ANALYSIS OF ALGORITHMS

Giảng viên: PGS.TS. Lê Đình Duy
ThS. Nguyễn Thanh Sơn

Nhóm 5:

Vũ Quí San - 18520143
Phạm Mạnh Tiến - 18520166
Nguyễn Vương Thịnh - 18520367

Mục lục

1. Khái niệm về phân tích thuật toán

Phân tích thuật toán là xác định **độ phức tạp thời gian** và **độ phức tạp không gian** của một thuật toán dựa trên **kích thước của input.**

Thời gian thực thi T(n) của một chương trình hiện thực một thuật toán:

$$T(n) \approx c(op) \times C(n)$$

Trong đó:

- **c(op)**: thời gian thực thi một hoạt động cơ bản của thuật toán trên một máy tính cụ thể

- **C(n):** Số lượng hoạt động cơ bản cần thiết để thực thi một thuật toán

2. Tầm quan trọng của phân tích thuật toán

Số lượng bản ghi	10	20	50	100	1000	5000
Thuật toán 1	0.00s	0.01s	0.05s	0.47s	23.92s	47min
Thuật toán 2	0.05s	0.05s	0.06s	0.11s	0.78s	14.22s

Thuật toán 2 sẽ hoạt động tốt hơn với số bản ghi >=1000

+ Số bản ghi <1000 thì dùng (1)

+ Số bản ghi >=1000 thì dùng (2)

Nguồn:https://vnoi.info/wiki/translate/topcoder/Computational-Complexity-Section-1.md?fbclid=IwAR0E1WhdygSAT2q9PntLosL9tbDoBR4s5F_kENku7_CaCqDjCCSEq8vpVio

2. Tầm quan trọng của phân tích thuật toán

1. **Tác dụng của phân tích:**
   + Phân tích vấn đề: hiểu được cốt lõi của bài toán
   + Phân tích cách xử lý: cải tiến hoặc chọn ra cách tốt hơn

2. **Hiểu rõ được ưu, nhược điểm của thuật toán:**

   + Cải tiến thuật toán cho hiệu quả hơn
   + So sánh các thuật toán giải quyết cùng một vấn đề để chọn ra thuật toán tốt hơn

3. Các phương pháp phân tích thuật toán

	Phân tích dựa trên **lý thuyết** (Asymptotic analysis)	Phân tích dựa trên **thực nghiệm** (Empirical metrics)
Khái niệm	Tính toán, *ước lương* thời gian chạy và bộ nhớ của các thuật toán và so sánh	Căn cứ vào thời gian chạy *thực tế* của máy tính và so sánh
Ưu điểm	Tổng quát độ phức tạp của thuật toán với mọi độ lớn của input.	Số liệu trực quan, dễ đánh giá, so sánh
Nhược điểm	Không so sánh được các thuật toán có thời gian xấp xỉ nhau.	Không thể tổng quát mọi độ lớn của input.

4. Asymptotic Notation

- Asymptotic Notation là phương pháp toán học giúp biểu diễn độ phức tạp thời gian (time complexity) của thuật toán cho việc phân tích tiệm cận hiệu quả của thuật toán (asymptotic analysis)

- Có 3 asymptotic notation thường được sử dụng để biểu diễn độ phức tạp thời gian của thuật toán: **Θ** (Big-Theta), **O** (Big-O), **Ω** (Big-Omega)

- Gọi f(n) là hàm số biểu diễn thời gian thực thi của thuật toán. g(n) là hàm số tiệm cận f(n).

⇒ f, g là các hàm số dương, không giảm trên tập số nguyên dương.

4. Asymptotic Notation

- Kí pháp **O** (Big - O): Hàm số f(n) được gọi thuộc **O**(g(n)), (kí hiệu $f(n) \in O(g(n))$) nếu tồn tại các hằng số c và $n_0 \geq 0$ thỏa điều kiện sau:

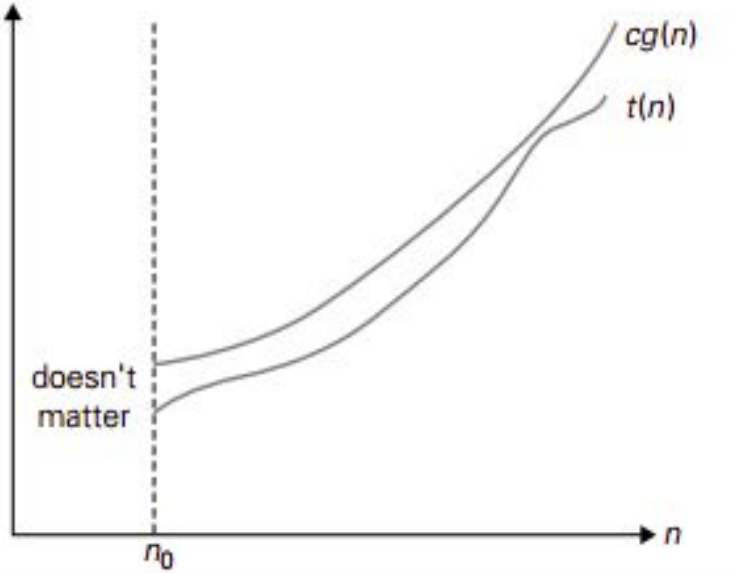
$$f(n) \leq c * g(n), \forall\ n \geq n_0$$

Ví dụ: $f(n) = 100n^2 + 50$

Ta có: $100n^2 + 50 \leq 100n^2 + 50n \leq 100n^2 + 50n^2 = 150n^2$

Với $c = 150$ và $n_0 = 1 \Rightarrow f(n) \in O(n^2)$, $g(n) = n^2$

$\Rightarrow$ Big - O thường được sử dụng để xác định **cận trên** của độ phức tạp thời gian của thuật toán



Hình 4.1: $f(n) \in O(g(n))$

4. Asymptotic Notation

- Kí pháp **Θ** (Big - Theta): Hàm số f(n) được gọi thuộc **Θ**(g(n)), (kí hiệu f(n) ∈ **Θ**(g(n))) nếu tồn tại các hằng số c_1, c_2 và $n_0 \geq 0$, thỏa điều kiện sau:

$$c_2 * g(n) \leq f(n) \leq c_1 * g(n), \ \forall \ n \geq n_0$$

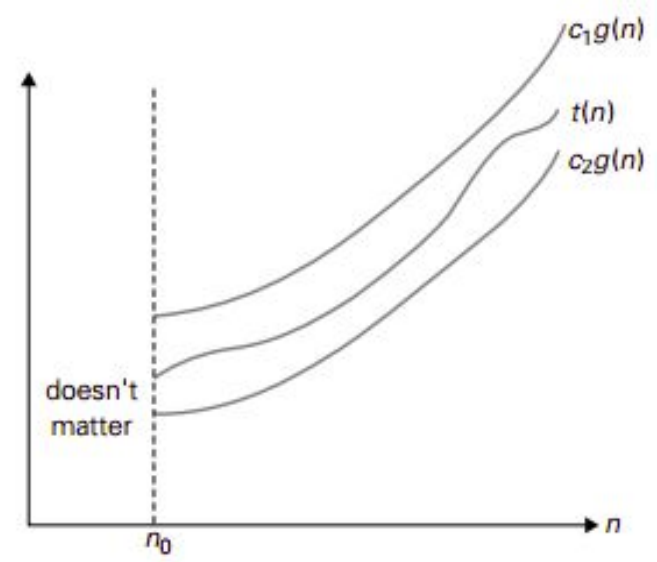
Ví dụ: Chứng minh $2n(n-1) \in \Theta(n^2)$

Ta có: $2n(n-1) = 2n^2 - 2n \leq 2n^2 \ \forall \ n \geq n_0$

$$2n(n-1) = 2n^2 - 2n \geq 2n^2 - 2n \times 0.5n = n^2$$

Với $c_2 = 1$, $c_1 = 2$ và $n_0 = 0 \Rightarrow f(n) = \Theta(n^2)$, $g(n) = n^2$

$\Rightarrow$ Big - Theta (**Θ**) cho chúng ta biết gần đúng tiệm cận của độ phức tạp thuật toán



Hình 4.2: f(n) ∈ **Θ**(g(n))

4. Asymptotic Notation

- Kí pháp **Ω** (Big - Omega): Hàm số f(n) được gọi thuộc **Ω**(g(n)), (kí hiệu f(n) $\in$ **Ω**(g(n))) nếu tồn tại các hằng số c và $n_0 \geq 0$ thỏa điều kiện sau:

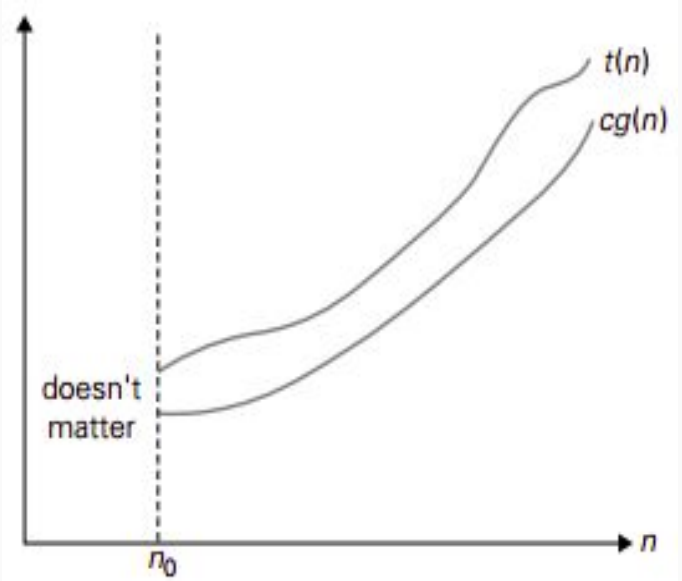
$$f(n) \geq c * g(n), \forall\ n \geq n_0$$

Ví dụ: Chứng minh $n^3 \in \mathbf{\Omega}(n^2)$

Ta có: $n^3 \geq \mathbf{\Omega}(n^2)$

Với c = 1 và $n_0 = 0 \Rightarrow f(n) = \mathbf{\Omega}(n^2), g(n) = n^2$

$\Rightarrow$ Big - Omega thường được dùng để xác định cận dưới độ phức tạp thuật toán



Hình 4.3: f(n) $\in$ **Θ**(g(n))

4. Asymptotic Notation

Bốn quy tắc cơ bản để tính độ phức tạp thuật toán:

1. Quy tắc bỏ hằng số
2. Quy tắc lấy max
3. Quy tắc cộng
4. Quy tắc nhân

Phần chứng minh bốn quy tắc xem tại đây

4. Asymptotic Notation

1. Quy tắc bỏ hằng số:

Bỏ qua các hằng số trong công thức.

Ví dụ:

Cho $T(n) = 3n^3 + 2n^2 + 4000$

Ta bỏ qua các hằng số $3, 2, 4000$ và thu được:

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

4. Asymptotic Notation

2. Quy tắc lấy max:

Giữ lại số hạng (term) lớn nhất.

Ví dụ:

Với $T(n) = O(n^3 + n^2)$

Vì $n^3 > n^2$ nên ta giữ lại n^3: $T(n) = O(n^3 + n^2) = O(n^3)$

4. Asymptotic Notation

3. Quy tắc cộng:

Nếu hai đoạn chương trình P1 và P2 có thời gian thực hiện lần lượt là $T_1(n) = O(f(n))$ và $T_2(n) = O(g(n))$ thì thời gian thực hiện P1 rồi đến P2 là:

$$T_1(n) + T_2(n) = O(\max(f(n), g(n)))$$

Ví dụ nếu $T_1(n) = O(n^2)$ và $T_2(n) = O(n)$ thì:

$$T_1(n) + T_2(n) = O(max(n^2, n)) = O(n^2)$$

4. Asymptotic Notation

4. Quy tắc nhân:

Nếu P có thời gian thực hiện $T(n) = O(f(n))$, thực hiện $k(n)$ lần đoạn chương trình P với $k(n) = O(g(n))$ thì độ phức tạp tính toán sẽ là $O(g(n) \times f(n))$

4. Asymptotic Notation

Ví dụ:

Cho $T_1(n) = n^3 + n^2 + n + 1$ và $T_2(n) = n^2 + n + 1$

Tính $T_1(n) \times T_2(n)$?

Cách 1:

$T_1(n) \times T_2(n) = (n^3 + n^2 + n + 1) \times (n^2 + n + 1) = n^5 + 2n^4 + 3n^3 + 3n^2 + 2n + 1 = O(n^5)$

Cách 2:

Ta có $T_1(n) = O(n^3)$ và $T_2(n) = O(n^2)$

$\Rightarrow T_1(n) \times T_2(n) = O(n^3 \times n^2) = O(n^5)$

4. Asymptotic Notation

Tính độ phức tạp vòng lặp:

- O(n^c): Độ phức tạp của **c** vòng lặp lồng nhau

+ **c = 0:** đoạn chương trình không có vòng lặp, đpt vòng lặp trở thành O(1).

+ **c = 1:** đoạn chương trình chỉ có một vòng lặp thực hiện **n** lần, do đó đpt là O(n).

```
for (int i = 1; i <=n; i += c) {
    for (int j = 1; j <=n; j += c) {
       // some O(1) expressions
    }
}

for (int i = n; i > 0; i -= c) {
    for (int j = i+1; j <=n; j += c) {
       // some O(1) expressions
}
```

Đoạn chương trình có độ phức tạp $O(n^2)$

4. Asymptotic Notation

Tính độ phức tạp vòng lặp:

- **O(Logn):** Biến lặp bị nhân (hay chia) bởi một hằng số sau mỗi vòng lặp

  + Thường có tại các thuật toán liên quan đến cấu trúc **cây** như Binary Tree Search.

```
for (int i = 1; i <=n; i *= c) {
    // some O(1) expressions
}
for (int i = n; i > 0; i /= c) {
    // some O(1) expressions
}
```

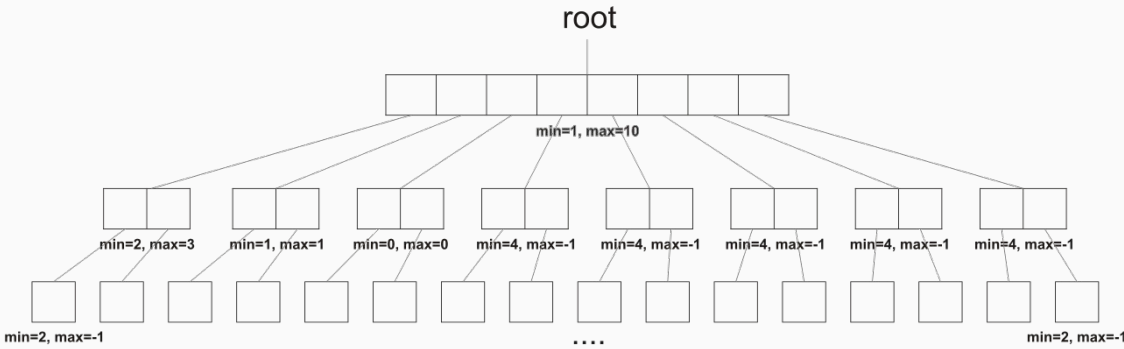
4. Asymptotic Notation

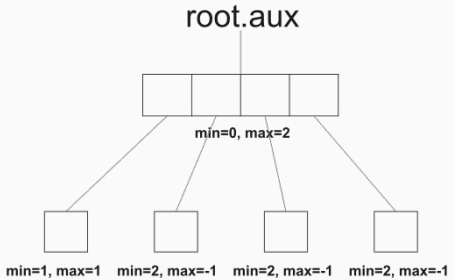
Tính độ phức tạp vòng lặp:

- O(LogLogn): Biến lặp bị nhân (hay chia) bởi **lũy thừa của một hằng** số sau mỗi vòng lặp

+ Nhanh hơn **O(Logn)** và thường ít gặp

+ Vd: Cấu trúc *van Emde Boas tree* (hình bên)





4. Asymptotic Notation

Tính độ phức tạp vòng lặp:

-Trường hợp hai **vòng lặp liên tiếp nhau** (xem hình bên), ta tính tổng độ phức tạp của chúng. Chú ý đối với vòng lặp thì **không áp dụng quy tắc cộng** đã đề cập ở trên:

$$O(m) + O(n) = O(m + n)$$

```
for (int i = 1; i <=m; i += c) {
    // some O(1) expressions
}
for (int i = 1; i <=n; i += c) {
    // some O(1) expressions
}
```

4. Asymptotic Notation

Tính độ phức tạp thuật toán đệ quy:

Có ba phương pháp chính:

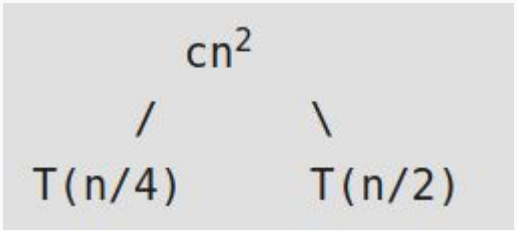
- Substitution Method: Ta dự đoán ĐPT rồi dùng toán học để chứng minh dự đoán đó là đúng.
- Recurrence Tree Method: Vẽ cây mô tả các tầng đệ quy và tính tổng thời gian của tất cả các tầng đó.
- Master Method: được rút ra từ pp Recurrence Tree Method.

Recurrence Tree Method

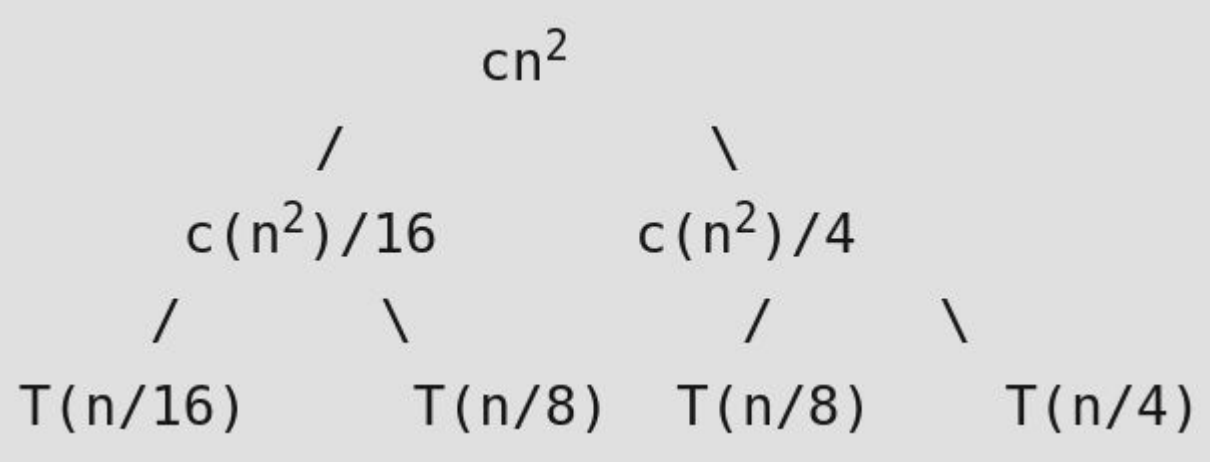
Một hàm đệ quy với tham số n có thời gian $T(n)$ như sau:

$$T(n) = T(n/4) + T(n/2) + cn^2$$

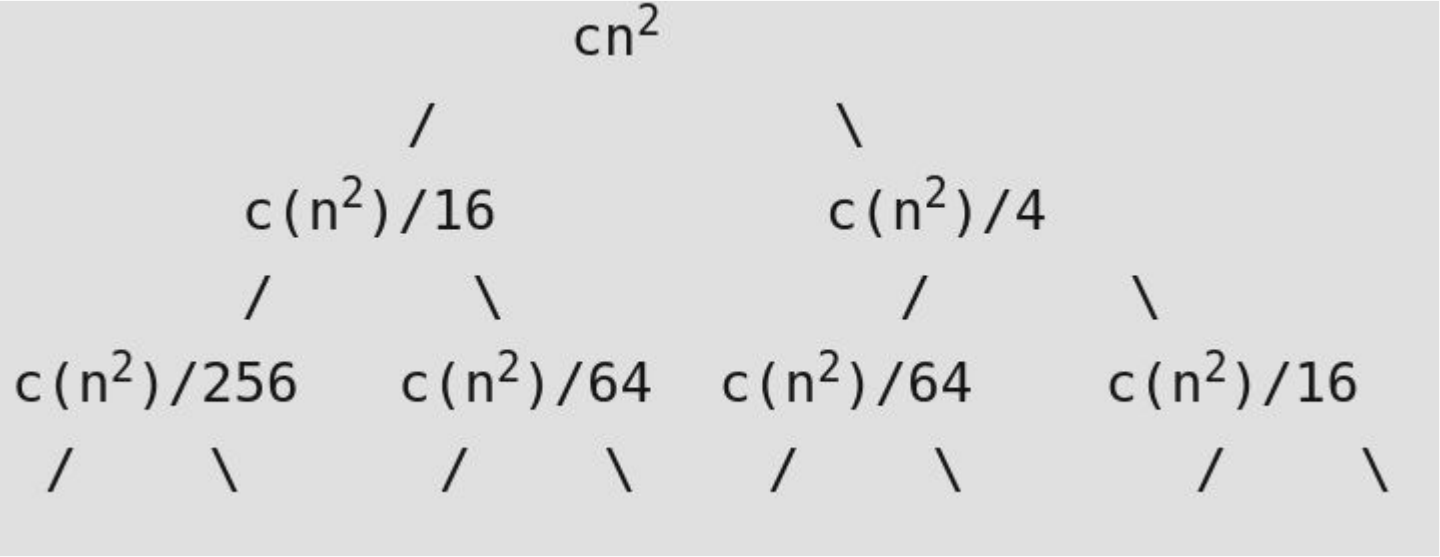
Ta có cây đệ quy:



Recurrence Tree Method



Recurrence Tree Method



4. Asymptotic Notation

Recurrence Tree Method

- Cứ tiếp tục khai triển cây, ta sẽ thu được một cây vô hạn.

- Tổng của tất cả các nút trên cây này chính là tổng của một dãy vô hạn và hội tụ.

- Bằng phép biến đổi toán học, ta có được:

$$T(n) \leq c \times \frac{n^2}{1-\frac{5}{16}} \Rightarrow T(n) = O(n^2)$$

4. Asymptotic Notation

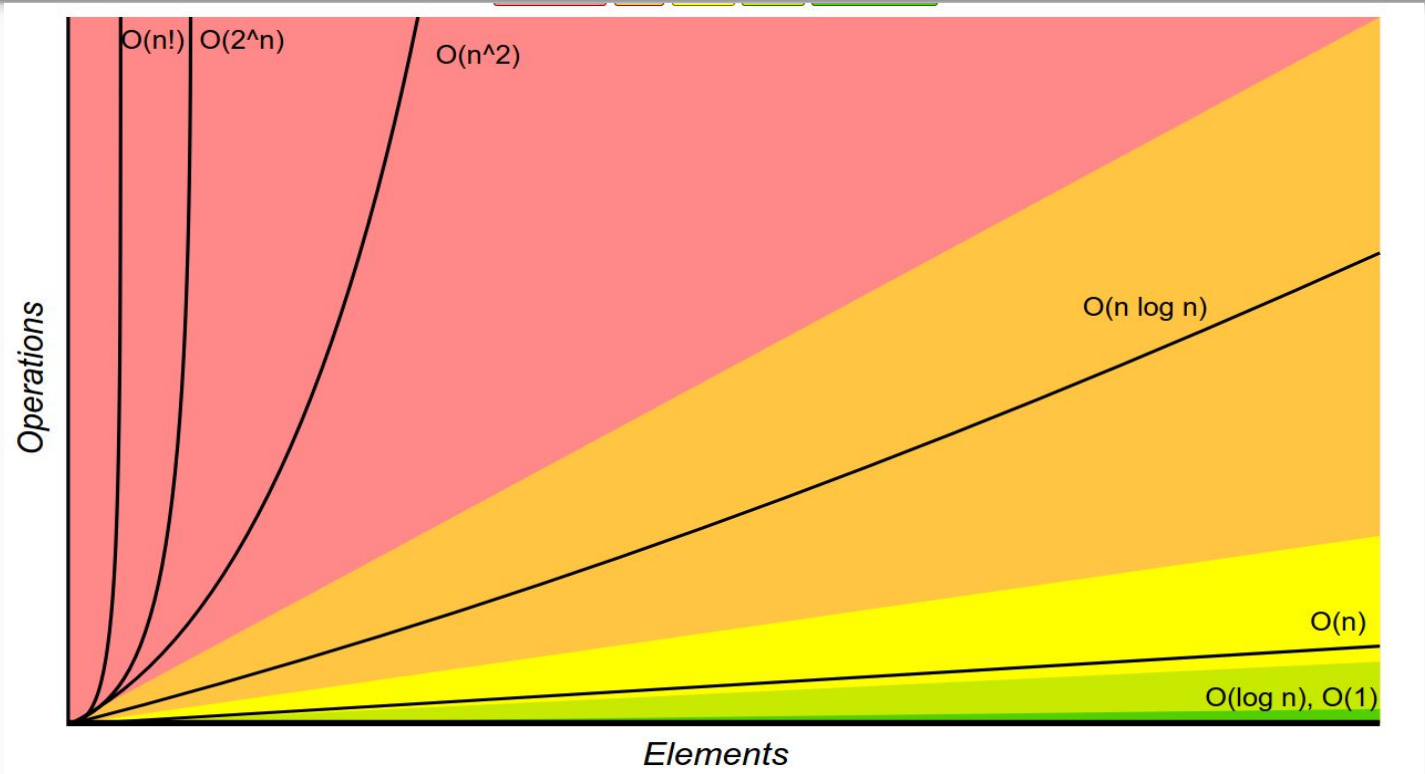
Kiến thức cần nhớ khi tính độ phức tạp thuật toán:

- 4 quy tắc cơ bản để tính độ phức tạp thuật toán (loại bỏ hằng số, lấy max, cộng, nhân).

- Cách tính độ phức tạp vòng lặp.

- Cách tính độ phức tạp thuật toán đệ quy.

4. Asymptotic Notation



Nguồn: https://www.bigocheatsheet.com/

4. Asymptotic Notation

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

Nguồn: https://zaxrosenberg.com/must-know-sorting-algorithms-in-python/

5. Tổng kết

1. Khái niệm và sự cần thiết của phân tích thuật toán
2. Các phương pháp phân tích thuật toán
3. Các kí pháp và 4 quy tắc tính độ phức tạp thuật toán
4. Tính độ phức tạp của thuật toán đệ quy

Tài liệu tham khảo

1. https://en.wikipedia.org/wiki/Analysis_of_algorithms
2. https://www.geeksforgeeks.org/fundamentals-of-algorithms/#AnalysisofAlgorithms
3. Levitin Anany. Introduction to the design analysis of algorithms Pearson, 2019
4. https://book.huihoo.com/data-structures-and-algorithms-with-object-oriented-design-patterns-in-java/html/page62.html
5. Giải thuật & Lập trình - Lê Minh Hoàng.
6. https://www.bigocheatsheet.com/

CÁM ƠN THẦY VÀ CÁC BẠN ĐÃ LẮNG NGHE

6. Câu hỏi mở

Đặt ra câu hỏi

Có thể tự trả lời câu hỏi đó hoặc không

4. Asymptotic Notation

Xét ví dụ thuật toán sắp xếp trộn Merge Sort có mã giả dưới đây:

```
MergeSort(mảng S) {
  if (số phần tử của S <= 1) return S;
  chia đôi S thành hai mảng con S1 và S2 với số phần tử gần bằng nhau;
  MergeSort(S1);
  MergeSort(S2);
  trộn S1 và S2 đã sắp xếp để thu được S mới đã sắp xếp;
  return S mới;
}
```

Ta thấy chỉ cần O(n) hoặc O(1) (tùy theo cách cài đặt) để chia 1 mảng N phần tử thành mảng con. Trộn hai mảng con đã sắp xếp có thể thức thi trong Θ(N).

Như vậy, tổng độ phức tạp để có thể Merge Sort một mảng có n phần tử là Θ (N) cộng với thời gian thực thi 2 lệnh đệ quy.

4. Asymptotic Notation

Gọi f(n) là độ phức tạp của thuật toán Merge Sort ở trên. Theo suy luận trên ta có:

$$f(n) = f([N/2]) + f([n/2]) + p(N)$$

Với p là hàm tuyến tính biểu thị tổng chi phí tính toán dành cho việc chia đôi mảng ban đầu và trộn hai mảng vào kết quả cuối.

Hay ta viết lại công thức truy hồi trên theo cách đơn giản hơn:

$$f(n) = f([N/2]) + f([N/2]) + \Theta(N) \ (*)$$

4. Asymptotic Notation

Subtition Method (Phương pháp thay thế)

Phương pháp này có thể tổng kết bằng một câu: dự đoán cận trên tiệm cận của hàm số f(n) và (cố gắng) chứng minh bằng quy nạp.

Để minh họa ta sẽ chứng minh hàm f(n) của pt (*) là O(NlogN). Từ (*) ta có:

$$\forall\ N,\ f(N) \leq 2f(N/2) + cN$$

với 1 hằng số c nào đó, ta sẽ c/m tồn tại một hằng số d đủ lớn nào đó mà $\forall$ N ta có: $f(N) \leq dN\lg N$. Ta sẽ c/m bằng quy nạp:

4. Asymptotic Notation

Subtition Method (Phương pháp thay thế)

Giả sử $f(N/2) \leq d(N/2)\lg(N/2)$. Ta có:

$$f(N) \leq 2f(N/2) + cN$$

$$\leq 2d(N/2)\lg(N/2) + cN$$

$$= dN(\lg N - \lg 2) + cN$$

$$= dN\lg N - dN + cN$$

Bất đẳng thức đúng khi $d > c$. Và ta luôn có thể chọn d đủ lớn để thỏa mãn điều kiện này

Recurrence Tree Method (Phương pháp cây đệ quy)

Ta có thể biểu diễn các bước thực thi của chương trình đệ quy trên một bộ đầu vào bằng một cây có gốc xác định. Mỗi đỉnh trên cây sẽ tương ứng với một bài toán con mà chương trình đang giải. Xét một đỉnh bất kì trên cây. Nếu giải bài toán thuộc đỉnh đó cần phải gọi đệ quy, đỉnh đó sẽ có các đỉnh con tương ứng với các bài toán nhỏ hơn nữa. Gốc của cây là bộ đầu vào, các lá tương ứng với các bài toán cơ bản có thể giải trực tiếp bằng các thuật toán thông thường (không đệ quy).

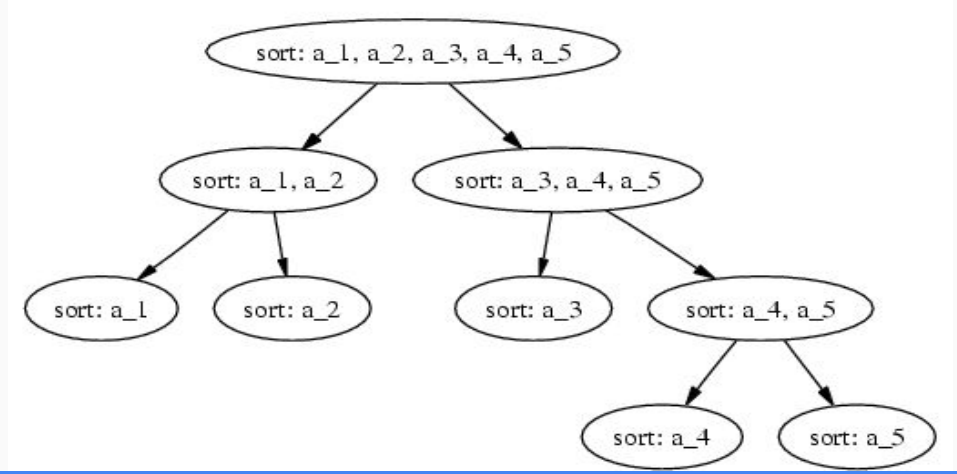
4. Asymptotic Notation

Recurrence Tree Method (Phương pháp cây đệ quy)

Giả sử ta đánh dấu mỗi đỉnh bằng một nhãn biểu thị độ phức tạp nội tại cần có trên đỉnh đó (không tính tới độ phức tạp của lệnh gọi đệ quy). Rõ ràng là độ phức tạp của bài toán gốc bằng tổng tất cả các nhãn của đỉnh

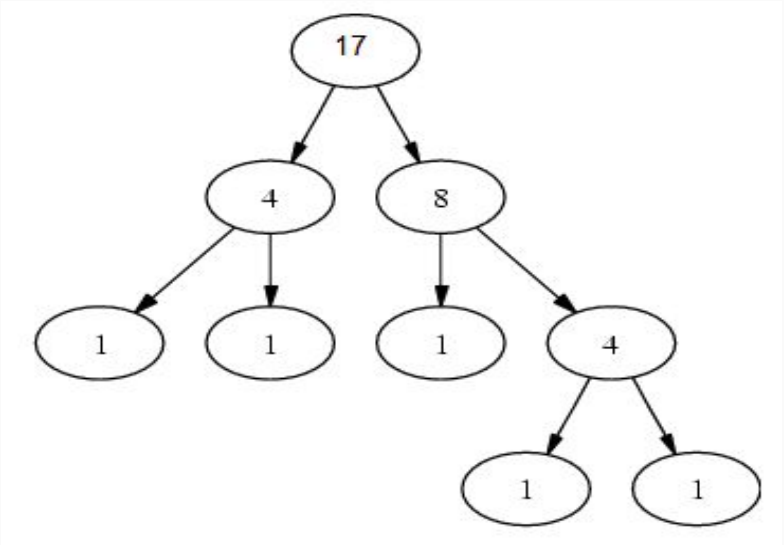
Ví dụ: Cây đệ quy cho thuật toán

Merge Sort với 5 phần tử



Recurrence Tree Method (Phương pháp cây đệ quy)

Cây đệ quy cho công thức truy hồi tương ứng của Merge Sort. Số trong mỗi đỉnh biểu thị số bước mà thuật toán thực thi tại đỉnh đó



ANALYSIS OF ALGORITHMS

CÁM ƠN THẦY VÀ CÁC BẠN ĐÃ LẮNG NGHE
CHÚC THẦY VÀ CÁC BẠN NHIỀU SỨC KHỎE!