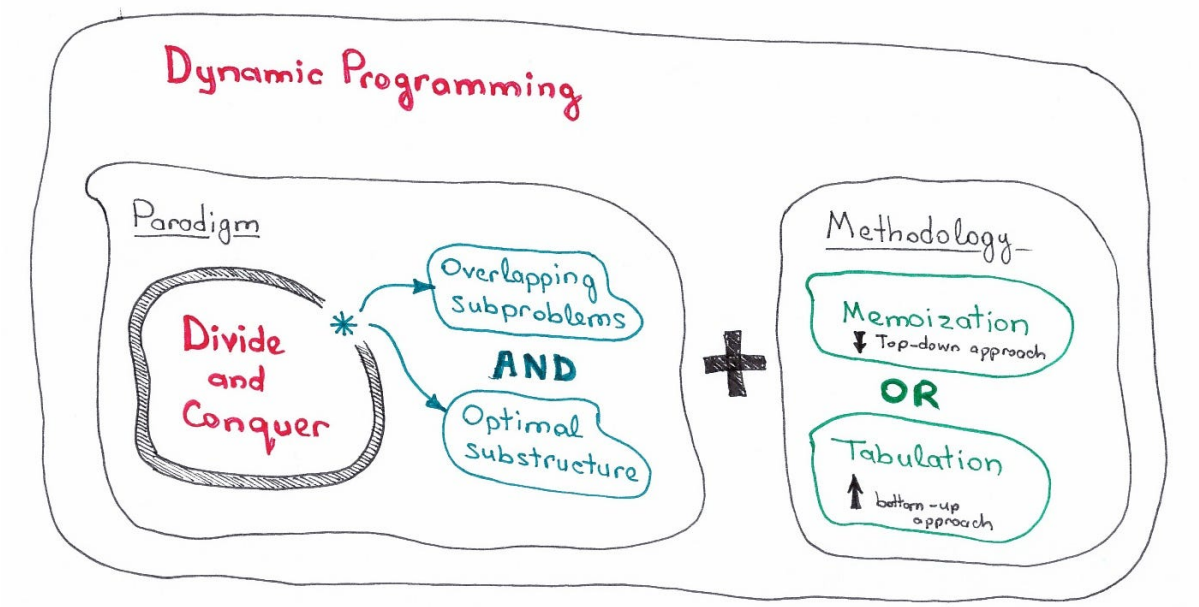
CS112 - Phương pháp thiết kế thuật toán: DYNAMIC PROGRAMMING (2)

Dynamic Programming:



I. Memory Function:

Trong quy hoạch động, Memory Functions (hay còn gọi là hàm nhớ) là một khái niệm được sử dụng để tối ưu hóa việc tính toán các bài toán con chồng chéo trong quá trình giải quyết bài toán lớn hơn. Khi ta giải quyết một bài toán lớn thành các bài toán con nhỏ hơn, có thể có sự trùng lặp của các bài toán con này. Memories Functions giúp chúng ta lưu trữ kết quả của các bài toán con đã được tính toán trước đó, từ đó tránh việc tính toán lại các bài toán con đã được giải quyết.

Cách thức hoạt động của Memories Functions thường được thực hiện bằng cách sử dụng một bảng (table) hoặc một mảng để lưu trữ các giá trị đã tính toán. Ban đầu, tất cả các giá trị trong bảng được khởi tạo với giá trị đặc biệt (ví dụ: null hoặc -1) để chỉ ra rằng chúng chưa được tính toán.

Khi cần tính toán một giá trị mới cho một bài toán con, trước tiên, ta kiểm tra xem giá trị đó đã được tính toán trước đó chưa bằng cách kiểm tra giá trị tương ứng trong bảng. Nếu giá trị đã tồn tại (khác giá trị đặc biệt), ta có thể truy xuất giá trị đó từ bảng mà không cần tính toán lại. Ngược lại, nếu giá trị chưa được tính toán, ta tính toán nó bằng cách sử dụng các giá trị đã được tính toán trước đó và lưu kết quả vào bảng để sử dụng cho các lần truy xuất sau này.

Bằng cách sử dụng Memories Functions, chúng ta có thể tránh việc tính toán lại các bài toán con đã được giải quyết, giúp tăng hiệu suất của thuật toán và giảm thời gian thực thi. Việc sử dụng hàm nhớ phổ biến trong nhiều bài toán quy hoạch động, chẳng hạn như

bài toán cái túi (knapsack problem) hay tìm dãy con tăng dài nhất (longest increasing subsequence).

ALGORITHM *MFKnapsack*(i, j)

```
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
//       items being considered and a nonnegative integer j indicating
//       the knapsack capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights[1..n], Values[1..n],
//and table F[0..n, 0..W] whose entries are initialized with −1's except for
//row 0 and column 0 initialized with 0's
if F[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1, j),
                    Values[i] + MFKnapsack(i − 1, j − Weights[i]))
    F[i, j] ← value
return F[i, j]
```

EXAMPLE 1 Let us consider the instance given by the following data:

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity $W = 5$.

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3, v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2, v_4 = 15$	4	0	—	—	—	—	**37**

FIGURE 8.6 Example of solving an instance of the knapsack problem by the memory function algorithm.

II. Optimal binary search tree:

Sử dụng kỹ thuật quy hoạch động để xây dựng cây nhị phân tìm kiếm tối ưu. Cụ thể, ta sử dụng một mảng 2 chiều để lưu trữ các giá trị tối ưu cho các cây con của cây nhị phân tìm kiếm.

Cây nhị phân tìm kiếm (binary search tree) là một cây nhị phân có tính chất sau:

1. *Mỗi nút là một khóa tìm kiếm*
2. *Với mỗi cây con, khóa của nút gốc lớn hơn khóa của mọi nút của cây con trái và nhỏ hơn khóa của mọi nút của cây con phải*

=> Để tìm cây nhị phân tối ưu thì các cây con của nó cũng phải tối ưu. Tức là ở đây chúng ta đang chia nhỏ bài toán ra để tìm các cây con trước, sau đó tìm được cây tối ưu lớn.

Problem 4: Cho một mảng K[1,2,…,n] đã sắp xếp theo chiều tăng dần trong đó các phần tử đôi một khác nhau. Mỗi phần tử K[i] có tần số tìm kiếm f[i]. Tìm cây nhị phân với khóa là các phần tử của mảng K sao cho tổng số lượng các phép so sánh là nhỏ nhất.

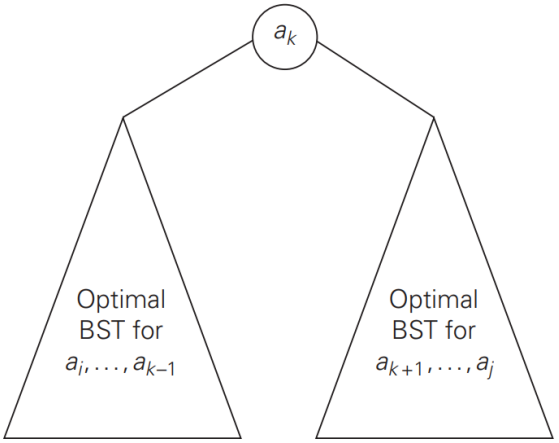


FIGURE 8.8 Binary search tree (BST) with root a_k and two optimal binary search subtrees T_i^{k-1} and T_{k+1}^{j}.

- Các bước để xác định OBST:
  + Xác định root: cho lần lượt các nút là root. Cho mỗi trường hợp tính tổng chi phí tìm kiếm.

+ Sử dụng mảng 2 chiều C[n,n] để lưu trữ kết quả, tránh việc tính toán lại. C[0][n-1] sẽ giữ kết quả cuối cùng. Trong quá trình triển khai là tất cả các giá trị đường chéo phải được điền trước, sau đó đến các giá trị nằm trên đường ngay phía trên đường chéo. Nói cách khác, trước tiên chúng ta phải điền tất cả các giá trị C[i][i], sau đó là tất cả các giá trị C[i][i+1], rồi tất cả các giá trị C[i][i+2].
+ Một bảng lưu trữ nút gốc, điền tương tự bảng trên bắt đầu với R(i, i) = i, 1 ≤ i ≤ n. Khi bảng được lấp đầy, các mục của nó chỉ ra các chỉ số về gốc của các cây con tối ưu, giúp có thể tái tạo lại một cây tối ưu cho toàn bộ tập hợp đã cho.

Chúng ta có công thức: (Tổng chi phí tìm kiếm = tổng chi phí tìm kiếm của cây con bên trái + tổng chi phí tìm kiếm của cây con bên phải + tổng các xác xuất p.)

$$C(i, j) = \min_{i \le k \le j} \{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \le i \le j \le n.$$

Với:

- C[i,j] là tổng chi phí tìm kiếm cho cây tối ưu = số phép so sánh của cây nhị phân tìm kiếm tối ưu cho mảng con K[i,...j].
- p là xác suất, tần số tìm kiếm f[i]

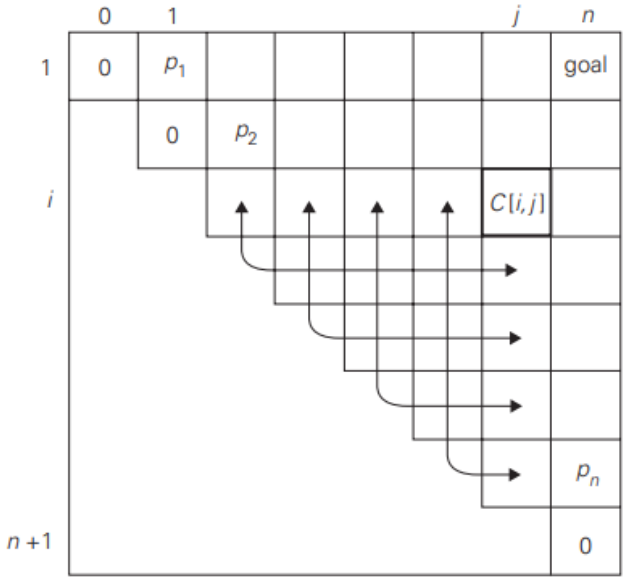


FIGURE 8.9 Table of the dynamic programming algorithm for constructing an optimal binary search tree.

ALGORITHM *OptimalBST*$(P[1..n])$

```
//Finds an optimal binary search tree by dynamic programming
//Input: An array P[1..n] of search probabilities for a sorted list of n keys
//Output: Average number of comparisons in successful searches in the
//        optimal BST and table R of subtrees' roots in the optimal BST
for i ← 1 to n do
    C[i, i − 1] ← 0
    C[i, i] ← P[i]
    R[i, i] ← i
C[n + 1, n] ← 0
for d ← 1 to n − 1 do    //diagonal count
    for i ← 1 to n − d do
        j ← i + d
        minval ← ∞
        for k ← i to j do
            if C[i, k − 1] + C[k + 1, j] < minval
                minval ← C[i, k − 1] + C[k + 1, j];   kmin ← k
        R[i, j] ← kmin
        sum ← P[i];   for s ← i + 1 to j do sum ← sum + P[s]
        C[i, j] ← minval + sum
return C[1, n], R
```

Ex:

key	*A*	*B*	*C*	*D*
probability	0.1	0.2	0.4	0.3

The initial tables look like this:

main table

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

main table

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					

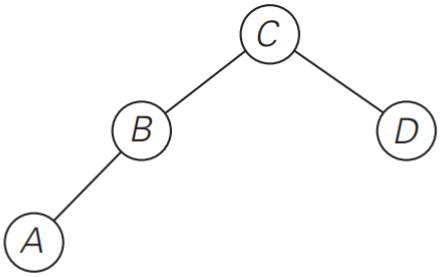


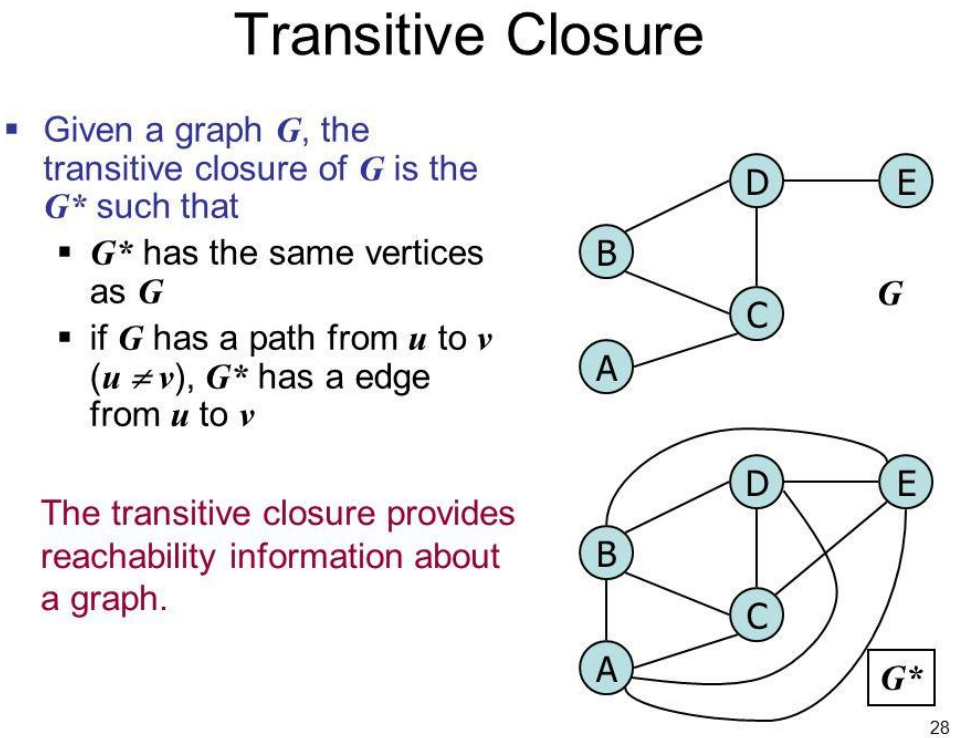
FIGURE 8.10 Optimal binary search tree for the example.

- References:
  https://www.geeksforgeeks.org/optimal-binary-search-tree-dp-24/
  https://www.giaithuatlaptrinh.com/?tag=optimal-binary-search-tree
  https://www.slideshare.net/GrokkingVN/grokking-techtalk-27-optimal-binary-search-tree
  https://www.javatpoint.com/optimal-binary-search-tree

III. Warshall's algorithm:

1. Transitive Closure

Transitive closure là một đồ thị mới được tạo ra từ đồ thị ban đầu bằng cách thêm tất cả các cạnh cần thiết để đảm bảo rằng mọi cặp đỉnh trong đồ thị đều có đường đi giữa chúng.



Chúng ta có thể tạo ra transitive closure của đồ thị bằng cách tìm kiếm theo chiều sâu hoặc tìm kiếm theo chiều rộng.

Hạn chế của cách làm này là ta phải duyệt qua cùng một đồ thị nhiều lần. Vì vậy, chúng ta cần một thuật toán hiệu quả hơn: Warshall's Algorithm.

2. Warshall's Algorithm

Thuật toán Warshall là một thuật toán để tìm đường đi ngắn nhất trong đồ thị có hướng hoặc vô hướng.

Thuật toán này sử dụng một ma trận kề để lưu trữ thông tin về đồ thị và tính toán các giá trị mới của ma trận đó để tìm ra đường đi ngắn nhất giữa các đỉnh.

ALGORITHM *Warshall*$(A[1..n, 1..n])$
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
return $R^{(n)}$

Các bước thực hiện:

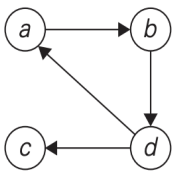
Khởi tạo ma trận kề ban đầu của đồ thị. Ma trận này có kích thước nxn, với n là số lượng đỉnh trong đồ thị. Giá trị của phần tử (i,j) trong ma trận là 1 nếu có cạnh từ đỉnh i đến đỉnh j, ngược lại là 0.

Sử dụng phương pháp lặp để tìm kiếm đường đi giữa các cặp đỉnh. Với mỗi lần lặp, thuật toán sẽ cập nhật ma trận kề bằng cách thêm các cạnh mới tìm được. Cụ thể, với mỗi đỉnh k trong đồ thị, thuật toán kiểm tra tất cả các cặp đỉnh i,j, nếu có thể tìm thấy đường đi từ i đến j qua đỉnh k, thì ta đánh dấu phần tử (i,j) của ma trận là 1. Duyệt qua công thức:

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \quad \text{or} \quad \left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)}\right).$$

Lặp lại bước 2 cho đến khi ma trận kề không còn thay đổi sau một lần lặp.

Thuật toán trả về ma trận kề với các giá trị phần tử (i,j) thể hiện có tồn tại đường đi từ đỉnh i đến đỉnh j hay không.



$R^{(0)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$R^{(1)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$R^{(2)}$ =

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$R^{(3)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a, b,* and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$R^{(4)}$ =

	a	b	c	d
a	**1**	1	**1**	1
b	**1**	**1**	**1**	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a, b, c,* and *d* (note five new paths).

Thuật toán Warshall có độ phức tạp O(n), với n là số đỉnh của đồ thị.

Mặc dù độ phức tạp này khá cao, nhưng thuật toán Warshall là một trong những thuật toán đơn giản nhất để tìm đường đi ngắn nhất và có thể được sử dụng trong nhiều bài toán thực tế.

IV. Floyd's algorithm for the all-pairs shortest-paths problem:

- Đồ thị liên thông có trọng số (vô hướng hoặc có hướng), bài toán tìm đường đi ngắn nhất từ mỗi đỉnh đến tất cả các đỉnh khác.. Đây là một trong nhiều biến thể của bài toán liên quan đến đường đi ngắn nhất trong đồ thị.
[Do các ứng dụng quan trọng của nó đối với thông tin liên lạc, mạng lưới giao thông vận tải và nghiên cứu hoạt động nên nó đã được nghiên cứu kỹ lưỡng trong nhiều năm. Trong số các ứng dụng gần đây của bài toán đường đi ngắn nhất cho tất cả các cặp là tính toán trước khoảng cách để lập kế hoạch chuyển động trong trò chơi máy tính.]

Floyd cũng tạo ma trận khoảng cách bằng một thuật toán rất giống với thuật toán của Warshall. Điểm khác là nó có thể áp dụng cho cả đồ thị có trọng số vô hướng và có hướng với điều kiện là chúng không chứa chu trình có độ dài âm. Thuật toán có thể được nâng cao để tìm không chỉ độ dài của các đường đi ngắn nhất cho tất cả các cặp đỉnh mà còn tìm cả các đường đi ngắn nhất

- Tương tự thuật toán Dijstrka.

Bước thực hiện:

+ Tạo ma trận A kích thước n*n, n là số đỉnh của đồ thị. Mỗi ô A[i][j] được điền khoảng cách từ đỉnh thứ i đến đỉnh thứ j. Nếu không có đường đi từ đỉnh thứ i đến đỉnh thứ j thì điền vô cực.
+ Ý tưởng là chọn từng đỉnh một và cập nhật tất cả các đường đi ngắn nhất bao gồm đỉnh đã chọn làm đỉnh trung gian trong đường đi ngắn nhất.
+ Khi chọn đỉnh số k làm đỉnh trung gian tức là ta đã coi các đỉnh {0, 1, 2, .. k-1} là đỉnh trung gian.
+ Với mọi cặp (i, j) của đỉnh nguồn và đích tương ứng, có hai trường hợp có thể xảy ra.
    - k không phải là đỉnh trung gian trên đường đi ngắn nhất từ i đến j. Chúng ta giữ nguyên giá trị của dist[i][j] .
    - k là đỉnh trung gian trên đường đi ngắn nhất từ i đến j. Chúng ta cập nhật giá trị của dist[i][j] theo công thức :

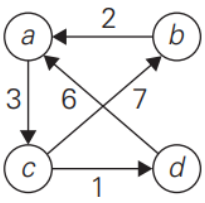
$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1,\ \ d_{ij}^{(0)} = w_{ij}.$$

ALGORITHM *Floyd*$(W[1..n, 1..n])$

```
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ← W  //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D
```

Độ phức tạp: Có ba vòng lặp. Mỗi vòng lặp có độ phức tạp không đổi. Vì vậy, độ phức tạp thời gian của thuật toán Floyd-Warshall là O(n3).

Độ phức tạp không gian của thuật toán Floyd-Warshall là O(n2).



$D^{(0)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$D^{(1)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	**5**	∞
c	∞	7	0	1
d	6	∞	**9**	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$D^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	**9**	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and *b* (note a new shortest path from *c* to *a*).

$D^{(3)} =$

	a	b	c	d
a	0	**10**	3	**4**
b	2	0	5	**6**
c	9	7	0	1
d	6	**16**	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a, *b*, and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$D^{(4)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	**7**	7	0	1
d	6	16	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

V. So sánh DP vs Chia để trị

Dynamic Programming	Divide and Conquer
DP sử dụng output của một vấn đề phụ nhỏ hơn và tìm cách để tối ưu hóa một vấn đề phụ lớn hơn và sử dụng Memoization để ghi nhớ output của các vấn đề phụ đã được giải quyết	Các giải pháp được tổng hợp để thu về một giải pháp chung cho bài toán

Dynamic Programming	Divide and Conquer

Gồm có trình tự 4 bước: Đặc trưng hóa cấu trúc của lời giải tối ưu Định nghĩa giá trị của lời giải tối ưu một cách đệ quy Tính giá trị của các lời giải tối ưu Xây dựng lời giải tối ưu từ thông tin được tính toán	Giải quyết với 3 bước tại mỗi lần đệ quy: Chia bài toán thành các bài toán con Giải quyết các bài toán con Tổng hợp kết quả từ các bài toán con và tìm ra lời giải cho bài toán ban đầu
Giải các bài toán con và lưu trữ một kết quả trong bảng, không có sự trùng lặp	Giải các bài toán con và xét tất cả các trường hợp, có sự trùng lặp
Các bài toán con chồng chéo lên nhau (phụ thuộc nhau)	Các bài toán con độc lập nhau
hỗ trợ cả hai chiều (top-down và bottom-up)	chỉ xử lý bài toán theo chiều từ trên xuống (top-down)

VI. Công thức truy hồi trong Quy hoạch động:

Để có được công thức truy hồi trong quy hoạch động, chúng ta thường làm theo các bước sau:

- **Xác định các bài toán con**: *Xác định các bài toán con tạo thành bài toán lớn hơn.*
- **Xác định các trường hợp cơ sở**: *đó là các bài toán con nhỏ nhất có thể được giải trực tiếp mà không cần chia nhỏ thêm.*
- **Định nghĩa quan hệ truy hồi:** *Biểu thị lời giải cho một bài toán con lớn hơn dưới dạng các lời giải cho các bài toán con nhỏ hơn của nó. Đây là bước quan trọng trong quy hoạch động vì đó là nơi chúng ta có thể tiết kiệm đáng kể thời gian và không gian bằng cách tránh tính toán dư thừa.*
- **Xác định thứ tự và đánh giá:** *Đây có thể là cách tiếp cận từ dưới lên hoặc cách tiếp cận từ trên xuống. Trong cách tiếp cận từ dưới lên, chúng ta bắt đầu với các trường hợp cơ bản và tiến dần đến các bài toán con lớn hơn. Trong cách tiếp cận từ trên xuống, chúng ta bắt đầu với bài toán lớn hơn và giải đệ quy các bài toán con nhỏ hơn nếu cần.*