🎓 HANDOUT BUỔI 5 – PHƯƠNG PHÁP ĐẾM (PHẦN CUỐI)

📍 Chủ đề: Bài toán liệt kê và Hệ thức truy hồi

Mục tiêu buổi học:

- Vận dụng kiến thức tổ hợp để giải quyết bài toán sinh, đếm cấu hình.
- Áp dụng kỹ thuật đệ quy – truy hồi để biểu diễn các cấu trúc lặp trong giải thuật.
- Phân tích độ phức tạp thông qua hệ thức truy hồi trong giải thuật.

🎯 I. GIỚI THIỆU CHUNG

1. Vì sao "liệt kê" và "truy hồi" là nền tảng trong phát triển phần mềm và giải thuật?

Trong Khoa học Máy tính, nhiều bài toán yêu cầu xử lý **tập hợp các khả năng, trạng thái hoặc cấu hình**. Khi đó, hai kỹ thuật quan trọng và thường xuyên được sử dụng là:

- **Liệt kê (enumeration)**: Là quá trình tạo ra **tất cả** các cấu hình có thể xảy ra theo một quy luật hay điều kiện cho trước.
- **Truy hồi (recursion / recurrence)**: Là phương pháp xây dựng lời giải của bài toán **dựa trên lời giải của các bài toán nhỏ hơn**.

Hai kỹ thuật này không chỉ giúp xây dựng các thuật toán hiệu quả mà còn tạo nền tảng để:

- Tư duy giải bài toán có tính **chia để trị (divide and conquer)**.
- Hiểu và tối ưu **hiệu suất** (thời gian chạy, bộ nhớ).
- Thiết kế phần mềm có cấu trúc rõ ràng, dễ bảo trì.

---

2. Mối liên hệ với các loại giải thuật trong phát triển phần mềm

📌 a. Sinh cấu hình trong giải thuật vét cạn, tham lam và backtracking

Các giải thuật này cần **liệt kê hoặc duyệt qua** nhiều khả năng khác nhau của một bài toán:

- **Vét cạn (Brute-force)**: Thử tất cả các cấu hình có thể → ví dụ: tìm chuỗi con trong mật khẩu.
- **Backtracking** (quay lui): Sinh từng cấu hình một cách **có kiểm soát**, loại bỏ sớm các cấu hình không hợp lệ.
    - ➤ Ứng dụng: giải Sudoku, n-Queens, bài toán tổ hợp trong AI.
- **Tham lam (Greedy)**: Không sinh toàn bộ cấu hình, nhưng **so sánh giữa các lựa chọn cục bộ** → vẫn cần hiểu cấu hình có thể để chọn bước tối ưu.

➤ Minh họa:

Giải bài toán tìm tất cả tập con con của $\{1, 2, 3\}$ → ta cần liệt kê:

$\emptyset$, $\{1\}$, $\{2\}$, $\{3\}$, $\{1, 2\}$, $\{1, 3\}$, $\{2, 3\}$, $\{1, 2, 3\}$.

→ Có tổng cộng $2^3 = 8$ cấu hình.

📌 b. Phân tích thuật toán đệ quy

Khi giải bài toán lớn bằng cách **chia nhỏ ra**, ta thường xây dựng lời giải theo công thức đệ quy.

▶ **Ví dụ điển hình:**

Thuật toán	Cách hoạt động	Hệ thức truy hồi
Binary Search	Chia đôi mảng mỗi lần	$+1$
Merge Sort	Chia và trộn hai mảng con	$+n$
Quicksort	Phân tách theo pivot	$+T(n-k-1)+n$ (trung bình)

Phân tích các hệ thức này giúp ta hiểu:

- Thuật toán có chạy nhanh không? (Logarit, tuyến tính, đa thức, hay luỹ thừa?)
- Tối ưu hóa thuật toán như thế nào?
- Có thể cài đặt tương tự bằng **đệ quy hoặc lặp lại (iteration)**?

Như vậy:

- "Liệt kê" giúp ta **xây dựng và duyệt không gian trạng thái** của bài toán – rất quan trọng trong lập trình, giải bài toán tổ hợp, hoặc AI.
- "Truy hồi" giúp ta **thiết kế giải thuật ngắn gọn, tự nhiên**, và là chìa khóa để phân tích thời gian chạy – đặc biệt trong các bài toán phân rã bài toán lớn thành bài toán nhỏ.

👉 Đây chính là **hai kỹ thuật cốt lõi** để lập trình các bài toán phức tạp trong thực tế, từ giải câu đố logic, xây dựng AI cho trò chơi, đến phân tích và tối ưu thuật toán trong các hệ thống lớn.

🔢 II. BÀI TOÁN LIỆT KÊ (Enumeration Problems)

1. Khái niệm và ứng dụng trong Công nghệ Thông tin (CNTT)

📌 1.1. Liệt kê cấu hình là gì?

Liệt kê (Enumeration) là quá trình **tạo ra tất cả các cấu hình khả dĩ** của một đối tượng tổ hợp, thỏa mãn các điều kiện nhất định.

Trong bối cảnh khoa học máy tính, "cấu hình" có thể hiểu là:

- Một tập hợp các giá trị đầu vào.
- Một chuỗi nhị phân, tổ hợp, hoán vị.
- Một cách phân phối tài nguyên, trạng thái của hệ thống, lịch làm việc...

Mục tiêu chính: Liệt kê tất cả các khả năng một cách **có hệ thống**, **không lặp** và **bao phủ toàn bộ không gian tìm kiếm**.

Việc liệt kê này là bước cơ bản trong nhiều ứng dụng, đặc biệt trong các bài toán **tìm kiếm, tối ưu hóa, kiểm thử phần mềm, học máy và trí tuệ nhân tạo.**

💡 1.2. Ứng dụng trong Công nghệ Thông tin

🔐 a. Sinh password / test case tự động

- **Bài toán:** Kiểm thử bảo mật hệ thống với mọi chuỗi có độ dài k từ bảng chữ cái {a, b, c, ..., z, 0–9}.
- **Vai trò của liệt kê:**
  - Tạo tất cả các mật khẩu có thể để brute-force kiểm tra.

- Sinh toàn bộ các đầu vào có thể để kiểm thử phần mềm (đặc biệt là fuzz testing).

Ví dụ: Liệt kê tất cả các chuỗi nhị phân độ dài 4: 0000, 0001, ..., 1111 → dùng làm tập test input cho hệ thống login.

📅 b. Lập lịch trình (Task Scheduling)

- **Bài toán:** Xếp lịch dạy cho giảng viên, lịch chạy máy, lịch trực nhân viên...
- **Liên quan đến tổ hợp:** Phân phối các công việc cho các thời điểm/sự kiện.
- **Liệt kê giúp gì?**
  - Sinh tất cả các lịch khả dĩ.
  - Dùng để chọn lịch **tối ưu** (ít trùng, ít vi phạm ràng buộc...).

Ứng dụng thực tế:

- Dùng thuật toán sinh hoán vị để kiểm tra tất cả cách sắp xếp công việc.
- Kết hợp với hàm chi phí để chọn lịch "tốt nhất".

🔍 c. Tìm kiếm toàn cục (Brute-force Search)

- **Bài toán:** Tìm lời giải cho một vấn đề bằng cách **thử tất cả khả năng**.
- **Vai trò của liệt kê:**
  - Tạo toàn bộ cấu hình đầu vào.
  - Duyệt từng cấu hình và kiểm tra điều kiện nghiệm.

Ví dụ: Tìm tập con của một tập S có tổng bằng T (Subset Sum Problem).

- Trong nhiều trường hợp, brute-force là phương pháp **cuối cùng nhưng chắc chắn**, đặc biệt khi số lượng cấu hình không quá lớn.

🧠 d. Sinh tập trạng thái trong AI / thuật toán heuristic

- **Bài toán:** Trong các hệ thống tự động (robot, game, trí tuệ nhân tạo), cần xây dựng **không gian trạng thái** để tìm đường đi, nước đi, hành động tốt nhất.
- **Liệt kê làm gì?**
  - Sinh tất cả trạng thái từ trạng thái ban đầu → dùng để tìm kiếm (BFS/DFS/A*).
  - Tạo danh sách các hành động có thể trong mỗi bước.
  - Giúp xây dựng cây tìm kiếm trạng thái (state-space tree).

Ví dụ:

- AI chơi cờ: Sinh tất cả nước đi hợp lệ → đánh giá và chọn nước tốt nhất.
- Bài toán 8 số (8-puzzle): Sinh tất cả các trạng thái chuyển động được từ trạng thái hiện tại.

✅ Tóm tắt kiến thức phần này

Ứng dụng thực tế	Dạng cấu hình cần liệt kê	Lợi ích mang lại
Sinh mật khẩu	Chuỗi ký tự	Tạo input toàn diện
Lập lịch	Hoán vị, tổ hợp	Đánh giá phương án tối ưu
Brute-force	Tập con, chuỗi	Tìm lời giải chính xác
Trí tuệ nhân tạo	Trạng thái hệ thống	Khám phá không gian hành động

📌 **Kỹ năng cần có:**
Sinh viên cần hiểu **cấu trúc tổ hợp** của từng bài toán, và biết cách **viết thuật toán sinh** (dùng đệ quy, lặp, hoặc bitmask).

🔢 II. BÀI TOÁN LIỆT KÊ

2. Các mô hình tổ hợp cần liệt kê

Trong các bài toán thực tế, việc liệt kê không phải là thao tác ngẫu nhiên, mà luôn tuân theo một **mô hình tổ hợp cụ thể**. Dưới đây là những mô hình phổ biến, đi kèm công thức đếm và ví dụ minh họa.

🟦 2.1. Chuỗi nhị phân độ dài *n*

🔷 **Mô tả:**

- Mỗi vị trí trong chuỗi nhận giá trị **0 hoặc 1**.
- Là mô hình cơ bản trong lưu trữ và mã hóa thông tin.

🔷 **Số lượng cấu hình:**

- Tổng số chuỗi:

🔷 **Ứng dụng:**

- Đại diện cho tập con của một tập hợp n phần tử (bitmask).
- Sinh tất cả input nhị phân để test thuật toán, mạch logic.

Ví dụ:

Với → các chuỗi là: 000, 001, 010, 011, 100, 101, 110, 111

🔄 2.2. Hoán vị (Permutation)

🔷 **Mô tả:**

- Sắp xếp lại **tất cả** các phần tử của một tập **không trùng lặp**.
- Ví dụ: hoán vị của {A, B, C} gồm ABC, ACB, BAC, BCA, CAB, CBA.

🔷 **Số lượng:**

- Với tập có phần tử → số hoán vị:

🔷 **Ứng dụng:**

- Lập lịch, sinh các thứ tự thực hiện công việc.
- Tối ưu hóa lộ trình (traveling salesman problem – TSP).

🔁 2.3. Chỉnh hợp (Arrangement / k-permutation)

🔷 **Mô tả:**

- Chọn **k phần tử khác nhau** từ n phần tử, sau đó **xếp theo thứ tự**.

🔷 **Có 2 loại:**

- **Không lặp:**
- **Có lặp:**

🔷 **Ứng dụng:**

- Sinh tất cả các mã có độ dài k từ bảng chữ cái n ký tự.
- Tạo không gian trạng thái cho bài toán phân phối tài nguyên.

Ví dụ:
Chỉnh hợp không lặp 2 phần tử từ {1, 2, 3}: 12, 13, 21, 23, 31, 32

🎯 2.4. Tổ hợp (Combination)

🔷 **Mô tả:**

- Chọn **k phần tử từ n phần tử**, **không xét thứ tự**.

🔷 **Có 2 loại:**

- **Không lặp:**
- **Có lặp:**

🔷 **Ứng dụng:**

- Chọn đội hình, chọn tập hợp tính năng.
- Sinh các tập hợp giá trị từ input cho hàm kiểm thử.

Ví dụ:
Tổ hợp 2 phần tử từ {A, B, C}: AB, AC, BC

📂 2.5. Tập con (Subset)

🔷 **Mô tả:**

- Là tập con của tập , có thể rỗng hoặc bằng .

🔷 **Số lượng:**

- (bằng số chuỗi nhị phân độ dài n)

🔷 **Ứng dụng:**

- Dùng để duyệt không gian tìm kiếm trong bài toán tối ưu.
- Bitmask thường dùng để biểu diễn.

Ví dụ:
Tập con của {1, 2}: ∅, {1}, {2}, {1, 2}

🧱 2.6. Phân hoạch (Partition)

🔷 **Mô tả:**

- Chia tập ban đầu thành các nhóm con không giao nhau.
- Tổng hợp tất cả cách chia tập thành các phần **rời nhau và bao phủ toàn bộ**.

🔷 **Số lượng:**

- Gọi là số phân hoạch của số nguyên → không có công thức đóng, nhưng có thể sinh dần hoặc dùng đệ quy.

🔷 **Ứng dụng:**

- Phân chia công việc, phân chia dữ liệu trong song song hóa.
- Mô hình phân tán, clustering trong AI.

Ví dụ:
Phân hoạch của 4:

- 4, 3+1, 2+2, 2+1+1, 1+1+1+1

📊 2.7. Bảng so sánh tổng hợp

Mô hình tổ hợp	Mô tả	Có xét thứ tự?	Có lặp không?	Số cấu hình
Chuỗi nhị phân	Dãy 0–1 độ dài n	Có	Có	
Hoán vị	Sắp xếp n phần tử khác nhau	Có	Không	
Chỉnh hợp không lặp	Chọn k phần tử rồi sắp xếp	Có	Không	
Chỉnh hợp có lặp	Chọn k phần tử từ n phần tử	Có	Có	
Tổ hợp không lặp	Chọn k phần tử	Không	Không	
Tổ hợp có lặp	Chọn k phần tử từ n phần tử	Không	Có	
Tập con	Tập hợp bất kỳ của tập n phần tử	Không	Không	
Phân hoạch	Chia n thành các tổng con	Không	Có	Không có công thức đóng

✅ Tóm lại

Nắm được **các mô hình tổ hợp cơ bản** là điều kiện tiên quyết để:

- Thiết kế thuật toán sinh cấu hình phù hợp.
- Biết chọn công thức đếm đúng.
- Giải quyết các bài toán phức tạp trong lập trình, tối ưu, và AI.

👉 **Gợi ý thực hành**: Hãy viết một hàm đệ quy để sinh các tổ hợp không lặp từ tập {1, 2, 3, 4} chọn ra 2 phần tử. Sau đó biến nó thành hàm sinh tổ hợp có lặp.

🔢 II. BÀI TOÁN LIỆT KÊ

3. Các kỹ thuật liệt kê chính

Trong thực tế, việc liệt kê hiệu quả không chỉ phụ thuộc vào việc hiểu mô hình tổ hợp, mà còn nằm ở **kỹ thuật sinh cấu hình**. Dưới đây là 3 kỹ thuật phổ biến và quan trọng nhất trong lập trình.

a. Đệ quy – Quay lui (Backtracking)

Mô tả:

Kỹ thuật sinh cấu hình bằng cách **duyệt sâu**, xây dựng cấu hình từ trái sang phải, mỗi bước **thử một lựa chọn hợp lệ**, sau đó **gọi đệ quy để xử lý tiếp phần còn lại**.

Cấu trúc chương trình tổng quát:

```
void Try(int i) {
   for (int j = candidate_min; j <= candidate_max; ++j) {
      s[i] = j;

      if (!isValid(i)) continue;  // Bỏ qua nếu không thỏa mãn ràng buộc

      if (i == n) {
         process();  // Xử lý cấu hình khi đã hoàn chỉnh
      } else {
         Try(i + 1);  // Gọi đệ quy để xây tiếp cấu hình
      }
   }
}
```

Điều kiện dừng:

- Khi đã sinh đủ độ dài cấu hình (i == n).
- Thực hiện hành động: in ra, đếm, lưu lại...

Kiểm tra ràng buộc (Constraint Checking):

- Giúp **cắt tỉa không gian tìm kiếm**, tránh sinh các cấu hình sai.
- Ví dụ: không chọn 2 phần tử trùng nhau, hoặc dãy phải tăng dần.

Ứng dụng thực tế:

- Giải bài toán Sudoku, n-Queens.
- Sinh tổ hợp, hoán vị, chỉnh hợp có điều kiện.
- AI: tìm đường, duyệt cây trạng thái.

Ví dụ: Sinh tất cả tổ hợp 3 phần tử tăng dần từ {1, 2, 3, 4, 5}
→ mỗi lần chỉ chọn số **lớn hơn phần tử trước đó**

b. Liệt kê theo thứ tự từ điển (Lexicographic Order)

Mô tả:

- Mỗi cấu hình được **sắp xếp theo thứ tự từ điển**.
- Dựa trên thuật toán **sinh cấu hình kế tiếp** (next).

Cách phát sinh cấu hình kế tiếp (next_permutation):

1. Tìm phần tử i **lớn nhất** sao cho s[i] < s[i+1].
2. Tìm phần tử j **lớn nhất** sao cho s[i] < s[j].
3. Đổi chỗ s[i] ↔ s[j].
4. Đảo ngược đoạn s[i+1..n].

Đây chính là cách std::next_permutation() trong C++ STL hoạt động.

💻 **Ứng dụng thực tiễn:**

- Sinh các đầu vào có thứ tự tăng/giảm → kiểm thử đầu vào biên.
- Sinh các tổ hợp, hoán vị phục vụ cho bài toán tối ưu.
- Thường dùng trong thuật toán **quay lui có trình tự rõ ràng**, dễ kiểm soát.

Ví dụ:

Tổ hợp 3 phần tử từ {1, 2, 3, 4} theo thứ tự từ điển là: 123, 124, 134, 234

🟦 c. Sinh cấu hình bằng Bitmask / Số nguyên

📌 **Mô tả:**

- **Bitmask** là cách dùng **số nhị phân n bit** để biểu diễn một tập con của tập n phần tử.
  - Bit i = 1 ⇔ chọn phần tử i
  - Bit i = 0 ⇔ không chọn phần tử i
- Với mỗi số từ 0 đến , ta có một cấu hình.

🔥 **Ưu điểm:**

- **Tối ưu về tốc độ và bộ nhớ.**
- **Dễ kiểm tra ràng buộc logic:** sử dụng toán tử bit: &, |, ^
- Có thể sinh tập con mà không dùng đệ quy.
- Rất phù hợp cho bài toán số lượng lớn cấu hình.

💡 **Ví dụ:**

```
// Duyệt qua tất cả các tập con của mảng a[] có n phần tử
for (int mask = 0; mask < (1 << n); ++mask) {
   // Bắt đầu in một tập con
   for (int i = 0; i < n; ++i) {
      if (mask & (1 << i)) {
         cout << a[i] << " ";  // Nếu bit i được bật, in phần tử a[i]
      }
   }
   cout << endl;  // Xuống dòng sau mỗi tập con
}
```

🧠 **Ứng dụng thực tế:**

- Sinh tập con để duyệt toàn bộ không gian tìm kiếm (trong DP, tối ưu hóa).
- Dùng trong bài toán phân vùng, chia nhóm, chọn tập con thỏa điều kiện.
- Tăng tốc các bài toán tổ hợp (so với đệ quy truyền thống).

Ví dụ:

Tập {A, B, C} có 3 phần tử → bitmask chạy từ 000 đến 111 tương ứng với 8 tập con.

✅ **Tổng kết so sánh**

Kỹ thuật	Ưu điểm	Hạn chế	Phù hợp với
Đệ quy – Quay lui	Linh hoạt, dễ kiểm soát logic	Dễ bị đệ quy sâu	Bài toán có ràng buộc phức tạp
Từ điển (Lexicographic)	Dễ tổ chức thứ tự sinh cấu hình	Cần cấu hình ban đầu	Sinh cấu hình có thứ tự tăng dần
Bitmask	Tốc độ cao, biểu diễn gọn	Khó hiểu với người mới	Tập con, kiểm tra tổ hợp lớn, tối ưu

4. Phân tích độ phức tạp

Khi xây dựng thuật toán liệt kê, một bước quan trọng là **phân tích độ phức tạp về thời gian và không gian**. Việc này giúp ta:

- Dự đoán **khả năng thực thi** của chương trình trên các bộ dữ liệu lớn.
- Xác định khi nào cần **tối ưu thuật toán**, khi nào có thể **dùng brute-force**.
- Phân biệt các bài toán thuộc loại **tính toán được** hay **quá tải tài nguyên**.

🟦 4.1. Đếm số lượng cấu hình

Đầu tiên, ta cần biết **có bao nhiêu cấu hình cần sinh ra**. Đây chính là **kích thước không gian tìm kiếm**.

Mô hình tổ hợp	Số lượng cấu hình	Tăng trưởng
Chuỗi nhị phân độ dài n		Hàm mũ (exponential)
Tập con của tập n phần tử		Hàm mũ
Hoán vị n phần tử	$n!$	Siêu hàm mũ (factorial)
Chỉnh hợp không lặp (k từ n)		Phụ thuộc vào k
Tổ hợp không lặp (k từ n)		Đa thức nếu k nhỏ
Tổ hợp có lặp		Lớn nếu k lớn

👉 Kết luận: Với n > 20, các mô hình như hoán vị, tập con đã trở nên **rất lớn**, cần xem xét kỹ về tính khả thi khi cài đặt.

⏱️ 4.2. Đánh giá độ phức tạp thời gian (Time Complexity)

Độ phức tạp thời gian phụ thuộc vào:

- **Số lượng cấu hình** cần liệt kê.
- **Chi phí xử lý mỗi cấu hình** (in ra, kiểm tra điều kiện, lưu trữ, v.v.).

Công thức chung:

$$T(n)=S$$

Ví dụ:

- In tất cả chuỗi nhị phân độ dài :
  (vì mỗi chuỗi dài n, cần in hoặc xử lý n ký tự).

- Với tổ hợp

📌 **Lưu ý**: Nếu chỉ **đếm** số cấu hình thay vì liệt kê, ta có thể dùng công thức tổ hợp — nhanh và không tốn thời gian sinh cấu hình.

💾 4.3. Đánh giá độ phức tạp không gian (Space Complexity)

Có 2 mức độ sử dụng bộ nhớ:

a. Bộ nhớ tạm thời (trong quá trình sinh)

- Đệ quy cần stack sâu tới *n* nếu gọi đệ quy .
- Cần mảng lưu cấu hình hiện tại: .

b. Bộ nhớ lưu toàn bộ cấu hình

- Nếu lưu toàn bộ cấu hình (ví dụ: trong vector 2D) → cần bộ nhớ:

Ví dụ nguy hiểm: Với n = 25 → triệu cấu hình → rất khó lưu trong RAM máy tính phổ thông.

☑️ 4.4. Chiến lược tối ưu hoá

Vấn đề	Giải pháp
Không đủ thời gian liệt kê	Chuyển sang thuật toán đếm
Không đủ RAM để lưu kết quả	Xử lý cấu hình theo dòng (stream)
Không cần in cấu hình	Chỉ dùng đếm (trả về số lượng)
Cấu hình có ràng buộc	Cắt tỉa cây tìm kiếm sớm (backtracking + pruning)
Tăng hiệu suất	Dùng bitmask thay vì mảng

✅ Tổng kết

- Phân tích độ phức tạp giúp đánh giá **khả năng thực thi** của thuật toán sinh cấu hình.
- Với bài toán có không gian tìm kiếm quá lớn (exponential), cần:
  - Giới hạn độ sâu tìm kiếm.
  - Tối ưu cấu trúc dữ liệu.
  - Ưu tiên thuật toán đếm thay vì liệt kê đầy đủ.

📌 **Thực hành gợi ý:**

- Viết hàm sinh tổ hợp k phần tử từ n phần tử, và đo thời gian chạy với n = 20, 25, 30.
- Quan sát sự **tăng trưởng thời gian và bộ nhớ**, từ đó rút ra bài học về đánh giá complexity.

5. Liên hệ thực tiễn trong IT

Mặc dù "liệt kê" nghe có vẻ là một bài toán toán học thuần túy, nhưng trên thực tế, kỹ thuật này đóng vai trò **rất thiết yếu** trong nhiều lĩnh vực của Công nghệ Thông tin (IT). Dưới đây là những liên hệ thực tế quan trọng giúp sinh viên thấy rõ giá trị ứng dụng của các kiến thức tổ hợp và phương pháp đếm:

🧠 5.1. Phân tích độ phức tạp thuật toán

- Khi xây dựng **thuật toán đệ quy hoặc tìm kiếm toàn cục**, việc **liệt kê toàn bộ cấu hình** thường là bước đầu tiên.
- Tuy nhiên, không phải lúc nào ta cũng có thể "liệt kê hết" – cần **phân tích số lượng cấu hình** để quyết định có nên:
    - Tối ưu thuật toán?
    - Dùng heuristic?
    - Dùng phép đếm thay vì sinh?

Ví dụ thực tiễn:

Trong thuật toán **Backtracking** giải bài toán n-Queens, nếu không giới hạn ràng buộc sớm, số cấu hình cần duyệt sẽ tăng rất nhanh → cần phân tích kỹ để cắt tỉa cây tìm kiếm.

🎯 **Kết luận:**

- Phân tích độ phức tạp giúp tránh viết những chương trình **"chết vì tốn tài nguyên"**.
- Kết hợp tư duy tổ hợp và kiến thức về thuật toán giúp nâng cao chất lượng phần mềm.

⚙️ 5.2. Tối ưu thuật toán nhờ hiểu công thức tăng trưởng

Việc hiểu bản chất của **tăng trưởng hàm đếm** (ví dụ như ,) cho phép:

- Dự đoán trước khi chạy chương trình.
- Tối ưu giải thuật bằng các chiến lược như:
    - Chỉ sinh cấu hình cần thiết (liệt kê có ràng buộc).
    - Dùng bitmask thay cho mảng → giảm chi phí không gian và thời gian.
    - Chuyển từ sinh sang đếm để tiết kiệm tài nguyên.

Ví dụ thực tiễn:

Khi viết chương trình kiểm tra hoán vị hợp lệ trong thuật toán sắp xếp, thay vì sinh tất cả hoán vị (n!) → chỉ sinh những hoán vị cần thiết thỏa điều kiện đã cho → tiết kiệm thời gian rất lớn.

🤖 5.3. Ứng dụng trong học máy (Machine Learning)

Trong học máy, kiến thức tổ hợp và phương pháp đếm được ứng dụng gián tiếp trong:

a. Biểu diễn không gian trạng thái

- Trong nhiều mô hình, không gian trạng thái (tập các giá trị đầu vào hoặc siêu tham số – hyperparameters) có thể được **liệt kê hoặc mô hình hóa** như một tập hợp tổ hợp.

- Việc liệt kê hoặc sampling từ không gian này giúp chọn ra tập hợp mô hình tốt nhất.

Ví dụ:

- Tìm tập siêu tham số tốt nhất cho SVM: chọn tổ hợp của {kernel, C, gamma} → cần sinh chỉnh hợp có lặp.

b. Hàm mất mát (Loss Function) qua nhiều epochs

- Mỗi epoch trong quá trình huấn luyện là một **trạng thái** của mô hình.
- Việc theo dõi sự thay đổi giá trị loss qua các epoch giống như đang **xem xét dãy số định nghĩa truy hồi**, vì:

Trong đó : giá trị loss tại epoch thứ n, phụ thuộc vào các yếu tố trước đó → ta có thể phân tích tiến trình học giống như hệ thức truy hồi.

c. Kỹ thuật sinh dữ liệu (Data Augmentation)

- Các kỹ thuật như hoán vị đặc trưng (feature permutation), chọn tập con dữ liệu, sinh chuỗi đầu vào (văn bản, âm thanh...) chính là **bài toán liệt kê** với ràng buộc.
- Hiểu đúng bản chất tổ hợp giúp sinh dữ liệu đa dạng hơn nhưng không dư thừa.

✅ Tổng kết ứng dụng

Ứng dụng	Vai trò của kiến thức liệt kê
Phân tích độ phức tạp	Ước lượng số cấu hình → đánh giá tính khả thi
Tối ưu thuật toán	Chọn cấu hình hợp lệ → tiết kiệm tài nguyên
Học máy	Mô hình hóa trạng thái học, chọn hyperparameters, biểu diễn tiến trình học
Test phần mềm	Sinh tập test đa dạng → kiểm thử toàn diện
Trí tuệ nhân tạo	Duyệt không gian hành động/trạng thái trong game, planning, robot

🎓 Kết nối kiến thức:

Phần này đặt nền tảng rất tốt cho các học phần sau như:

- **Phân tích và thiết kế thuật toán**
- **Machine Learning**
- **Phát triển hệ thống thông minh**
- **Kiểm thử phần mềm tự động**

🔁 III. HỆ THỨC TRUY HỒI (Recurrence Relations)

1. Định nghĩa và ý nghĩa

📌 1.1. Dãy số được định nghĩa đệ quy

Trong Toán học rời rạc, **hệ thức truy hồi (recurrence relation)** là một công thức cho phép xác định một **phần tử trong dãy số** dựa trên **các phần tử trước đó**.

Định nghĩa:
Một hệ thức truy hồi là công thức có dạng:

Trong đó:

- : phần tử thứ n của dãy.
- : bậc của hệ thức.
- : hàm số phụ thuộc vào các phần tử trước.

Để dãy được xác định đầy đủ, ta cần cung cấp **giá trị khởi đầu (initial conditions)**, ví dụ:

1.2. Biểu diễn thuật toán đệ quy bằng công thức toán học

Trong lập trình và giải thuật, nhiều bài toán được giải bằng **hàm đệ quy**. Khi đó, hàm đệ quy chính là một **dạng hệ thức truy hồi**, còn **độ phức tạp thời gian** của thuật toán cũng thường được biểu diễn dưới dạng hệ thức.

Ví dụ 1: Dãy Fibonacci

```
// Hàm đệ quy tính số Fibonacci thứ n
int fib(int n) {
    if (n <= 1) {
        return n;  // Trường hợp cơ sở: fib(0) = 0, fib(1) = 1
    }

    // Gọi đệ quy để tính hai số trước đó
    int fib_n1 = fib(n - 1);
    int fib_n2 = fib(n - 2);

    return fib_n1 + fib_n2;
}
```

Tương ứng với hệ thức:

Ví dụ 2: Đệ quy Merge Sort

```
// Hàm sắp xếp mảng a[] từ chỉ số l đến r bằng thuật toán Merge Sort
void mergeSort(int a[], int l, int r) {
    if (l >= r) return;  // Trường hợp cơ sở: mảng chỉ có 1 phần tử

    int m = (l + r) / 2;  // Tìm chỉ số giữa

    // Gọi đệ quy sắp xếp nửa trái và nửa phải
    mergeSort(a, l, m);
    mergeSort(a, m + 1, r);

    // Gộp hai nửa đã sắp xếp
```

```
    merge(a, l, m, r);  // merge có độ phức tạp O(n) }
```

→ Độ phức tạp thời gian $T(n)$ thỏa mãn hệ thức:

$$+O(n)$$

Đây là một **hệ thức truy hồi không thuần nhất**, có thể giải bằng phương pháp **Master Theorem**.

🧠 **Ý nghĩa trong IT và phân tích thuật toán**

Hệ thức truy hồi giúp chúng ta:

Mục tiêu	Vai trò của hệ thức
Hiểu thuật toán đệ quy	Diễn tả lại quá trình gọi hàm đệ quy dưới dạng toán học
Phân tích độ phức tạp	Từ hệ thức → tìm công thức tổng quát → đánh giá thời gian chạy
Thiết kế thuật toán tối ưu	Hiểu cấu trúc truy hồi giúp cắt giảm gọi lặp (memoization, DP)

Ví dụ thực tế:

- Tối ưu fib(n) từ $O(2^n)$ xuống O(n) bằng cách lưu giá trị đã tính.
- Biểu diễn hệ thức của quicksort giúp đánh giá thời gian chạy trung bình:

$$T(n)=T(k)+T(n-k-1)+O(n)$$

✅ **Kết luận**

- Hệ thức truy hồi là **cầu nối giữa toán học và lập trình đệ quy**.
- Hiểu hệ thức truy hồi giúp sinh viên:
  - Viết đệ quy đúng.
  - Phân tích thuật toán hiệu quả.
  - Tối ưu hoá code nhờ hiểu cấu trúc gọi hàm.

📌 Gợi ý mở rộng:

Bạn có thể thử viết hệ thức truy hồi cho các thuật toán khác như:

- Tìm số bước tối thiểu để giải tháp Hà Nội với n đĩa.
- Tìm số cách đi từ ô (0, 0) đến ô (m, n) trên lưới.

🔁 III. HỆ THỨC TRUY HỒI (Recurrence Relations)

2. Các loại hệ thức

Hệ thức truy hồi có thể được phân loại theo nhiều tiêu chí như: **cấp (bậc)**, **tính tuyến tính**, và sự có mặt của **vế phải** độc lập (non-homogeneous term). Việc phân loại giúp chọn **phương pháp giải thích hợp**.

🔷 2.1. Hệ thức truy hồi tuyến tính (Linear Recurrence Relation)

a. Định nghĩa:

Hệ thức được gọi là **tuyến tính** nếu mỗi số hạng

trong công thức xuất hiện dưới dạng **hệ số nhân với** , không có lũy thừa hay hàm phi tuyến.

Trong đó:

- là các hằng số.
- là vế phải (có thể là 0 hoặc khác 0).
- là **cấp (bậc)** của hệ thức.

b. Hệ thức thuần nhất (Homogeneous)

- Khi **vế phải** → hệ thức thuần nhất.
- Ví dụ:

⚙️ **Ứng dụng:**

- Các bài toán sinh dãy (Fibonacci, Lucas).
- Phân tích thuật toán không có chi phí ngoài (pure recursion).

c. Hệ thức không thuần nhất (Non-Homogeneous)

- Khi → hệ thức không thuần nhất.
- Ví dụ:

$$T(n)=2T(n-1)+1$$

⚙️ **Ứng dụng:**

- Merge Sort: $T(n)=2T(n/2)+n$
- Hanoi Tower: $H(n)=2H(n-1)+1$

📌 Trong các thuật toán đệ quy thường gặp, hệ thức không thuần nhất phổ biến hơn vì luôn có chi phí xử lý tại mỗi bước (ví dụ: +n, +1).

🔷 2.2. Hệ thức tuyến tính cấp 1 (First-order)

- Dạng tổng quát:

- Dễ giải, thường dùng phương pháp lặp lại (unfolding).

🔎 **Ví dụ:**

🔧 **Ứng dụng:**

- Tăng trưởng lũy thừa (lãi kép, cấp số nhân).
- Duyệt binary tree (số node nhân đôi mỗi cấp).

🔷 2.3. Hệ thức tuyến tính cấp k (Higher-order)

- Dạng tổng quát:

🔎 **Ví dụ:**

- Đây là hệ thức cấp 2, có nghiệm đặc trưng là

🔧 **Ứng dụng:**

- Fibonacci:
- Quicksort trung bình: $T(n)=T(k)+T(n-k-1)+cn$

📘 **Tóm tắt phân loại**

Loại hệ thức	Dạng tổng quát	Vế phải $f(n)$	Ứng dụng chính
Tuyến tính thuần nhất		0	Dãy số toán học, mô hình lặp thuần
Tuyến tính không thuần		$\neq 0$	Phân tích thuật toán, đệ quy thực tế
Cấp 1		Tuỳ	Lãi kép, tăng trưởng tuyến tính
Cấp k ≥ 2		Tuỳ	Fibonacci, Quicksort, DP

✅ **Gợi ý thực hành**

- Tìm và phân loại hệ thức truy hồi của các thuật toán sau:
  - Binary Search
  - Towers of Hanoi
  - Tính $n!$ bằng đệ quy
- Thử viết một hàm đệ quy và xác định hệ thức tương ứng → giải bằng tay hoặc dùng bảng.

📌 **Chuẩn bị cho phần tiếp theo:**

Ở phần sau, ta sẽ học cách **giải các hệ thức này** để tìm **công thức tổng quát** – bước quan trọng để:

- Dự đoán thời gian chạy của thuật toán.
- Kiểm tra tính đúng/sai của cài đặt đệ quy.
- Tối ưu hóa hiệu suất chương trình.

3. Kỹ thuật giải hệ thức truy hồi

Giải hệ thức truy hồi có nghĩa là **tìm ra công thức tổng quát** mà không cần tính tuần tự từng giá trị từ Việc này rất quan trọng trong:

- **Phân tích thuật toán đệ quy**.
- **Dự đoán thời gian chạy**.
- **Lập trình hiệu quả với quy hoạch động (dynamic programming)**.

🏵️ a. Diễn giải trực tiếp (Unfolding)

🔷 **Ý tưởng:**

- Mở rộng dần hệ thức bằng cách thay thế các giá trị truy hồi → rút ra quy luật.

🔎 **Ví dụ:**

Cho hệ thức:

$$T(n)=T(n-1)+3, T(0)=2$$

Ta có:

- $T(n)=T(n-1)+3$
- $=T(n-2)+3+3=T(n-2)+2\cdot 3$
- ...
- $=T(0)+3n=2+3n$

✅ Vậy:

$$T(n)=3n+2$$

✅ **Khi sử dụng:**

- Dễ áp dụng với hệ thức **cấp 1 đơn giản**.
- Không cần kiến thức giải phương trình.

📐 b. Phương pháp đặc trưng (Characteristic Equation)

🔷 **Dùng cho hệ tuyến tính thuần nhất cấp k, dạng:**

🔷 **Các bước:**

1. **Viết phương trình đặc trưng**:
2. **Giải phương trình**, tìm nghiệm .
3. **Lập công thức tổng quát** theo các nghiệm:
    - Nghiệm đơn:
    - Nghiệm kép:
    - Nghiệm phức: dùng công thức Euler (nâng cao)

🔎 **Ví dụ:**

Cho:

Bước 1:
Phương trình đặc trưng:

→ Nghiệm kép
Bước 2:
Công thức tổng quát:

Dùng điều kiện đầu:

-

-

→

✅ Vậy:

🎯 c. Đoán nghiệm + quy nạp

🔷 **Khi áp dụng:**

- Khi dãy có quy luật dễ nhận ra.
- Khi không áp dụng được phương pháp đặc trưng.

🔷 **Các bước:**

1. **Tính vài giá trị đầu** để tìm mẫu (pattern).
2. **Đoán công thức tổng quát**.
3. **Dùng quy nạp toán học để chứng minh**.

🔎 **Ví dụ:**

Cho:

$$T(n)=2T(n-1)+1, T(0)=0$$

Tính:

-
-
-
-

→ Đoán:

✔️ Chứng minh bằng quy nạp:

- **Cơ sở**: ✅
- **Giả sử** đúng với n, chứng minh với :

$$T(n+1)=2T(n)+1$$ ✅ Đúng.

⚙️ d. Dùng hàm sinh (Generating Functions) – (Nâng cao)

🔷 **Ý tưởng:**

- Biểu diễn dãy số dưới dạng **hàm sinh (power series)**.

- Dùng các phép biến đổi đại số để tìm biểu thức khép kín.

🔎 **Ví dụ kinh điển:**

Dãy Fibonacci:

→ Hàm sinh:

→ Phân tích mẫu số để tìm công thức tổng quát (nâng cao).

🔧 **Ứng dụng:**

- Phù hợp khi dãy có quy luật phức tạp.

- Dùng trong thuật toán nén, phân tích xác suất (combinatorics).
- Hữu ích trong lý thuyết ngôn ngữ hình thức, máy tự động, thuật toán nhanh.

📌 Phần này thường được dạy trong các học phần nâng cao như **Lý thuyết tổ hợp** hoặc **Kỹ thuật phân tích thuật toán**.

✅ **Tóm tắt các phương pháp**

Phương pháp	Ưu điểm	Khi dùng phù hợp
Diễn giải trực tiếp	Nhanh, trực quan	Hệ thức đơn giản, cấp 1
Phương trình đặc trưng	Giải hệ tuyến tính thuần nhất	Cấp 2 trở lên, không có vế phải
Đoán nghiệm + quy nạp	Linh hoạt, không cần công cụ phức tạp	Dãy dễ đoán, bài toán thi đấu
Hàm sinh (generating func)	Giải hệ thức phức tạp, tổ hợp	Nâng cao, dùng trong phân tích lý thuyết

4. Ứng dụng trong phân tích thuật toán

Phân tích đệ quy của MergeSort và Binary Search

Trong nhiều thuật toán chia để trị (Divide and Conquer), lời giải được xây dựng bằng cách chia bài toán lớn thành các bài toán con, giải từng bài toán con (thường bằng đệ quy), sau đó trộn kết quả lại. Phân tích độ phức tạp của các thuật toán này thường dẫn đến hệ thức truy hồi (recurrence relations).

🔍 **Ví dụ 1: MergeSort**

MergeSort là thuật toán sắp xếp gồm ba bước:

1. Chia mảng đầu vào có n phần tử thành hai mảng con có kích thước
2. Đệ quy sắp xếp hai mảng con
3. Gộp hai mảng đã sắp xếp thành mảng cuối

Hệ thức truy hồi mô tả thời gian chạy:

Giải thích:

thời gian để sắp xếp hai nửa mảng; : thời gian để trộn hai mảng đã sắp xếp

Giải hệ thức bằng phương pháp mở rộng (unfolding):

5. Liên hệ thực tiễn trong Công nghệ Thông tin (IT)

Hệ thức truy hồi không chỉ là kiến thức hàn lâm trong sách giáo khoa – chúng đóng vai trò quan trọng trong thực tiễn phát triển phần mềm và nghiên cứu các thuật toán hiện đại, đặc biệt trong những lĩnh vực như:

4. Tối ưu hóa
5. Học máy (Machine Learning)
6. Xử lý dữ liệu lớn (Big Data)
7. Thiết kế hệ thống hiệu năng cao

📊 **a. Phân tích độ phức tạp thuật toán**

Trong thực tế, một thuật toán có thời gian chạy kém hiệu quả không chỉ gây khó chịu mà còn có thể:

1. Làm tăng chi phí phần cứng
2. Gây lãng phí thời gian cho người dùng
3. Dẫn đến sập hệ thống khi dữ liệu tăng đột biến

🔍 Vai trò của hệ thức truy hồi:

- Dự đoán độ phức tạp thời gian hoặc số phép tính
- Chứng minh một thuật toán tối ưu hơn giải pháp khác
- So sánh hiệu quả giữa các thuật toán, ví dụ: MergeSort vs Bubble Sort

▶ Ví dụ thực tế:

Khi xây dựng tính năng sắp xếp dữ liệu người dùng trên website, nếu sử dụng Bubble Sort với độ phức tạp $O(n^2)$ thay vì MergeSort với $O(n \log n)$, thì thời gian phản hồi sẽ tăng mạnh khi xử lý hàng triệu bản ghi.

⚙️ b. Tối ưu thuật toán thông qua hiểu biết về công thức tăng trưởng

Công thức truy hồi mô tả cách một bài toán lớn được chia nhỏ thành các bài toán con. Việc hiểu rõ công thức này giúp:

- Tái cấu trúc mã: chuyển từ đệ quy sang memoization hoặc thuật toán lặp
- Cải tiến giải thuật: giảm số nhánh đệ quy, rút gọn đầu vào, hoặc giảm chi phí hợp nhất kết quả
- Tối ưu hiệu suất tổng thể: từ thuật toán → mã thực thi → tận dụng phần cứng hiệu quả hơn

💡 Ví dụ:

Quicksort có thể được cải tiến bằng cách chọn pivot theo phương pháp median-of-three, giúp tránh trường hợp xấu có độ phức tạp $O(n^2)$.

🧠 c. Ứng dụng trong Học máy (Machine Learning)

Trong học máy, quá trình tối ưu hóa hàm mất mát qua nhiều vòng lặp (epoch) thường được mô hình hóa bằng hệ thức truy hồi:

$$L_{t+1} = L_t - \eta \cdot \nabla L_t$$

Trong đó:

L_t : giá trị hàm mất mát tại epoch thứ t

- η: learning rate (tốc độ học)

∇L_t : gradient tại epoch t

📌 Đây là một hệ thức truy hồi không thuần nhất, mô tả quá trình cập nhật theo thời gian.

📈 Phân tích hệ thức này giúp:

- Dự đoán khả năng hội tụ của mô hình
- Điều chỉnh tốc độ học hợp lý
- Xác định số lượng epoch tối ưu

▶ Ứng dụng thực tế:

- Nâng cao hiệu suất của các mô hình AI trong nhận diện hình ảnh, dự đoán hành vi, v.v.
- Triển khai early stopping dựa trên phân tích truy hồi để tránh overfitting

🔧 Tổng kết

Ứng dụng IT	Vai trò của Hệ thức Truy hồi
Phân tích thuật toán	Dự đoán và đánh giá hiệu năng
Tối ưu thuật toán	Gợi ý cải tiến, tái cấu trúc và viết mã hiệu quả
Học máy	Mô hình hóa quá trình huấn luyện và tối ưu hóa học tập

👉 Kết luận:

Hiểu và vận dụng hệ thức truy hồi không chỉ là lý thuyết, mà còn là công cụ mạnh mẽ giúp bạn:

- Viết mã hiệu quả hơn
- Thiết kế những thuật toán mạnh mẽ
- Làm chủ các mô hình học máy hiện đại

BÀI TẬP CÓ ĐÁP ÁN

📓 PHẦN 1: BÀI TOÁN LIỆT KÊ & SINH CẤU HÌNH

🧪 Bài tập 1 – Liệt kê chỉnh hợp có ràng buộc

🎯 Đề bài:

Cho tập ký tự $\{A, B, C, D\}$, hãy:

- Liệt kê tất cả các chỉnh hợp không lặp độ dài 3
- Trong đó, đếm số cấu hình không chứa cặp "A-B" đứng liền nhau (A đứng trước B)

📌 Hướng dẫn triển khai:

🔷 **Phần 1: Tính tổng số chỉnh hợp không lặp**

Áp dụng công thức:

$$A_4^3 = 4 \times 3 \times 2 = 24$$

🔷 **Phần 2: Dùng kỹ thuật** đếm trừ

Bước 1: Tính số chỉnh hợp có cặp "AB" đứng liền nhau

- Gộp "AB" thành 1 khối → còn lại là {C, D}
- Sinh các chỉnh hợp còn lại kết hợp với "AB" như một phần tử

Bước 2: Trừ số lượng vi phạm ra khỏi tổng để thu được số cấu hình hợp lệ

🧠 Gợi ý bổ sung:

- Có thể duyệt tất cả chỉnh hợp bằng tay hoặc viết đoạn Python để sinh cấu hình và lọc những cấu hình vi phạm điều kiện.

🧪 Bài tập 2 – Sinh tập con có tổng chia hết cho 3

🎯 Đề bài:

Cho tập $S = \{1, 2, 3, 4, 5\}$, hãy:

- Sinh tất cả các tập con không rỗng của SS
- Liệt kê các tập con có tổng phần tử chia hết cho 3

📌 Hướng dẫn triển khai:

🔷 **Tổng số tập con không rỗng:**

$$2^5 - 1 = 31$$

🔷 **Duyệt từng tập con:**

- Tính tổng phần tử
- Kiểm tra: Tổng mod 3=0

🔧 Cách cài đặt:

- Bitmask: duyệt từ 1 đến $2^n - 1$
- Đệ quy: sinh tập con theo nhánh chọn/bỏ

📓 PHẦN 2: HỆ THỨC TRUY HỒI & ỨNG DỤNG

🟦 Bài tập 3 – Phân tích thuật toán chia để trị

🎯 Đề bài:

Một thuật toán hoạt động như sau:

- Chia đầu vào kích thước nn thành 2 phần bằng nhau
- Gọi đệ quy xử lý từng phần
- Quá trình trộn kết quả mất đúng $\frac{n^2}{2}$ bước

Yêu cầu:

- Viết hệ thức truy hồi cho $T(n)$
- Giải hệ thức
- So sánh với MergeSort T(n)=2T(n/2)+n và rút ra nhận xét

📌 Hướng dẫn triển khai:

🔷 **Hệ thức:**

$$T(n) = 2T(n\ /\ 2)$$

🔷 **Giải bằng Master Theorem:**

- $a = 2, b = 2, f(n) = \frac{n^2}{2}$
- So sánh với $n^{\log_b a} = n^{\log_2 2} = n \rightarrow$ Vì $f(n) = \Omega(n^{1+\epsilon})$ với $\epsilon = 1$, rơi vào trường hợp 3 của định lý:

$$T(n) = \Theta(n^2)$$

🧠 Nhận xét:

Chi phí trộn tăng từ $n \rightarrow \frac{n^2}{2}$ làm thuật toán chậm hơn MergeSort. Đây là ví dụ điển hình khi chi phí sau chia tăng làm tăng độ phức tạp tổng thể.

🤖 Bài tập 4 – Hàm mất mát trong học máy dưới dạng truy hồi

🎯 Đề bài:

Giả sử quá trình cập nhật hàm mất mát theo công thức:

$$L_{t+1} = 0.8 \cdot L_t, L_0 = 100$$

Yêu cầu:

- Viết hệ thức truy hồi và nghiệm tổng quát
- Tính giá trị loss sau 5 epochs
- Tìm số epoch cần để $L_t < 1$
- Nhận xét về tính hội tụ và liên hệ với Gradient Descent

📌 Hướng dẫn triển khai:

🔷 **Hệ thức tổng quát:**

$$L_t = L_0 \cdot (0.8 \quad t = 100 \cdot (0.8 \quad t))$$

🔷 **Sau 5 epochs:**

$$L_5 = 100 \cdot (0.8 \quad 5 = 100 \cdot 0.32768 = 32.768)$$

🔷 **Tìm epoch để** $L_t < 1$**:**

$$100 \cdot \left(0.8 \quad t < 1 \Rightarrow \left(0.8 \quad t < \frac{1}{100} \Rightarrow t > \frac{\log(1 \ / \ 100)}{\log(0.8)} \approx 20.6 \Rightarrow t = 21\right)\right)$$

🧠 Liên hệ:

Đây là mô hình đơn giản mô tả Gradient Descent, cho thấy quá trình hội tụ dần của hàm mất mát qua từng vòng huấn luyện (epoch).

✅ Tổng kết kỹ năng từ phần này:

Bài tập	Kỹ năng phát triển
Chỉnh hợp có ràng buộc	Tư duy tổ hợp, kỹ thuật đếm trừ
Sinh tập con chia hết cho 3	Bitmask, sinh tập con, lọc theo tính chất
Phân tích thuật toán chia để trị	Thiết lập hệ thức truy hồi, dùng Master Theorem
Mô hình hàm mất mát	Hiểu hệ thức truy hồi cấp số, phân tích hội tụ, liên hệ học máy

TRẮC NGHIỆM

🧭 I. GIỚI THIỆU CHUNG

Câu 1: Kỹ thuật nào sau đây giúp sinh toàn bộ các cấu hình có thể trong bài toán tổ hợp?
A. Phân tích đệ quy
B. Sinh bằng Bitmask
C. Lập lịch trình
D. Hàm mất mát

Câu 2: Trong thuật toán Binary Search, đặc điểm nào đúng?
A. Mỗi bước tăng độ dài của mảng
B. Luôn chia đôi không gian tìm kiếm
C. Có độ phức tạp O(n)
D. Duyệt toàn bộ mảng

🔢 II. BÀI TOÁN LIỆT KÊ

Câu 3: Có bao nhiêu chuỗi nhị phân độ dài 4?

A. 8
B. 12
C. 16
D. 24

Câu 4: Kỹ thuật nào giúp sinh cấu hình theo thứ tự tăng dần?
A. Backtracking
B. Bitmask
C. next_permutation
D. random_shuffle

Câu 5: Sinh chỉnh hợp không lặp của 3 phần tử từ tập {A, B, C, D}?
A. 6
B. 12
C. 24
D. 36

Câu 6: Ưu điểm chính của Bitmask là gì?
A. Giao diện đẹp
B. Tính ngẫu nhiên
C. Cấu trúc dữ liệu đơn giản, tiết kiệm bộ nhớ
D. Thích hợp cho hoán vị

🔁 III. HỆ THỨC TRUY HỒI

Câu 7: Công thức đệ quy của dãy Fibonacci?
A. $T(n) = 2T\left(\frac{n}{2}\right) + n$
B. F(n)=F(n−1)+F(n−2)
C. T(n)=T(n−1)+n
D. F(n)=2F(n−1)

Câu 8: Bước đầu khi giải $T(n) = 2T\left(\frac{n}{2}\right) + n$ bằng unfolding?
A. Tìm nghiệm tổng quát
B. Thử với $n = 1$
C. Viết lại $T(n)$ theo $T\left(\frac{n}{2}\right)$
D. Áp dụng đặc trưng

Câu 9: Hệ thức nào giải được bằng phương pháp đặc trưng?
A. T(n)=T(n−1)+n
B. F(n)=F(n−1)+F(n−2)
C. $T(n) = 2T\left(\frac{n}{2}\right) + n$

D. $T(n) = T\left(\frac{n}{2}\right) + 1$

Câu 10: Giải hệ thức $T(n) = T\left(\frac{n}{2}\right) + 1$ gần đúng với?

A. $O(n)$
B. $O(n^2)$
C. $O(\log\ n)$
D. $O(n\ \log\ n)$

Câu 11: Hệ thức độ phức tạp trung bình của QuickSort?
A. T(n)=T(n−1)+n
B. $T(n) = 2T\left(\frac{n}{2}\right) + n$
C. T(n)=T(k)+T(n−k−1)+n
D. $T(n) = T\left(\frac{n}{2}\right) + 1$

🧠 IV. ỨNG DỤNG TRONG CNTT

Câu 12: Vì sao cần phân tích hệ thức trước khi triển khai?
A. Tạo giao diện đẹp
B. Hiểu cấu trúc dữ liệu
C. Dự đoán hiệu năng khi dữ liệu tăng
D. Giảm lỗi logic

Câu 13: Trong học máy, cập nhật hàm mất mát theo dạng nào?
A. Chuỗi lặp vô hạn
B. Tích phân
C. Hệ thức truy hồi
D. Ma trận

Câu 14: Giảm độ phức tạp từ $O(n^2)$ xuống $O(n\ \log\ n)$ giúp:
A. Giao diện đẹp hơn
B. Ít cần RAM
C. Phản hồi người dùng nhanh
D. Cần nhiều CPU hơn

Câu 15: Công cụ nào giúp sinh hệ thức từ mô tả thuật toán?
A. Visual Studio
B. Codeforces
C. SymPy / WolframAlpha
D. GitHub Copilot

🧠 V. TỔNG HỢP & VẬN DỤNG

Câu 16: Nghiệm tổng quát của $T(n)=T(n-1)+n$?

A. $T(n) = n^2$

B. $T(n) = \log n$

C. $T(n) = \frac{n(n+1)}{2}$

D. $T(n) = 2^n$

Câu 17: Tư duy "chia để trị" liên quan đến?

A. Lập bảng

B. Cấu trúc cây

C. Cơ sở dữ liệu

D. Chuỗi Markov

Câu 18: Vì sao cần điều kiện dừng trong đệ quy?

A. Tăng tốc độ

B. Tránh in lặp

C. Tránh gọi vô hạn

D. Kiểm tra input

Câu 19: Trong bài toán "tối ưu lịch dạy", cần:

A. Sinh tổ hợp + kiểm tra ràng buộc

B. Hệ phương trình

C. Phân tích đa thức

D. Mã hóa dữ liệu

Câu 20: Độ phức tạp gần đúng của $T(n) = 2T\left(\frac{n}{2}\right) + n^2$ là?

A. $\Theta(n)$

B. $\Theta(n\log n)$

C. $\Theta(n^2)$

D. $\Theta(n^2 \log n)$

BÀI TẬP

🧪 **Bài 1:** Liệt kê tập con

Đề bài:

Cho tập $A = \{1, 2, 3, 4\}$. Liệt kê tất cả các tập con có đúng 2 phần tử.

📌 Gợi ý:

8. Sử dụng công thức tổ hợp $C(n, k)$ để xác định số lượng tập con.
9. Viết các cặp theo thứ tự tăng dần để tránh trùng lặp.

🧪 **Bài 2:** Sinh hoán vị có ràng buộc

Đề bài:
Với tập $\{A, B, C, D\}$, liệt kê tất cả các hoán vị không chứa cặp "A đứng liền trước B".

📌 Gợi ý:

4. Tổng số hoán vị: $4! = 24$
5. Dùng phép đếm bổ sung: đếm số hoán vị vi phạm điều kiện, sau đó trừ đi.
6. Có thể lập trình bằng Python để kiểm tra kết quả.

🧪 **Bài 3:** Viết hàm sinh chuỗi nhị phân
Đề bài:
Viết hàm đệ quy sinh tất cả chuỗi nhị phân độ dài n.

📌 Gợi ý:

- Mỗi bước chọn '0' hoặc '1'
- Dừng khi độ dài đạt n
- Kết quả có thể in ra hoặc lưu vào danh sách

🧪 **Bài 4:** Đếm tập con chia hết cho 3
Đề bài:
Cho tập $A = \{1, 2, 3, 4, 5, 6\}$. Đếm số tập con có tổng chia hết cho 3.

📌 Gợi ý:

- Dùng bitmask để duyệt tất cả tập con
- Với mỗi tập con, tính tổng và kiểm tra điều kiện **tổng mod 3 = 0**
- Có thể dùng brute-force để thử nghiệm

🧪 **Bài 5:** Viết hệ thức cho MergeSort
Đề bài:
Viết hệ thức truy hồi biểu diễn thời gian chạy của MergeSort với đầu vào kích thước n.
Vẽ cây truy hồi với $n = 8$.

📌 Gợi ý:

- Hệ thức: $T(n)=2T(n/2)+n$
  Vẽ cây có $\log_2 n$
- Ghi chú chi phí ở mỗi tầng để hiểu rõ cấu trúc thuật toán

🧪 **Bài 6:** Đếm số bước của Binary Search
Đề bài:
Với mảng đã sắp xếp gồm 1000 phần tử, Binary Search tối đa mất bao nhiêu bước?

📌 Gợi ý:

- Số bước tối đa là $\lceil \log_2 1000 \rceil \approx 10$
- Liên hệ đến việc tìm kiếm trong cơ sở dữ liệu lớn

🧪 **Bài 7:** Hệ thức trong thuật toán chia ba

Đề bài:
Thuật toán chia mảng nn phần tử thành 3 phần bằng nhau, xử lý từng phần đệ quy, tốn thêm nn thao tác để tổng hợp. Viết hệ thức và dự đoán độ phức tạp.

📌 Gợi ý:

- Hệ thức: $T(n)=3T(n/3)+n$
- Áp dụng Master Theorem:
  $\Rightarrow T(n)=\Theta(n\log n)$

🧪 **Bài 8:** So sánh hai thuật toán
Đề bài:
Cho $T_1(n) = n\log n$, $T_2(n) = n^2$. Với $n = 10000$n=10000, so sánh số thao tác.

📌 Gợi ý:

- Tính cụ thể: $T_1(10000) = 10000 \cdot \log_2 10000 \approx 10000 \cdot 13.3 = 133000$
  $T_2(10000) = 10000^2 = 100000000$
- Nhận xét: T_2 chậm hơn rất nhiều → ảnh hưởng lớn đến hiệu suất

🧪 **Bài 9:** Hệ thức giảm loss trong học máy
Đề bài:
Loss cập nhật theo: $L_{t+1} = 0.9 \cdot L_t$, với $L_0 = 100$
Viết hệ thức tổng quát và nhận xét về hội tụ.

📌 Gợi ý:

1. Đây là cấp số nhân: $L_t = 100 \cdot (0.9$ t) Nhận xét: $L_t \to 0$ khi $t \to \infty$
2. Ứng dụng trong tối ưu hóa deep learning

🧪 **Bài 10:** Cấu hình cây đệ quy trong n-Queens
Đề bài:
Cho thuật toán đệ quy giải bài toán n-Queens, vẽ cây truy hồi với $n = 4$. Ước lượng số cấu hình được xét.

📌 Gợi ý:

- Mỗi tầng có tối đa nn nhánh → số nút tối đa là n^n Với $n = 4$: $4^4 = 256$ cấu hình
- Thực tế ít hơn do cắt tỉa ràng buộc
- Ràng buộc trong backtracking giúp giảm đáng kể chi phí tính toán

DỰ ÁN

📓 DỰ ÁN 1: TRÌNH TẠO TEST CASE THÔNG MINH CHO THUẬT TOÁN TÌM ĐƯỜNG

🎯 Mục tiêu:
Thiết kế một chương trình sinh tất cả các lưới nhị phân kích thước $n \times n$n×n, trong đó:

- 0: ô trống (có thể đi qua)

- 1: vật cản (không thể đi qua)

⚠️ Ràng buộc: Phải tồn tại đúng một đường đi từ ô $(0, 0)$ đến $(n-1, n-1)$.

📌 Sau đó, sinh tối thiểu 10 cấu hình hợp lệ để dùng làm test case cho các thuật toán tìm đường như A* hoặc BFS.

🔧 Hướng dẫn triển khai:

🔷 **Bước 1 – Sinh lưới nhị phân bằng đệ quy/quay lui**

- Sử dụng backtracking để sinh các cấu hình grid[i][j]∈{0,1}
- Có thể dùng:
  - Bitmask nếu nn nhỏ
  - Hàm DFS(i, j) để điền dần giá trị cho lưới

🔷 **Bước 2 – Kiểm tra tính hợp lệ của lưới**

- Kiểm tra có tồn tại đường đi từ điểm bắt đầu đến điểm kết thúc
- Gợi ý thuật toán kiểm tra:
  - BFS (Breadth-First Search)
  - DFS
  - Flood-fill

🔷 **Bước 3 – Lưu lại test case**

- Với mỗi lưới hợp lệ:
  - Lưu dưới dạng file văn bản hoặc danh sách 2D
  - Có thể thiết lập mức độ khó: nhiều vật cản hơn → tìm đường khó hơn

💡 Gợi ý mở rộng:

- Cho phép sinh test case theo hướng ngược lại (từ đích về điểm xuất phát)
- Tự động sinh file đầu vào tương thích với chương trình tìm đường
- Tạo giao diện web nhỏ để sinh và xem trước test case

🧱 DỰ ÁN 2: PHÂN TÍCH ĐỘ PHỨC TẠP BẰNG TRỰC QUAN HÓA CÂY ĐỆ QUY

🎯 Mục tiêu:

Phát triển một công cụ trực quan hóa quá trình đệ quy của các thuật toán như:

- MergeSort
- Fibonacci

📌 Hiển thị:

- Cây lời gọi đệ quy
- Số lần gọi
- Dự đoán độ phức tạp theo thời gian thực

🔧 Hướng dẫn triển khai:

🔷 **Bước 1 – Cài đặt thuật toán đệ quy**

- Cài đặt thuật toán bằng Python (hoặc ngôn ngữ khác)
- Ghi lại log mỗi lần gọi, ví dụ:
  MergeSort(4) → MergeSort(2), MergeSort(2)

🔷 **Bước 2 – Vẽ cây truy hồi**

- Sử dụng thư viện đồ họa để trực quan hóa cây:

Ngôn ngữ	Thư viện đề xuất
Python	graphviz, matplotlib
JS	vis.js, D3.js

- Mỗi node đại diện cho một lời gọi $T(n)$
- Mỗi cạnh nối tới lời gọi con $T(n) \rightarrow T(n/2)$

🔷 **Bước 3 – Đếm và phân tích lời gọi**

- Đếm số node trong cây → số lần đệ quy thực hiện
- Phân tích mô hình tăng trưởng:

Thuật toán	Hệ thức	Độ phức tạp
MergeSort	$T(n)=2T(n/2)+n$	$\Theta(n \log n)$
Fibonacci	$F(n)=F(n-1)+F(n-2)$	$\Theta(2^n)$

💡 Gợi ý mở rộng:

- Cho phép người dùng nhập giá trị nn để thử nghiệm
- So sánh giữa đệ quy thường và đệ quy có nhớ (memoization)
- Ước lượng chi phí tài nguyên:
  - Bộ nhớ sử dụng
  - Thời gian thực thi
    → Từ đó đưa ra quyết định cải tiến thuật toán

✅ Lợi ích từ 2 dự án:

Dự án	Kỹ năng phát triển
Test Case tìm đường	Xử lý cấu hình, backtracking, thuật toán tìm đường (BFS, DFS, A*)
Trực quan hóa cây đệ quy	Hiểu sâu đệ quy, vẽ cây gọi, phân tích độ phức tạp và tài nguyên