Kiều Công Hậu – 18127259 – 18CLC1

**BÁO CÁO**

Đề tài: 12 thuật toán sắp xếp

1. **Readme**

* Để đơn giản, ta chỉ quan tâm đến việc sắp xếp các số nguyên không âm theo chiều tăng dần.
* Để phần mô tả ví dụ trở nên trực quan hơn thì:
* Các số màu đen là các số chưa được sắp xếp vào đúng vị trí.
* Các số màu xanh là các số đã được sắp xếp vào đúng vị trí.
* Các số màu đỏ là các số quan trọng (thường là min, max, pivot, left, right,…).
* Các thuật toán sắp xếp sắp được giới thiệu sau đây được sắp xếp theo trình tự có chủ đích, cho nên, để việc đọc hiểu dễ dàng hơn thì nên đọc theo đúng thứ tự từ trên xuống dưới.
* Về phần độ phức tạp thời gian (Time complexity) và thời gian chạy (Running time) của 12 thuật toán này đối với các kiểu dữ liệu khác nhau được thể hiện chi tiết trong file excel “Report\_18127259.xlsx”.
* Về phần source code được thể hiện trong thư mục “Source”.

1. **Selection sort:**
2. Mô tả thuật toán:

* Đây là một thuật toán rất đơn giản, nó hoạt động dựa trên nguyên lý liên tục tìm ra phần tử nhỏ nhất của mảng chưa được sắp xếp và đặt lên đầu mảng, sau đó thu nhỏ mảng lại (tức bỏ qua các phần tử nhỏ nhất vừa tìm được và được đặt lên đầu mảng) và cứ tiếp tục như vậy cho đến khi độ dài của mảng con chỉ còn 1.
* Thuật toán này không hiệu quả trên thực tế nên chỉ thường được sử dụng với mục đích giáo dục.

1. Ví dụ:

Cho mảng a có 5 phần tử: 3 5 2 1 9

* #1:

Tìm phần tử nhỏ nhất của mảng 5 phần tử, min = 1.

3 5 2 1 9

Sau đó hoán vị phần tử min này với phần tử đầu mảng là 3.

1 5 2 3 9

Vì 1 đã vào đúng vị trí nên ta thu nhỏ mảng lại, tức bỏ qua số 1, nên ta chỉ cần xét tiếp mảng 4 phần tử còn lại.

1 5 2 3 9

* #2:

Tìm phần tử nhỏ nhất của mảng 4 phần tử còn lại, min = 2.

1 5 2 3 9

Hoán vị phần tử min này với phần tử đầu mảng đang xét là 5.

1 2 5 3 9

Vì 2 đã vào đúng vị trí nên ta thu nhỏ mảng lại, tức bỏ qua số 2, nên ta chỉ cần xét tiếp mảng có 4 phần tử còn lại.

1 2 5 3 9

* #3: *(tương tự các bước trên)*

1 2 5 3 9 (min = 3)

1 2 3 5 9 (hoán vị 3 với 5)

1 2 3 5 9 (thu nhỏ mảng còn 2 phần tử)

* #4: *(tương tự các bước trên)*

1 2 3 5 9

1 2 3 5 9

1 2 3 5 9

* #5: *(tương tự các bước trên)*

1 2 3 5 9

1 2 3 5 9

1 2 3 5 9

Vậy mảng đã được sắp xếp: 1 2 3 5 9

1. **Insertion sort:**
2. Mô tả thuật toán:

* Thuật toán này chia mảng cần sắp xếp thành 2 mảng con, mảng trái và mảng phải, trong đó mảng trái được sắp xếp tăng dần, mảng phải là gồm các phần tử chưa được sắp xếp. Ở lần đầu tiên, mảng trái chỉ có một phần tử là phần tử đầu tiên của mảng, sau đó lần lượt chèn từng phần tử theo thứ tự từ trái sang phải của mảng phải vào mảng trái sao cho mảng trái vẫn giữ đúng tính chất tăng dần. Cứ liên tục như vậy cho đến khi phần tử cuối cùng của mảng phải được chèn vào đúng vị trí của mảng trái.
* Kỹ thuật chèn trong thuật toán này hoạt động như sau: lưu giá trị cần chèn ở đầu mảng phải vô mảng trái vào một biến *temp*, sau đó duyệt mảng trái theo thứ tự từ phải sang trái và so sánh từng phần tử với *temp*, nếu phần tử nào lớn hơn *temp* thì dịch sang phải 1 ô nhằm để tạo khoảng trống để chèn *temp* vào.
* Thuật toán này vô cùng nhanh nếu mảng đã được sắp xếp tăng dần (hoặc gần như đã được sắp xếp) và là tiền đề để phát triển thành các thuật toán tối ưu hơn khác như Shell sort, Flash sort,…

1. Ví dụ:

Cho mảng a có 5 phần tử: 5 3 2 1 4

* #1:

5 3 2 1 4

Chia mảng thành 2 phần, mảng trái được khởi tạo với phần tử đầu tiên là 5, mảng phải chứa 4 phần tử còn lại (mảng trái luôn đảm bảo tính chất tăng dần).

5 3 2 1 4

* #2:

5 3 2 1 4

Xét phần tử đầu tiên của mảng phải là 3 và chèn phần tử này mảng trái sao cho mảng trái luôn tăng dần. Nhưng trước hết hãy lưu 3 vào biến *temp*.

*temp* = 3

So sánh *temp* lần lượt với các phần tử của mảng trái theo thứ tự duyệt mảng từ phải sang trái, phần tử nào lớn hơn thì dịch sang phải 1 ô.

\_ 5 2 1 4 (vì 5 > *temp*)

Đến đây đã duyệt hết các phần tử của mảng trái, vậy ta chèn *temp* = 3 vào ô trống đã chừa ra sau các bước dịch.

3 5 2 1 4

Lúc này, mảng trái tăng lên 2 phần tử là 3 và 5; mảng phải còn 3 phần tử là 2, 1 và 4.

* #3:

3 5 2 1 4

Tiếp tục xét phần tử đầu tiên của mảng phải là 2 và chèn phần tử này vào mảng trái sao cho mảng trái luôn tăng dần. Nhưng trước hết hãy lưu 2 vào biến *temp*.

*temp* = 2

So sáng temp lần lượt với các phần tử của mảng trái theo thứ tự duyệt mảng từ phải sang trái, phần tử nào lớn hơn thì dịch sang phải 1 ô.

3 \_ 5 1 4 (vì 5 > *temp*)

\_ 3 5 1 4 (vì 3 > *temp*)

Đến đây đã duyệt hết các phần tử của mảng trái, vậy ta chèn *temp* = 3 vào ô trống đã chừa ra sau các bước dịch.

2 3 5 1 4

Lúc này, mảng trái tăng lên 3 phần tử là 2, 3 và 5; mảng phải còn 2 phần tử là 1 và 4.

* #4: (tương tự các bước trên)

2 3 5 1 4

temp = 1

2 3 \_ 5 4 (vì 5 > *temp*)

2 \_ 3 5 4 (vì 3 > *temp*)

\_ 2 3 5 4 (vì 2 > *temp*)

1 2 3 5 4 (chèn 1 vào ô trống)

* #5: (tương tự các bước trên)

1 2 3 5 4

temp = 4

1 2 3 \_ 5 (vì 5 > *temp*)

1 2 3 4 5 (chèn 4 vào ô trống)

Vậy là mảng đã được sắp xếp: 1 2 3 4 5

1. **Binary insertion sort:**
2. Mô tả thuật toán:

* Thuật toán này gần như tương tự Insertion sort, điều khác biệt duy nhất là Insertion sort tìm vị trí để chèn phần tử *temp* vào mảng trái bằng cách duyệt tuần tự tuyến tính từng phần tử của mảng trái theo thứ tự từ phải sang trái, còn Binary insertion sort thì sử dụng kỹ thuật tìm kiếm nhị phân để tìm ra vị trí cần chèn *temp* vào mảng trái (vì mảng trái đã được sắp xếp tăng dần nên việc áp dụng kỹ thuật tìm kiếm nhị phân là dễ hiểu).
* Kỹ thuật tìm kiếm nhị phân dùng để tìm vị trí cần chèn *temp* vào mảng trái sao cho mảng trái vẫn tăng dần được áp dụng trong thuật toán này hoạt động như sau: ban đầu, có 1 biến *left* để lưu index đầu tiên của mảng trái, 1 biến *right* để lưu index cuối cùng của mảng trái, 1 biến *mid = left + (right - left) / 2* và 1 biến *key* để lưu giá trị *temp*. Nếu a[*mid*] == *key* thì vị trí cần tìm là *mid*, a[*mid*] > *key* thì cập nhật *right* = *mid* – 1, nếu a[*mid*] < *key* thì cập nhật *left* = *mid* + 1, cứ lặp lại như thế cho tới khi nào a[*mid*] == *key* hoặc *left* == *right*. Nếu cho tới khi left == right mà vẫn không tìm được vị trí thì xét tiếp, nếu a[*left*] > *key* thì vị trí cần tìm *left*, nếu a[*left*] <= *key* thì vị trí cần tìm là *left* + 1.
* Binary insertion sort sẽ nhanh hơn đáng kể so với Insertion sort nếu mảng được sắp xếp không theo thứ tự nhất định và có số lượng phần tử vô cùng lớn.

1. Ví dụ:

Cho mảng a có 5 phần tử: 9 3 2 1 5

* #1:

9 3 2 1 5

Chia mảng thành 2 phần, mảng trái được khởi tạo với phần tử đầu tiên là 9, mảng phải chứa 4 phần tử còn lại (mảng trái luôn đảm bảo tính chất tăng dần).

9 3 2 1 5

* #2:

9 3 2 1 5

Xét phần tử đầu tiên của mảng phải là 3 và chèn phần tử này mảng trái sao cho mảng trái luôn tăng dần. Nhưng trước hết hãy lưu 3 vào biến *temp*.

*temp* = 3

Bây giờ, ta sẽ áp dụng kỹ thuật tìm kiếm nhị phân cho mảng trái để tìm vị trí cần chèn *temp*.

mảng trái: 9 (*left* = 0, *right* = 0, *key* = *temp*)

Vì *left* == *right* nên dừng quét mảng trái, mà a[*left*] > *key* nên vị trí cần chèn

là left, tức tại index 0.

Sau khi tìm được vị trí cần chèn, ta bắt đầu lần lượt dịch các phần tử của mảng trái theo thứ tự từ phải sang trái sang phải 1 ô cho đến khi vị trí cần chèn có 1 ô trống.

\_ 9 2 1 5

Sau khi dịch xong, ta chèn *temp* vào ô trống ấy.

3 9 2 1 5

Lúc này, mảng trái tăng lên 2 phần tử là 3 và 9; mảng phải còn 3 phần tử là 2, 1 và 5.

* #3:

3 9 2 1 5

Xét phần tử đầu tiên của mảng phải là 2 và chèn phần tử này mảng trái sao cho mảng trái luôn tăng dần. Nhưng trước hết hãy lưu 2 vào biến *temp*.

*temp* = 2

Bây giờ, ta sẽ áp dụng kỹ thuật tìm kiếm nhị phân cho mảng trái để tìm vị trí cần chèn *temp*.

mảng trái: 3 9 (*left* = 0, *right* = 1, *key* = *temp*)

Vì *left* < *right* nên *mid = left + (right - left) / 2* = 0, lúc này a[*mid*] là a[0] tức là 3, a[*mid*] > temp nên *right* = *mid* – 1 = 0.

Lúc này, *left* == *right*, mà a[*left*] tức là a[0] lại > *temp* nên vị trí cần chèn là *left*, tức tại index 0.

Sau khi tìm được vị trí cần chèn, ta bắt đầu lần lượt dịch các phần tử của mảng trái theo thứ tự từ phải sang trái sang phải 1 ô cho đến khi vị trí cần chèn có 1 ô trống.

3 \_ 9 1 5

\_ 3 9 1 5

Sau khi dịch xong, ta chèn *temp* vào ô trống ấy.

2 3 9 1 5

* #4: *(tương tự các bước trên)*

2 3 9 1 5

temp = 1

vị trí cần chèn là 0

2 3 \_ 9 5

2 \_ 3 9 5

\_ 2 3 9 5

1 2 3 9 5

* #5: *(tương tự các bước trên)*

1 2 3 9 5

temp = 5

vị trí cần chèn là 3

1 2 3 \_ 9

1 2 3 5 9

Vậy là mảng đã được sắp xếp: 1 2 3 5 9

1. **Shell sort:**
2. Mô tả thuật toán:

* Thuật toán này là một phiên bản cải tiến của Insertion sort và bao gồm 2 giai đoạn:

Giai đoạn 1: Phân hoạch sơ bộ lại mảng cần sắp xếp (mảng mẹ) bằng cách chia mảng mẹ thành nhiều mảng con và thực hiện sắp xếp trên từng mảng con này và ghép chúng lại. Điều đặc biệt ở việc chia mảng này là các phần tử của mỗi mảng con không phải là các phần tử kề nhau trên mảng mẹ mà là các phần tử cách nhau một khoảng *gap* trên mảng mẹ. Chúng ta sẽ tiến hành phân hoạch mảng mẹ với giá trị *gap* khởi đầu là *n / 2* (với *n* là số phần tử của mảng mẹ), các lần phân hoạch sau sẽ giảm giá trị của *gap* xuống một nửa (tức *gap = gap / 2*), liên tục phân hoạch mảng cho tới khi gap = 1 thì dừng lại.

Giai đoạn 2: Sau nhiều lần phân hoạch sơ bộ, mảng mẹ trở thành mảng gần như được sắp xếp hơn rất nhiều lần so với ban đầu. Vì vậy, để nhanh chóng, ta áp dụng thuật toán Insertion sort để sắp xếp lại tổng thể mảng mẹ 1 lần cuối.

* Shell sort tiên tiến hơn Insertion sort ở chỗ là nó có giai đoạn phân hoạch sơ bộ, giai đoạn này sẽ giúp các phần tử ở xa nhau được hoán vị với nhau. Điều này vô cùng có lợi khi các phần tử nhỏ được xếp ở cuối mảng.
* Việc sắp xếp các mảng con trong giai đoạn phân hoạch sơ bộ có thể dùng nhiều thuật toán sắp xếp khác nhau, tuy nhiên, trong bài báo cáo này, ta sẽ sử dụng luôn thuật toán Insertion sort để sắp xếp các mảng con ấy cho đồng bộ với giai đoạn 2.

1. Ví dụ:

Cho mảng a có 8 phần tử: 5 3 7 4 2 1 6 9

* #1: Phân hoạch với *gap = 8 / 2 = 4* (giai đoạn 1)

5 3 7 4 2 1 6 9

* Lúc này, *gap* = 4, tức ta chia mảng mẹ này thành các mảng con sao cho mỗi phần tử của mảng con này cách nhau 4 đơn vị trên mảng mẹ.

5 2 (mảng con 1)

3 1 (mảng con 2)

7 6 (mảng con 3)

4 9 (mảng con 4)

* Sau đó, sử dụng Insertion sort để sắp xếp từng mảng con này.

Mảng con 1: 5 2

5 2

temp = 2

\_ 5

2 5

Vậy mảng mẹ lúc này được cập nhật thành:

2 3 7 4 5 1 6 9

Mảng con 2: 3 1

3 1

temp = 1

\_ 3

1 3

Vậy mảng mẹ lúc này được cập nhật thành:

2 1 7 4 5 3 6 9

Mảng con 3: 7 6

7 6

temp = 6

\_ 7

6 7

Vậy mảng mẹ lúc này được cập nhật thành:

2 1 6 4 5 3 7 9

Mảng con 4: 4 9

4 9

temp = 9

4 9

Vậy mảng mẹ lúc này được cập nhật thành:

2 1 6 4 5 3 7 9

* #2: Phân hoạch với *gap* = 4 / 2 = 2 (giai đoạn 1)

2 1 6 4 5 3 7 9

* Lúc này, *gap* = 2, tức ta chia mảng mẹ này thành các mảng con sao cho mỗi phần tử của mảng con này cách nhau 2 đơn vị trên mảng mẹ.

2 6 5 7 (mảng con 1)

1 4 3 9 (mảng con 2)

* Sau đó, sử dụng Insertion sort để sắp xếp từng mảng con này.

Mảng con 1: 2 6 5 7

2 6 5 7

temp = 6

2 6 5 7

2 6 5 7

temp = 5

2 \_ 6 7

2 5 6 7

2 5 6 7

temp = 7

2 5 6 7

Vậy mảng mẹ lúc này được cập nhật thành:

2 1 5 4 6 3 7 9

Mảng con 2: 1 4 3 9

1 4 3 9

temp = 4

1 4 3 9

1 4 3 9

temp = 3

1 \_ 4 9

1 3 4 9

1 3 4 9

temp = 9

1 3 4 9

Vậy mảng mẹ lúc này được cập nhật thành:

2 1 5 3 6 4 7 9

* #3: Phân hoạch với *gap* = 2 / 2 = 1 (giai đoạn 2)

2 1 5 3 6 4 7 9

*temp* = 1

\_ 2 5 3 6 4 7 9

1 2 5 3 6 4 7 9

1 2 5 3 6 4 7 9

*temp* = 5

1 2 5 3 6 4 7 9

1 2 5 3 6 4 7 9

*temp* = 3

1 2 \_ 5 6 4 7 9

1 2 3 5 6 4 7 9

1 2 3 5 6 4 7 9

*temp* = 6

1 2 3 5 6 4 7 9

1 2 3 5 6 4 7 9

*temp* = 4

1 2 3 5 \_ 6 7 9

1 2 3 \_ 5 6 7 9

1 2 3 4 5 6 7 9

1 2 3 4 5 6 7 9

*temp* = 7

1 2 3 4 5 6 7 9

1 2 3 4 5 6 7 9

*temp* = 9

1 2 3 4 5 6 7 9

Vậy là mảng đã được sắp xếp: 1 2 3 4 5 6 7 9

1. **Bubble sort:**
2. Mô tả thuật toán:

* Thuật toán này tạm dịch là *Sắp xếp nổi bọt*, đúng như cái tên của nó, ý tưởng của thuật toán này là duyệt mảng tuần tự từ cuối mảng lên đầu mảng, phần tử nào nhỏ hơn phần tử liền trước nó thì hoán vị. Ở mỗi lần duyệt, sẽ có 1 phần tử nhỏ nhất trong mảng đang xét được “nổi” lên đầu mảng, sau đó, ta sẽ thu nhỏ mảng lại (tức loại bỏ các phần tử đã nổi lên đúng vị trí của nó) và thực hiện cho tới khi độ dài của mảng đang xét chỉ còn 1 (tức tất cả các phần tử đã nổi lên đúng vị trí của nó).
* Thuật toán này khá chậm nên thường chỉ được dùng với mục đích giáo dục.

1. Ví dụ:

Cho mảng a có 5 phần tử: 5 2 9 3 1

* #1:

5 2 9 3 1

Bắt đầu duyệt từ cuối mảng, 1 < 3 nên hoán vị 1 với 3.

5 2 9 1 3

1 < 9 nên hoán vị 1 với 9.

5 2 1 9 3

1 < 2 nên hoán vị 1 với 2.

5 1 2 9 3

1 < 5 nên hoán vị 1 với 5.

1 5 2 9 3

Lúc này, 1 đã nổi lên đúng vị trí của nó, mảng cần xét thu lại chỉ còn 4 phần tử cuối.

1 5 2 9 3

* #2:

1 5 2 9 3

Duyệt từ cuối mảng, 3 < 9 nên hoán vị 3 với 9.

1 5 2 3 9

3 > 2 nên bỏ qua.

1 5 2 3 9

2 < 5 nên hoán vị 5 với 2.

1 2 5 3 9

Lúc này, 2 đã nổi lên đúng vị trí của nó, mảng cần xét thu lại chỉ còn 3 phần tử cuối.

1 2 5 3 9

* #3:

1 2 5 3 9

Duyệt từ cuối mảng, 9 > 3 nên bỏ qua.

1 2 5 3 9

3 < 5 nên hoán vị 3 với 5.

1 2 3 5 9

Lúc này, 3 đã nổi lên đúng vị trí của nó, mảng cần xét thu lại chỉ còn 2 phần tử cuối.

1 2 3 5 9

* #4:

1 2 3 5 9

Duyệt từ cuối mảng, 5 > 3 nên bỏ qua.

Lúc này, 5 đã nổi lên đúng vị trí của nó, mảng cần xét thu lại chỉ còn 1 phần tử.

1 2 3 5 9

Phần tử còn lại cuối cùng là 9 hiển nhiên đã nằm đúng vị trí của nó.

Vậy là mảng đã được sắp xếp: 1 2 3 5 9

1. **Shaker sort:**
2. Mô tả thuật toán:

* Thuật toán này là một phiên bản cải tiến của Bubble sort, nếu Bubble sort duyệt mảng theo 1 chiều từ phải sang trái và đưa phần tử nhỏ nhất dần dần lên đầu mảng thì Shaker sort duyệt mảng theo 2 chiều, chiều từ trái sang phải đưa phần tử lớn nhất về dần cuối mảng, chiều từ phải sang trái đưa phần tử nhỏ nhất về dần đầu mảng.
* Nếu chỉ cải tiến như vậy thôi thì chả có gì khác biệt, điều đặc biệt ở sự cải tiến này là có thêm 3 biến *left*, *right* và *flag* nữa để làm cờ hiệu, *left* là biến giới hạn trái của mảng đang xét, *right* là biến giới hạn phải của mảng đang xét. Ban đầu, *left = 0* và *right = n – 1* (*n* là số lượng phần tử của mảng). Ở mỗi lần duyệt từ trái sang phải, cờ hiệu *flag* được gán mặc định ở *left,* nếu có sự hoán vị nào xảy ra thì ngay lập tức gán cờ hiệu *flag* vào vị trí xảy ra sự hoán vị ấy, sau khi duyệt tới *right* thì gán *right* = *flag*. Tương tự, ở mỗi lần duyệt từ phải sang trái, flag được gán mặc định ở *right*, nếu có sự hoán vị nào xảy ra thì ngay lập tức gán cờ hiệu *flag* vào ngay vị trí xảy ra hoán vị, sau khi duyệt tới left thì gán *left* = *flag*. Như vậy, nếu trong quá trình duyệt mỗi chiều mà không có sự hoán vị nào thì có nghĩa mảng đang xét đã được sắp xếp rồi, ta không cần phải duyệt thêm chi nữa.
* Nếu *left* == *right* thì mảng đã được sắp xếp.

1. Ví dụ:

Cho mảng a có 7 phần tử: 5 4 6 7 8 2 1

Phần tử được tô vàng là phần tử được gắn *flag*, 2 phần tử màu đỏ là phần tử tại *left* và phần tử tại *right*.

* #1:

Ban đầu, *left* sẽ tại 5, *right* tại 1 và *flag* mặc định tại *left*.

5 4 6 7 8 2 1

Duyệt từ phải sang trái, 5 > 4 nên hoán vị 5 và 4, flag tại 4.

4 5 6 7 8 2 1

5 < 6 nên bỏ qua.

6 < 7 nên bỏ qua.

7 < 8 nên bỏ qua.

8 > 2 nên hoán vị 8 và 2, *flag* tại 2.

4 5 6 7 2 8 1

8 < 1 nên hoán vị 8 và 1, *flag* tại 1.

4 5 6 7 2 1 8

Lúc này, 8 đã vào đúng vị trí nên *right* sẽ chạy tới *flag*.

4 5 6 7 2 1 8

* #2:

Giờ thì duyệt từ phải sang trái, *left* tại 4, *right* tại 1 và *flag* mặc định tại *right*.

1 < 2 nên hoán vị 1 và 2, flag tại 2.

4 5 6 7 1 2 8

1 < 7 nên hoán vị 1 và 7, flag tại 7.

4 5 6 1 7 2 8

1 < 6 nên hoán vị 1 và 6, flag tại 6.

4 5 1 6 7 2 8

1 < 5 nên hoán vị 1 và 5, flag tại 5.

4 1 5 6 7 2 8

1 < 4 nên hoán vị 1 và 4, flag tại 4.

1 4 5 6 7 2 8

Lúc này, 1 đã vào đúng vị trí nên *left* sẽ chạy tới *flag*.

1 4 5 6 7 2 8

* #3:

Duyệt từ trái sang phải, *left* tại 4, *right* tại 2 và *flag* mặc định tại *left*.

4 < 5 nên bỏ qua.

5 < 6 nên bỏ qua.

6 < 7 nên bỏ qua.

7 > 2 nên hoán vị 7 và 2, flag tại 2.

1 4 5 6 2 7 8

Lúc này, 7 đã vào đúng vị trí nên *right* sẽ chạy tới *flag*.

1 4 5 6 2 7 8

* #4:

Duyệt từ phải sang trái, *left* tại 4, *right* tại 2 và *flag* mặc định tại *right*.

2 < 6 nên hoán vị 2 và 6, *flag* tại 6.

1 4 5 2 6 7 8

2 < 5 nên hoán vị 2 và 5, *flag* tại 5.

1 4 2 5 6 7 8

2 < 4 nên hoán vị 2 và 4, *flag* tại 4.

1 2 4 5 6 7 8

Lúc này, 2 đã vào đúng vị trí nên *left* chạy tới *flag*.

1 2 4 5 6 7 8

* #5:

Duyệt từ trái sang phải, *left* tại 4, *right* tại 6, *flag* mặc định tại *left*.

4 < 5 nên bỏ qua.

5 < 6 nên bỏ qua.

Lúc này, 6 đã vào đúng vị trí nên *right* chạy tới *flag*.

1 2 4 5 6 7 8

Vì left trùng right nên dừng ở đây.

Vậy mảng đã được sắp xếp: 1 2 4 5 6 7 8

1. **Counting sort:**
2. Mô tả thuật toán:

* Thuật toán này hoạt động dựa trên nguyên lý đếm tần suất xuất hiện của mỗi phần tử phân biệt trong mảng. Sau đó thực hiện một số tính toán để tìm ra vị trí chính xác của từng phần tử. Chi tiết sẽ được thực hiện ở phần Ví dụ dưới đây.
* Đây là thuật toán không so sánh.
* Hạn chế của thuật toán này là chỉ áp dụng được với số nguyên.

1. Ví dụ:

Cho mảng a có 7 phần tử: 6 0 1 2 5 6 2

Tạo một mảng f có 7 phần tử dùng để đếm tần suất xuất hiện của mỗi phần tử riêng biệt trong mảng a.

Mảng f: index 0 1 2 3 4 5 6

f[index] 1 1 2 0 0 1 2

Sau đó duyệt mảng từ *i* = 1 đến 6, lấy *f[i] = f[i] + f[i – 1]*.

Mảng f: index 0 1 2 3 4 5 6

f[index] 1 2 4 4 4 5 7

Như vậy, mảng f lúc này sẽ lưu giữ vị trí cuối cùng của mỗi loại phần tử phân biệt trong mảng sau khi được sắp xếp.

Tạo một mảng b có cùng kích thức với a, ta sẽ sắp lần lượt từng phần tử của mảng a vào mảng b bằng cách kết hợp với mảng f.

b \_ \_ \_ \_ \_ \_ \_

Duyệt mảng a từ phải sang trái, xét lần lượt từng phần tử một.

a 6 0 1 2 5 6 2

Xét phần tử 2, ta thấy f[2] = 4 nên index của 2 trong mảng b sẽ là 4 – 1 = 3.

index 0 1 2 3 4 5 6

b[index] \_ \_ \_ 2 \_ \_ \_

Cập nhật lại giá trị của f[2] = f[2] – 1.

index 0 1 2 3 4 5 6

f[index] 1 2 3 4 4 5 7

a 6 0 1 2 5 6 2

Xét tiếp tới phần tử 6, ta thấy f[6] = 7 nên index của 6 trong mảng b sẽ là 7 – 1 = 6.

index 0 1 2 3 4 5 6

b[index] \_ \_ \_ 2 \_ \_ 6

Cập nhật lại giá trị của f[6] = f[6] – 1.

index 0 1 2 3 4 5 6

f[index] 1 2 3 4 4 5 6

a 6 0 1 2 5 6 2

Xét tiếp tới phần tử 5, ta thấy f[5] = 5 nên index của 5 trong mảng b sẽ là 5 – 1 = 4.

index 0 1 2 3 4 5 6

b[index] \_ \_ \_ 2 5 \_ 6

Cập nhật lại giá trị của f[5] = f[5] – 1.

index 0 1 2 3 4 5 6

f[index] 1 2 3 4 4 4 6

a 6 0 1 2 5 6 2

Xét tiếp tới phần tử 2, ta thấy f[2] = 3 nên index của 2 trong mảng b sẽ là 3 – 1 = 2.

index 0 1 2 3 4 5 6

b[index] \_ \_ 2 2 5 \_ 6

Cập nhật lại giá trị của f[2] = f[2] – 1.

index 0 1 2 3 4 5 6

f[index] 1 2 2 4 4 4 6

a 6 0 1 2 5 6 2

Xét tiếp tới phần tử 1, ta thấy f[1] = 2 nên index của 1 trong mảng b sẽ là 2 – 1 = 1.

index 0 1 2 3 4 5 6

b[index] \_ 1 2 2 5 \_ 6

Cập nhật lại giá trị của f[1] = f[1] – 1.

index 0 1 2 3 4 5 6

f[index] 1 1 2 4 4 4 6

a 6 0 1 2 5 6 2

Xét tiếp tới phần tử 0, ta thấy f[0] = 1 nên index của 0 trong mảng b sẽ là 1 – 1 = 0.

index 0 1 2 3 4 5 6

b[index] 0 1 2 2 5 \_ 6

Cập nhật lại giá trị của f[0] = f[0] – 1.

index 0 1 2 3 4 5 6

f[index] 0 1 2 4 4 4 6

a 6 0 1 2 5 6 2

Xét tiếp tới phần tử 6, ta thấy f[6] = 6 nên index của 6 trong mảng b sẽ là 6 – 1 = 5.

index 0 1 2 3 4 5 6

b[index] 0 1 2 2 5 6 6

Cập nhật lại giá trị của f[6] = f[6] – 1.

index 0 1 2 3 4 5 6

f[index] 0 1 2 4 4 4 5

Sau đó, gán mảng b qua lại mảng a.

Vậy mảng đã được sắp xếp: 0 1 2 2 5 6 6

1. **Radix sort:**
2. Mô tả thuật toán:

* Thuật toán này là một thuật toán không so sánh. Ý tưởng của nó là dựa vào tính chất “ổn định” của Counting sort, nó sẽ sắp xếp mảng bằng thuật toán Counting sort áp dụng trên từng chữ số theo thứ tự hàng đơn vị, hàng chục, hàng trăm,… Chi tiết sẽ được thể hiện ở phần Ví dụ dưới đây.

1. Ví dụ:

Cho mảng a có 6 phần tử: 123 21 11 9 5 225

Vì số lớn nhất là 225, có 3 chữ số nên ta thực hiện 3 lần Counting sort lần lượt cho chữ số hàng đơn vị, chữ số hàng chục và chữ số hàng trăm.

* #1: Sắp xếp mảng bằng Counting sort dựa trên chữ số hàng đơn vị.

123 021 011 009 005 225

Tạo mảng f có 10 phần tử, đếm tần suất xuất hiện của các chữ số hàng đơn vị của mỗi phần tử.

Mảng f: i 0 1 2 3 4 5 6 7 8 9

f[i] 0 2 0 1 0 2 0 0 0 1

Duyệt mảng từ i = 1 tới 9, cho f[i] = f[i] + f[i – 1]

f[i] 0 2 2 3 3 5 5 5 5 6

Theo Counting sort, mảng a sẽ được sắp xếp như sau:

\_ \_ \_ \_ 225 \_

\_ \_ \_ 005 225 \_

\_ \_ \_ 005 225 009

\_ 011 \_ 005 225 009

021 011 \_ 005 225 009

021 011 123 005 225 009

* #2: Sắp xếp mảng bằng Counting sort dựa trên chữ số hàng chục.

021 011 123 005 225 009

Tạo mảng f có 10 phần tử, đếm tần suất xuất hiện của các chữ số hàng đơn vị của mỗi phần tử.

Mảng f: i 0 1 2 3 4 5 6 7 8 9

f[i] 2 1 3 0 0 0 0 0 0 0

Duyệt mảng từ i = 1 tới 9, cho f[i] = f[i] + f[i – 1]

f[i] 2 3 6 6 6 6 6 6 6 6

Theo Counting sort, mảng a sẽ được sắp xếp như sau:

\_ 009 \_ \_ \_ \_

\_ 009 \_ \_ \_ 225

005 009 \_ \_ \_ 225

005 009 \_ \_ 123 225

005 009 011 \_ 123 225

005 009 011 021 123 225

* #3: Sắp xếp mảng bằng Counting sort dựa trên chữ số hàng trăm.

005 009 011 021 123 225

Tạo mảng f có 10 phần tử, đếm tần suất xuất hiện của các chữ số hàng đơn vị của mỗi phần tử.

Mảng f: i 0 1 2 3 4 5 6 7 8 9

f[i] 4 1 1 0 0 0 0 0 0 0

Duyệt mảng từ i = 1 tới 9, cho f[i] = f[i] + f[i – 1]

f[i] 4 5 6 0 0 0 0 0 0 0

Theo Counting sort, mảng a sẽ được sắp xếp như sau:

\_ \_ \_ \_ \_ 225

\_ \_ \_ \_ 123 225

\_ \_ \_ 021 123 225

\_ \_ 011 021 123 225

\_ 009 011 021 123 225

005 009 011 021 123 225

Vậy mảng đã được sắp xếp: 5 9 11 21 123 225

1. **Flash sort:**
2. Mô tả thuật toán:
3. Ví dụ:
4. Quick sort
5. Heap sort
6. Merge sort