# BUỔI 2: CHƯƠNG 1 - THUẬT TOÁN (PHẦN 2)

*Mục tiêu:*

- Hiểu thuật toán xử lý số.
- Nắm vững các thuật toán sắp xếp.
- Đánh giá độ phức tạp của thuật toán.

## PHẦN 1: LÝ THUYẾT

### 1.3. Thuật toán xử lý số

- Các bài toán số học: UCLN, BCNN, số nguyên tố, số chính phương, phương trình bậc hai
- Thuật toán xử lý số: Sàng Eratosthenes, Euclid
- Thực hành: Cài đặt & so sánh hiệu suất

### 1.4. Thuật toán tìm kiếm & sắp xếp

- Thuật toán tìm kiếm: Linear Search, Binary Search, Interpolation Search
- Thuật toán sắp xếp: Insertion Sort, Bubble Sort, Selection Sort
- Thực hành: Cài đặt & phân tích hiệu suất

### 1.5. Độ phức tạp của thuật toán

- Khái niệm & phân loại: O(1), O(n), O(log n), O($n^2$), O($2^n$)
- Phân tích độ phức tạp: Vòng lặp, công thức truy hồi
- Thực hành: Cài đặt đệ quy giai thừa, Fibonacci & so sánh hiệu suất

### Trắc nghiệm và Bài tập

- Ôn tập & bài tập lập trình thuật toán

## 1.3. Thuật toán xử lý số

### 1.3.1. Các bài toán xử lý số

Trong lập trình và khoa học máy tính, các bài toán xử lý số đóng vai trò quan trọng trong việc giải quyết các vấn đề liên quan đến số học, tối ưu hóa và thuật toán mã hóa. Dưới đây là ba bài toán cơ bản trong xử lý số:

### *1.3.1.1. Tìm ước chung lớn nhất (UCLN) và bội chung nhỏ nhất (BCNN)*

- **UCLN (GCD - Greatest Common Divisor):** Là số nguyên dương lớn nhất mà cả hai số đều chia hết cho nó.
- **BCNN (LCM - Least Common Multiple):** Là số nguyên dương nhỏ nhất chia hết cho cả hai số.
- **Ứng dụng:** Tối giản phân số, đồng bộ hóa chu kỳ lặp, mã hóa RSA.

**Thuật toán Euclid để tìm UCLN**

Thuật toán Euclid dựa trên tính chất:

| |
|---|
| UCLN(a,b)=UCLN(b,a mod b)<br>Nếu b=0, thì UCLN(a,b)=a. |

**Pseudocode của thuật toán Euclid:**

```
Function GCD(a, b):
    While b ≠ 0:
        temp ← b
        b ← a mod b
        a ← temp
    Return a
```

**Tính BCNN từ UCLN:**

$$BCNN(a,b) = \frac{|a \times b|}{UCLN(a,b)}$$

**Pseudocode:**

```
Function LCM(a, b):
    Return (a * b) / GCD(a, b)
```

### *1.3.1.2. Kiểm tra số nguyên tố và số chính phương*

- **Số nguyên tố:** Là số tự nhiên lớn hơn 1, chỉ chia hết cho 1 và chính nó.
- **Số chính phương:** Là số có căn bậc hai là một số nguyên.
- **Ứng dụng:** Mã hóa dữ liệu, kiểm tra tính toàn vẹn của số liệu.

**Thuật toán kiểm tra số nguyên tố**

- Cách tiếp cận thông thường: Kiểm tra số đó có ước số nào khác 1 và chính nó không.
- Tối ưu: Chỉ kiểm tra đến $\sqrt{n}$ để giảm số phép toán.

**Pseudocode kiểm tra số nguyên tố:**

```
Function isPrime(n):
    If n ≤ 1:
        Return False
    If n ≤ 3:
        Return True
    If n mod 2 = 0 or n mod 3 = 0:
        Return False
    i ← 5
    While i * i ≤ n:
        If n mod i = 0 or n mod (i + 2) = 0:
            Return False
        i ← i + 6
    Return True
```

**Thuật toán Sàng Eratosthenes (tìm tất cả số nguyên tố ≤ n)**

- Dùng mảng đánh dấu bội số của các số nguyên tố.
- Độ phức tạp: O(nloglogn).

**Pseudocode Sàng Eratosthenes:**

```
Function Sieve(n):
    Create array prime[0..n] initialized as True
    prime[0] ← prime[1] ← False
    For i from 2 to sqrt(n):
        If prime[i] is True:
            For j from i² to n step i:
                prime[j] ← False
```

```
    Return all indices i where prime[i] is True
```

### *1.3.1.3. Giải phương trình bậc hai*

Phương trình bậc hai có dạng:

$$ax^2+bx+c=0$$

- **Nghiệm:** Dựa vào công thức:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- **Phân loại nghiệm:**
    - $\Delta=b^2-4ac$
    - Nếu $\Delta>0$: Hai nghiệm phân biệt.
    - Nếu $\Delta=0$: Một nghiệm kép.
    - Nếu $\Delta<0$: Không có nghiệm thực.

**Pseudocode giải phương trình bậc hai:**

```
Function solveQuadratic(a, b, c):
    delta ← b*b - 4 * a * c
    If delta > 0:
        x1 ← (-b + sqrt(delta)) / (2 * a)
        x2 ← (-b - sqrt(delta)) / (2 * a)
        Return x1, x2
    Else If delta = 0:
        x ← -b / (2 * a)
        Return x
    Else:
        Return "No Real Solution"
```

### 1.3.2.3. So sánh hiệu suất của các thuật toán xử lý số

Để đánh giá hiệu suất của các thuật toán, ta có thể thử nghiệm trên tập dữ liệu lớn và đo thời gian thực thi:

#### *1. Thử nghiệm với tập dữ liệu lớn*

- Kiểm tra số nguyên tố cho dãy số lớn ($n>10^6$).
- So sánh thời gian giữa kiểm tra số nguyên tố thông thường và **Sàng Eratosthenes**.
- Tìm UCLN/BCNN cho dãy số lớn.

#### *2. Đánh giá thời gian chạy*

- Phương pháp kiểm tra số nguyên tố thông thường: $O(\sqrt{n})$.
- Sàng Eratosthenes: O(nloglogn).
- Thuật toán Euclid tìm UCLN: O(logb).

### Thực hành lập trình

1. Cài đặt thuật toán kiểm tra số nguyên tố bằng vòng lặp thông thường và Sàng Eratosthenes.
2. Viết chương trình tìm UCLN bằng thuật toán Euclid và tính BCNN từ UCLN.
3. Giải phương trình bậc hai với các trường hợp khác nhau.
4. So sánh hiệu suất các thuật toán khi xử lý tập dữ liệu lớn.

   **Hướng dẫn**:

   - **Chạy từng thuật toán trên tập dữ liệu lớn** (hàng triệu số).
   - **Ghi nhận thời gian thực thi** từng phương pháp.

   **Code Python**:

```
import time

# Hàm đo thời gian chạy

def measure_time(func, *args):

    start = time.time()

    result = func(*args)

    end = time.time()

    return result, end - start
```

```
# So sánh thuật toán kiểm tra số nguyên tố
print("Testing prime checking methods:")
print("Naive:", measure_time(is_prime, 10000019))
print("Sieve:", measure_time(sieve, 1000000))

# So sánh UCLN, BCNN
print("GCD:", measure_time(gcd, 123456789, 987654321))
print("LCM:", measure_time(lcm, 123456789, 987654321))
```

### 1.4.1. Thuật toán tìm kiếm

Tìm kiếm là quá trình xác định xem một phần tử có tồn tại trong danh sách hay không và nếu có, trả về vị trí của nó. Có nhiều phương pháp tìm kiếm khác nhau, mỗi phương pháp có ưu điểm riêng tùy theo tình huống sử dụng.

### 1. Tìm kiếm tuần tự (Linear Search)

👉 **Hướng dẫn**:

- **Ý tưởng**: Duyệt từng phần tử trong danh sách từ đầu đến cuối và so sánh với giá trị cần tìm.
- **Ứng dụng thực tế**:
    - Tìm kiếm trong danh sách học sinh khi điểm danh.
    - Tìm kiếm trong mảng nhỏ hoặc danh sách không sắp xếp.

👉 **Mã giả (Pseudocode)**:

```
Function LinearSearch(arr, key):
  For i from 0 to length(arr) - 1:
    If arr[i] == key:
      Return i  # Trả về vị trí tìm thấy
  Return -1  # Không tìm thấy
```

⏳ **Độ phức tạp**: O(n) - Tốc độ chậm khi danh sách lớn.

## 2. Tối ưu với Sentinel Search

👉 **Hướng dẫn**:

- **Cải tiến Linear Search** bằng cách thêm **Sentinel (phần tử chốt)** để giảm số lần kiểm tra điều kiện trong vòng lặp.
- Khi tìm thấy phần tử, có thể dừng ngay mà không cần kiểm tra ngoài phạm vi.

👉 **Mã giả (Pseudocode)**:

```
Function SentinelSearch(arr, key):
    Append key to end of arr (sentinel)
    i ← 0
    While arr[i] ≠ key:
        i ← i + 1
    If i < length(arr) - 1:
        Return i  # Tìm thấy
    Else:
        Return -1  # Không tìm thấy
```

⏳ **Độ phức tạp**: O(n), nhưng giảm số lần so sánh.

## 3. Tìm kiếm nhị phân (Binary Search)

👉 **Hướng dẫn**:

- **Điều kiện**: Dữ liệu **phải được sắp xếp** trước.
- **Ý tưởng**:
    - Tìm phần tử ở giữa danh sách.
    - Nếu phần tử cần tìm nhỏ hơn, tìm bên trái; nếu lớn hơn, tìm bên phải.
    - Lặp lại cho đến khi tìm thấy hoặc hết phần tử.

👉 **Mã giả (Pseudocode)**:

```
Function BinarySearch(arr, key, low, high):
    While low ≤ high:
        mid ← (low + high) / 2
        If arr[mid] == key:
            Return mid
        Else If arr[mid] < key:
            low ← mid + 1
        Else:
            high ← mid - 1
    Return -1  # Không tìm thấy
```

⏳ **Độ phức tạp**: O(logn) - Nhanh hơn Linear Search.

### 4. Tìm kiếm nội suy (Interpolation Search)

👉 **Hướng dẫn**:

- **Cải tiến Binary Search** khi dữ liệu **phân bố đều**.
- Thay vì chia đôi, nội suy chọn vị trí ước tính dựa trên giá trị của phần tử đầu và cuối.

👉 **Mã giả (Pseudocode)**:

```
Function InterpolationSearch(arr, key):
    low ← 0, high ← length(arr) - 1
    While low ≤ high and key ≥ arr[low] and key ≤ arr[high]:
        pos ← low + ((key - arr[low]) * (high - low) / (arr[high] - arr[low]))
        If arr[pos] == key:
            Return pos
        Else If arr[pos] < key:
            low ← pos + 1
        Else:
            high ← pos - 1
```

```
Return -1  # Không tìm thấy
```

⏳ **Độ phức tạp**: O(loglogn), nhanh hơn Binary Search khi dữ liệu đều.

### Thực hành lập trình

1. Cài đặt và chạy các thuật toán tìm kiếm:
    - Linear Search.
    - Sentinel Search.
    - Binary Search.
    - Interpolation Search.
2. So sánh hiệu suất trên danh sách lớn.
3. Chạy thử nghiệm với dữ liệu đã sắp xếp và chưa sắp xếp.

👉 **So sánh thời gian thực thi**:

```
import time


def measure_time(func, arr, key):
    start = time.time()
    result = func(arr, key)
    end = time.time()
    return result, end - start


arr = list(range(1, 1000000, 3))  # Danh sách lớn


print("Linear Search:", measure_time(linear_search, arr, 999997))
print("Binary Search:", measure_time(binary_search, arr, 999997))
print("Interpolation Search:", measure_time(interpolation_search, arr, 999997))
```

### 1.4.2. Bài toán sắp xếp dãy số

### 1. Mô tả bài toán sắp xếp

- **Bài toán**: Sắp xếp một danh sách số theo thứ tự tăng dần (hoặc giảm dần).
- **Ví dụ**:
  **Input**: [5, 2, 9, 1, 5, 6]
  **Output**: [1, 2, 5, 5, 6, 9]
- **Ứng dụng thực tế**:
  - **Quản lý dữ liệu**: Sắp xếp danh sách sinh viên theo điểm số.
  - **Tối ưu hóa xử lý thông tin**: Hỗ trợ tìm kiếm nhanh hơn.
  - **AI và Machine Learning**: Tiền xử lý dữ liệu để cải thiện độ chính xác mô hình.

⏳ **Mục tiêu**: Cài đặt và phân tích hiệu suất của 3 thuật toán sắp xếp cơ bản.

### 1.4.3. Các thuật toán sắp xếp cơ bản

#### 1. Thuật toán sắp xếp chèn (Insertion Sort)

👉 **Hướng dẫn**:

- Lấy từng phần tử, chèn vào vị trí đúng trong phần đã sắp xếp.
- **Ưu điểm**: Hiệu quả với danh sách gần như đã sắp xếp.
- **Nhược điểm**: Chậm với danh sách lớn.

👉 **Mã giả (Pseudocode)**:

```
Function InsertionSort(arr):
    For i from 1 to length(arr) - 1:
        key ← arr[i]
        j ← i - 1
        While j ≥ 0 and arr[j] > key:
            arr[j + 1] ← arr[j]
            j ← j - 1
        arr[j + 1] ← key
```

⏳ **Độ phức tạp**: $O(n^2)$.

#### 2. Thuật toán sắp xếp nổi bọt (Bubble Sort)

👉 **Hướng dẫn**:

- So sánh từng cặp phần tử và đổi chỗ nếu cần.
- **Ưu điểm**: Dễ hiểu, dễ triển khai.
- **Nhược điểm**: Kém hiệu quả với danh sách lớn.

👉 **Mã giả (Pseudocode)**:

```
Function BubbleSort(arr):
    For i from 0 to length(arr) - 1:
        For j from 0 to length(arr) - i - 1:
            If arr[j] > arr[j + 1]:
```

```
            Swap(arr[j], arr[j + 1])
```

⏳ **Độ phức tạp**: $O(n^2)$ (Chậm với danh sách lớn).

### 3. Thuật toán sắp xếp chọn (Selection Sort)

👉 **Hướng dẫn**:

- Tìm phần tử nhỏ nhất và đổi chỗ với phần tử đầu tiên.
- **Ưu điểm**: Ít đổi chỗ hơn Bubble Sort.
- **Nhược điểm**: Không nhanh hơn Insertion Sort.

👉 **Mã giả (Pseudocode)**:

```
Function SelectionSort(arr):
  For i from 0 to length(arr) - 1:
    min_index ← i
    For j from i + 1 to length(arr) - 1:
      If arr[j] < arr[min_index]:
        min_index ← j
    Swap(arr[i], arr[min_index])
```

⏳ **Độ phức tạp**: $O(n^2)$, tốt hơn Bubble Sort nhưng chậm hơn Insertion Sort khi danh sách gần như đã sắp xếp.

### Thực hành (Lập trình & Phân tích hiệu suất)

1. Cài đặt và chạy các thuật toán sắp xếp

- Insertion Sort.
- Bubble Sort.
- Selection Sort.

2. So sánh hiệu suất trên danh sách lớn (lên đến 100.000 phần tử)

👉 **Hướng dẫn**:

- Sinh viên sẽ đo thời gian chạy của từng thuật toán trên danh sách có kích thước lớn.

## 1.5. Độ phức tạp của thuật toán

### 1.5.1. Giới thiệu

**Tại sao cần đánh giá thuật toán?**

Thuật toán có thể có nhiều cách triển khai khác nhau, nhưng không phải cách nào cũng tối ưu. Đánh giá độ phức tạp giúp:

- Xác định thuật toán nào chạy nhanh hơn với đầu vào lớn.
- Tiết kiệm tài nguyên hệ thống, giảm thời gian thực thi.
- Dự đoán hiệu suất trước khi triển khai thực tế.

**Ảnh hưởng của độ phức tạp thuật toán đến hiệu suất chương trình thực tế**

Độ phức tạp xác định cách thuật toán mở rộng khi dữ liệu tăng. Ví dụ:

- **Tìm kiếm tuyến tính (O(n))** mất **1 triệu phép so sánh** cho 1 triệu phần tử.
- **Tìm kiếm nhị phân (O(log n))** chỉ mất khoảng **20 phép so sánh** cho cùng lượng dữ liệu.

Trong sắp xếp:

- **Bubble Sort ($O(n^2)$)** có thể mất **vài giờ** với dữ liệu lớn.
- **Merge Sort (O(n log n))** có thể giảm thời gian xuống **vài giây**.

Việc chọn thuật toán phù hợp giúp cải thiện hiệu suất đáng kể, đặc biệt trong các hệ thống xử lý dữ liệu lớn.

### 1.5.2. Khái niệm độ phức tạp của thuật toán

**Phân loại độ phức tạp thời gian và không gian**

- **Độ phức tạp thời gian**: Đo lường số bước thực hiện của thuật toán theo kích thước đầu vào nnn.
- **Độ phức tạp không gian**: Đánh giá lượng bộ nhớ bổ sung mà thuật toán cần sử dụng.

Tối ưu hóa thời gian thường quan trọng hơn, nhưng trong một số trường hợp, cân bằng giữa thời gian và bộ nhớ là cần thiết để đạt hiệu suất tốt nhất.

**Các lớp độ phức tạp phổ biến**

| Độ phức tạp | Mô tả | Ví dụ thuật toán |
|---|---|---|
| **O(1) – Hằng số** | Thời gian chạy không thay đổi dù đầu vào lớn hay nhỏ. | Truy xuất phần tử trong mảng arr[i]. |
| **O(n) – Tuyến tính** | Thời gian chạy tỷ lệ thuận với kích thước đầu vào. | Duyệt qua danh sách (Linear Search). |
| **O(logn) – Logarit** | Giảm số bước cần thiết bằng cách chia nhỏ dữ liệu. | Tìm kiếm nhị phân (Binary Search). |
| **O($n^2$) – Bậc hai** | Thời gian tăng theo bình phương kích thước đầu vào. | Sắp xếp chèn (Insertion Sort), sắp xếp chọn (Selection Sort). |
| **O($2^n$) – Mũ** | Thời gian tăng rất nhanh, thường không tối ưu. | Giải bài toán Tháp Hà Nội, sinh tập con. |

Chọn thuật toán có độ phức tạp thấp giúp tăng hiệu suất, đặc biệt khi xử lý dữ liệu lớn.

### 1.5.3. Đánh giá thuật toán

**Phương pháp đánh giá độ phức tạp bằng phân tích vòng lặp**

Phân tích vòng lặp là một phương pháp quan trọng để đánh giá độ phức tạp của thuật toán. Khi thực thi một chương trình, phần lớn thời gian xử lý thường nằm trong các vòng lặp. Vì vậy, việc xác định số lần lặp của một thuật toán có thể giúp ước lượng chính xác hơn độ phức tạp của nó.

1. **Xác định vòng lặp chính**: Xác định số lần thực hiện vòng lặp bằng cách xem xét phạm vi và cách thay đổi của biến điều khiển.
2. **Phân tích độ phức tạp của từng vòng lặp**: Với mỗi vòng lặp, tính toán số lần lặp dựa trên cách biến điều khiển thay đổi và điều kiện dừng.
3. **Tính toán tổng thể**: Nếu có nhiều vòng lặp lồng nhau, cần xác định tổng số lần thực hiện các lệnh trong vòng lặp ngoài cùng và kết hợp với các vòng lặp bên trong để tính toán tổng thời gian thực thi.
4. **Sử dụng ký hiệu độ phức tạp**: Sau khi xác định được số lần thực hiện các câu lệnh trong vòng lặp, ta có thể sử dụng ký hiệu độ phức tạp như O(n), O($n^2$), hoặc O(logn) để biểu diễn sự tăng trưởng của thời gian chạy theo kích thước đầu vào.

Ví dụ, xét thuật toán sắp xếp chèn (Insertion Sort). Trong trường hợp xấu nhất, nó có một vòng lặp chính chạy nnn lần, với vòng lặp bên trong chạy trung bình n/2 lần, dẫn đến tổng số lần thực hiện phép so sánh và hoán đổi là khoảng O($n^2$).

### So sánh thực tế giữa các thuật toán thông qua đo thời gian chạy

Mặc dù phân tích độ phức tạp giúp dự đoán hiệu suất của thuật toán theo lý thuyết, việc đo thời gian chạy thực tế cũng là một phương pháp quan trọng để so sánh các thuật toán.

1. **Chạy thực nghiệm**: Thực thi các thuật toán trên tập dữ liệu thực tế hoặc giả lập với các kích thước khác nhau.
2. **Ghi lại thời gian thực thi**: Đo thời gian chạy bằng các công cụ như System.nanoTime() trong Java hoặc timeit trong Python.
3. **So sánh giữa các thuật toán**: Chạy nhiều thuật toán khác nhau trên cùng một tập dữ liệu và so sánh thời gian chạy trung bình của chúng.
4. **Phân tích kết quả**: Kiểm tra xem thời gian thực tế có phù hợp với phân tích lý thuyết không, đặc biệt là với các thuật toán có độ phức tạp khác nhau như O(n),O(nlogn), và $O(n^2)$.

Ví dụ, so sánh thuật toán Quicksort và Mergesort. Trên thực tế, Quicksort thường chạy nhanh hơn Mergesort khi dữ liệu ngẫu nhiên, mặc dù cả hai đều có độ phức tạp trung bình là O(nlogn). Tuy nhiên, nếu dữ liệu đã gần như được sắp xếp trước, Mergesort có thể có hiệu suất ổn định hơn.

Phân tích vòng lặp và đo thời gian chạy giúp hiểu rõ hơn về hiệu suất thuật toán, từ đó lựa chọn phương pháp phù hợp với từng bài toán cụ thể.

### 1.5.4. Đánh giá thuật toán đệ quy

**Ví dụ về thuật toán đệ quy**

1. **Hàm đệ quy tính giai thừa**
   Hàm giai thừa được định nghĩa theo công thức truy hồi:

$$n! = \begin{cases} 1, & \text{n} = 0 \\ n \times (n-1)!, & \text{n} > 0 \end{cases}$$

Cài đặt bằng ngôn ngữ lập trình:

```
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

Độ phức tạp thời gian của thuật toán này là O(n), vì mỗi lần gọi đệ quy giảm n đi 1 cho đến khi đạt giá trị cơ sở.

2. **Dãy Fibonacci và cách tối ưu bằng quy hoạch động**
   Dãy Fibonacci được định nghĩa bằng công thức truy hồi:

$$F(n) = \begin{cases} 0, & \text{n} = 0 \\ 1, & \text{n} = 1 \\ F(n-1) + F(n-2), & \text{n} > 1 \end{cases}$$

Cài đặt đệ quy thông thường:

```
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Hàm này có độ phức tạp **lũy thừa** $O(2^n)$ do có nhiều phép gọi đệ quy lặp lại.

### Tối ưu bằng quy hoạch động (Memoization)

Sử dụng một mảng để lưu kết quả tính toán trước đó, giảm số lần tính toán lặp lại:

```
int fibonacciMemo(int n, int[] memo) {
    if (n <= 1) return n;
    if (memo[n] != 0) return memo[n];
    memo[n] = fibonacciMemo(n - 1, memo) + fibonacciMemo(n - 2, memo);
    return memo[n];
}
```

Độ phức tạp được giảm xuống **tuyến tính** O(n).

### Phân tích độ phức tạp của thuật toán đệ quy (sử dụng công thức truy hồi)

Công thức truy hồi là một phương pháp quan trọng để phân tích độ phức tạp của thuật toán đệ quy. Dưới đây là ba phương pháp phổ biến:

1. **Phương pháp mở rộng trực tiếp**
   Thay thế từng bước giá trị của hàm đệ quy cho đến khi tìm ra mẫu tổng quát.
2. **Phương pháp cây đệ quy**
   Minh họa cách gọi hàm đệ quy bằng cách vẽ cây đệ quy và đếm số lượng lời gọi.

3. **Định lý Master**

Định lý Master là công cụ mạnh mẽ để phân tích độ phức tạp của các thuật toán đệ quy chia để trị có dạng công thức truy hồi như sau:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

Trong đó:

- $T(n)$— độ phức tạp của bài toán kích thước n.
- $a$ — số lượng bài toán con (subproblems).
- $\frac{n}{b}$ — kích thước của mỗi bài toán con.
- $f(n)$ — chi phí (thời gian) để chia bài toán thành các phần nhỏ hơn và kết hợp kết quả.

**Ba trường hợp của Định lý Master**

Định lý Master gồm 3 trường hợp để đánh giá độ phức tạp của $T(n)$:

Trường hợp 1:
Nếu $f(n) = O(n^c)$ với $c < \log_b(a)$
$\rightarrow T(n) = O\left(n^{\log_b(a)}\right)$

Trường hợp 2:
Nếu $f(n) = O(n^c)$ với $c = \log_b(a)$
$\rightarrow T(n) = O(n^c \cdot \log(n))$

Trường hợp 3:
Nếu $f(n) = O(n^c)$ với $c > \log_b(a)$
$\rightarrow T(n) = O(f(n))$

**Ví dụ minh họa từ bài học**

Chúng ta sẽ lấy ví dụ từ tìm kiếm nhị phân (Binary Search).

Ví dụ: Tìm kiếm nhị phân

```
def binary_search(arr, low, high, x):
```

```
    if low <= high:
        mid = (low + high) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            return binary_search(arr, mid + 1, high, x)
        else:
            return binary_search(arr, low, mid - 1, x)
    return -1
```

**Phân tích công thức truy hồi**

Giả sử: Kích thước đầu vào là $n$.

Ở mỗi bước, chúng ta chia mảng làm đôi, kích thước $\frac{n}{2}$, nên:

a = 1 (một bài toán con cần giải)

b = 2 (chia đôi mảng)

$f(n) = O(1)$ (so sánh và chia đôi mảng)

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

**Áp dụng Định lý Master**

$\log_b(a) = \log_2(1) = 0$

$f(n) = O(1) = O(n^0)$

So sánh $c$ và $\log_b(a)$

$c = 0; \log_b(a) = 0$

Thỏa Trường hợp 2: $c = \log_b(a)$

Do đó, áp dụng trường hợp 2:

$$T(n) = O(n^0 \cdot \log(n)) = O(log(n))$$

Kết luận: Tìm kiếm nhị phân có độ phức tạp **O(log n)**.

### Thực hành (Lập trình & So sánh hiệu suất)

1. **Cài đặt và chạy thử nghiệm**
    - Cài đặt các phiên bản của hàm tính giai thừa và Fibonacci.
    - Đo thời gian chạy bằng các công cụ như System.nanoTime() trong Java hoặc timeit trong Python.
2. **So sánh hiệu suất**
    - Hàm giai thừa đệ quy so với phiên bản dùng vòng lặp.
    - Hàm Fibonacci đệ quy thông thường so với phiên bản sử dụng quy hoạch động.

**Kết luận:**

- Các thuật toán đệ quy có thể đơn giản về mặt logic nhưng có thể kém hiệu quả nếu không tối ưu.
- Sử dụng quy hoạch động hoặc chuyển đổi sang dạng lặp giúp cải thiện đáng kể hiệu suất của thuật toán.

### 1.5.5. Cách tính độ phức tạp của thuật toán

Để tính độ phức tạp của một thuật toán, chúng ta cần hiểu **độ phức tạp thời gian** (time complexity) và **độ phức tạp không gian** (space complexity). Phổ biến nhất là tính **độ phức tạp thời gian**, được biểu diễn bằng ký hiệu **Big O** - O(n). Dưới đây là hướng dẫn chi tiết từng bước.

### Bước 1: Xác định thao tác cơ bản nhất

- Xác định thao tác cơ bản nhất của thuật toán — thao tác này lặp đi lặp lại nhiều lần nhất và có ảnh hưởng chính đến thời gian thực thi.
- Ví dụ: trong vòng lặp duyệt mảng, thao tác chính là **so sánh** hoặc **gán giá trị**.

### Bước 2: Đếm số lần thực hiện thao tác cơ bản

- Xem xét số lần lặp của vòng lặp, các câu lệnh điều kiện, và các thao tác gán giá trị.
- Các vòng lặp lồng nhau sẽ **nhân số lần thực hiện**. Ví dụ, hai vòng lặp lồng nhau thì số lần thực hiện là **n × n = n²**.

### Bước 3: Loại bỏ hằng số và hệ số không đáng kể

- Khi tính độ phức tạp, chúng ta **bỏ qua hằng số** và các hệ số nhỏ hơn.
- Ví dụ: nếu có O(3n + 5), ta coi là O(n). Hoặc O($n^2$ + n) thì coi là O($n^2$).

### Bước 4: Xem xét trường hợp xấu nhất

- Thông thường, độ phức tạp được xét ở trường hợp xấu nhất (worst case).
- Trường hợp tốt nhất (best case) và trung bình (average case) cũng có thể được xem xét nhưng không phổ biến bằng.

### 📝 Ví dụ minh họa

***Ví dụ 1: Thuật toán duyệt mảng***

```
void findMax(int[] arr) {
   int max = arr[0];                // O(1)
   for (int i = 1; i < arr.length; i++) { // O(n)
      if (arr[i] > max) {           // O(n)
         max = arr[i];  }}          // O(n)
   System.out.println(max);    }     // O(1)
```

- Bước 1: Thao tác chính là **so sánh** và **gán**.
- Bước 2: Vòng lặp chạy **n-1** lần.

- Bước 3: Bỏ hằng số => O(n).
- **Độ phức tạp: O(n)**.

### *Ví dụ 2: Thuật toán duyệt hai vòng lặp lồng nhau*

```
void printPairs(int[] arr) {
    for (int i = 0; i < arr.length; i++) {          // O(n)
        for (int j = i + 1; j < arr.length; j++) {  // O(n)
            System.out.println(arr[i] + ", " + arr[j]); }}}    // O(1)
```

- Vòng lặp ngoài chạy **n** lần, vòng lặp trong chạy tối đa **n - 1** lần.
- Số lần in ra là **$n(n-1)/2 \approx O(n^2)$**.
- **Độ phức tạp: $O(n^2)$**.

### *Ví dụ 3: Thuật toán tìm kiếm nhị phân*

```
int binarySearch(int[] arr, int x) {
    int left = 0, right = arr.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] < x)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}
```

- Ở mỗi bước, mảng bị chia đôi → số bước là $\log_2(n)$.
- **Độ phức tạp: O(log(n))**.

## 🎯 Tóm tắt các mức độ phức tạp phổ biến

| Độ phức tạp | Tên gọi | Ví dụ |
|---|---|---|
| **O(1)** | Hằng số | Truy xuất phần tử trong mảng |
| **O(log(n))** | Logarit | Tìm kiếm nhị phân |
| **O(n)** | Tuyến tính | Duyệt mảng, tìm kiếm tuyến tính |
| **O(n log(n))** | Tuyến tính nhân logarit | Sắp xếp trộn (Merge Sort) |
| **O($n^2$)** | Bình phương | Sắp xếp nổi bọt, duyệt hai vòng lặp |
| **O($2^n$)** | Hàm mũ | Thuật toán đệ quy không tối ưu |
| **O(n!)** | Giai thừa | Xếp thứ tự tất cả phần tử |

### Làm thế nào để tính độ phức tạp cho thuật toán bất kỳ?

1. **Xem xét số vòng lặp**: Vòng lặp đơn O(n), vòng lặp lồng nhau O($n^2$).
2. **Xem xét điều kiện rẽ nhánh**: Xem số lần thực hiện trong trường hợp xấu nhất.
3. **Phân tích đệ quy**: Viết lại công thức đệ quy, sử dụng công thức **Master Theorem** nếu có thể.
4. **Loại bỏ yếu tố không đáng kể**: Bỏ hằng số, bỏ các thành phần không ảnh hưởng lớn.

## PHẦN 2: TRẮC NGHIỆM

### 1. Thuật toán xử lý số

**Câu 1: Thuật toán Euclid dùng để tìm:**

A. Số nguyên tố

B. Bội chung nhỏ nhất (BCNN)

C. Ước chung lớn nhất (UCLN)

D. Dãy Fibonacci

**Câu 2: Độ phức tạp của thuật toán tìm UCLN bằng Euclid là:**

A. $O(n)$

B. $O(\log n)$

C. $O(n^2)$

D. $O(2^n)$

**Câu 3: Thuật toán Sàng Eratosthenes có độ phức tạp:**

A. $O(n)$

B. $O(n^2)$

C. $O(n\log\log n)$

D. $O(\log n)$

**Câu 4: Để kiểm tra một số nguyên $n$ có phải là số nguyên tố hay không, ta cần kiểm tra đến:**

A. $n/2$

B. $n-1$

C. $\sqrt{n}$

D. $n/3$

**Câu 5: Thuật toán kiểm tra số chính phương hoạt động bằng cách:**

A. Tính căn bậc hai của số đó và kiểm tra xem nó có phải số nguyên hay không.

B. Kiểm tra số đó có chia hết cho 2 không.

C. Tính tổng các chữ số của số đó.

D. Kiểm tra số đó có chia hết cho chính nó hay không.

## 2. Thuật toán tìm kiếm & sắp xếp

**Câu 6: Độ phức tạp trung bình của thuật toán tìm kiếm tuần tự (Linear Search) là:**

A. $O(1)$

B. $O(n)$

C. $O(\log n)$

D. $O(n^2)$

**Câu 7: Khi nào thuật toán tìm kiếm nhị phân có thể được áp dụng?**

A. Khi dữ liệu được lưu trữ dưới dạng cây.

B. Khi dữ liệu đã được sắp xếp.

C. Khi dữ liệu có số lượng phần tử nhỏ.

D. Khi dữ liệu không có thứ tự.

**Câu 8: Trong các thuật toán sắp xếp sau, thuật toán nào có độ phức tạp tốt nhất trong trường hợp trung bình?**

A. Bubble Sort

B. Selection Sort

C. Quick Sort

D. Insertion Sort

**Câu 9: Độ phức tạp của thuật toán Sắp xếp chọn (Selection Sort) trong trường hợp xấu nhất là:**

A. $O(n)$

B. $O(\log n)$

C. $O(n^2)$

D. $O(n\log n)$

**Câu 10: Trong sắp xếp chèn (Insertion Sort), nếu dãy đầu vào đã được sắp xếp, độ phức tạp sẽ là:**

A. $O(n^2)$

B. $O(n\log n)$

C. $O(n)$

D. $O(\log n)$

### 3. Độ phức tạp của thuật toán

**Câu 11: Lớp độ phức tạp nào sau đây là tốt nhất về hiệu suất?**

A. $O(n^2)$

B. $O(n\log n)$

C. $O(2^n)$

D. $O(n!)$

**Câu 12: Nếu một thuật toán có độ phức tạp $O(1)$, điều đó có nghĩa là:**

A. Thuật toán luôn chạy nhanh nhất.

B. Thời gian chạy của thuật toán không phụ thuộc vào kích thước đầu vào.

C. Thuật toán có độ phức tạp tuyến tính.

D. Thuật toán có thời gian chạy theo cấp số nhân.

**Câu 13: Trong phân tích độ phức tạp của thuật toán, phương pháp nào dùng để đánh giá số lần lặp của vòng lặp?**

A. Phân tích toán học

B. Định lý Master

C. Phân tích vòng lặp

D. Sử dụng cây đệ quy

**Câu 14: Thuật toán nào dưới đây có độ phức tạp tốt hơn so với các thuật toán còn lại trong trường hợp trung bình?**

A. Sắp xếp nổi bọt

B. Sắp xếp chọn

C. Sắp xếp chèn

D. Sắp xếp nhanh

**Câu 15: Nếu một thuật toán có độ phức tạp $O(2^n)$, điều đó có nghĩa là:**

A. Nó chạy nhanh hơn thuật toán $O(n\log n)$

B. Nó chạy nhanh hơn thuật toán $O(n^2)$

C. Nó rất không hiệu quả khi $n$ lớn.

D. Nó có thể chạy trong thời gian hằng số.

### 4. Đánh giá thuật toán đệ quy

**Câu 16: Độ phức tạp thời gian của thuật toán tính giai thừa bằng đệ quy là:**

A. $O(n)$

B. $O(\log n)$

C. $O(n^2)$

D. $O(2^n)$

**Câu 17: Độ phức tạp của thuật toán Fibonacci đệ quy không tối ưu là:**

A. $O(n)$

B. $O(\log n)$

C. $O(2^n)$

D. $O(n^2)$

**Câu 18: Khi sử dụng quy hoạch động để tối ưu thuật toán Fibonacci, độ phức tạp giảm xuống:**

A. $O(n)$

B. $O(\log n)$

C. $O(n^2)$

D. $O(2^n)$

**Câu 19: Phương pháp nào dưới đây có thể dùng để phân tích độ phức tạp của thuật toán đệ quy?**

A. Phân tích vòng lặp

B. Định lý Master

C. Quy hoạch động

D. Tìm kiếm nhị phân

**Câu 20: Đối với thuật toán đệ quy có công thức truy hồi $T(n) = 2T(n-1)$+O(1), độ phức tạp của thuật toán là:**

A. $O(n)$

B. $O(\log n)$

C. $O(n^2)$

D. $O(2^n)$

## PHẦN 3: BÀI TẬP LUYỆN TẬP

Dưới đây là 10 bài tập sắp xếp từ dễ đến trung bình, chỉ hướng dẫn các bước thực hiện, yêu cầu sinh viên tự viết code.

### Chủ đề 1: Thuật toán xử lý số

#### Bài 1: Tính UCLN và BCNN

**Yêu cầu:**
Viết chương trình nhập hai số nguyên dương và tính UCLN, BCNN của chúng bằng thuật toán Euclid.

**Hướng dẫn:**

1. Nhập hai số nguyên dương a,b.
2. Áp dụng thuật toán Euclid để tìm UCLN:
    - Lặp lại phép chia lấy dư cho đến khi số dư bằng 0.
3. Tính BCNN bằng công thức:

$$BCNN(a,b) = \frac{|a \times b|}{UCLN(a,b)}$$

4. In kết quả ra màn hình.

#### Bài 2: Kiểm tra số nguyên tố

**Yêu cầu:**
Viết chương trình kiểm tra xem một số nnn có phải là số nguyên tố không.

**Hướng dẫn:**

1. Nhập số nguyên nnn.
2. Kiểm tra nếu nnn nhỏ hơn 2 thì không phải số nguyên tố.
3. Duyệt từ 2 đến $\sqrt{n}$, nếu nnn chia hết cho bất kỳ số nào trong khoảng này thì không phải số nguyên tố.
4. Nếu không tìm thấy ước số nào, kết luận nnn là số nguyên tố.
5. In kết quả ra màn hình.

## Chủ đề 2: Thuật toán tìm kiếm

### Bài 3: Tìm kiếm tuần tự

**Yêu cầu:**
Viết chương trình tìm kiếm một phần tử trong danh sách bằng tìm kiếm tuần tự.

**Hướng dẫn:**

1. Nhập danh sách số nguyên.
2. Nhập số cần tìm.
3. Duyệt từng phần tử trong danh sách, nếu tìm thấy thì trả về vị trí.
4. Nếu hết danh sách mà chưa tìm thấy, thông báo phần tử không tồn tại.

### Bài 4: Tìm kiếm nhị phân

**Yêu cầu:**
Viết chương trình tìm kiếm nhị phân trên danh sách đã sắp xếp.

**Hướng dẫn:**

1. Nhập danh sách số nguyên đã sắp xếp.
2. Nhập số cần tìm.
3. Chia đôi danh sách, so sánh phần tử giữa với số cần tìm:
    - Nếu số cần tìm nhỏ hơn, tiếp tục tìm trong nửa bên trái.
    - Nếu số cần tìm lớn hơn, tiếp tục tìm trong nửa bên phải.
    - Nếu trùng khớp, in ra vị trí của phần tử.
4. Nếu không tìm thấy, thông báo kết quả.

## Chủ đề 3: Thuật toán sắp xếp

### Bài 5: Sắp xếp chèn (Insertion Sort)

**Yêu cầu:**
Viết chương trình sắp xếp danh sách số nguyên bằng thuật toán sắp xếp chèn.

**Hướng dẫn:**

1. Nhập danh sách số nguyên.
2. Duyệt từng phần tử từ vị trí thứ hai đến cuối danh sách.
3. Chèn từng phần tử vào vị trí thích hợp trong danh sách con đã sắp xếp.
4. Tiếp tục cho đến khi toàn bộ danh sách được sắp xếp.
5. In danh sách sau khi sắp xếp.

### Bài 6: Sắp xếp nổi bọt (Bubble Sort)

**Yêu cầu:**
Viết chương trình sắp xếp danh sách số nguyên bằng thuật toán sắp xếp nổi bọt.

**Hướng dẫn:**

1. Nhập danh sách số nguyên.
2. Duyệt qua danh sách nhiều lần, trong mỗi lần duyệt:
    - So sánh từng cặp phần tử kề nhau.
    - Nếu phần tử trước lớn hơn phần tử sau, hoán đổi chúng.
3. Tiếp tục lặp lại cho đến khi danh sách được sắp xếp hoàn toàn.
4. In danh sách sau khi sắp xếp.

## Chủ đề 4: Độ phức tạp của thuật toán

### Bài 7: Đếm số lần lặp trong vòng lặp lồng nhau

**Yêu cầu:**
Xác định số lần lặp của vòng lặp lồng nhau với độ phức tạp $O(n^2)$.

**Hướng dẫn:**

1. Nhập số nguyên n.
2. Sử dụng hai vòng lặp lồng nhau:

- Vòng lặp ngoài chạy từ 0 đến n−1.
- Vòng lặp trong cũng chạy từ 0 đến n−1.

3. Tăng biến đếm mỗi khi vòng lặp trong chạy.
4. In ra số lần lặp tổng cộng.

## Chủ đề 5: Thuật toán đệ quy

### Bài 8: Đệ quy tính giai thừa

**Yêu cầu:**
Viết chương trình tính giai thừa n! bằng đệ quy.

**Hướng dẫn:**

1. Nhập số nguyên n.
2. Xây dựng hàm đệ quy:
   - Nếu n=0 trả về 1 (điều kiện dừng).
   - Nếu n>0, trả về n×giaiThừa(n−1).
3. Gọi hàm và in kết quả.

### Bài 9: Đệ quy tính Fibonacci

**Yêu cầu:**
Viết chương trình tính số Fibonacci thứ n bằng đệ quy.

**Hướng dẫn:**

1. Nhập số nguyên n.
2. Xây dựng hàm đệ quy:
   - Nếu n=0 hoặc n=1, trả về n.
   - Nếu n>1, trả về tổng của Fibonacci thứ n−1 và Fibonacci thứ n−2.
3. Gọi hàm và in kết quả.

### Bài 10: Tối ưu hóa Fibonacci bằng quy hoạch động

**Yêu cầu:**
Viết chương trình tối ưu hóa thuật toán Fibonacci bằng cách lưu kết quả trung gian (memoization).

**Hướng dẫn:**

1. Nhập số nguyên nnn.
2. Dùng một danh sách (mảng) hoặc từ điển để lưu giá trị đã tính trước đó.
3. Xây dựng hàm đệ quy:
    - Nếu giá trị Fibonacci đã được lưu, trả về ngay mà không tính lại.
    - Nếu chưa có, tính toán và lưu kết quả vào danh sách.
4. Gọi hàm và in kết quả.

## PHẦN 4: BÀI TẬP DỰ ÁN

**Số lượng sinh viên thực hiện: 2**
**Mô tả dự án**

Sinh viên sẽ xây dựng một chương trình so sánh hiệu suất của ba thuật toán tìm kiếm:

1. Tìm kiếm tuần tự (Linear Search)
2. Tìm kiếm nhị phân (Binary Search)
3. Tìm kiếm nội suy (Interpolation Search)

Chương trình sẽ thực hiện các thuật toán trên **dữ liệu ngẫu nhiên có kích thước lớn** và **ghi nhận thời gian thực thi** để đánh giá thuật toán nào hoạt động tốt nhất với từng loại dữ liệu đầu vào.

**Yêu cầu thực hiện**

1. **Chuẩn bị dữ liệu:**
    - Sinh viên tạo một danh sách số nguyên ngẫu nhiên có kích thước **100.000 phần tử**.
    - Danh sách này được sử dụng cho Linear Search, Binary Search và Interpolation Search.
    - Phải tạo thêm **một danh sách đã sắp xếp** để kiểm thử Binary Search và Interpolation Search.

2. **Cài đặt thuật toán:**
    - Viết ba thuật toán tìm kiếm với cùng giao diện hàm.
    - Đảm bảo chương trình có thể thực thi từng thuật toán với cùng một tập dữ liệu.
3. **Đo thời gian thực thi:**
    - Sử dụng thư viện hỗ trợ đo thời gian (như time trong Python hoặc System.nanoTime() trong Java).
    - Chạy mỗi thuật toán **nhiều lần** để lấy thời gian trung bình.
4. **Phân tích kết quả:**
    - So sánh hiệu suất giữa các thuật toán.
    - Xác định trường hợp nào mỗi thuật toán hoạt động hiệu quả nhất.
    - Viết báo cáo mô tả **tình huống sử dụng thực tế** của từng thuật toán.