Nhóm 5: Phân tích độ phức tạp của thuật toán không đệ qui

1. Khung/Sườn để phân tích hiệu năng (The Analysis Framework)

Trong phần này ta sẽ vẽ ra một cái sườn để phân tích hiệu năng của thuật toán. Như đã biết thì ta có 2 loại hiệu năng: Hiệu năng thời gian (**time efficiency**) và hiệu năng không gian (**space efficiency**). Hiệu năng thời gian hay độ phức tạp thời gian (**time complexity**) sẽ xác định thời gian chạy nhanh hay chậm của thuật toán. Hiệu năng không gian hay độ phức tạp không gian (**space complexity**) đề cập đến lượng đơn vị bộ nhớ mà thuật toán yêu cầu thêm ngoài không gian cần thiết cho input và output của nó.

Ta sẽ tập trung chủ yếu vào độ phức tạp thời gian, những vẫn sẽ xét đến độ phức tạp không gian.

1.1 Đo lường kích thước input (Measuring an Input's size)

Rõ ràng là hầu hết các thuật toán sẽ chạy chậm hơn đối với input lớn hơn. Ví dụ như sẽ mất thời nhiều thời gian hơn để sắp xếp các mảng lớn hơn, nhân các ma trận lớn hơn… Do đó, ta có thể khảo sát thuật toán như một hàm với tham số *n* nào đó (có thể thuật toán sẽ yêu cầu nhiều hơn 1 tham số để biểu thị kích thước của input).

Sự lựa chọn một đơn vị đo kích thước phù hợp có thể bị ảnh hưởng bởi cách hoạt động của thuật toán. Ví dụ, ta có thuật toán kiểm tra lỗi chính tả, nếu thuật toán kiểm tra từng ký tự thì ta nên đo kích thước input dựa trên số lượng ký tự, nếu thuật toán xử lý từng từ thì ta nên đếm số lượng từ.

Một lưu ý đặc biệt, khi đo kích thước input cho các thuật toán để giải các bài toán như kiểm tra *n* có phải là số nguyên tố không thì ở đây input chỉ là 1 con số *n*. Trong trường hợp này, ta nên đo kích thước input bằng số lượng bit *b* khi biểu diễn *n* dưới dạng nhị phân: $b = \lfloor \log_2 n \rfloor + 1$.

1.2 Đơn vị đo thời gian chạy (Units for measuring running time)

Ta không thể dùng đơn vị đo thời gian tiêu chuẩn như second hay millisecond để đo thời gian chạy của thuật toán bởi vì nó còn phụ thuộc vào nhiều yếu tố: tốc độ của máy tính, chất lượng của chương trình triển khai thuật toán, compiler… Do đó, ta cần một đơn vị đo mà không phụ thuộc vào các yếu tố đó.

Một cách là đếm số lần thuật toán thực thi một operation. Nhưng cách này cực kì khó và không cần thiết. Điều cần làm là xác định operation quan trọng nhất của thuật toán, gọi là **basic operation**, là operation đóng góp nhiều nhất vào thời gian chạy của thuật toán, và đếm số lần thực thi của một basic operation.

Basic operation của một thuật toán thường là operation tốn nhiều thời gian nhất và ở vòng lặp trong cùng của thuật toán. Ví dụ, hầu hết các thuật toán sắp xếp hoạt động bằng cách so sánh các phần tử thì basic operation ở đây chính là sự so sánh các phần tử. Một ví dụ khác, trong 4 phép tính cộng, trừ, nhân, chia thì phép chia là operation tốn nhiều thời gian nhất, sau đó là phép nhân và cuối cùng là cộng với trừ.

1.3 Cấp độ tăng (Orders of growth)

Một sự khác nhau về thời gian chạy trên input kích thước nhỏ không thực sự phân biệt được thuật toán hiệu quả và không hiệu quả.

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

1.4 Hiệu năng trường hợp xấu nhất, tốt nhất và trung bình (Worst-Case, Best-Case, and Average-Case Efficiencies)

Như ở trên đã nói, ta đo lường hiệu năng của thuật toán như một hàm có tham số biểu diễn input của thuật toán. Nhưng có nhiều thuật toán mà thời gian chạy của nó không chỉ phụ thuộc vào kích thước input mà còn phụ thuộc vào các chi tiết cụ thể của một input cụ thể.

Ví dụ, ta có thuật toán tìm kiếm tuần tự:

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element in A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Rõ ràng thời gian chạy của thuật toán này có thể khác nhau trên cùng một mảng có n phần tử. Trong trường hợp xấu nhất, phần tử cần tìm có thể nằm ở cuối mảng hoặc không tồn tại, thuật toán sẽ thực hiện số lần so sánh phần tử nhiều nhất có thể trên input có kích thước n: $C_{worst}(n) = n$.

Hiệu năng trong trường hợp xấu nhất (**worst-case efficiency**) của thuật toán được xác định bằng cách khá rõ ràng: phân tích xem kiểu input nào làm cho số lần thực thi basic operation $C(n)$ nhiều nhất trên input kích thước n và tính giá trị $C_{worst}(n)$.

Hiệu năng trong trường hợp tốt nhất (**best-case efficiency**) là trường hợp thuật toán chạy nhanh nhất trên input có kích thước n. Tức là $C(n)$ sẽ nhỏ nhất. Đối với thuật toán tìm kiếm tuần tự ở trên: $C_{best}(n) = 1$ (phần tử cần tìm nằm ngay ở đầu mảng).

Tuy nhiên, phân tích trường hợp xấu nhất và tốt nhất không thể hiện được hành vi của thuật toán trong trường hợp input mang tính "đặc trưng" hay "ngẫu nhiên". Đây chính là thông tin mà hiệu năng trong trường hợp trung bình (**average-case efficiency**) muốn cung cấp. Để phân tích hiệu năng trong trường hợp của thuật toán, ta cần phải đưa ra các giả định về các loại input khả thi có kích thước là n.

Quay lại thuật toán tìm kiếm tuần tự, ta có các giả định sau:

- Xác suất tìm kiếm thành công là p $(0 \leq p \leq 1)$ (phần tử cần tìm có tồn tại).
- Xác suất tìm thấy tại vị trí thứ i là như nhau với mọi i.

Dưới các giả định này, ta có thể tính $C_{avg}(n)$ như sau:

- Trong trường hợp tìm kiếm thành công, xác suất tìm thấy tại vị trí thứ i là p/n với mọi i.
- Trong trường hợp tìm kiếm không thành công, số lần thực hiện so sánh là n với xác suất là $(1 - p)$.

$$C_{avg}(n) = \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}\right] + n \cdot (1 - p)$$

$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p)$$

$$= \frac{p}{n}\frac{n(n+1)}{2} + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p).$$

Nếu tồn tại phần tử cần tìm, tức là $p = 1$, thì $C_{avg}(n) = (n + 1)/2$. Nếu không tồn tại phần tử cần tìm, tức là $p = 0$, thì $C_{avg}(n) = n$.

Ngoài ra còn có hiệu năng khấu hao/trừ dần (**amortized efficiency**). Không chỉ áp dụng cho một lần chạy thuật toán mà cho cả một chuỗi các operation thực hiện trên cùng một cấu trúc dữ liệu. (Xem thêm tài liệu).

1.5 Tóm tắt framework phân tích hiệu năng

- Cả hiệu năng thời gian và không gian đều được đo như các hàm của kích thước input của thuật toán.
- Hiệu năng thời gian được đo bằng cách đếm số lần thuật toán thực thi một basic operation. Hiệu năng không gian được đo bằng số lượng đơn vị bộ nhớ mà thuật toán yêu cầu thêm.
- Hiệu năng của thuật toán có thể thay đổi một cách đáng kể trên các kiểu input khác nhau có cùng kích thước. Đối với những thuật toán như thế, ta cần phải phân biệt hiệu năng của các trường hợp xấu nhất, tốt nhất và trung bình.
- Trọng tâm của framework nằm ở cấp độ tăng (order of growth) của thời gian chạy của thuật toán (các đơn vị bộ nhớ bổ sung đã được tiêu thụ) khi kích thước input tiến đến vô cùng.

2. Các ký hiệu tiệm cận và phân lớp hiệu năng (Asymptotic Notations and Basic Efficiency Classes)

Để so sánh và xếp hạng các cấp độ tăng (orders of growth), người ta dùng 3 ký hiệu tiệm cận: O (big-oh), Ω (big-omega) và Θ (big-theta).

2.1 Ký hiệu tiệm cận O (chặn dưới)

Định nghĩa. Ta nói hàm $t(n)$ thuộc $O\big(g(n)\big)$, ký hiệu $t(n) \in O\big(g(n)\big)$, nếu $t(n)$ bị chặn trên bởi bội số không đổi của $g(n)$ với mọi n lớn. Tức là tồn tại một hằng số c và số nguyên không âm n_0 sao cho:

$$t(n) \leq cg(n) \quad \text{với mọi } n \geq n_0.$$

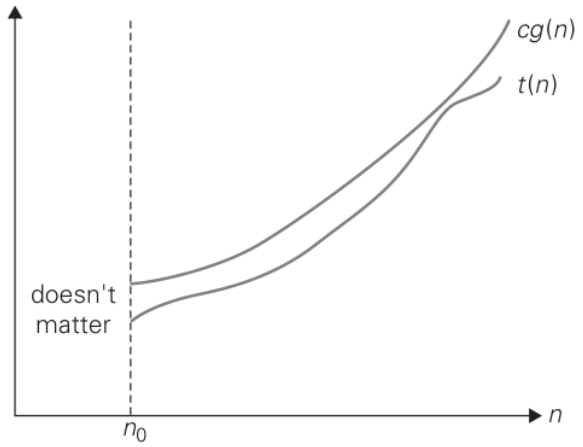

FIGURE 2.1 Big-oh notation: $t(n) \in O(g(n))$.

Ví dụ, chứng minh $100n + 5 \in O(n^2)$.

$$100n + 5 \leq 100n + n = 101n \leq 101n^2 \quad \text{với mọi } n \geq 5.$$

Như vậy, ta có thể chọn $c = 100$ và $n_0 = 5$. Lưu ý rằng, định nghĩa cho ta rất nhiều tự do trong việc lựa chọn các hằng số c và n_0. Ta có thể làm như sau:

$$100n + 5 \leq 100n + 5n = 105n \quad \text{với mọi } n \geq 1$$

và hoàn tất chứng minh với $c = 5$ và $n_0 = 1$.

2.2 Ký hiệu tiệm cận Ω (chặn trên)

Định nghĩa. Ta mói hàm $t(n)$ thuộc $\Omega\big(g(n)\big)$, ký hiệu $t(n) \in \Omega\big(g(n)\big)$, nếu $t(n)$ bị chặn dưới bởi bội số không đổi của $g(n)$ với mọi n lớn. Tức là tồn tại một hằng số c và số nguyên không âm n_0 sao cho:

$$t(n) \geq cg(n) \quad \text{với mọi } n \geq n_0.$$

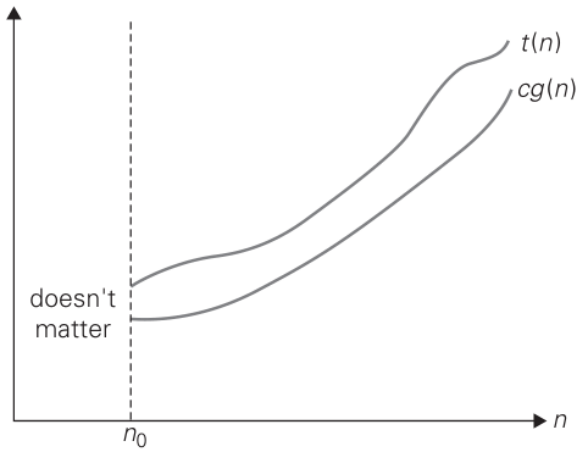

FIGURE 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$.

Ví dụ, chứng minh rằng $n^3 \in \Omega(n^2)$. Ta có thể chọn $c = 1$ và $n_0 = 0$.

$$n^3 \geq n^2 \quad \text{với mọi } n \geq 0.$$

2.3 Ký hiệu tiệm cận Θ (chặn trên và chặn dưới)

Định nghĩa. Ta mói hàm $t(n)$ thuộc $\Theta\big(g(n)\big)$, ký hiệu $t(n) \in \Theta\big(g(n)\big)$, nếu $t(n)$ bị chặn trên và bị chặn dưới bởi bội số không đổi của $g(n)$ với mọi n lớn. Tức là tồn tại các hằng số c_1 và c_1 và số nguyên không âm n_0 sao cho:

$$c_2 g(n) \leq t(n) \leq c_1 g(n) \quad \text{với mọi } n \geq n_0.$$

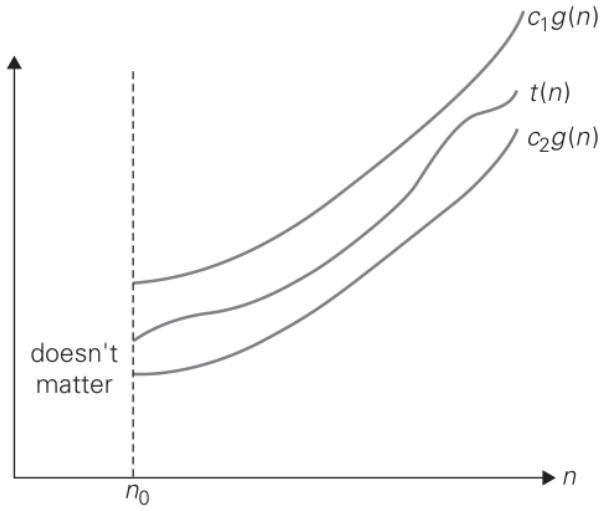

FIGURE 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$.

Ví dụ, ta sẽ chứng minh rằng $\frac{1}{2}n(n-1) \in \Theta(n^2)$.

- Đầu tiên, ta sẽ chứng minh chặn trên:

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \qquad \forall n \geq 0.$$

- Tiếp theo, chứng minh chặn dưới:

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n = \frac{1}{4}n^2 \qquad \forall n \geq 2.$$

Như vậy, ta có thể chọn $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$ và $n_0 = 2$.

2.4 Tính chất liên quan đến các ký hiệu tiệm cận

Định lý. Nếu $t_1(n) \in O\big(g_1(n)\big)$ và $t_2(n) \in O\big(g_2(n)\big)$, thì

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

Điều này cũng đúng với ký hiệu tiệm cận Ω và Θ.

Chứng minh. (Xem thêm tài liệu)

2.5 Dùng giới hạn (limits) để so sánh cấp độ tăng

Ta có thể so sách cấp độ tăng giữa 2 hàm bằng cách tính giới hạn tỷ lệ của chúng.

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\begin{cases}0\\c\\\infty\end{cases}$$

Trong đó,

0 nghĩa là $t(n)$ có cấp độ tăng nhỏ hơn $g(n)$.

Hằng số c nghĩa là $t(n)$ có cùng cấp độ tăng với $g(n)$.

∞ nghĩa là $t(n)$ có cấp độ tăng lớn hơn $g(n)$.

Ta có thể tận dụng quy tắc L'Hôspital:

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\lim_{n\to\infty}\frac{t'(n)}{g'(n)}$$

và công thức/xấp xỉ Stirling (đối với giá trị n lớn):

$$n!\approx\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Ví dụ 1. So sánh cấp độ tăng của $\frac{1}{2}n(n-1)$ và n^2.

$$\lim_{n\to\infty}\frac{\frac{1}{2}n(n-1)}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2}=\frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right)=\frac{1}{2}.$$

Vậy cấp độ tăng của 2 hàm là như nhau, hay có thể viết $\frac{1}{2}n(n-1)\in\Theta(n^2)$.

Ví dụ 2. So sánh cấp độ tăng của $\log_2 n$ và $\sqrt{n}$.

$$\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=\lim_{n\to\infty}\frac{(\log_2 n)'}{(\sqrt{n})'}=\lim_{n\to\infty}\frac{\frac{1}{n\ln 2}}{\frac{1}{2\sqrt{n}}}=\lim_{n\to\infty}\frac{\log_2 e}{n}2\sqrt{n}=2\log_2 e\lim_{n\to\infty}\frac{1}{\sqrt{n}}=0.$$

Vậy $\log_2 n$ có cấp độ tăng nhỏ hơn $\sqrt{n}$.

Ví dụ 3. So sánh cấp độ tăng của $n!$ và 2^n. Sử dụng xấp xỉ Stirling:

$$\lim_{n\to\infty}\frac{n!}{2^n}=\lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}=\lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n=\infty.$$

Vậy, 2^n tăng rất nhanh nhưng $n!$ tăng còn nhanh hơn. Ta có thể viết là $n!\in\Omega(2^n)$, tuy nhiên, Ω vẫn không thể loại trừ khả năng $n!$ và 2^n có cùng cấp độ tăng.

2.6 Phân lớp hiệu năng cơ bản (Basic efficiency classes)

Hiệu năng của một số lượng lớn các thuật toán sẽ rơi vào các phân lớp như trong bảng.

TABLE 2.2 Basic asymptotic efficiency classes

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	*linearithmic*	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

3. Phân tích độ phức tạp của các thuật toán không hồi qui

3.1 Hướng chung để phân tích độ phức tạp thời gian

- Xác định một hay nhiều tham số biểu thị kích thước của input.
- Xác định basic operation của thuật toán. (Về nguyên tắc, nó sẽ nằm ở vòng lặp trong cùng.)
- Kiểm tra xem số lần thuật toán thực thi basic operation có chỉ phụ thuộc vào kích thước input hay không. Nếu nó còn phụ thuộc vào một vài yếu tố khác thì ta cần phải khảo sát riêng từng trường hợp xấu nhất, trung bình và trường hợp tốt nhất nếu cần thiết.
- Thiết lập một tổng thể hiện số lần thuật toán thực thi basic operation.
- Dùng các công thức và quy tắc biến đổi tổng để cho ra một công thức rút gọn thể hiện được số lần thuật toán thực thi basic operation hoặc ít nhất là thiết lập được cấp độ tăng của thuật toán.

Ta có 2 quy tắc thường sử dụng:

$$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i.$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i + \sum_{i=l}^{u} b_i.$$

và 2 công thức tổng:

$$\sum_{i=l}^{u} 1 = u - l + 1.$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).$$

(Xem thêm phần APPENDIX A trong tài liệu)

Ví dụ 1. Tìm giá trị lớn nhất trong mảng có n phần tử.

ALGORITHM *MaxElement*$(A[0..n-1])$
//Determines the value of the largest element in a given array
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in A
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
    if $A[i] > maxval$
        $maxval \leftarrow A[i]$
return $maxval$

Kích thước input là n ,số phần tử trong mảng.

Basic operation là phép so sánh A[i] > maxval, bởi vì nó luôn được thực thi sau mỗi lần lặp còn phép gán maxval ← A[i] thì không.

Rõ ràng số lần thực thi basic operation đối với mọi mảng có kích thước n là như nhau. Do đó, không cần phải phân biệt các trường hợp xấu nhất, tốt nhất và trung bình.

Thuật toán lặp từ chỉ số 1 đến chỉ số cuối của mảng. Do đó:

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

Ví dụ 2. Kiểm tra xem các phần tử trong mảng có đôi một khác nhau hay không.

ALGORITHM *UniqueElements*$(A[0..n-1])$
//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

Kích thước input là n ,số phần tử trong mảng.

Basic operation là phép so sánh A[i]=A[j] ở vòng lặp trong cùng.

Số lần thực thi basic operation không chỉ phụ thuộc vào n mà còn phụ thuộc có hay không các phần tử bằng nhau, nếu có thì vị trí các phần tử đó ở đâu. Ta sẽ chỉ khảo sát trường hợp xấu nhất, $C_{worst}(n)$.

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$