# Flash sort

Flash Sort là một thuật toán sắp xếp dựa trên sự phân bố của dữ liệu, cụ thể là phân bố đều, được Karl-Dietrich Neubert phát minh năm 1998. Nếu chúng ta không xác định được phân bố của một tập dữ liệu, khi dữ liệu có khả năng phân bố không đều, thì độ phức tạp có thể trở thành O(n2).

## Ý tưởng

Thuật toán Flash Sort gồm ba công việc: phân lớp, hoán vị và sắp xếp cục bộ. Phân lớp giúp xác định kích thước của từng lớp phần tử. Hoán vị giúp hoán vị các phần tử giữa các lớp này. Sắp xếp cục bộ để sắp xếp các phần tử trong cùng một lớp. Ý tưởng chung của Flash Sort là chia để trị, tương tự như Merge Sort hay Quick Sort. Thay vì chia thành hai mảng con đến khi nào không chia được nữa thì Flash Sort chia thành m phân lớp.

### Phân lớp

Giả sử các phần tử trong danh sách là **a[i]** và được phân bố đều rải rác. Cần tính vị trí của phần tử **a[i]** trong một lớp dữ liệu. Việc tính toán này có thể tính dựa trên giá trị của phần tử đó mà không cần thực hiện các phép so sánh.

Đầu tiên, ta cần đi tìm hệ số **m**, là số lượng các lớp phần tử. Ta có công thức sau, với **n** là số lượng phần tử:

Ảnh có chứa văn bản

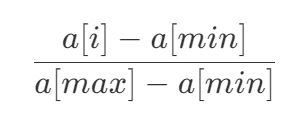
Mô tả được tạo tự động

Tiếp theo, ta cần đi tìm giá trị lớn nhất (**a[max]**) và nhỏ nhất (**a[min]**) của danh sách các phần tử. Công việc này cần phải quét hết tất cả các phần tử nên độ phức tạp là O(n).

Ảnh có chứa văn bản, cửa shoji

Mô tả được tạo tự động

Sau đó, ta lấy đoạn màu xanh dương trên hình trừ cho đoạn màu đỏ để ra đoạn màu xanh lá. Nói cách khác, đoạn màu xanh lá có biểu thức:



Cuối cùng, ta nhân biểu thức này cho **(m – 1)** với **m** là số lượng các phân lớp, lý do trừ 1 vì các phân lớp bắt đầu tại 0.

Ta suy ra được biểu thức tìm vị trí **k**:

Ảnh có chứa bàn

Mô tả được tạo tự động

Kết quả sẽ có giá trị từ 0 đến **m**. Nếu muốn index bắt đầu là 1 thì ta chỉ cần cộng thêm 1 vào biểu thức trên. Trung bình sẽ có **n/m** phần tử ở mỗi lớp.

Khi có biểu thức trên, ta cần xét hết tất cả các phần tử và đếm xem mỗi lớp có bao nhiêu phần tử. Rồi chúng ta tính chỉ số index cuối cùng của mỗi lớp, việc này sẽ phục vụ cho công việc hoán vị tiếp theo. Chỉ số này sẽ được lưu trong mảng gồm **m** phần tử, là mảng **L** vốn trước đó lưu số phần tử của mỗi phân lớp. Mảng **L** này sẽ lưu giá trị cuối cùng của các phân lớp.

### Hoán vị

Ý tưởng của công việc hoán vị là di chuyển một phần tử đang ở vị trí không đúng lớp đến nơi mà nó thuộc về.

Vị trí có thể di chuyển của một lớp phụ thuộc vào mảng **L** ở trên, với **L[0]** là vị trí có thể di chuyển đến cho phần tử **a[j]** (Ta dùng index **j** trong phần hoán vị này) bất kỳ của lớp 0. Tương tự **L[1]** là lớp 1 và **L[k]** là của lớp **k**. Sau khi di chuyển đến, ta giảm giá trị **L[k]** xuống một giá trị để chuyển sang vị trí khác.

Bất cứ khi nào di chuyển một phần tử **a[j]** đến lớp **m**, ta sẽ hoán vị một phần tử của lớp **m** đó cho phần tử **a[j]** cần di chuyển. Ví dụ dưới đây di chuyển **a[j]** đến lớp 2, ta đẩy **a[k]** sang vị trí trước đó của **a[j]**. Sau đó, giá trị của **L[2]** trừ đi một để chuyển đến vị trí dời chỗ tiếp theo. Việc tính toán xem phần tử **a[j]** thuộc lớp nào cũng phụ thuộc vào biểu thức tính k ở trên.

Tiếp tục xét xem **a[k]** thuộc lớp nào và đổi chỗ với phần tử thuộc lớp đó.

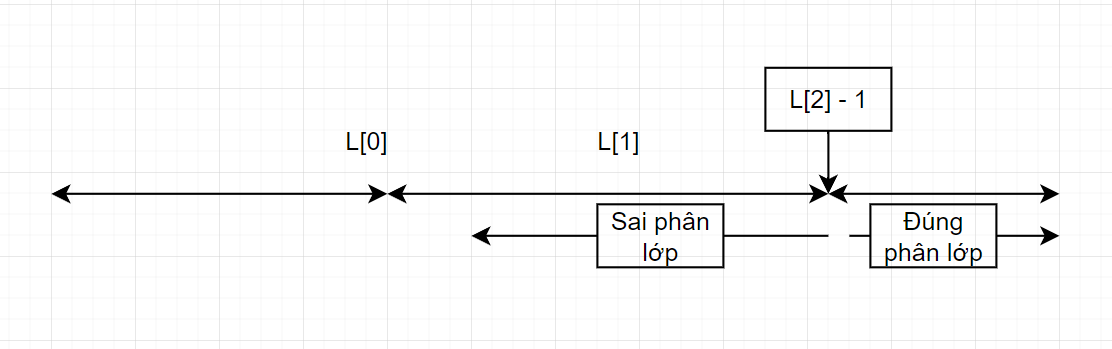
Ảnh có chứa cửa shoji, trong nhà, bẩn

Mô tả được tạo tự động

**Các lưu ý để cài đặt thuật toán:**

Khoảng vị trí của một phân lớp **k** là từ **L[k]** trở đi. Mỗi lần thêm một phần tử vào lớp **k** bất kỳ thì giá trị của **L[k]** giảm đi một. Khi lớp đó đã chứa đầy các phần tử thì giá trị của **L[k]** = **L[k – 1] + 1** (tức là vị trí sau vị trí cuối cùng của phân lớp trước).

Từ đó, ta suy ra được rằng với vị trí **j** bất kỳ lớn hơn **L[k] – 1** (**L[k]** trở đi) thì phần tử đó đã thuộc phân lớp của nó, do vậy ta bỏ qua và xét vị trí **j** tiếp theo. Ngược lại, phần tử có vị trí **j** bé hơn hoặc bằng **L[k] – 1** thì nó vẫn chưa nằm đúng phân lớp, ta cần tìm phân lớp của **a[j]** và chuyển nó đến đó.

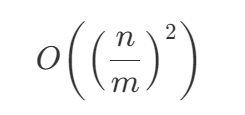


Khi đã tìm được vị trí **j** để bắt đầu một chu trình hoán vị đưa các phần tử về đúng phân lớp, ta sẽ lặp đến khi tìm được một vị trí mà ở đó **j** không thỏa vị trí mà phân lớp của nó chỉ định. Tức là **j** khác **L[k]**. Trong vòng lặp này ta sẽ thực hiện đổi chỗ và tăng biến đếm số lần hoán vị lên.

Khi số lần hoán vị là **n** thì ta dừng quá trình hoán vị và sang sắp xếp cục bộ. Đặc biệt, khi chỉ còn một phần tử chưa được hoán vị thì ta biết rằng nó đã nằm đúng vị trí, nên điều kiện dừng của bước hoán vị sẽ là **n – 1**.

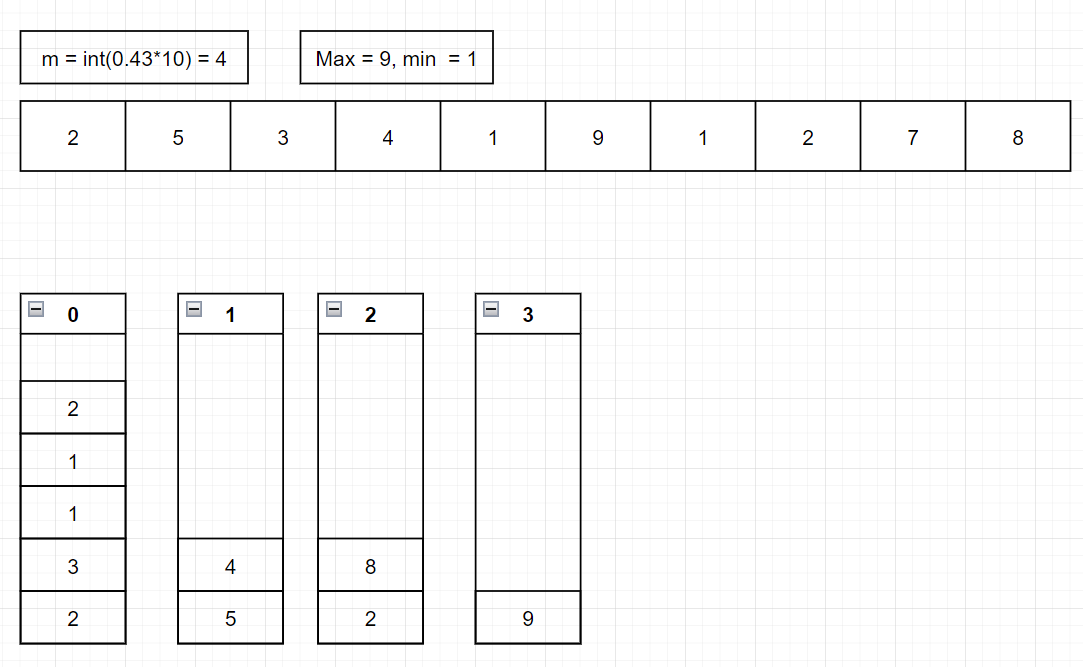
### Sắp xếp cục bộ

Khi đã hoán vị các phần tử về đúng phân lớp của nó, ta sẽ tiến hành sắp xếp cục bộ trong từng phân lớp. Ở đây chúng ta sử dụng sắp xếp chèn (Insertion Sort), và độ phức tạp thuật toán của mỗi phân lớp là:



**Ví dụ:**

Cho mảng dưới đây, các số đã được phân vào 4 lớp phù hợp.



Tiến hành hoán vị, ta thấy 9 thuộc lớp 3, ta đổi chỗ với 8. Sau đó 8 thuộc lớp 2, đổi chỗ với 7.

Ảnh có chứa bàn

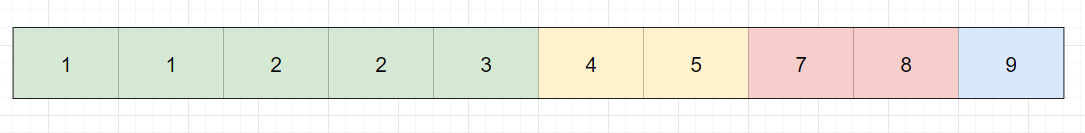
Mô tả được tạo tự động

Cứ như vậy đến khi hoán vị tất cả các phần tử về đúng lớp.

Ảnh có chứa bàn

Mô tả được tạo tự động

Cuối cùng ta tiến hành Insertion Sort cho từng lớp để được kết quả.



## Tối ưu hóa thời gian chạy thuật toán

Bằng cách điều chỉnh số phân lớp, chúng ta có thể điều chỉnh thời gian chạy tổng thể của một thuật toán. Nếu chúng ta giảm số phân lớp, thời gian cho quá trình phân lớp sẽ giảm đi. Đổi lại, thời gian dành cho quá trình sắp xếp cục bộ lại tăng lên (do mỗi phân lớp có thêm nhiều phần tử).

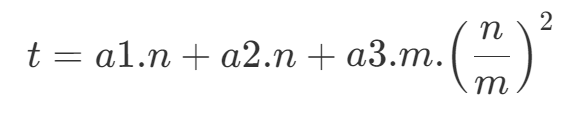
Sau đây là cách mà tác giả của thuật toán tiến hành tìm ra hằng số C để chia số phân lớp nhằm tối ưu thuật toán.

Tác giả phân tích rằng các bước trong thuật toán sẽ đóng góp vào tổng thời gian chạy thuật toán như sau:

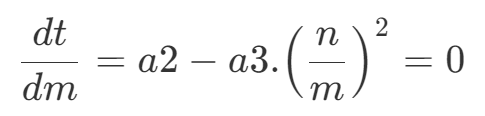
Ảnh có chứa văn bản

Mô tả được tạo tự động

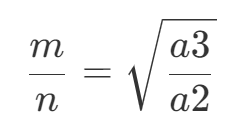
Tổng thời gian chạy thuật toán là:



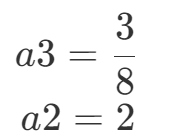
Lấy đạo hàm để tìm cực tiểu:



Giải tìm m, ta được nghiệm là:



Theo một số tính toán của các nhà nghiên cứu thì ước tính được:



Nên ta suy ra hằng số C là 0.43 và công thức tính số phân lớp sẽ là:

Ảnh có chứa văn bản

Mô tả được tạo tự động

Tuy nhiên con số này chỉ là lý thuyết, trên thực tế thì có thể sai khác tùy vào loại máy.

## Độ phức tạp thuật toán

Như đã phân tích sơ lược ở phần ý tưởng, độ phức tạp thuật toán của các bước là:

Ảnh có chứa văn bản

Mô tả được tạo tự động

Nếu số lượng phân lớp là tuyến tính theo O(n), mỗi phân lớp sẽ có kích thước hằng số gần giống nhau, thì việc sắp xếp một phân lớp như vậy với thuật toán có độ phức tạp là O(n2) như Insertion Sort sẽ có độ phức tạp là O(12) = O(1).

Như vậy, độ phức tạp thời gian của thuật toán là O(n).

Độ phức tạp thời gian sẽ tăng lên nếu có ít không gian bộ nhớ được sử dụng và sẽ giảm đi nếu có nhiều không gian bộ nhớ được sử dụng. Nói cách khác, độ phức tạp thời gian và không gian là tỉ lệ nghịch. Mặc dù vậy, nếu dữ liệu đầu vào của thuật toán là một danh sách đã sắp xếp, thì thời gian thực hiện thuật toán cũng không giảm.

Với m = 0.1n, Flash Sort luôn nhanh hơn Heap Sort và nhanh hơn Quick Sort nếu n > 80. Hơn thế nữa, với n = 10000, Flash Sort sẽ nhanh hơn Quick Sort xấp xỉ hai lần. Mặt khác, với m = 0.2n, Flash Sort nhanh hơn khi m = 0.1n xấp xỉ 15%. Với m = 0.05n, nó chậm hơn 30% nhưng khi n > 200 thì nó vẫn nhanh hơn Quick Sort.

Số phân lớp m = O(n) chính là độ phức tạp không gian dùng trong thuật toán và độ phức tạp không gian này có thể sai khác phụ thuộc nhiều vào hằng số C được tính ở trên.

Tổng kết lại:

|  |  |  |
| --- | --- | --- |
| Best case | Average case | Worst case |
| O(n) | O(n) | O(n2) |

**Điểm mạnh**:

1. Có thể dùng để sắp xếp các số không phải số nguyên (non – integer). Để sắp xếp dữ liệu dạng string thì cần chuyển sang hệ cơ số 256.
2. Thời gian chạy thuật toán rất nhanh khi chọn m thích hợp.

**Điểm yếu**:

1. Nếu dữ liệu không phân bố ngẫu nhiên đều khắp danh sách và phân bố tập trung thành một cụm (skewed), thì số phân lớp sẽ suy biến về O(1). Dẫn đến độ phức tạp là O(n2).
2. Không phải là thuật toán stable.
3. Khó cài đặt code.

# Heap Sort

Heap Sort là một thuật toán dựa trên các phép so sánh. Có thể xem nó như một phiên bản cải tiến của Selection Sort. Heap Sort được phát minh bởi J. W. J. Williams năm 1964, cũng là thời điểm ra đời của Heap.

## Ý tưởng

Thuật toán Heap Sort được chia làm hai phần là xây dựng Max Heap và sắp xếp.

### Xây dựng Max Heap

Một số khái niệm:

**Heap** là cây nhị phân hoàn chỉnh.

**Cây nhị phân hoàn chỉnh** là cấu trúc cây nhị phân hoàn hảo nhưng trừ mức cuối, ở mức cuối các node lá dồn hết qua trái càng xa càng tốt.



Hai phần tử con của một phần tử **a[i]** bất kỳ sẽ nằm ở vị trí **2i + 1** và **2i + 2** (nếu phần tử đầu là 0) hoặc **2i** và **2i + 1** (nếu phần tử đầu là 1). Các phần tử này gọi là các **phần tử liên đới**. Các số đánh trong hình trên chính là vị trí index của các phần tử khi thể hiện dưới dạng mảng. Nếu duyệt cây theo các mức (Level Order Traversal), ta sẽ thu được một danh sách tăng dần các vị trí index từ 1 đến n (hoặc từ 0 đến n – 1).

**Vun đống** (Heapify) là một thao tác kiểm tra xem node đang xét có giá trị lớn hơn giá trị của node con nó hay chưa. Chúng ta sẽ hoán vị một phần tử với node con của nó nếu giá trị của node đó bé hơn node con. Và nếu cả hai node con đều lớn hơn, ta sẽ hoán vị với node con lớn nhất.

**Max Heap** là một heap sao cho các node đều lớn hơn các node con của nó. Để xây dựng Max Heap, ta sẽ vun đống từ giữa mảng. Ta không cần phải đi xây dựng một cấu trúc cây nhị phân trừu tượng mà sẽ thao tác trực tiếp trên mảng dữ liệu đầu vào.

Ví dụ minh họa

Ví dụ cho một mảng như sau:

Ảnh có chứa văn bản

Mô tả được tạo tự động

Cây nhị phân xây dựng từ mảng này là:

Ảnh có chứa bầu trời

Mô tả được tạo tự động

Để tìm vị trí của phần tử giữa mảng, ta lấy phần nguyên của (**n – 1)/2**, với mảng ở trên ta có (7 – 1)/2 = 3.

Vậy ta sẽ xét vun đống ở phần tử **a[3]**, có giá trị là 2.

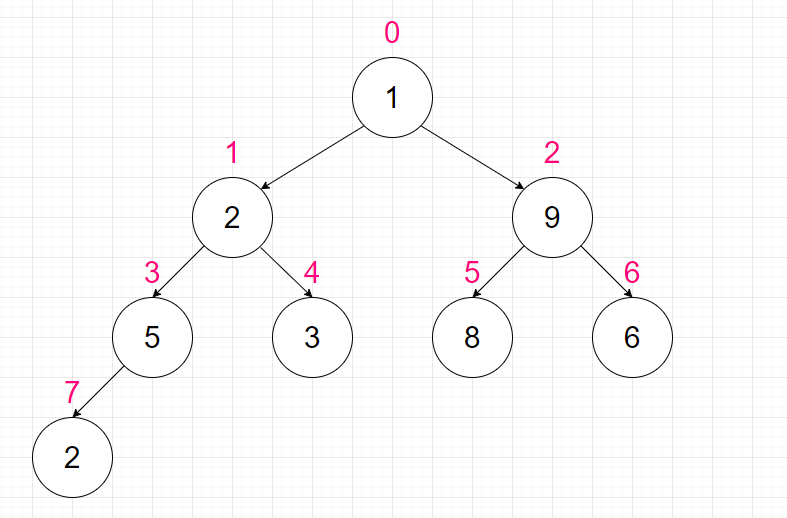
Ta hoán vị **a[3]** = 2 với **a[7]** = 5.

Các phần tử bị hoán vị phải được vun đống ở vị trí mà nó hoán vị đến. Việc này để đảm bảo các node luôn có giá trị lớn hơn node con của nó sau khi có sự thay đổi gây ra bởi công việc hoán vị. Cụ thể, với ví dụ trên ta sẽ xét vun đống tại **a[7]**, nhưng không có gì xảy ra vì nốt này không có node con.

Ảnh có chứa bầu trời, văn phòng phẩm, phong bì

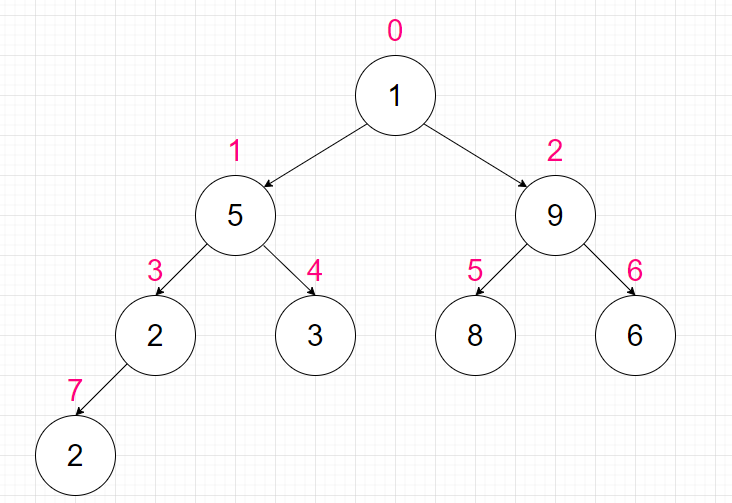
Mô tả được tạo tự động

Sau khi đã xét **a[3]** thì ta tiếp tục xét đến **a[2]**, **a[1]** và **a[0]** (tiến dần về đầu mảng). Khi xét tại **a[2]** = 6, nhận thấy có **a[6]** = 9 lớn hơn, nên ta sẽ hoán vị hai node này. Kết quả là:



Tương tự cũng phải xét vun đống tại **a[6]**, nhưng vẫn không có gì xảy ra vì tại vị trí này không có node con.

Xét **a[1]** = 2 và hoán vị với **a[3]** = 5. Rồi xét vun đống tại **a[3]** nhưng không xảy ra hoán vị vì **a[3]** = **a[7]** = 2.



Cuối cùng ta xét **a[0]** = 1, hoán vị với **a[2]** = 9 rồi hoán vị tiếp tục với **a[5]** = 8. Ta thu được một Max Heap hoàn chỉnh:

Ảnh có chứa bầu trời, văn phòng phẩm, phong bì

Mô tả được tạo tự động

Nhận thấy tại mỗi node, giá trị của nó luôn lớn hơn giá trị của các node con.

Tổng quát hóa cho quá trình xây dựng Max Heap:

* Xét vun đống Max Heap tại vị trí giữa mảng. Lý do chọn vị trí này vì các vị trí sau sẽ không tồn tại bất kỳ node con nào.
* Khi giá trị của node con lớn hơn node đang xét, hoán vị chúng với nhau. Nếu hai node con đều lớn hơn, chọn node con lớn nhất.
* Khi xảy ra hoán vị, cần xét vun đống tại vị trí được hoán vị tới của node đang xét. Ví dụ hoán vị giá trị của node tại vị trí **i** với vị trí **j**, thì ta cần phải xét vun đống tại vị trí **j**.
* Lặp lại quá trình xét cho đến phần tử đầu mảng.

### Sắp xếp

Sau khi xây dựng được Max Heap, ta sẽ lặp lại quá trình sắp xếp n lần như sau:

* Hoán vị phần tử đầu với phần tử cuối mảng, sau đó loại bỏ phần tử cuối mảng ra khỏi phạm vi vun đống. Vị trí cuối mảng xem như đã được sắp xếp.
* Tiến hành vun đống cho vị trí đầu tiên trong mảng, do các vị trí còn lại đều đã được vun đống.
* Lặp lại quá trình này khi chỉ còn một phần tử trong mảng hay nói cách khác là tất cả các phần tử đều đã chuyển đến cuối mảng và được sắp xếp.

## Độ phức tạp thuật toán

Phần vun đống, mỗi khi muốn xét một phần tử nào đó xem có thỏa mãn tính chất của Max Heap hay chưa, ta có thể phải đào sâu xuống các node con bên dưới để kiểm tra và vun đống nếu cần thiết. Mà chiều cao của cây nhị phân hoàn chỉnh với n node là O(log2n). Trường hợp xấu nhất khi phải đào đến các lá thì độ phức tạp của phần này O(log2n).

Trong phần xây dựng Max Heap, ta chỉ xét từ giữa mảng, tức là chỉ duyệt qua n/2 phần tử. Mỗi phần tử đều phải vun đống và có độ phức tạp thuật toán là O(log2n) như đã tìm ra ở trên. Vì vậy tổng độ phức tạp của việc xây dựng Max Heap là:

Ảnh có chứa văn bản

Mô tả được tạo tự động

Đối với phần sắp xếp, cần phải sử dụng một vòng lặp duyệt qua n phần tử trong mảng để có thể chuyển nó về cuối và vun đống ngay sau đó. Mỗi lần như vậy cần tốn chi phí O(log2n) để vun đống. Từ đó suy ra phần này có độ phức tạp là O(n.log2n).

Độ phức tạp từng phần:

1. Heapify: O(log2n).
2. Xây dựng Max Heap: O(n.log2n)
3. Sắp xếp kèm heapify: O(n.log2n)

Độ phức tạp không gian:O(n).

Tổng kết lại:

|  |  |  |
| --- | --- | --- |
| Best case | Average case | Worst case |
| O(n.log2n) | O(n.log2n) | O(n.log2n) |

**Điểm mạnh:**

1. Một thuật toán Inplace không dùng thêm bộ nhớ phụ.
2. Có độ phức tạp ổn định, sẽ tốt hơn Worst Case của Quick Sort.
3. Nếu dữ liệu tổ chức trên Heap, đây là một thuật toán sử dụng hiệu quả.
4. Có thể sử dụng để giải quyết vấn đề top k phần tử như Selection Sort.

**Điểm yếu:**

1. Là một thuật toán Non – Stable, thứ tự của các phần tử cùng lớp giá trị bị xáo trộn.
2. Chậm hơn rất nhiều so với Quick Sort hay Merge Sort khi dùng trong thực tế (hoặc khi dữ liệu quá lớn).
3. Khó cài đặt code.

## Biến thể

### Thuật toán Floyd

Thuật toán này dùng để xây dựng một Max Heap nhưng lại chỉ có độ phức tạp là O(n) thay vì thông thường là O(n.log2n).

**Ý tưởng:**

1. Duyệt Heap từ cuối mảng lên đầu mảng.
2. Khi nào gặp một phần tử không thỏa mãn tính chất Max Heap (hoặc Min Heap) thì sẽ hoán đổi giá trị với các node con của nó.
3. Tiếp tục xét cho các node con vừa mới bị hoán đổi. Ta gọi bước hai và ba là sàng lọc (percolate)

### Bottom – up

Đây là một phiên bản khác của Heap Sort với phần vun đống (heapify) được tinh chỉnh để giảm thiểu số lần so sánh.

**Ý tưởng:**

So sánh hai node con của node đang xét vun đống với nhau và lưu lại node con có giá trị lớn hơn, tạm gọi là node A. Tiếp tục đào xuống để so sánh hai node con của node A và tìm ra node con lớn hơn.

Cứ như thế cho đến khi đào đến node lá nào đó. Nếu node lá đó có anh em, thì cũng phải so sánh để tìm ra node con lớn nhất, tạm gọi là B. Khi đã có node B này, ta xét từ node gốc (node đang xét vun đống) trở xuống node lá và tìm node đầu tiên lớn hơn node gốc.

Sao chép node gốc đến node cuối cùng lớn hơn node gốc. Sau đó di chuyển từng phần tử lên một vị trí về phía node gốc.

# Merge Sort

Là một thuật toán dựa trên sự so sánh và phương pháp chia để trị do John Von Neumann phát minh ra 1945.

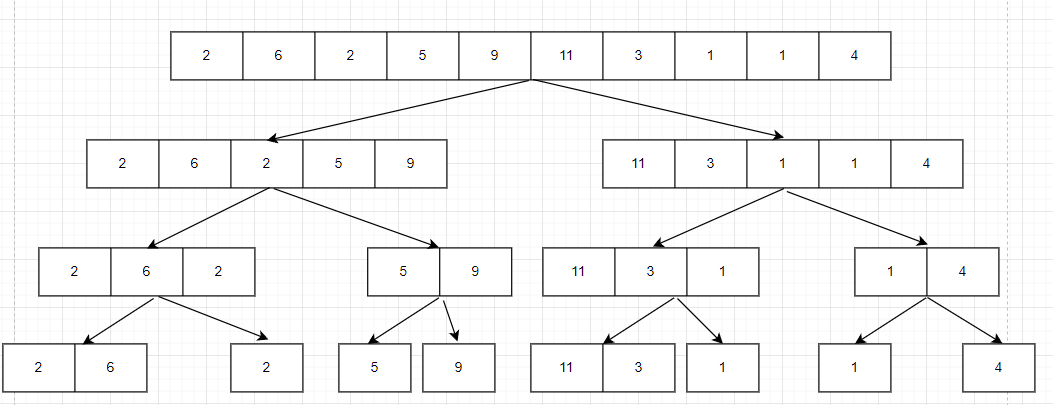
## Ý tưởng

Thuật toán Merge Sort là một thuật toán ứng dụng phương pháp chia để trị, thuật toán này gồm hai phần.

### Chia mảng

Phần đầu tiên, chia các mảng thành hai không gian con, nếu các không gian con này có nhiều hơn một phần tử thì tiếp tục chia đôi. Ngược lại có duy nhất một phần tử hoặc không có phần tử nào (trong trường hợp dãy lẻ) thì bắt đầu trộn lại (gọi đệ qui).

Xét ví dụ cho mảng dưới đây



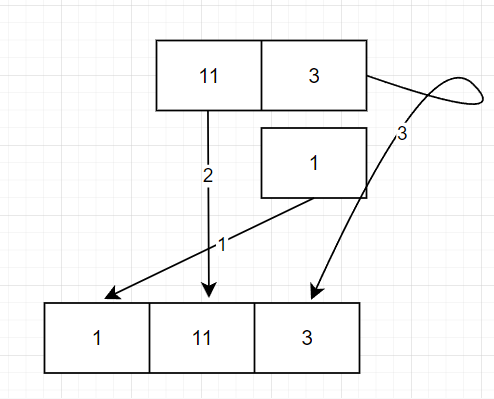
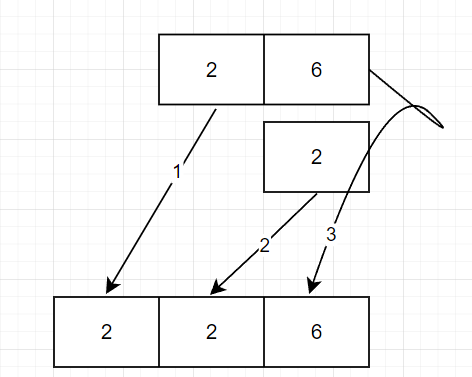
### Trộn mảng

Phần thứ hai chính là quá trình trộn, trong quá trình trộn sẽ kết hợp sắp xếp mảng.

Trộn 2 mảng con được thực hiện như sau:

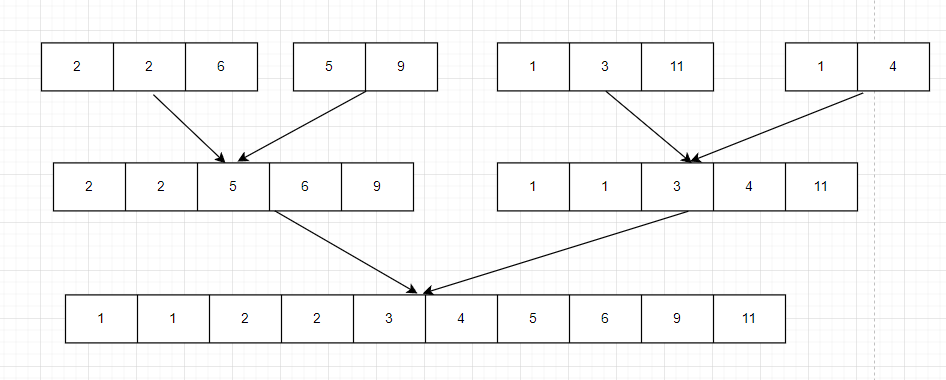
1. Chọn phần tử min ở vị trí đầu của một trong hai mảng, xếp vào mảng cần trộn.
2. Phần tử nào đã xếp vào thì xóa đi, vị trí đầu của mảng là phần tử tiếp theo.
3. Nếu chưa đến cuối mảng thì lặp lại bước 1. Nếu đã đến cuối của một mảng (luôn xảy ra một mảng đã sắp hết và một mảng thì chưa), thì thêm toàn bộ mảng kia vào mảng cần sắp.

Ví dụ xét vài mảng con đầu tiên:



Thứ tự của việc trộn được đánh số như trên hình.

Tương tự cho đến khi trộn hết tất cả các mảng con:



## Độ phức tạp thuật toán

Ta có 2x là lũy thừa của cơ số 2. Với x là số lần nhân tích lũy các số 2 với nhau. Nếu một mảng có n phần tử và ta cho nó bằng 2x, thì:

Ảnh có chứa văn bản

Mô tả được tạo tự động

Nói cách khác, x là số lần nhân đôi số lượng node hiện có để đạt được n node.

Nếu chúng ta làm ngược lại, tìm số lần để chia một mảng n node thành 1 node đơn lẻ, ta sẽ lấy logarith của hai vế theo cơ số 2:

Ảnh có chứa văn bản

Mô tả được tạo tự động

Như vậy, số lần cần thiết để chia nhỏ mảng ra đến khi không chia được nữa là log2n.

Từ đó, phần chia mảng trong thuật toán Merge Sort có độ phức tạp là O(log2n).

Ngoài ra, khi trộn mảng, ta phải duyệt qua từng phần tử của hai mảng con bất kỳ. Như vậy độ phức tạp phần trộn là O(n + m), với n và m là kích thước của hai mảng con.

Bên cạnh đó, không gian phụ mà thuật toán sử dụng là rất lớn phụ thuộc tuyến tính vào số phần tử đầu vào, nên độ phức tạp không gian là O(n). Nếu như dùng Merge Sort trên danh sách liên kết thì sẽ không cần dùng thêm không gian phụ.

Tuy nhiên không gian bộ nhớ cần dùng cho các lời gọi đệ quy lưu trong Stack ở cả trường hợp dùng mảng và danh sách liên kết đều là O(log2n). Nên không gian tối thiểu mà Merge Sort sử dụng là O(log2n). Chỉ là trong trường hợp dùng Merge Sort trên mảng, không gian phụ là O(n) áp đảo không gian O(log2n) cho việc lưu Stack khi n tiến về một số rất lớn, nên trường hợp này sẽ là O(n).

Từ đó ta có độ phức tạp thuật toán là:

|  |  |  |
| --- | --- | --- |
| Best case | Average case | Worst case |
| O(n.log2n) | O(n.log2n) | O(n.log2n) |

**Điểm mạnh:**

1. Khi hai phần tử có cùng giá trị, phần tử nào có chỉ số index nhỏ hơn sẽ đứng trước. Điều này làm thuật toán trở nên Stable trong một số phiên bản.
2. Độ phức tạp ổn định
3. Nhanh hơn Quick Sort khi dữ liệu đầu vào lớn.

**Điểm yếu:**

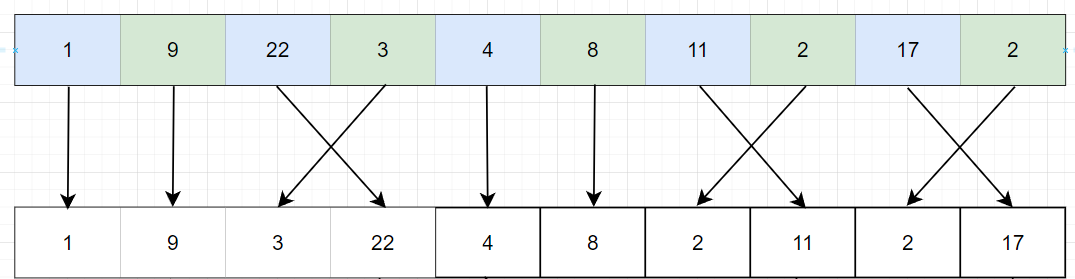
1. Khó cài đặt code
2. Chậm khi dữ liệu gần như được sắp xếp

## Biến thể

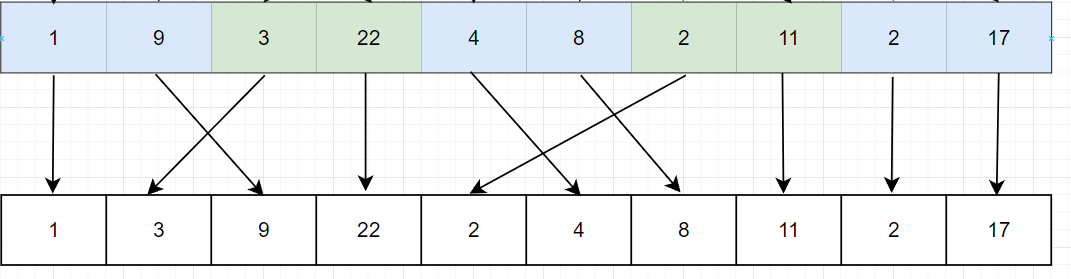
### Bottom – Up

Ý tưởng của thuật toán này là thay vì chia mảng ra nhiều mảng con, thì lại xem xét mảng dưới góc độ là chứa nhiều mảng con. Rồi tiến hành trộn các mảng con liền kề lại với nhau.

Xét ví dụ sau đây, ta tiến hành trộn hai các mảng con kích thước là 1 lại với nhau.



Sau đó trộn các mảng con kích thước là 2.

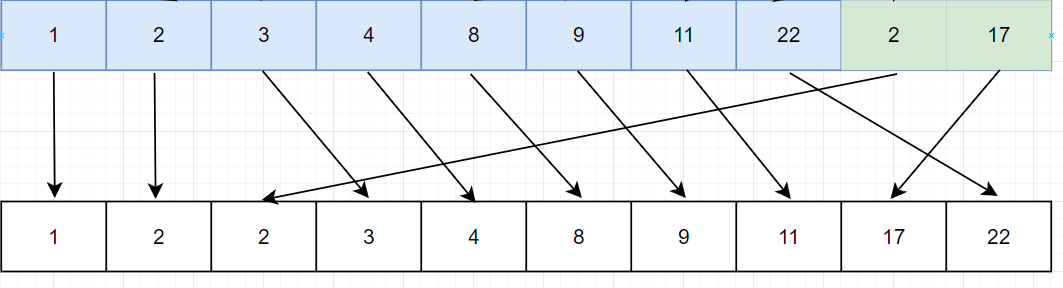


Trộn các mảng con có kích thước là 4.

Ảnh có chứa văn bản, trong nhà

Mô tả được tạo tự động

Và cuối cùng là trộn các mảng có kích thước là 8 lại với nhau và hoàn thành thuật toán.



Chú ý rằng thứ tự của hai số 2 vẫn giữ nguyên, do đó phiên bản này của Merge Sort vẫn là stable.

### Natural Merge Sort

Bởi vì Merge Sort thông thường không nhận biết được một dãy đã sắp xếp nên trong thực tế người ta không dùng Merge Sort thuần để ứng dụng. Thay vào đó họ sử dụng Natural Merge Sort tận dụng các đường chạy để tối ưu thuật toán.

Đường chạy là một dãy tăng không giảm. Ví dụ dãy 7 8 1 2 5 6 4 3 sẽ có bốn đường chạy là (7, 8), (1, 2, 5, 6), (4) và (3). Khi chọn ra được các đường chạy trong cùng một mảng, chúng ta sẽ tiến hành trộn tuần tự các đường chạy đó lại với nhau. Ví dụ sắp xếp cho dãy trên, ta có:

Ảnh có chứa bàn

Mô tả được tạo tự động

Trộn hai đường chạy đầu tiên lại với nhau:

Ảnh có chứa văn bản, trong nhà, cửa sổ, ảnh chụp màn hình

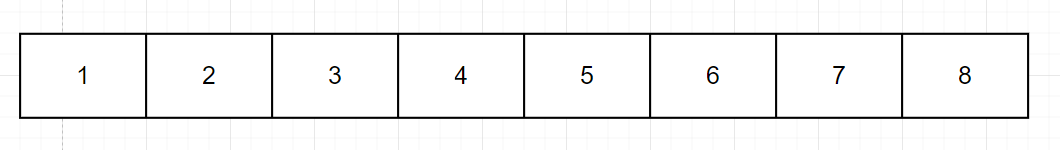
Mô tả được tạo tự động

Tiếp tục trộn:

Ảnh có chứa bàn

Mô tả được tạo tự động

Trộn đường chạy cuối cùng, ta được kết quả là dãy đã sắp xếp.



Thay vì phân ra quá nhiều mảng con, phiên bản này của Merge Sort chỉ phân ra k đường chạy nhất định rồi trộn chúng lại với nhau.

### 