Phần 2

DSA

Cẩm Nang Toàn Tập

(Data Structures & Algorithms)

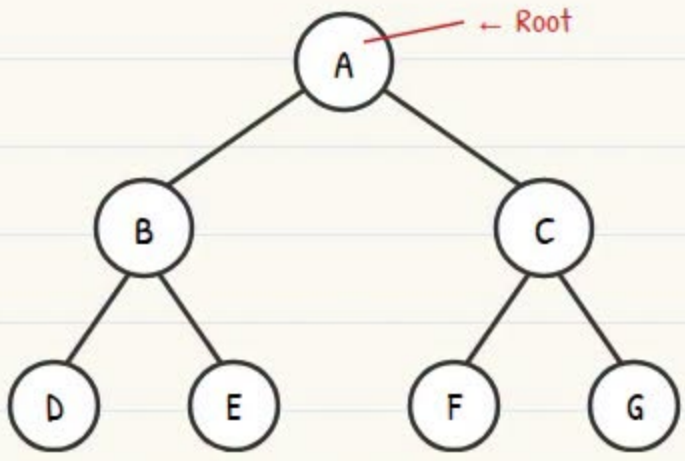
Từ Cơ Bản Đến Nâng Cao

Vuốt Để Xem Tiếp →

Tree Cơ Bản

① Tree Là Gì?

Tree là một cấu trúc dữ liệu phi tuyến tính (non-linear), gồm các node được nối với nhau bằng các cạnh (edges). Nó có cấu trúc phân cấp với một node gốc (Root) duy nhất và không hoặc nhiều cây con (subtree).



② Thuật Ngữ

- **Root**: Node trên cùng. (A)
- **Parent**: Node có một hoặc nhiều con. (A là parent của B và C)
- **Child**: Node được nối trực tiếp với parent của nó. (B và C là con của A)
- **Leaf Node**: Node không có con. (D, E, F, G)
- **Sibling**: Các node có cùng parent. (B và C là sibling)
- **Depth**: Số cạnh từ Root đến node đó. (Depth của E = 2)
- **Height**: Số cạnh trên đường đi dài nhất từ node xuống một leaf. (Height của A = 2)

③ Các Loại Tree

✓ **General Tree**: Một node có thể có bất kỳ số lượng con nào.

✓ **Binary Tree**: Một node có tối đa 2 con.

✓ **Binary Search Tree (BST)**: Với mọi node, left < node < right.

✓ **Full Binary Tree**: Mỗi node có 0 hoặc 2 con.

✓ **Complete Binary Tree**: Mọi tầng đều được lấp đầy hoàn toàn, trừ tầng cuối có thể chưa đầy.

✓ **Perfect Binary Tree**: Mọi tầng đều đầy và mọi leaf node đều ở cùng một tầng.

✓ **Balanced Binary Tree**: Chênh lệch chiều cao giữa cây con trái và phải ≤ 1 với mọi node.

④ Ứng Dụng Thực Tế

📁 File System trong OS (Directories & Subdirectories)

🌐 Cấu trúc HTML DOM (trang Web)

👥 Sơ đồ tổ chức (Employees & Managers)

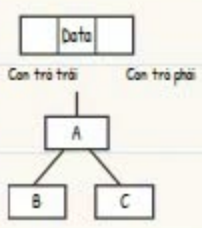
🗄️ Đánh chỉ mục Database (B-Trees, BST)

</> Thiết kế Compiler (Expression Trees)

🧠 AI & Ra quyết định (Game Trees)

⑤ Cách Biểu Diễn Tree

1) Linked Representation (Phổ biến)



✓ Kích thước động

✓ Sử dụng bộ nhớ hiệu quả

✓ Chèn/xóa dễ dàng

2) Array Representation (Chủ yếu cho Binary Tree)

Index: 0 1 2 3 4 5 6

A	B	C	D	E	F	G

Với node tại index i:

- Left Child → 2*i + 1
- Right Child → 2*i + 2
- Parent → (i - 1) / 2 (chia nguyên)

✓ Truy cập nhanh

✗ Lãng phí bộ nhớ ở cây thưa (sparse tree)

⭐ Điểm Cần Nhớ

Tree biểu diễn quan hệ phân cấp. Nắm vững thuật ngữ rất quan trọng – đây là nền tảng cho mọi thuật toán về tree!

⚠️ Lỗi Thường Gặp

✗ Nhầm lẫn giữa Depth và Height.

✗ Nghĩ rằng Leaf có thể có 1 con.

✗ Sai quan hệ parent/child khi biểu diễn bằng array.

💡 Thực Hành Tốt

✓ Vẽ tree ra để hiểu cấu trúc.

✓ Xác định Root, Leaf, và các tầng trước.

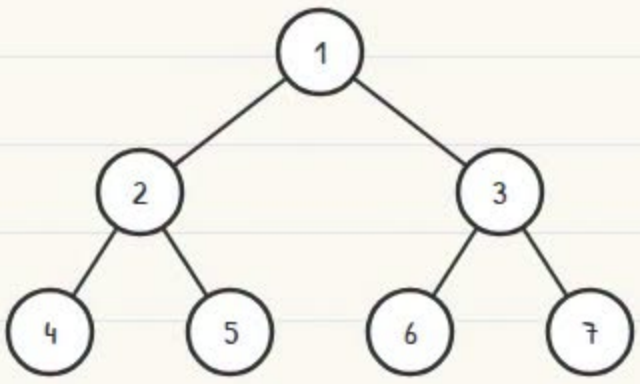
✓ Chọn cách biểu diễn phù hợp với use case.

🎯 Mẹo Phỏng Vấn

Câu hỏi về Tree rất phổ biến trong phỏng vấn. Hãy luyện tập traversal, height, balance, và các thao tác BST thường xuyên!

Duyệt Binary Tree

① Sơ Đồ Duyệt



Chú giải

L → Left R → Right N → Node

② Các Loại Duyệt

① Preorder (N – L – R)

Thăm Node → Left → Right

Thứ tự: 1, 2, 4, 5, 3, 6, 7

② Inorder (L – N – R)

Thăm Left → Node → Right

Thứ tự: 4, 2, 5, 1, 6, 3, 7

③ Postorder (L – R – N)

Thăm Left → Right → Node

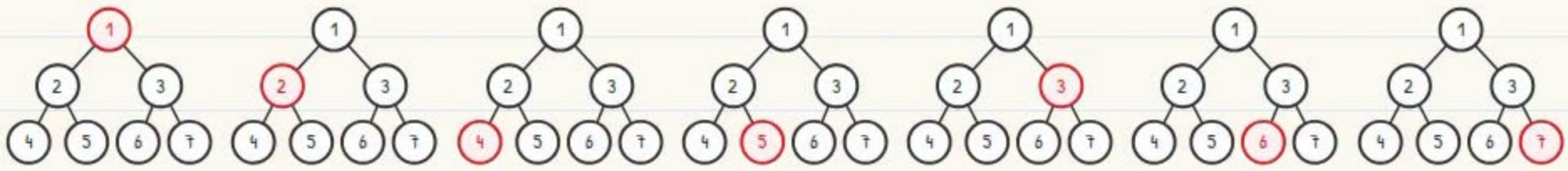
Thứ tự: 4, 5, 2, 6, 7, 3, 1

④ Level Order / BFS

Thăm từng tầng (trái sang phải)

Thứ tự: 1, 2, 3, 4, 5, 6, 7

③ Ví Dụ Từng Bước (Preorder)



④ Độ Phức Tạp

Duyệt	Time	Space
Preorder	O(n)	O(h)
Inorder	O(n)	O(h)
Postorder	O(n)	O(h)
Level Order (BFS)	O(n)	O(w)

n = số lượng node

h = chiều cao của tree

w = độ rộng tối đa của tree (số node nhiều nhất ở một tầng)

⭐ Ghi Chú Quan Trọng

- ✓ Mọi phép duyệt đều thăm mỗi node đúng một lần.
- ✓ Với tree cân bằng, h = O(log n)
- ✓ Trường hợp xấu nhất (tree lệch), h = O(n)
- ✓ Level Order dùng queue.

🎯 Mẹo Phỏng Vấn

Nếu đề bài yêu cầu xử lý node theo thứ tự cụ thể, hãy xác định phép duyệt phù hợp: Preorder, Inorder, Postorder hoặc Level Order.

⭐ Khi Nào Nên Dùng?

- **Preorder** → Copy tree, biểu thức Prefix
- **Inorder** → Xuất dữ liệu đã sắp xếp trong BST
- **Postorder** → Xóa tree, biểu thức Postfix
- **Level Order** → Đường đi ngắn nhất trong unweighted tree, xử lý theo tầng

🎯 Liên Hệ Thực Tế

Duyệt tree giống như các cách khám phá một tòa nhà khác nhau:

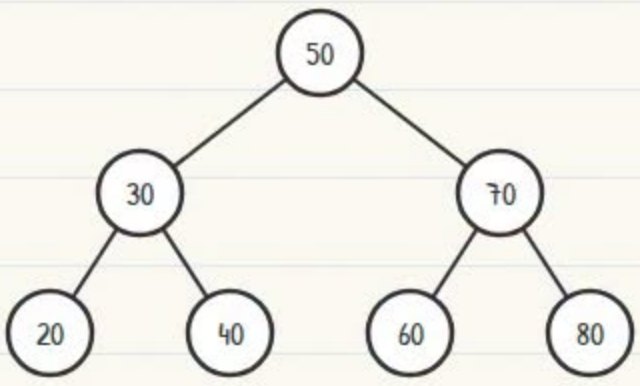
- **Preorder** → Vào tầng, rồi các phòng bên trái, rồi bên phải.
- **Inorder** → Khám phá phòng bên trái trước, rồi đến tầng, rồi bên phải.
- **Postorder** → Khám phá hết các phòng, rồi mới đánh dấu đã thăm tầng.
- **Level Order** → Khám phá từng tầng một.

Binary Search Tree (BST)

① Tính Chất BST

Với mọi node N:

- Mọi key trong cây con trái < N.key
- Mọi key trong cây con phải > N.key
- Không có key trùng lặp (trong BST chuẩn).
- Cây con trái và phải cũng là các BST.
- Duyệt Inorder của BST cho kết quả đã sắp xếp.



Inorder: 20, 30, 40, 50, 60, 70, 80

② Tìm Kiếm (Key = 60)

50 → 60 > 50 → Đi Phải

70 → 60 < 70 → Đi Trái

60 → Tìm thấy!

Pseudocode (Tìm kiếm)

```
while (node != null)
  if (key == node.key) return true;
  else if (key < node.key) node = node.left;
  else node = node.right;
return false;
```

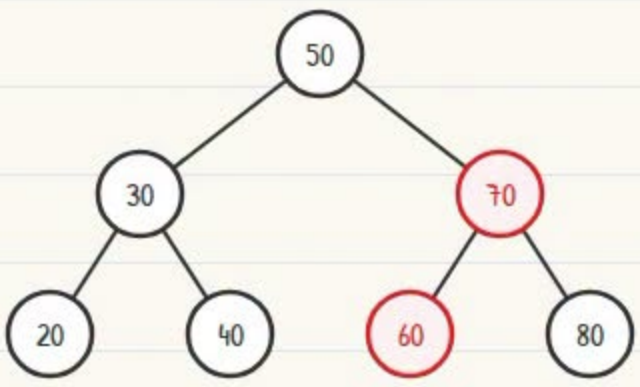
③ Chèn (Key = 65)

65 > 50 → Đi Phải

65 < 70 → Đi Trái

65 > 60 → Đi Phải

Chèn làm con phải của 60



④ Xóa (Key = 30)

Các trường hợp:

- ① Không con → Xóa node
- ② Một con → Thay bằng con đó
- ③ Hai con → Thay bằng Inorder Successor (nhỏ nhất trong cây con phải) hoặc Inorder Predecessor (lớn nhất trong cây con trái)

Ví dụ (Xóa 30)

Trước: 50(30,70) → 30(20,40)

Sau khi xóa 30: 50(40,70) → 40 nhận con 20

⑤ Ưu Điểm

✓ Search, Insert, Delete hiệu quả → Trung bình O(log n)

✓ Duyệt Inorder cho dữ liệu đã sắp xếp.

✓ Hữu ích trong hệ thống cần dữ liệu có thứ tự với thao tác nhanh.

✓ Nền tảng cho nhiều cấu trúc nâng cao (AVL Tree, Red-Black Tree, v.v.)

⑥ Độ Phức Tạp Thời Gian & Không Gian

Operation	Trung bình (BST cân bằng)	Xấu nhất (BST lệch)	Space
Search	O(log n)	O(n)	O(h)
Insert	O(log n)	O(n)	O(h)
Delete	O(log n)	O(n)	O(h)
Inorder Traversal	O(n)	O(n)	O(h)

Trong đó h = chiều cao của tree.

⭐ Điểm Cần Nhớ

BST duy trì thứ tự: Left < Root < Right, giúp thao tác nhanh khi cây cân bằng.

⚠️ Lỗi Thường Gặp

✗ Cho phép trùng lặp mà không có quy tắc.

✗ Không xử lý đủ 3 trường hợp khi xóa.

✗ Tạo cây lệch (BST suy biến).

💡 Thực Hành Tốt

✓ Giữ tree cân bằng.

✓ Dùng cách tiếp cận lặp (iterative) để tránh tràn stack.

✓ Kiểm tra tính hợp lệ của BST trước khi thao tác.

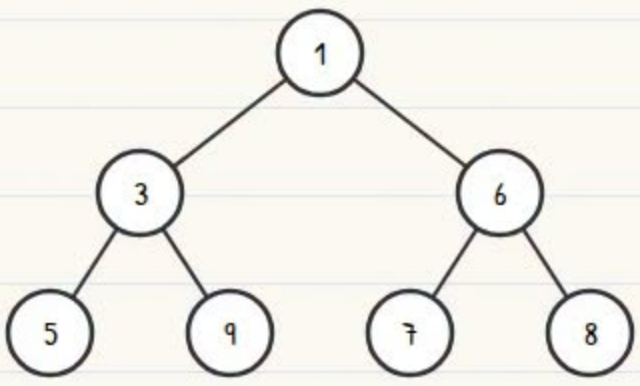
🎯 Mẹo Phỏng Vấn

Luôn xử lý cẩn thận trường hợp xóa node có 2 con – đây là trường hợp quan trọng nhất khi phỏng vấn!

Heap & Priority Queue

① Min Heap

- Complete Binary Tree.
- Giá trị mỗi node nhỏ hơn hoặc bằng con của nó.

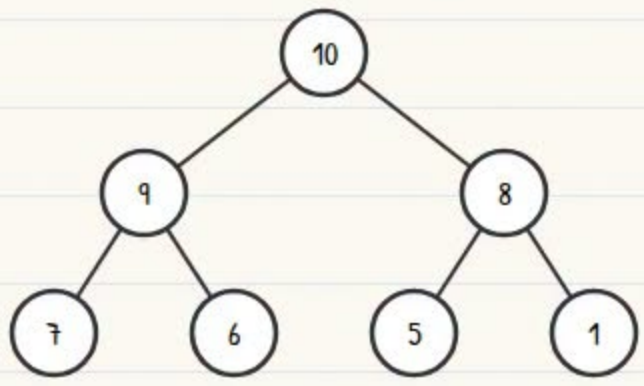


Array Representation (0-indexed)

1	3	6	5	9	7	8

② Max Heap

- Complete Binary Tree.
- Giá trị mỗi node lớn hơn hoặc bằng con của nó.



Array Representation (0-indexed)

10	9	8	7	6	5	1

③ Các Thao Tác Trên Heap

1) Insert (Push)

Thêm phần tử vào cuối. Heapify Up (Bubble Up) để giữ đúng tính chất heap.

2) Extract Top (Pop)

Xóa phần tử gốc. Đưa phần tử cuối lên gốc và Heapify Down.

3) Peek / Top

Trả về phần tử gốc mà không xóa nó.

4) Heapify (Up / Down)

Up: dùng sau khi insert. Down: dùng sau khi delete.

④ Priority Queue

- Một ADT mà mỗi phần tử có một priority.
- Các phần tử được lấy ra dựa trên priority (không theo thứ tự chèn vào).
- Có thể cài đặt bằng Heap.

Ví dụ (Min Priority Queue)

Insert: (5), (1), (3), (2)
Thứ tự lấy ra: 1 → 2 → 3 → 5

Lưu ý: Trong Min PQ, phần tử nhỏ nhất có priority cao nhất. Trong Max PQ, phần tử lớn nhất có priority cao nhất.

⑤ Ứng Dụng

📅 Task Scheduling (CPU / Processes)
📍 Dijkstra's Algorithm (Shortest Path)
→ Merge K Sorted Lists
🌲 Huffman Coding
⭐ Top K Elements (Stream / Logs / Data)

🌐 Liên Hệ Thực Tế

Hãy nghĩ đến phòng cấp cứu bệnh viện → bệnh nhân được điều trị theo mức độ ưu tiên (nguy cấp trước), không theo thứ tự đến.

⑥ Độ Phức Tạp

Operation	Min Heap	Max Heap	Space
Insert (Push)	O(log n)	O(log n)	O(n)
Extract Top (Pop)	O(log n)	O(log n)	O(n)
Peek / Top	O(1)	O(1)	O(n)
Heapify (Up/Down)	O(log n)	O(log n)	O(n)
Build Heap	O(n)	O(n)	O(n)

n = số phần tử trong heap

⭐ Điểm Cần Nhớ

Heap là Complete Binary Tree có tính chất heap. Min Heap → nhỏ nhất trên đỉnh. Max Heap → lớn nhất trên đỉnh. Priority Queue được xây trên nền Heap.

⚠️ Lỗi Thường Gặp

✗ Nhầm lẫn Min Heap với Max Heap.

✗ Quên heapify sau insert/delete.

✗ Không dùng công thức 0-index khi làm việc với array.

💡 Thực Hành Tốt

✓ Dùng cài đặt bằng array để tiết kiệm bộ nhớ.

✓ Luôn duy trì tính chất heap sau mỗi lần cập nhật.

✓ Dùng PQ cho các bài toán tối ưu hóa.

🎯 Mẹo Phỏng Vấn

Heap và Priority Queue thường xuất hiện trong bài toán scheduling, graph, và optimization!

Trie (Prefix Tree)

① Cấu Trúc

- Trie là cấu trúc dữ liệu dạng cây, dùng để lưu một tập hợp động các chuỗi.
- Mỗi node có tối đa 26 con (cho a-z).
- Kết thúc một từ được đánh dấu bằng một flag (End = true).

* Kết thúc từ (isEnd = true)
  ○ Node (isEnd = false)

② Chèn (Insert)

Chèn các từ: **cat, car, care, cart**

- Bắt đầu từ Root.
- Với mỗi ký tự: nếu con đã tồn tại → đi đến đó, nếu không → tạo node mới.
- Sau ký tự cuối, đánh dấu isEnd = true.

Pseudocode

```
node = root
for ch in word:
  idx = ch - 'a'
  if node.children[idx] is null:
    node.children[idx] = new Node()
  node = node.children[idx]
node.isEnd = true
```

③ Tìm Kiếm (Từ Chính Xác)

Tìm từ: **care**

- Duyệt qua từng ký tự.
- Nếu thiếu bất kỳ ký tự nào → từ không tồn tại.
- Sau ký tự cuối → kiểm tra isEnd.

care → Tìm thấy (isEnd = true)

④ Tìm Tiền Tố (Prefix Search)

Tiền tố: **ca**

- Duyệt đến ký tự cuối của tiền tố.
- Nếu đường đi tồn tại → tiền tố tồn tại.
- Không cần isEnd = true.

ca → Tồn tại (prefix exists)

⑤ Ứng Dụng

🔍 Autocomplete / Gợi ý tìm kiếm

🌐 Kiểm tra chính tả (Spell Checker)

🌍 IP Routing (Longest Prefix Match)

📖 Từ điển / Trò chơi chữ

🗜️ Nén dữ liệu

🔎 Tìm kiếm trong tập dữ liệu chuỗi lớn

Ví dụ

Words = {cat, car, care, cart}

Autocomplete cho "ca" → {cat, car, care, cart}

Autocomplete cho "car" → {car, care, cart}

⑥ Độ Phức Tạp

Operation	Time	Space
Insert	O(L)	O(N*L)
Search	O(L)	O(1)
Prefix Search	O(L)	O(1)

N = số từ được lưu, L = độ dài trung bình của từ

Lưu ý: Trie rất hiệu quả khi làm việc với nhiều chuỗi và các truy vấn dựa trên tiền tố.

⭐ Điểm Cần Nhớ

Trie lưu chuỗi dưới dạng cây. Rất mạnh cho các bài toán về tiền tố. Thời gian phụ thuộc độ dài từ (L), không phải tổng số từ (N).

⚠️ Lỗi Thường Gặp

✗ Đánh dấu isEnd không chính xác.

✗ Cấp phát lãng phí 26 con trỏ (nên dùng HashMap cho Trie thưa).

✗ Quên xử lý con trỏ null.

✅ Thực Hành Tốt

✓ Dùng array[26] cho bảng chữ nhỏ (a-z).

✓ Dùng HashMap cho tập ký tự lớn/thưa.

✓ Tái sử dụng node, tránh trùng lặp.

🎯 Mẹo Phỏng Vấn

Hãy thử các bài: Implement Trie, Word Search II, Longest Common Prefix, Replace Words.

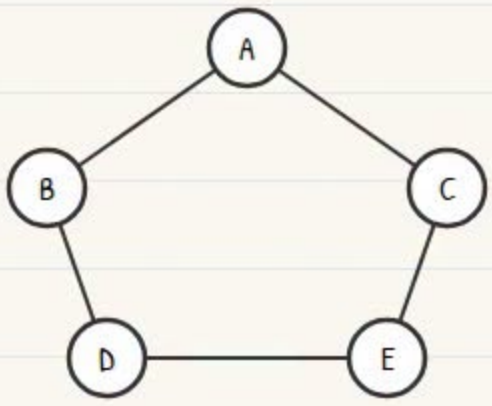
Graph Cơ Bản

① Vertices & Edges

Một Graph G = (V, E) gồm:

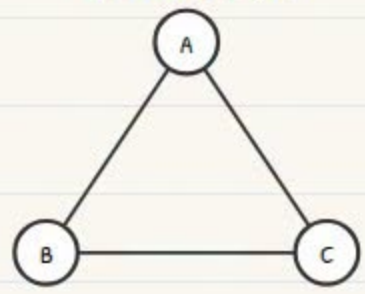
- V: Tập các vertex (node)
- E: Tập các edge (kết nối) giữa các cặp vertex.

Thứ tự (u, v) trong một edge có thể quan trọng hoặc không, tùy loại graph.



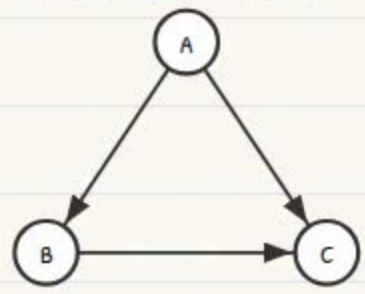
② Directed vs Undirected

Undirected Graph



Edge không có hướng.
(A, B) giống (B, A)

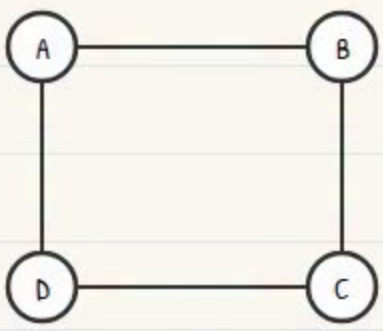
Directed Graph (Digraph)



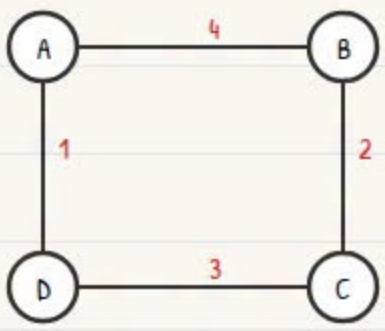
Edge có hướng.
(A, B) khác (B, A)

③ Weighted vs Unweighted

Unweighted Graph



Weighted Graph



Mọi edge có trọng số bằng nhau (thường coi là 1). Edge có trọng số/chi phí, dùng biểu diễn khoảng cách, chi phí, dung lượng, v.v.

④ Cách Biểu Diễn Graph

1) Adjacency List

Vertex	Neighbors
A	B, C
B	A, D
C	A, D, E
D	B, C, E
E	C, D

2) Adjacency Matrix

	A	B	C	D	E
A	0	1	1	0	0
B	1	0	0	1	0
C	1	0	0	1	1
D	0	1	1	0	1
E	0	0	1	1	0

0 → Không có edge. 1 → Có edge (với đồ thị không trọng số). Với đồ thị có trọng số, lưu trọng số thay vì 1.

⑤ Ứng Dụng Của Graph

- 📍 Mạng lưới đường / Bản đồ (Shortest Path)
- 👥 Mạng xã hội (Người dùng & kết nối)
- 🌐 Trang Web (Liên kết giữa các trang)
- 🖥️ Mạng máy tính (Router & kết nối)
- 📅 Lập lịch dự án (Dependencies / DAG)
- 🔀 Hệ thống gợi ý (User-item graph)

⭐ Điểm Cần Nhớ

- ✓ Graph mô hình hóa các quan hệ theo cặp.
- ✓ Hiểu rõ loại graph trước khi áp dụng thuật toán.
- ✓ Chọn cách biểu diễn dựa trên mật độ của graph.

⚠️ Lỗi Thường Gặp

✗ Nhầm lẫn directed và undirected.

✗ Bỏ qua trọng số edge trong weighted graph.

✗ Dùng adjacency matrix cho graph quá thưa.

💡 Thực Hành Tốt

✓ Dùng Adjacency List cho graph thưa (E << V^2).

✓ Dùng Adjacency Matrix cho graph dày hoặc cần tra cứu edge nhanh.

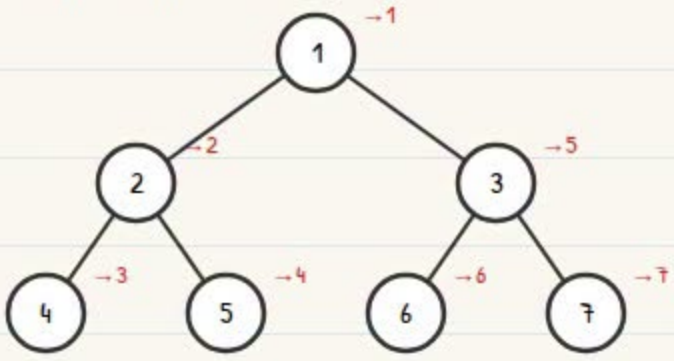
🎯 Mẹo Phỏng Vấn

Hãy sẵn sàng chuyển đổi giữa các cách biểu diễn và xác định phương pháp phù hợp cho từng bài toán graph khác nhau!

Duyệt Graph (DFS & BFS)

① DFS (Depth First Search)

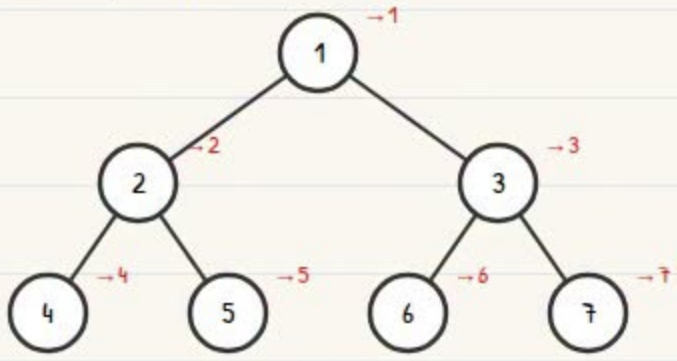
- Đi sâu nhất có thể theo một nhánh trước khi quay lui.
- Dùng Stack (hoặc Recursion).
- Time Complexity: O(V + E)
- Space Complexity: O(V) (cho recursion stack)



DFS Order (bắt đầu từ 1): 1 → 2 → 4 → 5 → 3 → 6 → 7

② BFS (Breadth First Search)

- Duyệt hết các neighbor ở tầng hiện tại trước khi qua tầng kế tiếp.
- Dùng Queue.
- Time Complexity: O(V + E)
- Space Complexity: O(V) (cho queue)



BFS Order (bắt đầu từ 1): 1 → 2 → 3 → 4 → 5 → 6 → 7

③ Queue vs Stack

DFS dùng STACK (LIFO)
Last In First Out
Đi sâu trước
Cài đặt bằng stack hoặc recursion

BFS dùng QUEUE (FIFO)
First In First Out
Duyệt theo từng tầng
Cài đặt bằng queue

④ Ví Dụ Duyệt (Bắt Đầu = 1)

Adjacency List:

1: [2, 3] 2: [4, 5] 3: [6, 7] 4,5,6,7: []

DFS Order: 1 → 2 → 4 → 5 → 3 → 6 → 7

BFS Order: 1 → 2 → 3 → 4 → 5 → 6 → 7

⑤ Tóm Tắt Độ Phức Tạp

Duyệt	Time	Space	Cấu trúc dùng
DFS	O(V + E)	O(V)	Stack / Recursion
BFS	O(V + E)	O(V)	Queue

V = số Vertex (Node), E = số Edge

⑥ Khi Nào Nên Dùng?

Dùng DFS khi:

✓ Cần khám phá đường đi sâu (backtracking, maze, n-queens).
✓ Phát hiện chu trình trong directed graph.
✓ Kiểm tra tính liên thông.
✓ Topological Sort (dùng stack).

Dùng BFS khi:

✓ Cần đường đi ngắn nhất trong unweighted graph.
✓ Duyệt theo tầng.
✓ Tìm số bước tối thiểu.
✓ Web crawling / broadcasting.

⭐ Điểm Cần Nhớ

DFS đi sâu bằng stack. BFS đi rộng bằng queue. Cả hai đều thăm mọi node và edge đúng một lần.

⚠️ Lỗi Thường Gặp

✗ Không đánh dấu node đã thăm → gây lặp vô hạn.
✗ Nhầm lẫn DFS (stack) với BFS (queue).
✗ Sai thứ tự đẩy các neighbor vào.

💡 Thực Hành Tốt

✓ Luôn dùng visited[] để tránh thăm lại.
✓ Với DFS, đánh dấu visited khi đi sâu.
✓ Với BFS, đánh dấu visited khi đưa vào queue.

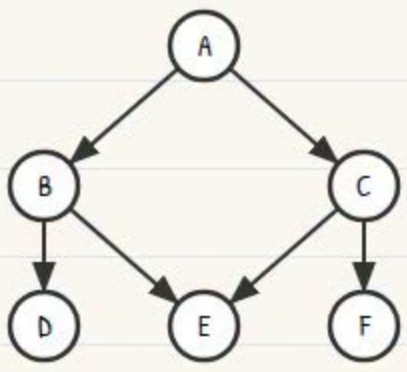
🎯 Mẹo Phỏng Vấn

Hãy nắm rõ sự khác biệt giữa DFS và BFS, khi nào nên dùng cái nào. Luyện tập trên nhiều dạng graph khác nhau!

Topological Sort

① DAG (Directed Acyclic Graph)

- Topological Sort chỉ được định nghĩa cho DAG.
- Là một thứ tự tuyến tính của các vertex sao cho với mọi edge có hướng u → v, u đứng trước v.
- Hữu ích trong lập lịch, sắp xếp các task có phụ thuộc.



Một thứ tự Topological hợp lệ: A, B, C, D, E, F

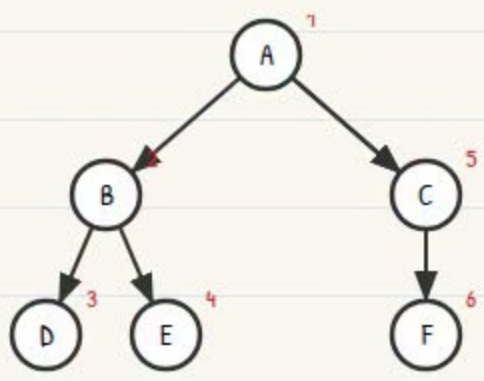
② Kahn's Algorithm (dựa trên BFS)

1. Tính in-degree của mọi node.
2. Đưa mọi node có in-degree 0 vào queue.
3. Trong khi queue không rỗng: lấy 1 node, thêm vào kết quả, giảm in-degree các neighbor, nếu in-degree về 0 thì thêm vào queue.
4. Nếu kích thước kết quả == V → hợp lệ, ngược lại → có chu trình.

```
compute inDegree[]
queue all nodes with inDegree 0
while queue not empty:
  node = queue.pop()
  add node to result
  for each neighbor n of node:
    inDegree[n]--
    if inDegree[n] == 0: queue.push(n)
if result size != V: return "Cycle Exists"
```

③ Phương Pháp DFS (Reverse Postorder)

1. Chạy DFS từ mọi node chưa thăm.
2. Sau khi thăm hết neighbor, đẩy node vào stack.
3. Đảo ngược stack → Topological Order.




```
for each node:
  if not visited: DFS(node)
// trong DFS
for each neighbor:
  if not visited: DFS(neighbor)
push node to stack
reverse(stack) → result
```

④ Ứng Dụng

📋 Lập lịch Task (Task có ràng buộc thứ tự)

</> Build System (Biên dịch source file)

🎓 Lập kế hoạch môn học tiên quyết

🗄️ Database Schema Migrations

⚙️ Giải quyết Dependency trong package

🖥️ Instruction Scheduling trong CPU

⑤ Độ Phức Tạp

Algorithm	Time	Space
Kahn's Algorithm	O(V + E)	O(V)
DFS Method	O(V + E)	O(V)

V = số Vertex (Node), E = số Edge

⑥ Ghi Chú Quan Trọng

✓ Topological Sort chỉ khả thi với DAG.

✓ Nếu có chu trình → không có thứ tự hợp lệ.

✓ Thứ tự không duy nhất; có thể có nhiều thứ tự hợp lệ.

✓ Kahn's (BFS) có tính lặp, DFS có tính đệ quy.

Ghi Nhớ

BFS (Kahn's) → Tốt để phát hiện chu trình dễ dàng.

DFS → Đơn giản và gọn với stack.

⭐ Điểm Cần Nhớ

Topological Sort cho ra thứ tự tuyến tính của DAG. Mọi dependency (u → v) đều được tôn trọng.

⚠️ Lỗi Thường Gặp

✗ Áp dụng Topological Sort trên graph có chu trình.

✗ Quên kiểm tra result size == V.

✗ Cập nhật sai in-degree trong Kahn's Algo.

💡 Thực Hành Tốt

✓ Dùng Kahn's Algorithm cho graph lớn.

✓ Dùng DFS khi ưu tiên đệ quy.

✓ Luôn xác nhận là DAG trước khi Topological Sort.

🎯 Mẹo Phỏng Vấn

Hãy sẵn sàng cài đặt cả Kahn's (BFS) và DFS Topological Sort. Giải thích cách phát hiện chu trình trong directed graph!

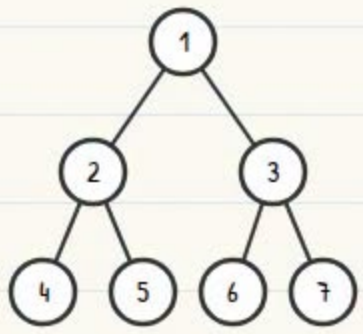
Union Find (Disjoint Set Union)

① Union Find Là Gì?

- Union Find (DSU) là cấu trúc dữ liệu theo dõi một tập hợp phần tử được chia thành các tập con rời nhau (không giao nhau).
- Hỗ trợ hiệu quả hai thao tác chính: Find (tìm đại diện của tập) và Union (gộp hai tập).

② Find Operation

Find(x) trả về đại diện (root) của tập chứa x.

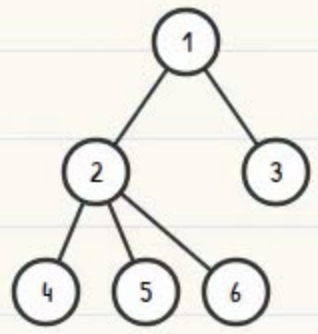


Ví dụ: Find(5) → 1 Find(7) → 1

③ Union Operation

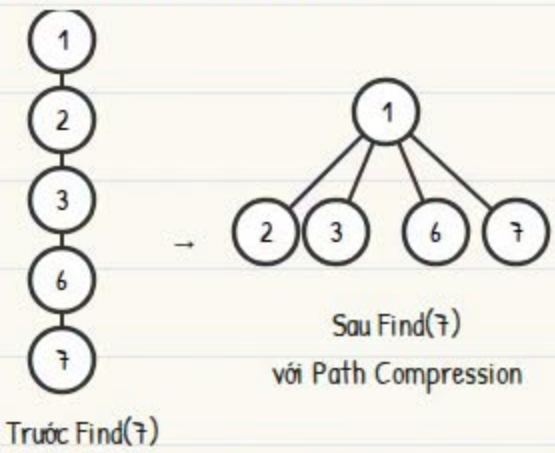
Union(x, y) gộp các tập chứa x và y.

Ví dụ: Trước Union(5, 6) → Sau Union(5, 6)



④ Path Compression

Trong lúc Find(x), cho mọi node trên đường đi trỏ thẳng đến root. Giúp giảm mạnh chiều cao của tree.



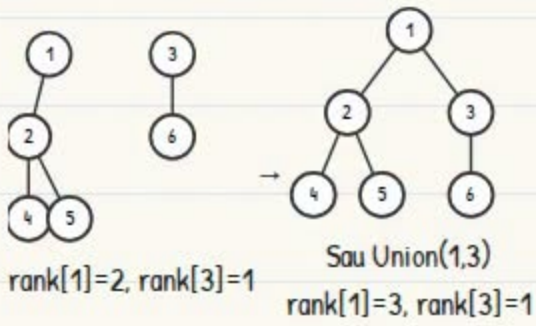
Trước Find(7)

Sau Find(7) với Path Compression

Lưu ý: Các thao tác sau sẽ nhanh hơn nhiều khi tree trở nên phẳng hơn.

⑤ Union By Rank (hoặc Size)

Luôn gắn tree nhỏ hơn dưới root của tree lớn hơn (theo rank/size). Giữ chiều cao tree nhỏ.



rank[1]=2, rank[3]=1

Sau Union(1,3) rank[1]=3, rank[3]=1

⑥ Ứng Dụng

- 🔺 Kruskal's Minimum Spanning Tree
- 🔗 Phát hiện chu trình trong Undirected Graph
- 👥 Connected Components trong Graph
- 🌐 Dynamic Connectivity (truy vấn online)
- 🖼️ Xử lý ảnh (gộp vùng)
- 📊 Phân tích Mạng / Mạng xã hội

⑦ Độ Phức Tạp

Operation	Không heuristic	Với Path Compression + Union by Rank	Amortized
Find(x)	O(N)	α(N)	O(α(N))
Union(x, y)	O(N)	α(N)	O(α(N))

N = số phần tử. α(N) = hàm Ackermann nghịch đảo (≤ 4 với mọi N thực tế)

💻 Cài Đặt Điển Hình (Arrays)

```
int parent[N], rank[N];
// Initialize
for (int i = 1; i <= N; i++) {
  parent[i] = i; // mỗi node tự làm parent
  rank[i] = 0; // rank (xấp xỉ chiều cao)
}
```

parent[i] → parent của i rank[i] → rank/chiều cao α(N) → rất nhỏ (gần như hằng số)

⭐ Điểm Cần Nhớ

Union Find quản lý hiệu quả các tập rời nhau. Path Compression + Union by Rank cho thời gian gần hằng số.

⚠️ Lỗi Thường Gặp

✗ Quên Path Compression → tree quá cao.

✗ Không dùng union by rank/size → tree mất cân bằng.

✗ Khởi tạo sai mảng parent/rank.

💡 Thực Hành Tốt

✓ Luôn dùng cả Path Compression và Union by Rank.

✓ Dùng Find() dạng lặp để tránh tràn recursion stack với N lớn.

✓ Giữ array 1-indexed để dễ cài đặt.

🎯 Mẹo Phỏng Vấn

Hiểu vì sao dùng α(N) và luyện cài đặt DSU từ đầu. Thường được hỏi trong bài MST, phát hiện chu trình và connectivity!

Thuật Toán Greedy

① Greedy Là Gì?

- Greedy là một chiến lược thuật toán xây dựng lời giải từng bước, luôn chọn phương án tốt nhất hiện tại.
- Không quay lui (backtrack), đưa ra lựa chọn tối ưu cục bộ với hy vọng đạt tối ưu toàn cục.

💡 Ý Tưởng Chính

Chọn phương án tốt nhất ngay bây giờ và tiến về phía trước.

② Greedy Choice Property

- Một bài toán có tính chất Greedy Choice nếu tối ưu toàn cục có thể đạt được bằng cách đưa ra lựa chọn tối ưu cục bộ.
- Nghĩa là lựa chọn hiện tại không ảnh hưởng đến tính tối ưu của phần lời giải còn lại.

Ví dụ: Bài toán Activity Selection

Chọn activity có thời gian kết thúc sớm nhất (tối ưu cục bộ). Nó để lại nhiều thời gian nhất cho các activity còn lại → tối ưu toàn cục.

③ Các Bài Toán Thường Giải Bằng Greedy

📅 Activity Selection

🎒 Fractional Knapsack

💰 Huffman Coding

📍 Dijkstra's Shortest Path (dùng min-heap)

🌲 Minimum Spanning Tree (Prim's, Kruskal's)

📋 Job Sequencing With Deadlines

🪙 Bài toán đổi tiền (hệ thống chuẩn)

Ví dụ: Activity Selection

Cho các activity đã sắp xếp theo thời gian kết thúc:

Activity	Start	End
A1	1	3
A2	2	5
A3	4	6
A4	6	7
A5	5	8
A6	8	9

Chiến lược Greedy: Luôn chọn activity có thời gian kết thúc sớm nhất mà tương thích với activity đã chọn trước đó.

Activities đã chọn: A1 → A3 → A4 → A6

④ Ưu Điểm

✓ Đơn giản và dễ cài đặt.

✓ Hiệu quả – thường nhanh hơn các cách tiếp cận khác.

✓ Dùng ít bộ nhớ hơn (không backtracking).

✓ Hoạt động tốt cho các bài toán optimization.

⑤ Hạn Chế

✗ Không phải lúc nào cũng cho lời giải tối ưu.

✗ Bài toán phải thỏa Greedy Choice Property và Optimal Substructure.

✗ Không phù hợp với bài toán có overlapping subproblems.

⭐ Điểm Cần Nhớ

Greedy hoạt động khi lựa chọn tốt nhất cục bộ dẫn đến lời giải tốt nhất toàn cục. Luôn kiểm chứng greedy choice property trước khi áp dụng.

⚠️ Lỗi Thường Gặp

✗ Áp dụng greedy cho bài toán không phù hợp.

✗ Không chứng minh greedy choice property.

✗ Bỏ qua các edge case.

💡 Thực Hành Tốt

✓ Cố gắng chứng minh tính đúng đắn (Greedy Choice + Optimal Substructure).

✓ Kiểm thử trên ví dụ cụ thể.

✓ Dùng đúng cấu trúc dữ liệu (sort, heap, priority queue).

🎯 Mẹo Phỏng Vấn

Nếu được hỏi "Bài này có giải được bằng Greedy không?", trước tiên hãy kiểm tra: Có thể đưa ra lựa chọn tốt nhất cục bộ không? Lựa chọn đó có dẫn đến tối ưu toàn cục không?

Dynamic Programming (DP)

① Dynamic Programming Là Gì?

- DP là kỹ thuật giải bài toán bằng cách chia nhỏ thành các subproblem chồng lấp (overlapping) và lưu lại kết quả của chúng.
- Chủ yếu dùng cho bài toán optimization khi brute force / recursion trở nên quá chậm.

💡 Ý Tưởng Chính

Giải mỗi subproblem một lần và tái sử dụng kết quả (tránh tính toán lại).

② Recursion vs DP

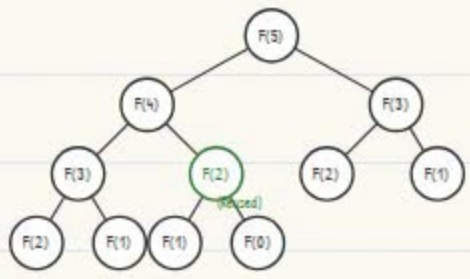
Recursion (Naïve)	Dynamic Programming
Tính lại subproblem nhiều lần.	Lưu kết quả subproblem.
Thời gian mũ (exponential).	Thời gian đa thức (polynomial).
Độ phức tạp cao.	Độ phức tạp thấp.
Không lưu kết quả.	Dùng Memoization hoặc Tabulation.

③ Memoization (Top-Down)

Giải bằng recursion + cache (memo). Lưu kết quả trong array/map.

Ví dụ: Fibonacci (n=5)

F(n) = F(n-1) + F(n-2)



④ Tabulation (Bottom-Up)

- Giải lặp (iteratively) từ base case đến kết quả cuối.
- Dùng bảng (thường là array) để lưu kết quả.

Ví dụ: Fibonacci (n=5) – F(n) = F(n-1) + F(n-2)

n	0	1	2	3	4	5
F(n)	0	1	1	2	3	5

Thứ tự tính: 0 → 1 → 2 → 3 → 4 → 5

⑤ State & Transition

1) State: Xác định ý nghĩa của một phần tử DP.

Ví dụ: dp[i] → đáp án cho subproblem kích thước i.

2) Transition: Quan hệ / công thức để tính state hiện tại từ các state nhỏ hơn.

Ví dụ Fibonacci: dp[i] = dp[i-1] + dp[i-2] (i ≥ 2)

Ghi Nhớ: Xác định state rõ ràng + transition đúng = lời giải DP thành công.

⑥ Các Bài Toán DP Thường Gặp

- ⭐ Fibonacci Series
- ⭐ Climbing Stairs
- ⭐ 0/1 Knapsack
- ⭐ Coin Change (Min Coins)
- ⭐ Longest Common Subsequence (LCS)
- ⭐ Longest Increasing Subsequence (LIS)
- ⭐ Matrix Chain Multiplication
- ⭐ Edit Distance

🌐 Liên Hệ Thực Tế

DP giống như leo núi. Thay vì khám phá lại từng đường đi nhiều lần, ta ghi nhớ độ cao đã đạt được ở mỗi điểm và chọn đường đi tốt nhất tiếp theo.

⑦ So Sánh Độ Phức Tạp

Cách tiếp cận	Time	Space	Ghi chú
Naïve Recursion	Exponential (vd: O(2^n))	O(n)	Tính lại subproblem
Memoization (Top-Down)	O(N · States)	O(N · States)	Dùng recursion + cache
Tabulation (Bottom-Up)	O(N · States)	O(N · States)	Lặp, an toàn stack

N = kích thước input, States = số subproblem duy nhất

⭐ Điểm Cần Nhớ

DP tránh tính toán lại. Xác định Overlapping Subproblems. Định nghĩa State rõ ràng. Viết đúng Transition. Chọn Memoization hoặc Tabulation hợp lý.

⚠️ Lỗi Thường Gặp

✗ Định nghĩa sai state.

✗ Sai công thức transition.

✗ Quên base case.

✗ Dùng recursion khi lặp đã đủ (nguy cơ TLE).

💡 Thực Hành Tốt

✓ Vẽ recursion tree trước.

✓ Quan sát overlapping trong tree.

✓ Viết rõ state & transition.

✓ Bắt đầu với memoization cho rõ ràng, sau đó tối ưu bằng tabulation.

🎯 Mẹo Phỏng Vấn

Giải thích tư duy của bạn: 1. Định nghĩa state 2. Viết relation (transition) 3. Xử lý base case 4. Chọn Top-Down hay Bottom-Up

Backtracking

① Backtracking Là Gì?

- Backtracking là kỹ thuật giải bài toán bằng cách thử mọi lời giải khả dĩ và bỏ (backtrack) khỏi những hướng đi không dẫn đến lời giải hợp lệ.
- Khám phá lựa chọn từng bước và hoàn tác lựa chọn cuối khi gặp ngõ cụt.

💡 Ý Tưởng Chính

Thử → Khám phá → Backtrack → Thử lựa chọn khác.

② Decision Tree

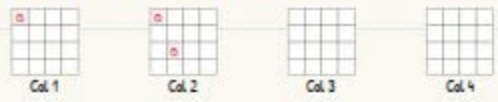
Biểu diễn mọi lựa chọn (state) và kết quả có thể.

✓ = Lời giải hợp lệ ✗ = Không hợp lệ / Ngõ cụt

③ State Space Tree

Mỗi node là một state. Cạnh là các lựa chọn. Độ sâu = số quyết định đã đưa ra.

Ví dụ: N-Queens (N = 4)



Lưu ý: Nhiều nhánh bị loại bỏ sớm khi chúng vi phạm ràng buộc (không tấn công nhau).

④ Backtracking Hoạt Động Thế Nào?

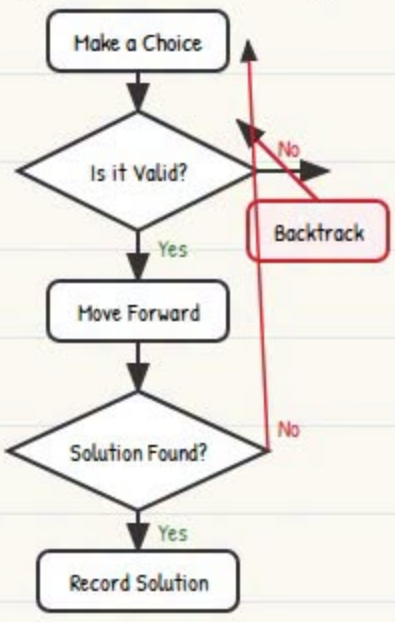
✓ Chọn một candidate.

✓ Kiểm tra tính hợp lệ.

✓ Nếu hợp lệ → thực hiện lựa chọn và tiến lên.

✓ Nếu dẫn đến ngõ cụt → hoàn tác lựa chọn (backtrack) và thử candidate tiếp theo.

✓ Lặp lại đến khi khám phá hết mọi khả năng.



⑤ Ứng Dụng / Bài Toán Thường Gặp

♛ Bài toán N-Queens

🧩 Giải Sudoku

🔗 Hamiltonian Path / Cycle

{ } Sinh mọi Subset / Permutation

🌀 Giải Maze

🔤 Tìm từ trong lưới (Word Search)

💰 Subset Sum / Bài toán Partition

🎨 Tô màu Graph (Graph Coloring)

🌐 Liên Hệ Thực Tế

Thử một con đường, nếu gặp ngõ cụt, quay lại và thử đường khác.

⑥ Phân Tích Độ Phức Tạp

Backtracking khám phá state space trong trường hợp xấu nhất. Nếu branching factor = b và độ sâu tối đa = d → Số state = O(b^d) (có tính mũ).

Tham số	Complexity
Time Complexity	O(b^d) (Xấu nhất)
Space Complexity	O(d) (Recursion Stack)

b = branching factor, d = độ sâu recursion (số quyết định)

⑦ Điểm Cần Nhớ

✓ Backtracking xây lời giải từng bước.

✓ Loại bỏ các lựa chọn không dẫn đến lời giải hợp lệ.

✓ Dùng DFS kèm pruning.

✓ Hữu ích cho bài toán combinatorial (tổ hợp).

✓ Không phù hợp cho không gian tìm kiếm quá lớn nếu thiếu pruning/tối ưu.

Mẹo giảm không gian tìm kiếm

✓ Pruning (loại bỏ sớm) ✓ Thứ tự lựa chọn hợp lý ✓ Kiểm tra ràng buộc sớm ✓ Dùng heuristic

💡 Thực Hành Tốt

✓ Luôn backtrack sau khi return từ recursion.

✓ Giữ code sạch và module hóa.

✓ Trực quan hóa decision tree.

⚠️ Lỗi Thường Gặp

✗ Không hoàn tác lựa chọn khi backtrack.

✗ Thiếu base case hoặc điều kiện dừng.

✗ Không pruning sớm, gây TLE.

⭐ Ghi Nhớ

Backtracking = DFS + Undo Choice

Explore → Check → Recurse → Backtrack (Undo)

🎯 Mẹo Hay

Kết hợp Backtracking với memoization (trong một số bài) hoặc bitmasking để tăng hiệu năng.

Chiến Lược Divide & Conquer

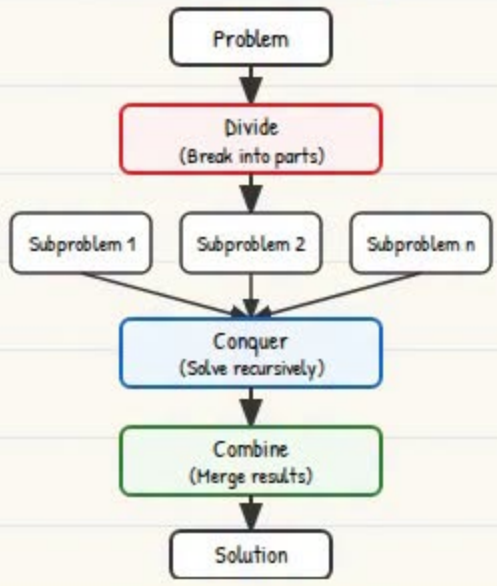
① Divide & Conquer Là Gì?

- Chia bài toán thành các subproblem nhỏ hơn cùng loại.
- Giải các subproblem bằng đệ quy.
- Kết hợp kết quả để có lời giải cuối cùng.

💡 Ý Tưởng Chính

Divide → Conquer → Combine
(Chia → Trị → Kết hợp)

Quy Trình Chung



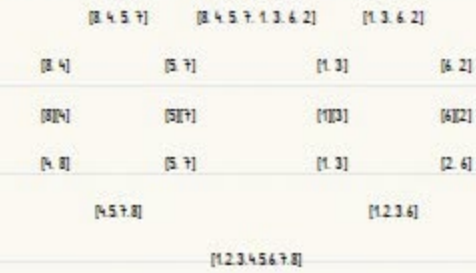
② Merge Sort

Divide: Chia array thành 2 nửa.

Conquer: Sắp xếp đệ quy 2 nửa.

Combine: Merge 2 nửa đã sắp xếp.

Ví dụ: Arr = [8, 4, 5, 7, 1, 3, 6, 2]



③ Quick Sort

Divide: Chọn một pivot. Phân hoạch (partition) array thành: nhỏ hơn pivot, bằng pivot, lớn hơn pivot.

Conquer: Sắp xếp đệ quy phần trái và phải.

Ví dụ: Arr = [10, 7, 8, 9, 1, 5], Pivot = 7

[10,7,8,9,1,5] Pivot=7 → [1,5] [7] [10,8,9]
[1,5] → P=5 → [1][5] [10,8,9] → P=8 → [8][9,10]... →
Kết quả: 1,5,7,8,9,10

④ Binary Search

Hoạt động trên array đã sắp xếp. So sánh phần tử giữa với target và loại bỏ nửa không gian tìm kiếm.

Ví dụ: Arr = [2, 4, 6, 8, 10, 12, 14], Target = 10

2	4	6	8	10	12	14

10 > 8 → tìm nửa phải → Mid=5 → 10 < 12 → tìm nửa trái → Found!

⑤ Ưu Điểm

✓ Giảm đáng kể độ phức tạp thời gian.

✓ Tận dụng xử lý đa lõi/song song.

✓ Thường cho lời giải thanh lịch và hiệu quả.

✓ Phù hợp với bài toán có optimal substructure.

⑥ Độ Phức Tạp

Algorithm	Trung bình	Xấu nhất	Space
Merge Sort	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	O(n^2)	O(log n) tb
Binary Search	O(log n)	O(log n)	O(1)

n = số phần tử

⑦ Khi Nào Nên Dùng?

🗄️ Dữ liệu lớn

📊 Bài toán có subproblem rõ ràng

⏱️ Khi brute force quá tốn kém

🖥️ Bài toán có thể song song hóa

⭐ Điểm Cần Nhớ

Chia bài toán, giải các subproblem, và kết hợp kết quả để có lời giải tốt nhất.

⚠️ Lỗi Thường Gặp

✗ Không xử lý base case đúng cách.

✗ Kết hợp kết quả không chính xác.

✗ Chọn pivot kém trong Quick Sort (dẫn đến trường hợp xấu nhất).

💡 Thực Hành Tốt

✓ Luôn định nghĩa rõ base case.

✓ Chọn pivot tốt (ngẫu nhiên hoặc median-of-three).

✓ Phân tích độ phức tạp thời gian & không gian.

🎯 Mẹo Phỏng Vấn

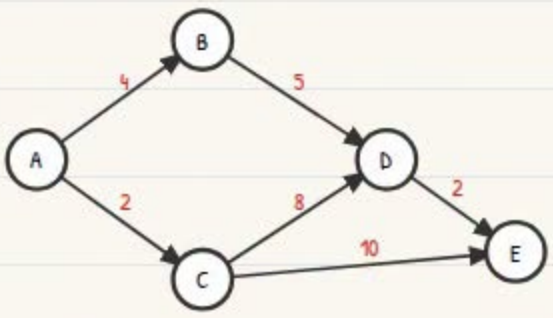
Hãy sẵn sàng viết code cho Merge Sort, Quick Sort, và Binary Search. Hiểu rõ các recurrence relation!

Thuật Toán Shortest Path

① Dijkstra's Algorithm

- Tìm đường đi ngắn nhất từ một nguồn (source) đến mọi vertex khác.
- Chỉ hoạt động với trọng số edge không âm.
- Cách tiếp cận Greedy dùng min-priority queue.

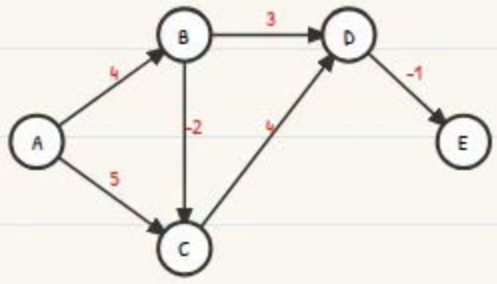
Ví dụ (Source = A)



Time: O((V+E) log V) (dùng Min Priority Queue)

② Bellman-Ford Algorithm

- Tìm đường đi ngắn nhất từ một nguồn.
- Hoạt động với trọng số âm.
- Có thể phát hiện chu trình trọng số âm.
- Relax mọi edge |V| - 1 lần.



Time: O(V × E) Space: O(V)

③ Floyd-Warshall Algorithm

- Tìm đường đi ngắn nhất giữa mọi cặp vertex.
- Hoạt động với trọng số edge âm.
- Không xử lý được chu trình âm.
- Dùng Dynamic Programming.

	A	B	C	D
A	0	3	7	2
B	5	0	6	1
C	8	11	0	4
D	6	9	10	0

Time: O(V^3) Space: O(V^2)

④ So Sánh

Feature	Dijkstra	Bellman-Ford	Floyd-Warshall
Source	Single Source	Single Source	All Pairs
Trọng số Edge	Không âm	Cho phép âm	Cho phép âm
Phát hiện chu trình âm	Không	Có	Không
Cách tiếp cận	Greedy + PQ	Relaxation	Dynamic Programming
Time	O((V+E) log V)	O(V × E)	O(V^3)
Space	O(V)	O(V)	O(V^2)
Phù hợp cho	Graph thưa, trọng số không âm	Graph có trọng số âm hoặc cần check chu trình	Graph dày, all-pairs

V = số vertex, E = số edge

⑤ Ứng Dụng

📍 Điều hướng GPS & định tuyến bản đồ

Tìm tuyến đường nhanh nhất giữa 2 địa điểm.

🖥️ Định tuyến mạng

Tìm đường đi tối ưu trong mạng máy tính.

💰 Tài chính & Arbitrage

Phát hiện chu trình âm cho cơ hội chênh lệch giá.

👥 Mạng xã hội

Tìm kết nối ngắn nhất giữa người dùng.

📅 Lập lịch dự án

Tìm critical path và dependencies.

⭐ Điểm Cần Nhớ

Dijkstra nhanh nhất cho trọng số không âm. Bellman-Ford phát hiện chu trình âm. Floyd-Warshall cho kết quả đường đi ngắn nhất mọi cặp.

⚠️ Lỗi Thường Gặp

✗ Dùng Dijkstra với trọng số âm.

✗ Quên kiểm tra chu trình âm trong Bellman-Ford.

✗ Khởi tạo sai bảng DP trong Floyd-Warshall.

💡 Thực Hành Tốt

✓ Dùng thuật toán phù hợp theo ràng buộc bài toán.

✓ Với graph thưa, ưu tiên Dijkstra.

✓ Với graph dày hoặc cần all-pairs, dùng Floyd-Warshall.

🎯 Mẹo Phỏng Vấn

Luôn hỏi: Có trọng số âm không? Cần single source hay all pairs? Có cần phát hiện chu trình không?

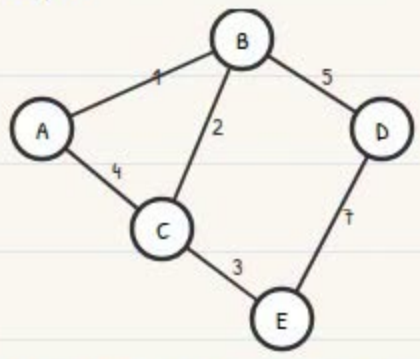
Minimum Spanning Tree (MST)

① Khái Niệm MST

Với một graph liên thông, không có hướng, có trọng số, MST là tập con các edge:

- ✓ Kết nối mọi vertex
- ✓ Không có chu trình
- ✓ Có tổng trọng số nhỏ nhất

Ví dụ Graph



💡 Ý Tưởng Chính

Chọn edge cẩn thận để kết nối mọi node với chi phí thấp nhất.

② Prim's Algorithm (Grow Tree)

1. Bắt đầu từ một vertex bất kỳ.
2. Chọn edge có trọng số nhỏ nhất nối tree với vertex mới.
3. Thêm vertex và edge vào tree.
4. Lặp lại đến khi mọi vertex được thêm.

Ví dụ (Bắt đầu từ A): Cây: {A} → Chọn AB(1) → {A,B} → Chọn BC(2) → {A,B,C} → Chọn CE(3) → {A,B,C,E} → Total Cost = 6

③ Kruskal's Algorithm (Pick Smallest)

1. Sắp xếp mọi edge theo trọng số tăng dần.
2. Chọn edge nhỏ nhất không tạo chu trình.
3. Lặp lại đến khi có (V - 1) edge.

Edge	Đưa vào?	Lý do
AB(1)	Có	Không chu trình
BC(2)	Có	Không chu trình
CE(3)	Có	Không chu trình
AC(4)	Không	Tạo chu trình
BD(5)	Có	Không chu trình

MST Edges: AB(1), BC(2), CE(3), BD(5) – Total Cost = 11

④ So Sánh

Feature	Prim's Algorithm	Kruskal's Algorithm
Chiến lược	Phát triển một tree duy nhất	Xây rừng, gộp các thành phần
Cấu trúc dữ liệu	Min Priority Queue	Disjoint Set (Union-Find)
Phù hợp cho	Graph dày ($E \approx V^2$)	Graph thưa ($E << V^2$)
Time	$O(E \log V)$	$O(E \log E) \approx O(E \log V)$
Space	$O(V)$	$O(V)$
Chọn Edge	Dựa trên edge kết nối nhỏ nhất	Dựa trên edge nhỏ nhất toàn cục
Phát hiện chu trình	Không cần	Dùng Union-Find

⑤ Ứng Dụng

🔀 Thiết kế mạng lưới

Kết nối mọi địa điểm với chi phí thấp nhất.

🔌 Kết nối cáp / Internet / điện

Lắp cáp/dây với tổng độ dài hoặc chi phí nhỏ nhất.

🛣️ Xây dựng đường

Xây đường kết nối các thành phố với chi phí thấp nhất.

🖥️ Thiết kế VLSI

Giảm thiểu độ dài dây nối giữa các thành phần.

🔗 Phân cụm (Single Link)

Dùng trong phân cụm dữ liệu phân cấp.

⭐ Điểm Cần Nhớ

MST kết nối mọi vertex với tổng trọng số nhỏ nhất. Có đúng (V-1) edge. Prim phát triển một tree. Kruskal xây rừng và gộp lại.

⚠️ Lỗi Thường Gặp

✗ Đưa vào edge tạo chu trình.

✗ Dừng trước khi kết nối hết mọi vertex.

✗ Không xử lý graph không liên thông (MST không khả thi).

💡 Thực Hành Tốt

✓ Dùng Prim với Min-Heap cho graph dày.

✓ Dùng Kruskal với Union-Find cho graph thưa.

✓ Luôn sắp xếp edge một lần trong Kruskal.

🎯 Mẹo Phỏng Vấn

Có thể được hỏi: Cài đặt Prim's/Kruskal's, tìm tổng chi phí MST của graph, hoặc phân biệt Prim và Kruskal.

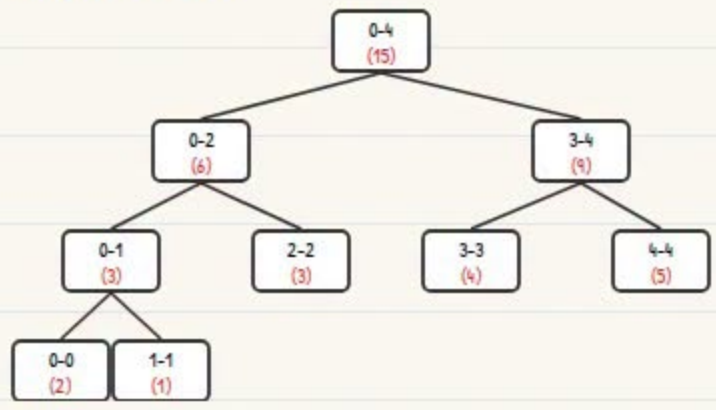
Ghi nhớ: MST ≠ Shortest Path Tree – MST tối thiểu hóa tổng trọng số mọi edge, không phải khoảng cách từ một nguồn.

Segment Tree & Fenwick Tree

① Segment Tree

- Segment Tree là một binary tree, mỗi node đại diện cho một khoảng (segment).
- Dùng để xử lý hiệu quả range query và point/range update.
- Hoạt động với các hàm kết hợp được (associative) như Sum, Min, Max, GCD.

Ví dụ: Array = [2, 1, 3, 4, 5]



Định dạng node: [start-end] giá trị (ở đây, giá trị = tổng các phần tử trong range đó)

Range Query (Sum từ L đến R)

1. Nếu range của node hoàn toàn nằm ngoài [L,R] → trả về 0
2. Nếu range của node hoàn toàn nằm trong [L,R] → trả về giá trị node
3. Ngược lại → query trái + query phải

Update (Index i, giá trị mới)

1. Cập nhật leaf node.
2. Cập nhật các node tổ tiên khi quay lui.

Time: O(log n) mỗi lần query/update Space: O(4n)

② Fenwick Tree (Binary Indexed Tree)

- Fenwick Tree là cấu trúc tối ưu bộ nhớ cho prefix sum.
- Tốt nhất cho point update và truy vấn prefix sum.
- Dùng thao tác bit để di chuyển giữa các range.

Ví dụ: Array = [2, 1, 3, 4, 5] (1-indexed)

Index (i)	1	2	3	4	5
BIT[i]	2	3	3	10	5

LSB (i & -i) = Last Set Bit. VD: 6(110) → 2(010), 4(100) → 4(100), 3(011) → 1(001)

Prefix Sum (1 đến i)

```
res = 0
while (i > 0):
  res += BIT[i]
  i -= (i & -i)
return res
```

Update (thêm delta tại index i)

```
while (i <= n):
  BIT[i] += delta
  i += (i & -i)
```

Time: O(log n) mỗi lần query/update Space: O(n)

③ Các Loại Range Query Hỗ Trợ

Cấu trúc	Query	Hỗ trợ?
Segment Tree	Range Sum/Min/Max	✓
	Range Update (Lazy)	✓
	Point Update	✓
Fenwick Tree (BIT)	Prefix Sum (1 đến i)	✓
	Point Update	✓
	Range Query (L đến R)	✗

④ So Sánh Độ Phức Tạp

Operation	Segment Tree	Fenwick Tree
Build	O(n)	O(n)
Point Update	O(log n)	O(log n)
Range Query	O(log n)	O(log n) (chỉ prefix)
Range Update	O(log n) (Lazy)	Không hỗ trợ
Space	O(4n)	O(n)

Segment Tree → Tổng quát hơn (mọi range query/update)
Fenwick Tree → Trường hợp đặc biệt (prefix sum), ít bộ nhớ hơn

⑤ Ứng Dụng

Segment Tree

- ✓ Range Minimum / Maximum Query
- ✓ Range Sum Query
- ✓ Range GCD / LCM
- ✓ Range Update với Lazy Propagation
- ✓ Order Statistics, K-th number

Fenwick Tree (BIT)

- ✓ Prefix Sum Queries
- ✓ Inversion Count
- ✓ Đếm phần tử nhỏ hơn
- ✓ Cập nhật bảng tần suất
- ✓ Competitive Programming

⭐ Điểm Cần Nhớ

Dùng Segment Tree cho range query/update phức tạp. Dùng Fenwick Tree cho bài toán liên quan prefix sum. Cả hai cho O(log n) mỗi thao tác.

⚠️ Lỗi Thường Gặp

✗ Xử lý sai index (0-index vs 1-index).

✗ Quên dùng Lazy Propagation trong range update.

✗ Dùng BIT cho range query ngoài prefix sum.

💡 Thực Hành Tốt

✓ Giữ các hàm module hóa (build, query, update).

✓ Test với trường hợp nhỏ trước khi input lớn.

✓ Hiểu rõ bài toán trước, rồi chọn cấu trúc.

🎯 Mẹo Phỏng Vấn

Hãy sẵn sàng: vẽ tree/BIT, giải thích update & query, phân tích độ phức tạp thời gian & không gian.

Các Pattern Phỏng Vấn DSA Nâng Cao

① Fast & Slow Pointer

- Dùng 2 con trỏ di chuyển với tốc độ khác nhau.
- Tốt cho phát hiện chu trình, tìm phần tử giữa, palindrome.

Ví dụ: Linked List Cycle

Slow = 1 bước, Fast = 2 bước. Nếu gặp nhau → có chu trình

② Monotonic Stack

- Giữ các phần tử theo thứ tự tăng/giảm trong stack.
- Hữu ích cho bài toán Next Greater/Smaller.

Ví dụ: Next Greater Element
Arr = [2, 1, 2, 4, 3]

Element	2	1	2	3
NGE	4	2	4	-1

Time: O(n) Space: O(n)

③ Monotonic Queue (Deque)

- Giữ phần tử theo thứ tự tăng/giảm trong deque.
- Tốt nhất cho sliding window max/min.

Ví dụ: Sliding Window Max
Arr = [1,3,-1,-3,5,3,6,7], k=3

1	3	-1	-3	5	3	6	7

Kết quả: 3, 3, 5, 5, 6, 7 Time: O(n) Space: O(k)

④ Merge Intervals

- Sắp xếp theo thời gian bắt đầu.
- Gộp các interval chồng lấp.

Ví dụ: Intervals = [[1,3],[2,6], [8,10],[15,18],[17,20]]

1.3	2.6	8.10	15.18	17.20

Kết quả: 1,6 8,10 15,20
Time: O(n log n) Space: O(1)

⑤ K-Way Merge

- Gộp k sorted lists/arrays dùng Min-Heap.
- Luôn lấy phần tử nhỏ nhất hiện tại.

Ví dụ: Gộp k sorted lists
L1: [1,4,5] L2: [1,3,4] L3: [2,6]

Kết quả: 1,1,2,3,4,4,5,6

Time: O(N log k) (N=tổng phần tử)
Space: O(k)

⑥ Top-K Elements

- Dùng Min-Heap kích thước k.
- Giữ k phần tử lớn nhất/nhỏ nhất.

Ví dụ: Top 3 lớn nhất
Arr = [5,1,8,2,7,3,9], k=3

Top 3 = {7, 8, 9}

Time: O(n log k) Space: O(k)

⑦ Islands (Grid DFS/BFS)

- Đếm connected components trong grid.
- Di chuyển theo 4 hướng (lên, xuống, trái, phải).

1	1	0	0	0
1	0	0	1	1
0	0	1	0	0
0	0	0	1	1

Islands = 3 Time: O(m × n)
Space: O(m × n)

⑧ Matrix Traversal

- Các pattern thường gặp: Spiral, Diagonal, Boundary, Zigzag.

Ví dụ: Spiral Order

1	2	3	4
5	6	7	8
9	10	11	12

Kết quả: 1,2,3,4,8,12,11,10,9,5,6,7

⑨ Hướng Dẫn Chọn Pattern

Loại bài toán / Mục tiêu	Dấu hiệu nhận biết	Pattern phù hợp nhất
Phát hiện chu trình / Tìm điểm giữa	Linked List. Loop. Cycle	Fast & Slow Pointer
Next Greater / Smaller	Next. Previous. Greater. Smaller	Monotonic Stack
Sliding Window Min/Max	Window. Subarray. Size k	Monotonic Queue
Gộp các range chồng lấp	Intervals. Meetings. Overlap	Merge Intervals
Gộp k sorted lists/arrays	k lists. Sorted arrays	K-Way Merge (Min-Heap)
Tìm top k lớn nhất/nhỏ nhất	Top K. Largest. Smallest	Top-K với Heap
Đếm components trong grid	Islands. Number of Regions	DFS / BFS
Spiral / Diagonal / Zigzag	Matrix. Traverse. Spiral. Diagonal	Matrix Traversal

Nắm vững các pattern này → Giải quyết hiệu quả hơn 80% bài toán phỏng vấn!

Mẹo Phỏng Vấn

- ✓ Nhận diện pattern trước khi bắt tay vào code.
- ✓ Luyện tập nhiều biến thể của mỗi pattern.
- ✓ Vẽ ví dụ nhỏ & chạy thử (dry run).
- ✓ Xử lý edge case sớm.
- ✓ Phân tích độ phức tạp thời gian & không gian.
- ✓ Giữ code sạch và module hóa.

Ghi Nhớ
Pattern là các viên gạch nền tảng. Càng nhận diện được nhiều, bạn càng giải quyết nhanh hơn!

⭐ Điểm Cần Nhớ

Pattern biến các bài toán phức tạp thành lời giải có thể lặp lại.

⚠️ Lỗi Thường Gặp

- ✗ **Không nhận diện pattern.**
- ✗ **Viết code mà không dry run trước.**
- ✗ **Bỏ qua edge case.**

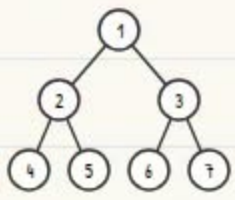
🎯 Sự Thật Thú Vị Khi Phỏng Vấn

Các công ty lớn thường tái sử dụng cùng pattern trong nhiều đề bài khác nhau.

Cẩm Nang Ôn Tập DSA Nâng Cao

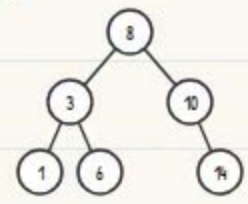
① Trees

- Cấu trúc phân cấp
- Root → Children → Subtree
- Duyệt: Preorder, Inorder, Postorder, Level Order



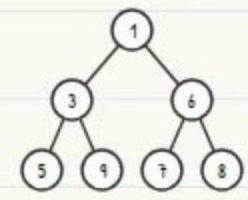
② BST (Binary Search Tree)

- Left < Root < Right
- Duyệt Inorder cho kết quả đã sắp xếp



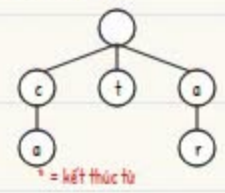
③ Heap (Min Heap)

- Complete Binary Tree
- Parent ≤ Children
- Insert, Extract: O(log n)



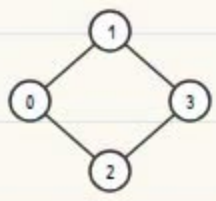
④ Trie

- Prefix Tree cho chuỗi
- Insert, Search, Prefix: O(L)
- L = độ dài chuỗi



⑤ Graph

- Vertices & Edges
- Biểu diễn bằng Adjacency List/Matrix



⑥ DFS

- Đi sâu trước
- Dùng Stack/Recursion
- Time: O(V+E) Space: O(V)

Thứ tự: 0 → 1 → 3 → 2 → 4

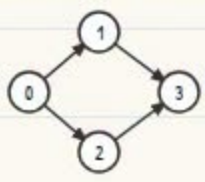
⑦ BFS

- Duyệt theo tầng
- Dùng Queue
- Time: O(V+E) Space: O(V)

Thứ tự: 0 → 1 → 2 → 3 → 4

⑧ Topological Sort

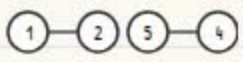
- Cho Directed Acyclic Graph (DAG)
- Indegree + Queue (Kahn's Algo)
- Time: O(V+E)



Một thứ tự: 5 → 0 → 2 → 3 → 1

⑨ Union Find

- Disjoint Set Union
- Find, Union, Path Compression + Union by Rank
- Time: O(α(n)) ≈ O(1)



⑩ Greedy

- Chọn tối ưu cục bộ mỗi bước
- Cần Greedy Choice + Optimal Substructure

VD: Activity Selection, Huffman Coding, Prim's, Kruskal's

⑪ Dynamic Programming

- Overlapping Subproblems + Optimal Substructure
- Top-Down (Memo) / Bottom-Up (Tab)

VD: 0/1 Knapsack, LIS, Matrix Chain

⑫ Backtracking

- Thử mọi khả năng
- Xây lời giải từng bước
- Backtrack khi thất bại

VD: N-Queens, Sudoku Solver, Permutations

⑬ Divide & Conquer

- Divide → Solve → Combine
- Recurrence Relation

VD: Merge Sort, Quick Sort, Binary Search, Strassen Matrix Mult.

⑭ Shortest Path Algorithms

Algorithm	Áp dụng cho	Trọng số âm	Time
Dijkstra	Trọng số không âm	Không	O((V+E) log V)
Bellman-Ford	Mọi trọng số	Có	O(V × E)
Floyd-Warshall	All pairs	Có	O(V^3)

⑮ Minimum Spanning Tree (MST)

Algorithm	Chiến lược	Time
Prim's	Phát triển tree (Min-Heap)	O(E log V)
Kruskal's	Chọn edge nhỏ nhất + Union Find	O(E log E)

⑯ Cấu Trúc Dữ Liệu Nâng Cao

Cấu trúc	Use Case	Operations
Segment Tree	Range Query/Update	O(log n)
Fenwick Tree (BIT)	Prefix Sum / Point Update	O(log n)

⑰ Pattern Phỏng Vấn Nâng Cao

⇄ Fast & Slow Pointer	Linked List Cycle. Palindrome
📊 Monotonic Stack	Next Greater Element. Daily Temperatures
📉 Monotonic Queue	Sliding Window Maximum. Minimum
🔀 Merge Intervals	Employee Free Time. Meeting Rooms
🔗 K-Way Merge	Merge k Sorted Lists. Smallest Range
⭐ Top-K Elements	Top K Frequent. Kth Largest
🏝️ Islands (DFS/BFS)	Number of Islands. Max Area of Island
🔲 Matrix Traversal	Spiral Order. Diagonal Traverse

⑱ Tóm Tắt Độ Phức Tạp

Algorithm/Operation	Best	Average	Worst	Space
Access (Array/Index)	O(1)	O(1)	O(1)	O(1)
Search (Linear)	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Sorting (Bubble)	O(n)	O(n^2)	O(n^2)	O(1)
Sorting (Merge)	O(n log n)	O(n log n)	O(n log n)	O(n)
Sorting (Quick)	O(n log n)	O(n log n)	O(n^2)	O(log n)
BST Operations	O(log n)	O(log n)	O(n)	O(h)
Heap Operations	O(log n)	O(log n)	O(log n)	O(1)
DFS / BFS	O(V+E)	O(V+E)	O(V+E)	O(V)
DP (Tabulation)	-	O(states × transitions)	-	O(states)

⭐ Ghi Nhớ

✓ Hiểu sâu bản chất bài toán.

✓ Chọn đúng pattern.

✓ Viết code sạch, module hóa.

✓ Phân tích độ phức tạp thời gian & không gian.

✓ Luyện tập, luyện tập, luyện tập!

⭐ Điểm Cần Nhớ

Thành thạo pattern + nền tảng vững chắc = Giải quyết tự tin các bài toán DSA phức tạp!

⚠️ Lỗi Thường Gặp

✗ Không hiểu rõ bài toán.

✗ Bỏ qua edge case.

✗ Phân tích độ phức tạp kém.

✗ Không tối ưu bộ nhớ.

💡 Thực Hành Tốt

✓ Chia nhỏ bài toán thành từng bước.

✓ Dùng sơ đồ & ví dụ minh họa.

✓ Bắt đầu đơn giản, tối ưu sau.

✓ Ôn tập thường xuyên.