Độ phức tạp thuật toán

Mục tiêu:

- Hiểu và giải thích được độ phức tạp thuật toán là gì và tại sao nó quan trọng.
- Sử dụng thành thạo ký hiệu Big O để mô tả hiệu năng của thuật toán.
- Phân tích và tính toán được độ phức tạp về thời gian và không gian cho các đoạn mã C#.
- Biết được độ phức tạp của các thao tác trên những cấu trúc dữ liệu phổ biến trong .NET (List<T>, Dictionary<TKey, TValue>, v.v.).
- Áp dụng kiến thức để lựa chọn thuật toán, cấu trúc dữ liệu phù hợp và tối ưu hóa hiệu năng cho ứng dụng trong thực tế.

- Giới thiệu về Độ phức tạp của thuật toán
- Big O
- Quy tắc tính độ phức tạp trên mã nguồn
- Phân tích các cấu trúc dữ liệu phổ biến trong C#
- Phân tích các thuật toán kinh điển

Giới thiệu về Độ phức tạp của thuật toán

Mỗi dòng code khi được thực thi đều phải "trả giá". Cái "giá" này không không hẳn là tiền mà là tài nguyên của máy tính (RAM, CPU, GPU...). Độ phức tạp của thuật toán tương ứng với cái giá mà chúng ta phải trả. Tính toán độ phức tạp của thuật toán đồng nghĩa với việc tính toán cái giá mà chúng ta phải trả.

Độ phức tạp của thuật toán là gì?

Độ phức tạp của thuật toán (Algorithmic Complexity) là hiểu đơn giản là cách mô tả lượng tài nguyên mà một thuật toán cần để chạy xét theo sự thay đổi của kích thước dữ liệu đầu vào.

Kích thước dữ liệu đầu vào (input size) thường được ký hiệu là *n.* Đây được coi là yếu tố cốt lõi.

Ví dụ:

- Trong bài toán sắp xếp một mảng, n là số lượng phần tử trong mảng.
- Trong bài toán xử lý chuỗi, n là độ dài chuỗi
- Trong bài toán duyệt đồ thị, n là số đỉnh hoặc số cạnh

Độ phức tạp cho chúng ta biết:" Khi n lớn dần lên, thuật toán của chúng ta sẽ chạy chậm đi hoặc tốn nhiều bộ nhớ hơn **theo quy luật nào**"

Tại sao cần quan tâm?

Trong môi trường học thuật, thuật toán chỉ cần cho ra kết quả đúng. Nhưng trong thực tế, môi trường product chuyên nghiệp một thuật toán chạy quá chậm hoặc tốn quá nhiều bộ nhớ là một thuật toán vô dụng.

Một thuật toán quá chậm có thể dẫn tới những vấn đề sau:

- Trải nghiệm người dùng tệ hại (thời gian chờ đợi lâu, đơ giật, crash..)
- Chi phí vận hành lớn (yêu cầu quá nhiều RAM, CPU, điện...)
- Khả năng mở rộng (Tốn quá nhiều tài nguyên dẫn đến khó mở rộng)

Độ phức tạp thời gian và Độ phức tạp không gian

?? ph?c t?p th?i gian (Time complexity)

Độ phức tạp thời gian mô tả thời gian chạy của thuật toán tăng lên như thế nào khi kích thước đầu vào tăng lên

Đây là yếu tố thường được quan tâm nhất. Thời gian ở đây không phải là giây, phút mà là số lượng thao tác cơ bản (gán, so sánh, cộng trừ...) mà máy tính phải thực hiện. Lý do là vì thời gian chạy thực tế còn phụ thuộc vào tốc độ phần cứng, ngôn ngữ lập trình... khi đó số thao tác cần thực hiện được coi là thước đo tuyệt đối.

?? ph?c t?p không gian (Space complexity)

Độ phức tạp không gian mô tả lượng bộ nhớ mà một thuật toán cần sử dụng tăng lên như thế nào khi kích thước đầu vào tăng.

Lượng bộ nhớ này bao gồm cả không gian lưu trữ dữ liệu đầu vào và bất kỳ không gian bổ sung nào mà thuật toán cần để hoạt động (VD mảng tạm, biến cục bộ)

Thông thường, có một sự **đánh đổi** giữa thời gian và không gian. Đôi khi bạn có thể làm cho thuật toán chạy nhanh hơn bằng cách sử dụng nhiều bộ nhớ hơn và ngược lại. Nhiệm vụ của một kỹ sư giỏi là hiểu rõ sự đánh đổi này và chọn giải pháp phù hợp nhất cho bài toán cụ thể.

Big O

Big O Notation là một quy ước toán học được các lập trình viên toàn thế giới sử dụng để mô tả hiệu năng của thuật toán một cách đồng nhất.

Big O là gì?

Big O mô tả **tốc độ tăng** của thời gian và không gian mà một thuật toán yêu cầu trong **trường hợp xấu nhất**, khi kích thước đầu vào n tiến đến vô cùng.

Tốc độ tăng: Big O không cho bạn biết thuật toán chạy chính xác trong thời gian bao lâu. Nó cho bạn biết quy luật mà thời gian chạy sẽ tăng lên.

Trường hợp xấu nhất: Khi phân tích thuật toán, chúng ta thường quan tâm đến kịch bản tệ nhất có thể xảy ra. Vì nó cho chúng ta một sự đảm bảo. Nếu chúng ta biết thuật toán của mình chạy không quá x giây trong trường hợp xấu nhất chúng ta có thể yên tâm rằng nó luôn đáp ứng được trong mọi tình huống thực tế.

Các loại độ phức tạp phổ biến

O(1) - Constant

Thời gian thực thi không phụ thuộc vào kích thước đầu vào n.

Trường hợp thường gặp:

- Truy cập một phần tử trong mảng bằng index
- Lấy một giá trị trong Dictionary thông qua key
- Thêm, xóa một phần tử ở cuối List

O(log n) - Logarithmic

Thời gian thực thi tăng rất chậm khi n tăng. Khi gấy đôi n, thời gian chỉ tăng thêm một lượng rất nhỏ.

Trường hợp thường gặp:

- Thuật toán tìm kiếm nhị phân trên tệp dữ liệu đã sắp xếp

O(n) - Linear

Thời gian thực thi tỷ lệ thuận với kích thước đầu vào n. Nếu n tăng gấp đôi thì thời gian cũng tăng gấp đôi

Trường hợp thường gặp:

- Duyệt qua tất cả các phần tử để tìm một giá trị
- In ra tất cả các phần tử trong mảng

O(n log n) - Linearithmic

Hiệu quả hơn O(n^2) nhưng kém hiệu quả hơn O(n). Đây là tiêu chuẩn vàng cho các thuật toán sắp xếp dựa trên so sánh.

O(n log n) tương đương với việc thực hiện một thao tác O(log n) với mỗi phần tử.

Trường hợp thường gặp:

- Các thuật toán sắp xếp như merge sort, quick sort.
- IntroSort được sử dụng bởi `List<T>.Sort()` trong C#

O(n^2) - Quadratic

Thời gian thực thi tăng theo bình phương của n. Nếu n tăng 10 lần thì thời gian/không gian tăng 100 lần

Trường hợp thường gặp:

- Các vòng lặp lồng nhau
- Các thuật toán Bubble Sort, Selection Sort

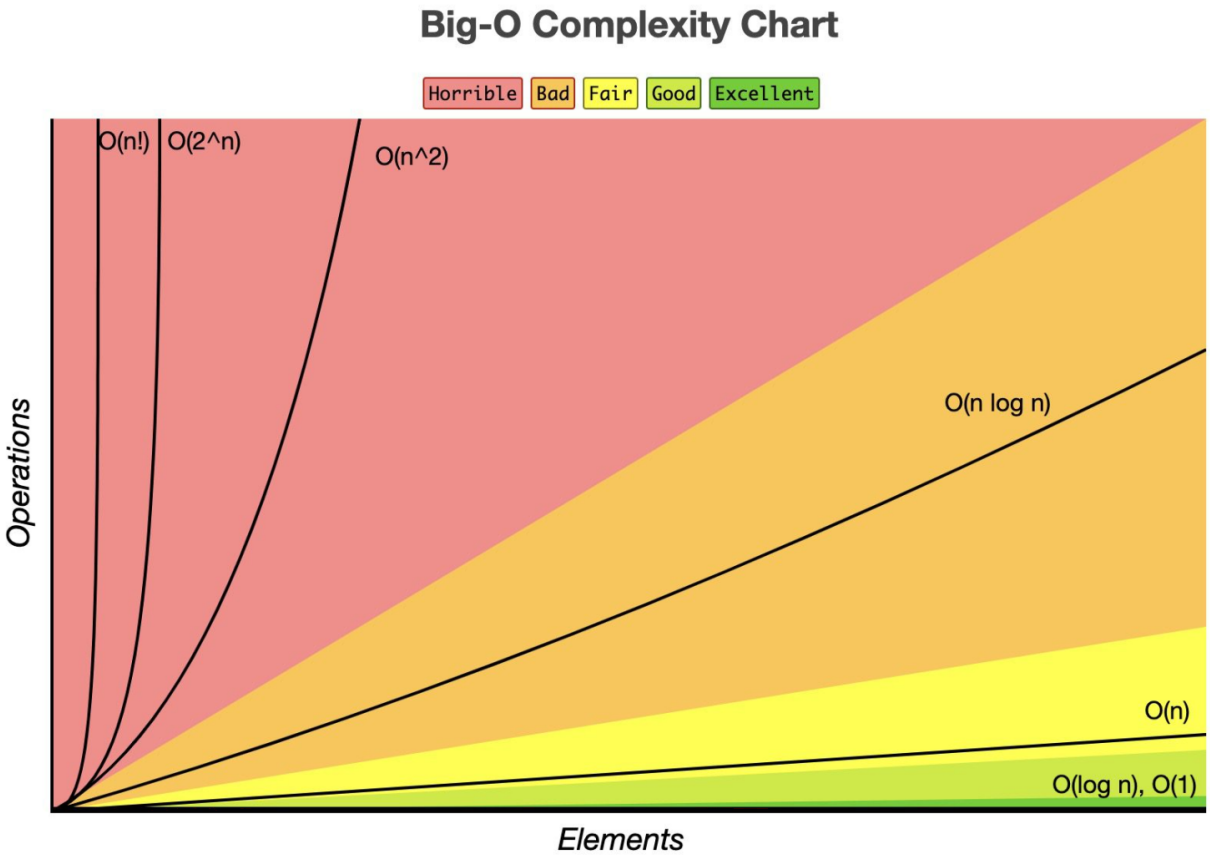
O(2^n) - Exponential và O(n!) Factorial

Cực kỳ chậm và nhanh chóng trở nên bất khả thi ngay cả với n tương đối nhỏ

Trường hợp thường gặp:

- O(2n) Tính số Fibonacci bằng đệ quy đơn giản
- O(n!) Giải quyết bài toán Traveling Salesman Problen bằng cách vét cạn

Xếp hạng trực quan


Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

Quy tắc tính độ phức tạp trên mã nguồn

Các quy tắc nên tảng

Quy t?c b? h?ng s?

Big O chỉ quan tâm đến tốc độ tăng trưởng khi n rất lớn. Do đó, các hệ số hằng số không có ý nghĩa

Ví dụ

- O(2n) → O(n)
- O(500) → O(1)
- O(n/3) → O(n)

Quy t?c l?y thành ph?n v??t tr?i

Khi một thuật toán có nhiều thành phần, độ phức tạp tổng thể được quyết định bởi thành phần chậm nhất

Ví dụ

- $O(n^2 + n) \rightarrow O(n^2)$
- $O(2^n + n^2) \rightarrow O(2^n)$
- O(n log n + n log n) → O(n log n)

Các thao tác c? b?n là O(1)

Một thao tác cơ bản không phụ thuộc vào kích thước đầu vào được coi là có độ phức tạp hằng số O(1)

Phân tích mã nguồn

Các câu l?nh tu?n t?

```
public void DoSomething(int[] numbers)
{
    // Phân tích t?ng dòng:
    Console.WriteLine("B?t ??u...");            // O(1)
```

```
    int sum = numbers[0] + numbers[1];          // O(1)

    for (int i = 0; i < numbers.Length; i++)    // O(n)
    {
        Console.WriteLine(numbers[i]);
    }

    Console.WriteLine("K?t thúc.");             // O(1)
}
```

Tổng độ phức tạp = O(1) + O(1) + O(n) + O(1) = O(n+3)

Áp dụng quy tắc loại bỏ hằng số và lấy phần vượt trội ta có độ phức tạp là O(n)

Vòng l?p

```
// n = items.Count
public void LoopExample(List<string> items)
{
    // Vòng l?p này ch?y n l?n
    foreach (var item in items)
    {
        // Kh?i l?nh bên trong ch? có các thao tác O(1)
        Console.WriteLine("Item: " + item); // O(1)
        int length = item.Length;           // O(1)
    }
}
```

Mỗi vòng lặp tốn O(1) + O(1) = O(1)

Vòng lặp này lặp lại n lần → độ phức tạp là O(n)

Vòng l?p l?ng nhau

```
// n = numbers.Count
public void NestedLoopExample(List<int> numbers)
{
    // Vòng l?p ngoài ch?y n l?n
    foreach (var num1 in numbers)
    {
        // Vòng l?p trong c?ng ch?y n l?n v?i m?i l?n l?p c?a vòng ngoài
        foreach (var num2 in numbers)
        {
            // Thao tác bên trong là O(1)
            Console.WriteLine($"{num1}, {num2}");
        }
    }
}
```

Độ phức tạp của mỗi vòng lặp trong là O(n)

Vòng lặp trong được lặp lại n lần.

Độ phức tạp của thuật toán là O(n*n) = O(n^2)

Câu l?nh ?i?u ki?n

```
public void ConditionalExample(int condition, int[] data)
{
    if (condition == 0)
    {
        // Nhánh này có ?? ph?c t?p O(1)
        Console.WriteLine("Condition is true.");
    }
    else if (condition ==  1)
    {
        // Nhánh này có ?? ph?c t?p O(n)
        foreach (var item in data)
        {
            Console.WriteLine(item);
        }
    }
    else
    {
        // Nhánh này có ?? ph?c t?p là O(n2)
        foreach (var num1 in numbers)
        {
            foreach (var num2 in numbers)
            {
                Console.WriteLine($"{num1}, {num2}");
            }
        }
    }
}
```

Với câu lệnh điều kiện độ phức tạp của thuật toán là độ phức tạp của nhánh tệ nhất.

Trong trường hợp này là $O(n2)$

Phân tích các cấu trúc dữ liệu phổ biến trong C#

Việc sử dụng cấu trúc dữ liệu với đúng công việc cũng là một kỹ năng quan trọng trong lập trình. Việc lựa chọn sai có thể khiến cho ứng dụng của bạn chậm đi đáng kể.

Tóm tắt

Kiểu dữ liệu	Truy cập (index/key)	Tìm kiếm	Thêm (cuối)	Thêm (giữa)	Xóa (giữa)
Array	O(1)	O(n)	O(1) *(chưa đầy)* O(n) *(đầy)*	O(n)	O(n)
List<T>	O(1)	O(n)	O(1)	O(n)	O(n)
Dictionary<TKey, TValua>	O(1)	O(1) *(theo key)* O(n) *(theo value)*	O(1)	x	O(1)
HashSet<T>	x	O(1)	O(1)	x	O(1)
Queue<T>	x	O(n)	O(1)	x	O(1)
Stack<T>	x	O(n)	O(1)	x	O(1)

Lưu ý khi sử dụng

- Sử dụng Array, List khi cần truy cập nhanh theo chỉ số.
  - Array nhẹ hơn do không có chi phí quản lý
  - List nhiều tiện ích (thêm, sửa, xoá, kiểm tra...)
- Sử dụng Dictionary khi cần tìm kiếm nhanh theo khoá.
- Sử dụng Queue, Stack khi cần thêm/xoá ở đầu hoặc cuối.
- Sử dụng HashSet cho danh sách không trùng lặp, không cần truy cập từng phần tử riêng lẻ.

Phân tích các thuật toán kinh điển

Thuật toán tìm kiếm

Tìm ki?m tuy?n tính (Linear Search)

- Linear Search được triển khai bằng cách duyệt từ đầu đến cuối tất cả các phần tử và so sánh nó với giá trị cần tìm. Thuật toán dừng lại khi duyệt hết danh sách hoặc tìm thấy phần tử cần tìm.
- Trường hợp xấu nhất là khi phần tử cần tìm nằm ở cuối danh sách hoặc không tìm thấy phần tử cần tìm. Khi này buộc phải duyệt qua tất cả danh sách.
- **Thuật toán có độ phức tạp O(n)**

Tìm ki?m nh? phân (Binary Search)

- Binary Search được thực hiện trên một danh sách đã được sắp xếp
- Mảng cần tìm sẽ được chia thành 2 phần left và right, phần tử ở giữa gọi là mid. Dựa vào mid ta xác định xem phần tử cần tìm nằm ở left hoặc right. Loại bỏ phía không chứa phần tử cần tìm rồi lặp lại cho đến khi tìm được phần tử.
- **Thuật toán có độ phức tạp là O(log n)**

Thuật toán sắp xếp

S?p x?p n?i b?t (Bubble Sort)

- Bubble Sort được thực hiện bằng cách lặp qua toàn bộ danh sách, so sánh 2 phần tử liền kề và đổi chỗ chúng nếu cần.
- Thuật toán này sử dụng 2 vòng lặp lồng nhau. Vòng ngoài xác định số lượt duyệt, vòng trong thực hiện so sánh và đổi chỗ
- Thuật toán có độ phức tạp là $O(n^2)$

S?p x?p nhanh (QuickSort)

- QuickSort chọn một phần tử làm pivot sau đó sắp xếp lại mảng sao cho các phần tử lớn hoặc bằng pivot nằm một phía và nhỏ hơn nằm một phía. Tiếp tục đệ quy với 2 danh sách nằm ở 2 phía của pivot. Quá trình được lặp lại đến khi các mảng con chỉ còn 1 phần tử.
- **Thuật toán có độ phức tạp là O(n log n)** xảy ra khi pivot được chọn gần với giá trị ở giữa danh sách

- Trong trường hợp xấu nhất **độ phức tạp là O(n2)** xảy ra khi pivot là phần tử lớn nhất hoặc nhỏ nhất.