ĐẠI HỌC QUỐC GIA TPHCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

MÔN HỌC: CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Báo cáo thực hành

Đề tài: Nghiên cứu và phân tích các thuật toán sắp xếp

Môn học: Cấu trúc dữ liệu và giải thuật

Sinh viên thực hiện:

Lê Anh Đức - 23120236

Nguyễn Văn Bình Dương - 23120242

Trần Nhật Dương - 23120243

Nguyễn Phúc Hậu - 23120252

Giáo viên hướng dẫn:

Phan Thị Phương Uyên

Ngày 6 tháng 12 năm 2024

Mục lục

Danh sách hình vẽ

1 Giới thiệu

Trong lĩnh vực khoa học máy tính, thuật toán đóng vai trò quan trọng trong việc giải quyết các bài toán phức tạp và tối ưu hóa các quy trình tính toán. Việc lựa chọn và áp dụng thuật toán phù hợp không chỉ giúp nâng cao hiệu suất xử lý mà còn đảm bảo tính chính xác và tiết kiệm tài nguyên tính toán. Một trong những vấn đề cơ bản và quan trọng trong khoa học máy tính là sắp xếp dữ liệu, vì hầu hết các thuật toán phức tạp khác đều có thể được tối ưu hóa thông qua việc sử dụng một thuật toán sắp xếp hiệu quả.

Mục tiêu của đồ án này là nghiên cứu và phân tích các thuật toán sắp xếp phổ biến, đánh giá hiệu suất của chúng qua các chỉ số thời gian thực thi và số lượng phép so sánh khi áp dụng vào các tập dữ liệu khác nhau. Chúng tôi sẽ tập trung vào các thuật toán sắp xếp cơ bản như Quick Sort, Merge Sort, Counting Sort, Radix Sort, các thuật toán đơn giản hơn như Bubble Sort, Selection Sort, và Insertion Sort hay là các thuật toán phức tạp, ít được biết đến như Flash Sort, Binary Insertion Sort, Shaker Sort, Shell Sort. Các thuật toán này sẽ được thử nghiệm trên nhiều kích thước dữ liệu khác nhau, bao gồm cả dữ liệu đã được sắp xếp, dữ liệu đảo ngược và dữ liệu gần như đã được sắp xếp, nhằm đánh giá sự thay đổi hiệu suất trong từng tình huống cụ thể.

Thông qua đồ án này, nhóm nghiên cứu chúng tôi đã rút ra những kết luận về đặc điểm, ưu nhược điểm của từng thuật toán trên lý thuyết lẫn trong các tình huống thực tế. Qua đó cung cấp cho người đọc những hiểu biết sâu sắc hơn về việc lựa chọn thuật toán tối ưu trong các ứng dụng cụ thể.

2 Thông tin

Tên đề tài:	Nghiên cứu và phân tích các thuật toán sắp xếp
Giảng viên hướng dẫn:	Phan Thị Phương Uyên
Nhóm nghiên cứu:	Nhóm 2 - 23CTT3
Thành viên nhóm:	Lê Anh Đức - 23120236
	Nguyễn Văn Bình Dương - 23120242
	Trần Nhật Dương - 23120243
	Nguyễn Phúc Hậu - 23120252
Khóa học:	Học kỳ I, năm học 2024 - 2025

Ngày hoàn thành: Ngày 06 tháng 12 năm 2024

3 Trình bày thuật toán

3.1 Selection sort

3.1.1 Core concept - Ý tưởng

Chọn ra phần tử nhỏ nhất trong mảng chưa sắp xếp. Sau đó, hoán vị với phần tử đầu mảng để đưa về vị trí đúng. Sau mỗi lần hoán vị số phần tử trong mảng chưa sắp xếp sẽ giảm đi 1. Tiếp tục lặp quá trình trên đến khi mảng sắp xếp xong

3.1.2 Step-by-step Explanations - Giải thích từng bước

Bước 1: i = 1;

Bước 2: Tìm phần tử a[min] nhỏ nhất trong dãy chưa sắp xếp từ a[i] đến a[N]

Bước 3: Hoán vị a[min] với a[i];

Bước 4: Nếu i $\leq$ N-1 thì i = i+1; Lặp lại bước 2

Ngược lại: Dừng [14]

Ví dụ:

Giả sử ta có mảng [7,2,19,10,5] cần được sắp xếp

Bước 1: Tìm phần tử nhỏ nhất trong mảng chưa sắp xếp

Sau khi đã đi qua hết phần tử thì ta tìm được phần tử nhỏ nhất là 2.

Bước 2: Hoán vị phần tử nhỏ nhất với phần tử đầu mảng

[7,2,19,10,5] $\Longrightarrow$ [2,7,19,10,5]

Bước 3: Lặp lại các bước trên với những phần tử chưa được sắp xếp.

Mảng hiện tại [2,7,19,10,5]

tìm phần tử nhỏ nhất trong mảng chưa sắp xếp: Phần tử nhỏ nhất là 5

Hoán vị với phần tử đầu mảng: [2,7,19,10,5] $\Longrightarrow$ [2,5,19,10,7]

Tiếp tục đến khi sắp xếp xong mảng: [2,5,19,10,7] $\Longrightarrow$ [2,5,7,10,19]

3.1.3 Complexity Analysis - Phân tích mức độ phức tạp

Time complexity

Giả sử có một mảng cần sắp xếp chứa n phần tử

Ta tìm phần tử nhỏ nhất trong n phần tử chưa sắp xếp. Số phép so sánh sẽ là n-1;

Sau mỗi vòng lặp, số lượng phần tử cần sắp xếp giảm đi 1.

Tiếp tục vòng lặp cho đến khi số phép so sánh là 1.

Số phép so sánh của thuật toán là cấp số cộng $= (n-1) + (n-2) + \cdots + 1 = \frac{n.(n-1)}{2}$

$\Longrightarrow$ Độ phức tạp về thời gian sẽ là $O(n^2)$

Độ phức tạp ở mọi trường hợp (tốt nhất, trung bình và tệ nhất) đều là $O(n^2)$. Lý do là khi tìm phần tử nhỏ nhất cho vị trí i, thuật toán phải khỏa sát các phần tử còn lại trong mảng chưa sắp xếp, dù phần tử i đã ở đúng vị trí.

Space complex

Thuật toán này chỉ dùng một lượng bộ nhớ cố định cho các biến tạm thời (temporary variable) để tìm phần tử nhỏ nhất và hoán đổi. Vì vậy, Độ phức tạp về không gian là O(1).

3.1.4 Variants and Optimizations - Các biến thể và tối ưu thuật toán

Double selection sort: hay cocktail sort, là 1 biến thể selection sort, tìm cả giá trị nhỏ nhất và lớn nhất trong danh sách trong mỗi lượt duyệt. Điều này yêu cầu ba phép so sánh cho mỗi hai phần tử (một cặp phần tử được so sánh, sau đó phần tử lớn hơn được so sánh với giá trị lớn nhất và phần tử nhỏ hơn được so sánh với giá trị nhỏ nhất), thay vì một phép so sánh trên mỗi phần tử như thuật toán selection sort thông thường. Tuy nhiên, nó chỉ cần thực hiện một nửa số lượt duyệt, mang lại lợi ích tiết kiệm 25% tổng thời gian thực hiện. [5]

Bingo sort: là 1 biến thể của selection sort tối ưu cho mảng có nhiều phần tử giống nhau, sau khi tìm được phần tử nhỏ nhất thì thuật toán sẽ đưa tất cả các phần tử có cùng giá tị nhỏ nhất về đúng vị trí, sau đó tiếp tục tìm phần tử nhỏ nhất tiếp theo.[5]

Heap sort: khắc phục nhược điểm của selection sort là thời gian tìm kiếm phần tử , tối ưu tìm kiếm phần tử lớn nhất/bé nhất bằng cách sử dụng cấu trúc dữ liệu heap, từ đó tối ưu Độ phức tạp về thời gian.

Trong các biến thể trên, heap sort là là biến thể tối ưu nhất, vì nó đã tối ưu được nhược điểm lớn

nhất là tìm kiếm phần tử. Biến thể này sẽ được trình bày chi tiết ở phần sau của báo cáo, để làm rõ việc nó đã tối ưu như thế nào.

3.2 Insertion sort

3.2.1 Core concept - Ý tưởng

Giả sử cần sắp xếp một dãy có **thứ tự tăng dần**. Cho một dãy ban đầu $\boldsymbol{a_0, a_1, \ldots, a_{n-1}}$, có thể xem như đã có một đoạn gồm một phần tử $\boldsymbol{a_1}$ đã có thứ tự. Sau đó thêm $\boldsymbol{a_2}$ vào đoạn $\boldsymbol{a_0}$ sẽ có đoạn $\boldsymbol{a_0, a_1}$ đã được sắp; tiếp tục thêm $\boldsymbol{a_2}$ vào đoạn $\boldsymbol{a_0, a_1}$ để có đoạn $\boldsymbol{a_0, a_1, a_2}$ được sắp; tiếp tục cho đến khi thêm xong $\boldsymbol{a_{n-1}}$ vào đoạn $\boldsymbol{a_0, a_1, \ldots, a_{n-2}}$ thì sẽ có dãy ban đầu mà đã được sắp xếp. [14]

3.2.2 Step-by-step Explanations - Giải thích từng bước [14]

Bước 1: $i = 1$; // giả sử có đoạn $a[0]$ đã được sắp

Bước 2: Gán $x = a[i]$; Tìm vị trí pos thích hợp trong đoạn $a[1]$ đến $a[i-1]$ để chèn $a[i]$.

Bước 3: Dời chỗ các phần tử từ $x = a[pos]$ đến $x = a[i-1]$ sang phải một vị trí để dành chỗ cho $x = a[i]$.

Bước 4: $a[pos] = x$; // có đoạn $a[1]..a[i]$ đã được sắp.

Bước 5: $i = i + 1$;

Nếu $i < n$: Lặp lại Bước 2.

Ngược lại: Dừng.

Mã giả :

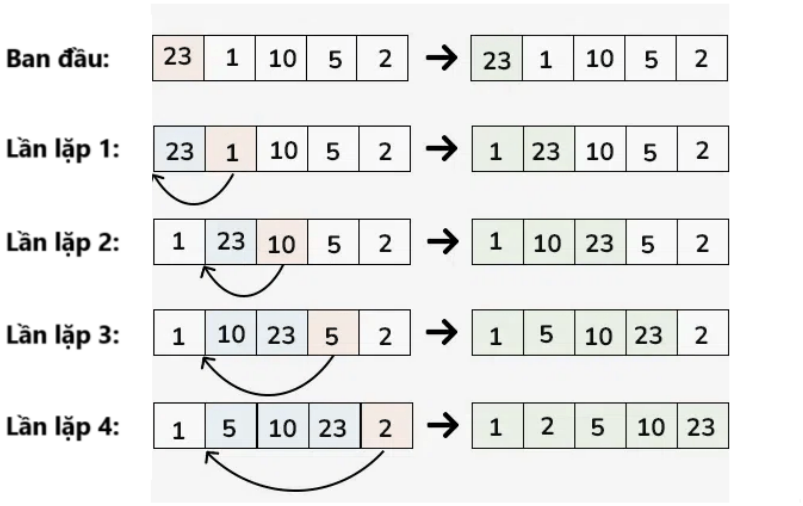
Input: mảng a có n phần tử chưa được sắp xếp

Output: toàn bộ phần tử của mảng a đã được sắp xếp theo thứ tự không giảm.

```
for i = 1 to n - 1 do
    x = a[i]
    pos = i - 1
    while pos ≥ 0 and a[pos] > x
            a[pos + 1] = a[pos]
            pos = pos − 1
    a[pos + 1] = x // có đoạn a[1]..a[i] đã được sắp
```

Ví dụ : Cho dãy số a:

$$23, 1, 10, 5, 2$$



Hình 1: Mô tả quá trình sắp xếp chèn. [4]

3.2.3 Complexity Analysis - Phân tích mức độ phức tạp

- **Độ phức tạp thời gian**

  Trong các thuật toán sắp xếp dựa trên so sánh (comparison-based algorithms), số lần so sánh giữa các phần tử thường là phép toán chiếm ưu thế và có tác động chính đến độ phức tạp thời gian. Vì vậy, trong báo cáo này, chúng tôi sẽ tập trung phân tích chi tiết số phép so sánh, thay vì số phép gán, nhằm đánh giá hiệu suất của các thuật toán một cách rõ ràng và chính xác hơn

  Đối với giải thuật chèn trực tiếp (insertion sort), các phép so sánh để tìm vị trí thích hợp **pos** xảy ra trong mỗi vòng lặp **while**, và mỗi lần xác định vị trí đang xét không thích hợp, sẽ dời chỗ phần tử a_{pos} tương ứng.

  Trong trường hợp xấu nhất (worst case), tức là phép so sánh $a_{pos} > x$, tìm vị trí thích hợp để chèn a_i, được thực hiện nhiều nhất. Khi đó $a_0 > a_1, a_1 > a_2, \ldots, a_{n-2} > a_{n-1}$. Nói cách khác, trường hợp xấu nhất là một dãy có các giá trị giảm dần nghiêm ngặt. Số lượng **key comparison** (phép so

sánh chính) đối với trường hợp trên là [11]

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} = \frac{n(n-1)}{2} \in \mathcal{O}(n^2)$$

Trong trường hợp tốt nhất (best case), dễ nhận ra đó là khi phép so sánh $a_{pos} > x$ chỉ được thực hiện duy nhất một lần tại mỗi vòng lặp **for**. Có nghĩa là $a_{i-1} \leq a_i$, $i \in \{1, 2, \ldots, n-1\}$. Vậy nên trường hợp tốt nhất chính là khi dãy đã có thứ tự đã thỏa mãn yêu cầu sắp xếp. Số lượng **key comparison** trong trường hợp này là:[11]

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \mathcal{O}(n)$$

Kết quả của Insertion sort trong trường hợp tốt nhất thường ít mang tính ứng dụng thực tiễn, bởi một dãy đã sắp xếp hoàn chỉnh là tình huống rất đặc biệt và hiếm khi xảy ra. Tuy nhiên, với các danh sách tập tin trong nhiều ứng dụng, dữ liệu thường xuất hiện ở trạng thái **gần như có thứ tự**. Do đó, Insertion sort vẫn thể hiện sự hiệu quả vượt trội trong những trường hợp này.[11]

Một phân tích nghiêm ngặt về **hiệu quả trung bình** (average) của thuật toán dựa trên việc tìm hiểu số lượng cặp phần tử không theo thứ tự. Nó cho thấy rằng trên các dãy có các giá trị được sắp xếp ngẫu nhiên, sắp xếp chèn thực hiện trung bình một nửa số lần so sánh so với trên các mảng giảm dần, tức là [11]

$$C_{\text{avg}}(n) \approx \frac{n^2}{4} \in \mathcal{O}(n^2).$$

Với chi phí trung bình chỉ bằng một nửa so với trường hợp xấu nhất, cùng khả năng xử lý hiệu quả các dãy có các giá trị gần như đã sắp xếp, Insertion sort thể hiện sự vượt trội so với các thuật toán cơ bản khác như Selection sort và Bubble sort. Từ phân tích về chi phí trung bình $C_{\text{avg}}(n)$ thì có thể đi đến kết luận **độ phức tạp thời gian** của Insertion sort là: [11]

$$T_{(n)} \in \mathcal{O}(n^2).$$

- **Độ phức tạp không gian**

Insertion sort là một thuật toán sắp xếp tại chỗ (in-place), có nghĩa là nó thao tác trực tiếp trên dãy các giá trị mà không cần sử dụng bộ nhớ ngoài hay bất kỳ cấu trúc dữ liệu bổ sung nào. Nhờ vậy, thuật toán này chỉ cần sử dụng một lượng bộ nhớ cố định, dẫn đến độ phức tạp không gian của nó là $\mathcal{O}(1)$.

3.2.4 Variants and Optimizations - Các biến thể và tối ưu thuật toán

Có một số biến thể khác của Insertion sort như là Divide-and-Conquer Insertion Sort, Binary Insertion sort và Shell sort. Đối với Binary Insertion sort, đúng như cái tên của nó, khi tìm kiếm vị trí cần thiết để chèn phần tử x thì sẽ dùng Binary Search - thuật toán tìm kiếm nhị phân trên dãy giá trị đã được sắp xếp trước đó. Còn đối với Shell sort, một thuật toán do nhà khoa học máy tính Donald. L. Shell, là một cải tiến vượt trội hơn của Insertion sort. Thay vì duyệt tuần tự từng phần tử như Insertion sort, thuật toán sẽ duyệt qua các phần tử cách nhau một khoảng h, giúp di chuyển các phần tử ở xa nhau đến gần vị trí đúng của chúng sớm hơn. Cả hai thuật toán trên đều sẽ được chúng tôi phân tích kỹ hơn ở phần 3.3 (Shell sort) và phần 3.11 (Binary Insertion sort) của tài liệu này.

Divide-and-Conquer Insertion Sort, là một biến thể của Insertion sort bằng cách dùng chiến lược chia để trị để phân dãy giá trị thành hai nửa, sau đó sắp xếp từng nửa một cách song song và hợp nhất chúng. Điểm đặc biệt là quá trình hợp nhất sử dụng logic của Insertion sort. Từ đó biến thể này cải thiện hiệu năng bằng cách giảm độ phức tạp tính toán xuống $O(n.log_2^2(n))$ nhờ tính song song và tận dụng bộ nhớ đệm (cache) hiệu quả hơn. [13]

3.3 Shell sort

3.3.1 Core concept - Ý tưởng

Shell sort là một phương pháp cải tiến của phương pháp Insertion sort, của nhà khoa học máy tính Donald. L. Shell. Ý tưởng của phương pháp sắp xếp là phân chia dãy ban đầu thành những dãy con gồm các phần tử ở cách nhau h vị trí. Tiến hành sắp xếp các phần tử trong cùng dãy con sẽ làm cho các phần tử được đưa về vị trí đúng tương đối, sau đó giảm khoảng cách h để tạo ra dãy con mới và tiếp tục sắp xếp các dãy con đó cho đến khi $h = 1$. [14]

3.3.2 Step-by-step Explanations - Giải thích từng bước

Yếu tố quyết định tính hiệu quả của thuật toán là cách chọn khoảng cách h trong từng bước sắp xếp và số bước sắp xếp. Giả sử quyết định sắp xếp k bước, các khoảng cách chọn phải thỏa điều kiện:

$$h_i > h_{i+1} \text{ và } h_k = 1 \text{ [14]}$$

Tuy nhiên đến nay vẫn chưa có tiêu chuẩn nào rõ ràng trong việc lựa chọn dãy giá trị

khoảng cách tốt nhất, một dãy số được Knuth đề nghị:

$$h_i = (h_{i-1})/3 \text{ và } h_k = 1, k = log_3 n - 1 \text{ [14]}$$

Ví dụ một dãy giá trị của h có thể là 127, 40, 13, 4, 1

hay

$$h_i = (h_{i-1})/2 \text{ và } h_k = 1, k = log_2 n - 1 \text{ [14]}$$

Ví dụ: 15, 7, 3, 1

Các bước tiến hành như sau: [14]

Bước 1: Chọn ; khoảng cách $h[1], h[2], ..., h[k]$; $i = 1$

Bước 2: Phân chia dãy ban đầu thành các dãy con cách nhau $h[i]$ khoảng cách. Sắp xếp từng dãy con bằng phương pháp chèn trực tiếp.

Bước 3: $i = i + 1$;

Nếu $i > k$: Dừng.

Nếu $i <= k$: Lặp lại Bước 2

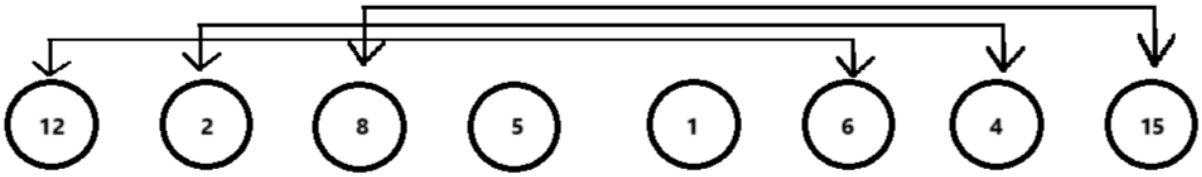
.

Ví dụ [14]: Cho dãy giá trị: Cho dãy giá trị:

$$12, 2, 8, 5, 1, 6, 4, 15$$

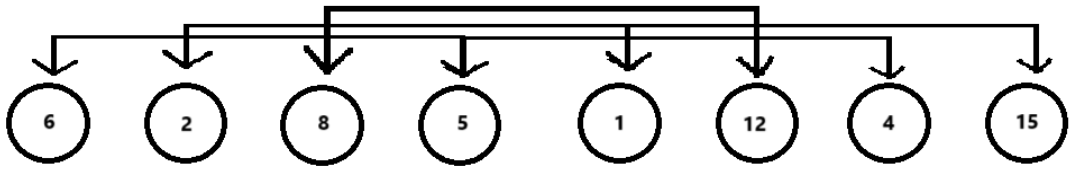
Giả sử chọn các khoảng cách là 5, 3, 1

$h = 5$: xem dãy ban đầu như các dãy con:



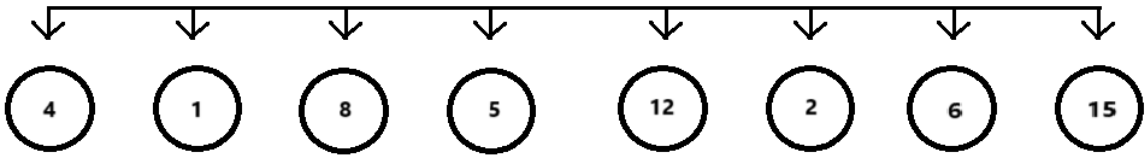
Hình 2: Mô tả quá trình sắp xếp các dãy con của Shell sort với $h = 5$.

$h = 3$: (sau khi đã sắp xếp các dãy con ở bước trước)



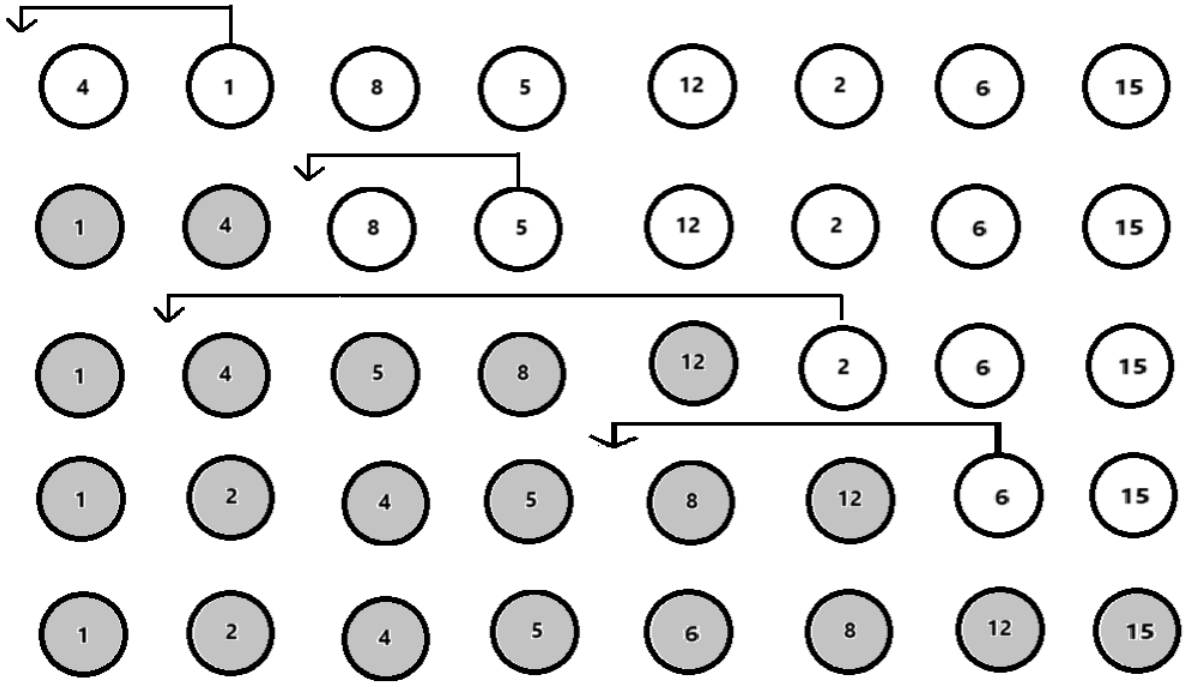
Hình 3: Tiếp tục quá trình sắp xếp các dãy con của Shell sort với $h = 3$.

$h = 1$: (sau khi đã sắp xếp các dãy con ở bước trước)



Hình 4: Tiếp tục quá trình sắp xếp các dãy con của Shell sort với $h = 1$.

Sắp xếp dãy ta có:



Hình 5: Tiếp tục quá trình sắp xếp như Insertion sort thông thường.

3.3.3 Complexity Analysis - Phân tích mức độ phức tạp

• Độ phức tạp thời gian

Hiện nay việc đánh giá giải thuật Shell sort dẫn đến những vấn đề toán học rất phức tạp, thậm chí một số chưa được chứng minh. Tuy nhiên hiệu quả của thuật toán còn phụ thuộc vào dãy các độ dài được chọn. Trong trường hợp chọn dãy độ dài theo công thức $h_i = (h_i - 1)/2$ và $h_k = 1, k = log_2 - 1$ thì giải thuật có độ phức tạp $\approx\ n^{1.2} << n^2$ [14]

• Độ phức tạp không gian

Shell Sort, tương tự như Insertion Sort, là một thuật toán sắp xếp tại chỗ (in-place), thực hiện thao tác trực tiếp trên mảng đầu vào mà không cần sử dụng bất kỳ bộ nhớ ngoài hoặc cấu trúc dữ liệu bổ sung nào. Mặc dù thuật toán này sử dụng thêm một mảng phụ để quản lý khoảng cách giữa các phần tử được so sánh, nhưng mảng này không làm tăng đáng kể yêu cầu bộ nhớ, vì nó chỉ chiếm một lượng không gian cố định. Do đó, Shell Sort cũng có độ phức tạp không gian là $O(1)$, làm nổi bật tính hiệu quả về mặt bộ nhớ của thuật toán, đặc biệt khi so sánh với các thuật toán sắp xếp cần sử dụng bộ nhớ phụ trợ như Merge Sort.

3.3.4 Variants and Optimizations - Các biến thể và tối ưu thuật toán [6]

Có một số biến thể của Shell Sort, trong đó sự khác biệt chính nằm ở cách chọn dãy khoảng cách h được chọn. Dưới đây là một số biến thể phổ biến và hiệu năng cao mà chúng tôi có thể kể tới :

• Dãy khoảng cách nguyên gốc - Original Sequence

Dãy nguyên gốc trong giải thuật Shell Sort được xác định bởi công thức: $h_i = \left\lfloor \frac{2^i}{n} \right\rfloor$, với i là chỉ số của bước sắp xếp. Với cách chọn này, khoảng cách giảm theo cấp số nhân, thường là $n/2, n/4, n/8, n/16, ..., 1$. Dãy này dễ triển khai do cấu trúc đơn giản và không yêu cầu tính toán phức tạp. Tuy nhiên, nó không tận dụng tối đa khả năng sắp xếp của thuật toán Shell sort. Trong trường hợp xấu nhất, độ phức tạp thời gian của giải thuật sử dụng dãy nguyên gốc là $O(n^2)$, làm giảm hiệu năng đối với các mảng lớn hoặc gần như sắp xếp ngược.

• Dãy của Sedgewick

Robert Sedgewick là một nhà khoa học máy tính nổi tiếng, ông đã đề xuất dãy khoảng cách được định nghĩa bởi công thức $h_i = 4^i + 3.2^{i-1} + 1$. có các giá trị đầu tiên là $1, 5, 19, 41, 109, ...$. Dãy Sedgewick có cấu trúc phức tạp hơn và được thiết kế để giảm số lượng phép so sánh và gán trong khi vẫn đảm bảo sắp xếp hiệu quả. Nghiên cứu lý thuyết và thực nghiệm cho thấy

độ phức tạp trung bình của giải thuật với dãy này là $O(n^{4/3})$, làm cho nó phù hợp với các mảng lớn hơn, nơi sự khác biệt về hiệu năng trở nên rõ rệt.

• Dãy của Tokuda

Takahiro Tokuda là một nhà nghiên cứu Nhật Bản, ông đã giới hiệu một cách chọn khoảng cách với công thức $h_i = \left\lceil \frac{9.(9/4)^i - 4}{5} \right\rceil$ cho các giá trị như $1, 4, 9, 20, ...$ có hiệu năng vượt trội trong nhiều trường hợp. Dãy này được thiết kế để tối ưu hóa hiệu năng sắp xếp dựa trên phân tích thực nghiệm. Dãy Tokuda được đánh giá cao vì sự cải thiện rõ rệt trong việc giảm số lượng phép so sánh và gán, đặc biệt đối với các mảng có kích thước lớn. Độ phức tạp trung bình của thuật toán khi sử dụng dãy Tokuda thường dao động quanh$O(n^{1.25})$ một sự cải thiện so với dãy nguyên gốc và gần tương đương với dãy Sedgewick.

• So sánh với dãy Knuth

Dãy của Knuth, với công thức $h_i = 3^k - 1$ cho các giá trị như $1, 4, 13, 40, ...$, có hiệu năng vượt trội trong nhiều trường hợp. Với độ phức tạp trung bình là $O(n^{1.2})$, thấp hơn một chút so với 2 dãy trên.

So với các dãy nguyên gốc, các dãy của Sedgewick, Tokuda, Knuth đều cải thiện hiệu năng nhờ lựa chọn khoảng cách không đều, giúp tối ưu hóa quá trình sắp xếp trên các mảng lớn và gần như ngẫu nhiên. Dãy Knuth tuy dễ triển khai nhưng không đạt đến mức hiệu năng cao nhất mà các dãy Sedgewick và Tokuda có thể mang lại.

3.4 Bubble sort

Bubble Sort hay còn được biết đến với tên gọi sắp xếp nổi bọt, tên gọi được bắt nguồn từ nguyên lý hoạt động của thuật toán giống với hiện tượng thực tế. Cụ thể, trong cuộc sống hàng ngày ta dễ dàng bắt gặp hình ảnh bọt khí nổi lên trong một cốc nước ngọt, bọt khí to chứa nhiều khí CO2 sẽ nổi lên trước còn những bọt khí nhỏ sẽ nổi theo sau. Tương tự như vậy, thuật toán sắp xếp nổi bọt sẽ đẩy phần tử thỏa mãn điều kiện lên trên đúng với vị trí của nó sau mỗi bước.

3.4.1 Core concept - Ý tưởng

Từ ý tưởng trên, người ta sẽ duyệt từ đầu (hoặc cuối) mảng kiểm tra hai phần tử nằm cạnh nhau. Nếu cặp phần tử này không thỏa mãn điều kiện sắp xếp (ví dụ, không theo thứ tự tăng dần), chúng sẽ được hoán đổi. Như vậy, sau mỗi bước duyệt, ta sẽ đưa được một phần tử về đúng vị trí của nó. Quá trình này được lặp đi lặp lại, với phạm vi duyệt giảm dần sau mỗi vòng lặp, vì các phần tử đã "nổi lên"đúng vị trí nên không cần kiểm tra lại.

3.4.2 Step-by-step Explanations - Giải thích từng bước

Trong báo cáo này, tôi lựa chọn triển khai thuật toán Bubble Sort để sắp xếp mảng dữ liệu tăng dần và bắt đầu từ trái sang phải.

Bước 1: Thực hiện lặp các bước sau đến khi số lượng phần tử chưa sắp xếp nhỏ hơn 2.

Bước 2: So sánh 2 phần tử liền kề nhau trong mảng chưa được sắp xếp.

Bước 3: Nếu 2 phần tử chưa đúng vị trí (phần tử phía trước lớn hơn phía sau) thì đổi chỗ hai phần tử này.

Nhằm giúp người đọc dễ dàng hình dung được các bước của thuật toán, đây là đoạn mã giả tương ứng với mô trả phía trên:

Input: Mảng A kích thước n

Output: Mảng A được sắp xếp tăng dần

For i = n down to 2:

    For j = 0 to i - 2:

        If A[j] > A[j+1]:

            Swap A[j] và A[j+1]

Tiếp theo, tôi sẽ thực hiện sắp xếp tăng dần mảng dữ liệu bằng thuật toán Bubble Sort với mảng A = [2,6,9,5,1]

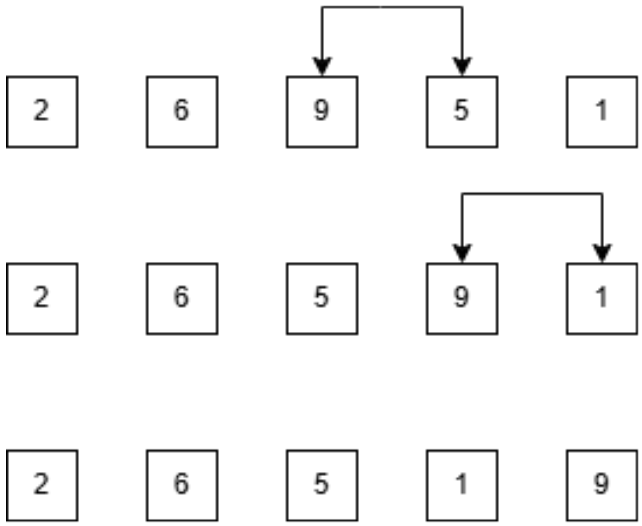
Lần duyệt thứ 1:

So sánh A[0] và A[1]: $2 < 6$ => không đổi.

So sánh A[1] và A[2]: $6 < 9$ => không đổi.

So sánh A[2] và A[3]: $9 > 5$ => đổi chỗ => A = [2,6,5,9,1].

So sánh A[3] và A[4]: $9 > 1$ => đổi chỗ => A = [2,6,5,1,9].



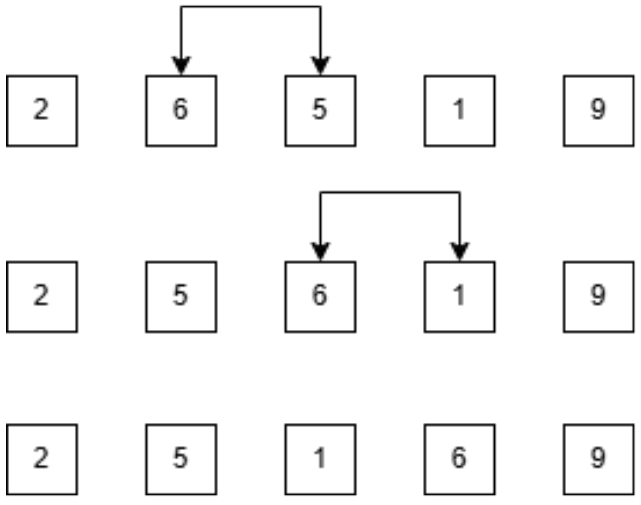
Hình 6: Lần duyệt thứ nhất Bubble Sort

Lần duyệt thứ 2:

So sánh A[0] và A[1]: $2 < 6$ => không đổi.

So sánh A[1] và A[2]: $6 > 5$ => đổi chỗ => A = [2,5,6,1,9].

So sánh A[2] và A[3]: $6 > 1$ => đổi chỗ => A = [2,5,1,6,9].

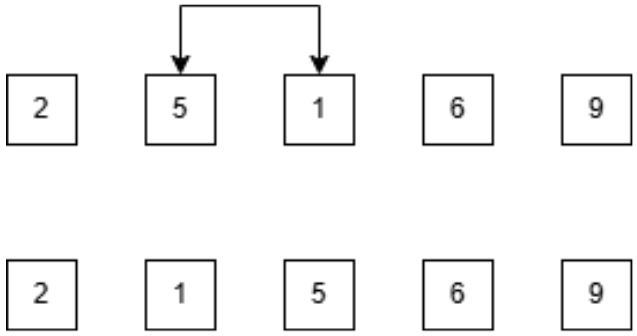


Hình 7: Lần duyệt thứ hai Bubble Sort

Lần duyệt thứ 3:

So sánh A[0] và A[1]: $2 < 5$ => không đổi.

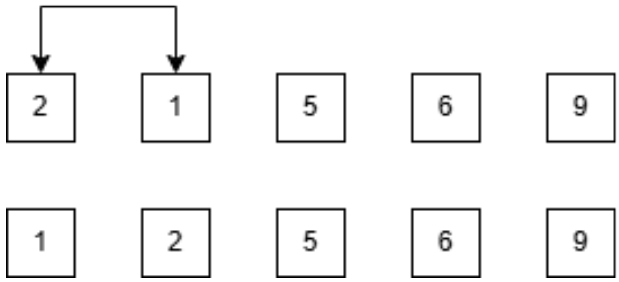
So sánh A[1] và A[2]: 5 > 1 => đổi chỗ => A = [2,1,5,6,9].



Hình 8: Lần duyệt thứ ba Bubble Sort

Lần duyệt thứ 4:

So sánh A[0] và A[1]: 2 > 1 => đổi chỗ => A = [1,2,5,6,9].



Hình 9: Lần duyệt thứ tư Bubble Sort

Như vậy, đối với mảng A = [2,6,9,5,1] thì thuật toán Bubble Sort cần 10 phép so sánh để hoàn thành việc sắp xếp.

3.4.3 Complexity Analysis - Phân tích mức độ phức tạp

Time Complexity

Đối với thuật toán Bubble Sort, số phép so sánh độc lập với dữ liệu, tức là đối với mọi kiểu dữ liệu có kích thước bằng nhau ta đều cần dùng cùng một số lượng phép so sánh. Ngược lại, phép hoán đổi vị trí của hai phần tử kề nhau lại phụ thuộc vào kết quả so sánh nên nó thay đổi tùy theo loại dữ liệu đầu vào.

Đối với mọi trường hợp thì số phép so sánh là $\sum_{i=1}^{n-1}(n-i+1) = \frac{n(n-1)}{2}$. Có thể hiểu công thức này một cách đơn giản như sau, với bộ dữ liệu gồm n phần tử, ta cần so sánh tất cả các cặp có thể có của

mảng. Số lượng cặp này chính là số cách chọn 2 phần tử từ n phần tử, áp dụng công thức tổ hợp ta có C_n^2 cách, mà $C_n^2 = \frac{n(n-1)}{2}$.

Đối với phép đổi chỗ hai vị trí thì ta có trường hợp tốt nhất chính là 0 khi mảng đã được sắp xếp rồi, ta không cần đổi chỗ bất kì phần tử nào. Trường hợp xấu nhất là ta luôn phải đổi chỗ với mỗi phép so sánh nên số phép đổi chỗ là $\frac{n(n-1)}{2}$ khi mảng được sắp xếp giảm dần.

Độ phức tạp của thuật toán Bubble Sort được đánh giá dựa trên phép so sánh vì trong thuật toán này nó là yếu tố trội hơn các yếu tố còn lại. Như vậy, độ phức tạp của Bubble Sort là O(n^2) vì số phép so sánh là $\frac{n(n-1)}{2} \in O(n^2)$.

Trường hợp tốt nhất: O(n^2)

Trường hợp tệ nhất: O(n^2)

Trường hợp trung bình: O(n^2)

Space Complexity

Thuật toán Bubble Sort chỉ yêu cầu một vùng nhớ cố định để lưu trữ mảng dữ liệu nên nó không phụ thuộc vào kích thước của mảng đầu vào. Ngoài ra, thuật toán sắp xếp trực tiếp trên mảng nên không cần sử dụng bộ nhớ phụ. Từ các nguyên nhân trên nên Bubble Sort có độ phức tạp không gian là O(1).

3.4.4 Variants and Optimizations - Các biến thể và tối ưu thuật toán

Thuật toán sắp xếp nổi bọt có thể được tối ưu so với phiên bản gốc bằng cách thêm một biến đánh dấu trước mỗi lần duyệt mảng các phần tử chưa được sắp xếp. Cụ thể, sau khi thực hiện hoán đổi vị trí của hai phần tử nằm kề nhau, ta thay đổi trạng thái của biến đánh dấu. Sau mỗi lần duyệt mảng, tiến hành kiểm tra biến đánh dấu nếu không thay đổi chứng tỏ mảng đã đúng với thứ tự cần sắp xếp và dừng thuật toán. Như vậy, việc cải thiện giúp giảm đi đáng kể thời gian chạy và kết thúc sớm hơn so với thuật toán gốc với một số trường hợp đã mô tả.

Đây là đoạn mã giả đã cải thiện so với thuật toán cũ:

Input: Mảng A kích thước n

Output: Mảng A được sắp xếp tăng dần

For i = n down to 2:

    Đặt biến flag = false

    For j = 0 to i-2:

        If A[j] > A[j+1]:

            Swap A[j] và A[j+1]

            flag = true

If flag == false:

Break

Đối với cải tiến này, Bubble Sort sẽ hoạt động hiệu quả hơn với các loại dữ liệu như nearly sorted và sorted với độ phức tạp rơi vào O(n).

Bubble Sort có một số biến thể như: Optimized Bubble Sort, Cocktail Shaker Sort (Shaker Sort),... Trong đó, Optimized Bubble Sort đã được trình bày phía trên, với ý tưởng tạo ra một biến lá cờ để kiểm tra xem mảng có thay đổi hay không nhằm dừng thuật toán sớm trong một số trường hợp. Ngoài ra, Shaker Sort − một biến thể trực tiếp của Bubble Sort sẽ được trình bày chi tiết ở phần sau của báo cáo này để làm rõ ý tưởng triển khai và những cải tiến vượt trội so với phiên bản gốc.

3.5 Shaker sort

Shaker Sort, hay còn gọi là Cocktail Shaker Sort, là một biến thể của Bubble Sort. Tên gọi của thuật toán được lấy cảm hứng từ kỹ thuật pha chế cocktail. Cụ thể, các bartender lắc chiếc cốc pha chế nằm ngang qua lại từ trái sang phải, rồi từ phải sang trái. Điểm đặc biệt trong kỹ thuật pha chế này là sau mỗi lần lắc, quỹ đạo của chiếc cốc sẽ dần thu hẹp do người pha chế giảm dần cường độ của lực lắc.

3.5.1 Core concept - Ý tưởng

Từ ý tưởng trên, Shaker Sort ra đời với nguyên lý hoạt động là duyệt từ phía này sang phía kia của dữ liệu, sau mỗi lần duyệt sẽ tìm ra hai vị trí đúng ở đầu và cuối dữ liệu từ đó thu hẹp phạm vi sắp xếp.

3.5.2 Step-by-step Explanations - Giải thích từng bước

Trong báo cáo này, tôi lựa chọn triển khai thuật toán Shaker Sort để sắp xếp mảng theo thứ tự tăng dần và bắt đầu từ bên trái.

Bước 1: Khởi tạo các giá trị đại diện cho vị trí trái nhất (left), phải nhất (right) của dữ liệu, đặc biệt là biến k lưu vị trí lần gần nhất thay đổi.

Bước 2: Thực hiện lặp với điều kiện biên trái còn nhỏ hơn biên phải.

Bước 3: So sánh lần lượt 2 phần tử liền kề từ trái sang phải. Nếu phần tử phía trước lớn hơn phần tử phía sau, đổi chỗ chúng và cập nhật vị trí k.

Bước 4: Gán biên phải bằng k.

Bước 5: So sánh lần lượt 2 phần tử liền kề từ phải sang trái. Nếu phần tử phía trước lớn hơn phần tử phía sau, đổi chỗ chúng và cập nhật vị trí k.

Bước 6: Gán biên trái bằng k.

Nhằm giúp người đọc dễ dàng hình dung được các bước của thuật toán, đây là đoạn mã giả tương ứng với mô trả phía trên:

Input: Mảng A kích thước n

Output: Mảng A được sắp xếp tăng dần

Đặt left = 0, right = n - 1, k = 0, i = 0

While left < right:

    For i = left to right - 1:

        If A[i] > A[i + 1]:

            Swap A[i] và A[i + 1]

            k = i

    right = k

    For j = right down to left + 1:

        If A[j] < A[j - 1]:

            Swap A[j] và A[j - 1]

            k = i

    left = k

Tiếp theo, tôi sẽ thực hiện sắp xếp tăng dần mảng dữ liệu bằng thuật toán Shaker Sort với mảng A = [2,6,9,5,1]

Lần duyệt thứ 1:

So sánh A[0] và A[1]: 2 < 6 => không đổi.

So sánh A[1] và A[2]: 6 < 9 => không đổi.

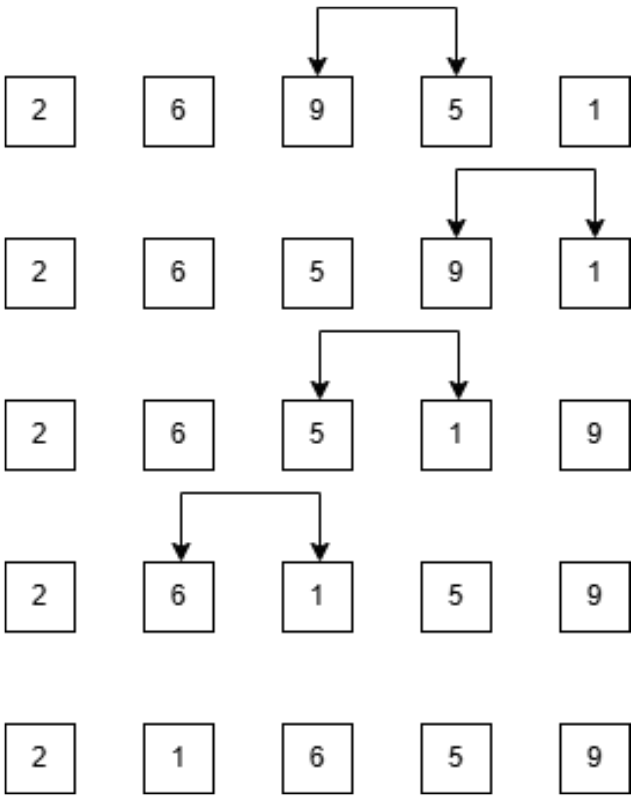
So sánh A[2] và A[3]: 9 > 5 => đổi chỗ => A = [2,6,5,9,1].

So sánh A[3] và A[4]: 9 > 1 => đổi chỗ => A = [2,6,5,1,9].

So sánh A[2] và A[3]: 5 > 1 => đổi chỗ => A = [2,6,1,5,9].

So sánh A[1] và A[2]: 6 > 1 => đổi chỗ => A = [2,1,6,5,9].

So sánh A[0] và A[1]: 2 > 1 => đổi chỗ => A = [1,2,6,5,9].



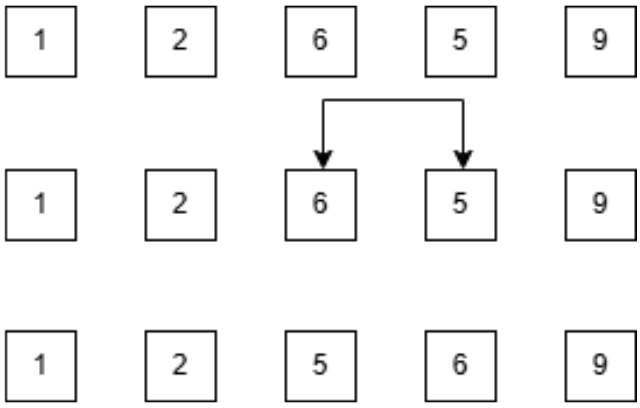
Hình 10: Lần duyệt thứ nhất Shaker Sort

Lần duyệt thứ 2:

So sánh A[1] và A[2]: $2 < 6 \Rightarrow$ không đổi.

So sánh A[2] và A[3]: $6 > 5 \Rightarrow$ đổi chỗ $\Rightarrow$ A = [1,2,5,6,9]..

So sánh A[1] và A[2]: $2 < 5 \Rightarrow$ không đổi.



Hình 11: Lần duyệt thứ hai Shaker Sort

Như vậy, đối với mảng A = [2,6,9,5,1] thì thuật toán Shaker Sort cần 10 phép so sánh để hoàn thành việc sắp xếp. So sánh với ví dụ này ở phần 3.4.2 ta nhận thấy đối với các mảng dữ liệu với thứ tự ngẫu nhiên thì số phép so sánh của Bubble Sort và Shaker Sort gần như giống nhau.

3.5.3 Complexity Analysis - Phân tích mức độ phức tạp

Mặc dù thuật toán Shaker Sort kế thừa tư tưởng từ Bubble Sort nhưng có một điểm khác biệt lớn chính là số phép so sánh không cố định với những loại dữ liệu khác nhau. Tuy nhiên, phép so sánh vẫn được coi là chiếm ưu thế vì phép gán chỉ hoạt động khi điều kiện so sánh xảy ra. Như vậy, trong phần này của báo cáo, tôi sẽ phân tích độ phức tạp dựa vào số phép so sánh.

Cụ thể, trường hợp tốt nhất xảy ra khi mảng được sắp xếp tăng dần, khi đó thuật toán chỉ cần 1 lần duyệt qua mảng và điều kiện dừng chính là left = right = k = 0. Số lượng phép so sánh là n vì sao khi duyệt từ trái sang phải mà không thực hiện bất kì phép gán nào thì right = k = 0.

Còn với trường hợp tệ nhất khi mảng được sắp xếp giảm dần thì ta luôn phải so sánh hai phần tử bất kỳ từ n phần tử. Như đã giải thích ở phần 3.4.3, ta được số phép so sánh là $\frac{n(n-1)}{2} \in O(n^2)$ nên Shaker Sort có độ phức tạp là $O(n^2)$.

Trường hợp tốt nhất: $O(n)$.

Trường hợp tệ nhất: $O(n^2)$.

Trường hợp trung bình: $O(n^2)$.

3.5.4 Variants and Optimizations - Các biến thể và tối ưu thuật toán

Cũng giống như thuật toán Bubble Sort, Shaker Sort cũng được cải tiến bằng cách thêm biến đánh đánh dấu trước mỗi lần duyệt mảng các phần tử chưa được sắp xếp. Cụ thể, sau khi thực hiện hoán đổi

vị trí của hai phần tử nằm kề nhau, ta thay đổi trạng thái của biến đánh dấu. Sau mỗi lần duyệt mảng, tiến hành kiểm tra biến đánh dấu nếu không thay đổi chứng tỏ mảng đã đúng với thứ tự cần sắp xếp và dừng thuật toán. Tuy nhiên, khác với Bubble Sort, ta cần kiểm tra hai lần biến đánh dấu, mỗi lần kiểm tra là ngay sau khi duyệt mảng từ trái sang phải và từ phải sang trái. Như vậy, việc cải thiện giúp giảm đi đáng kể thời gian chạy và kết thúc sớm hơn so với thuật toán gốc với một số trường hợp đã mô tả.

Điểm mạnh của nó là hoạt động nhanh hơn Bubble Sort ở loại dữ liệu gần được sắp xếp. Ví dụ đối với mảng [3,4,5,6,1] thì Bubble Sort cần 4 lần duyệt mảng để đẩy lần lượt các phần tử về đúng vị trí trong khi Shaker Sort chỉ cần 1 lần duyệt đẩy 1 về đầu.

Tuy nhiên, trong trường hợp mảng có ngẫu nhiên phần tử với thứ tự đảo lộn thì Bubble Sort và Shaker Sort cho thời gian sắp xếp gần tương đương nhau. Vì vậy, có thể nói rằng Shaker Sort ưu thế hơn Bubble Sort trong trường hợp các phần tử trong mảng gần có thứ tự.[7]

3.6 Heap sort

3.6.1 Core concept - Ý tưởng

Heap sort là cải tiến của selection sort, sử dụng cấu trúc dữ liệu Heap, tận dụng kết quả của lần sort trước để tối ưu quá trình tìm phần tử cần thiết. Từ đó khắc phục nhược điểm là thời gian tìm kiếm và tối ưu selection sort.

3.6.2 Step-by-step Explanations - Giải thích từng bước

Cấu trúc dữ liệu Heap: có 2 dạng là max-heap và min-heap, nhưng vì yêu cầu của đồ án là sắp xếp tăng dần nên ta chỉ trình bày max-heap.

Max-heap là cấu trúc dữ liệu cây nhị phân hoàn chỉnh có tính chất: Với mỗi nút, giá trị của nút đó luôn lớn hơn hoặc bằng con của nút đó.

Khi ở dạng mảng, max-heap là một mảng có tính chất: với mỗi phần tử a[i], giá trị của phần tử a[i] luôn lớn hơn hoặc bằng giả trị của phần tử a[i*2+1] và a[i*2+2].

Heap sort trải qua 2 giai đoạn:

Giai đoạn 1: Hiệu chỉnh dãy ban đầu thành heap.

Giai đoạn 2: Sắp xếp dãy số dựa trên heap:

Bước 1: Đưa phần tử lớn nhất về đúng ở cuối dãy:

r = n; Hoán vị(a_1, a_r);

Bước 2: Loại bỏ phần tử lớn nhất ra khỏi heap: r = r-1;

Hiệu chỉnh phần còn lại của dãy thành một heap

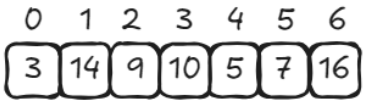
Bước 3: Nếu r>1 (heap còn phần tử): Lặp lại bước 2

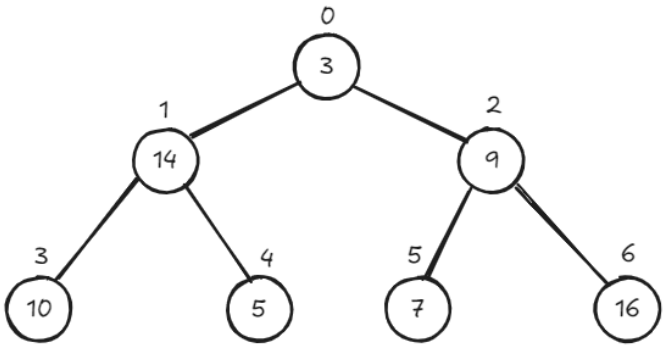
Ngược lại: Dừng[14]

Ví dụ:

Giả sử có mảng a có 7 phần tử [3, 14, 9, 10, 5, 7, 16].

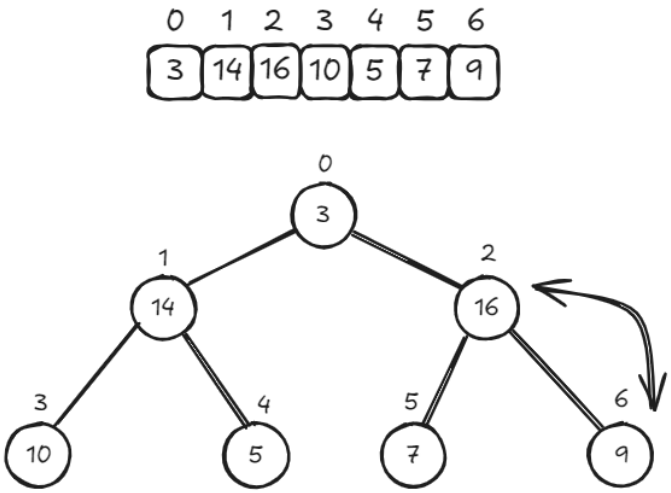
Giai đoạn 1: Hiệu chỉnh dãy ban đầu thành heap





Ta sẽ heapify nửa đầu của mảng từ n/2 – 1 đến 0 (2 $\longrightarrow$ 0) , Ta bắt đầu từ n/2 – 1 vì các phần tử nửa sau n/2 – 1 sẽ không có con. Cách xây dựng heap này được gọi là bottom-up heap construction.

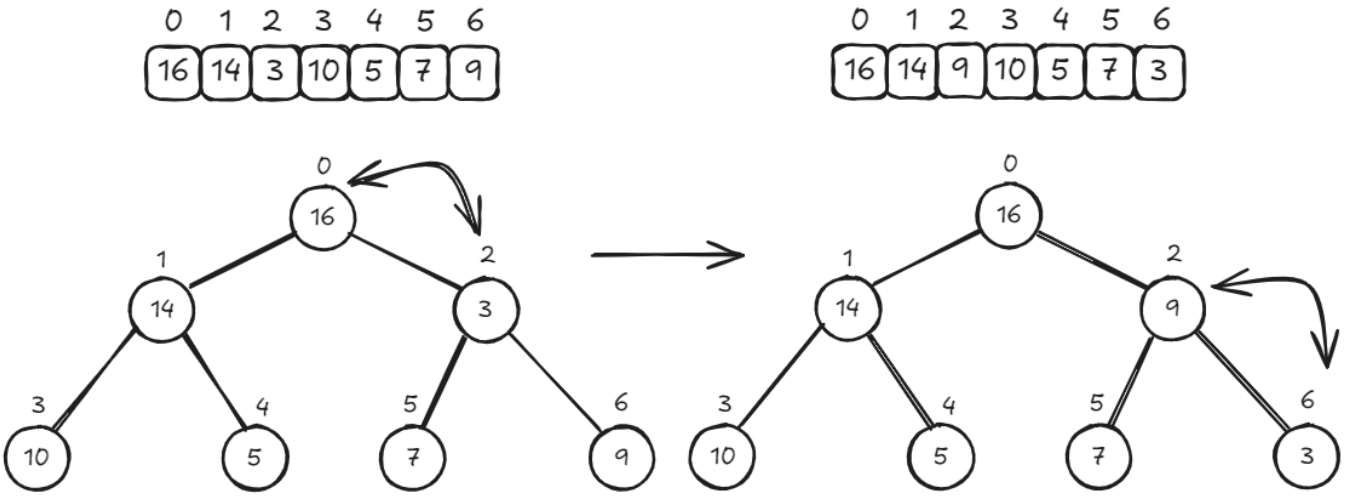
So sánh a[2] = 9 với con của a[2]: a[5]=7, a[6]=16. Ta thấy a[6] lớn nhất $\Longrightarrow$ Hoán vị a[2] và a[6]. Sau đó, kiểm tra heap ở a[6], mà a[6] không có con nên kết thúc.



Tiếp tục so sánh a[1] = 14 với con: a[3] = 10 , a[4] = 5. Ta thấy a[1] đã thỏa điều kiện heap.

Tiếp tục so sánh a[0] = 3 với con: a[1] = 14, a[2] = 16. Ta thấy a[2] lớn nhất $\longrightarrow$ Hoán vị a[0] và a[2];

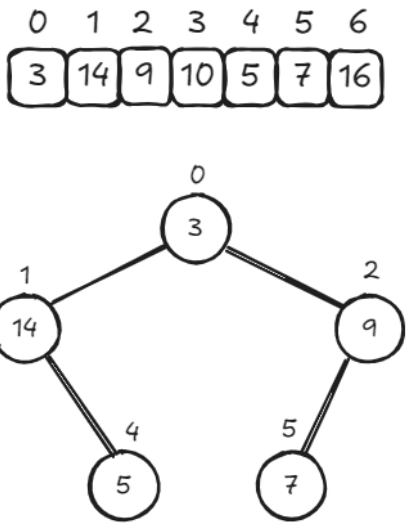
Sau khi hoán vị, kiểm tra heap ở a[2]: so sánh a[2]=3 với a[5]=7, a[6]=9. Ta thấy a[6] lớn nhất $\longrightarrow$ Hoán vị a[2] và a[6]. Sau đó, kiểm tra heap ở a[6], mà a[6] không có con nên kết thúc.



Như vậy, ta đã hiệu chỉnh xong một max-heap.

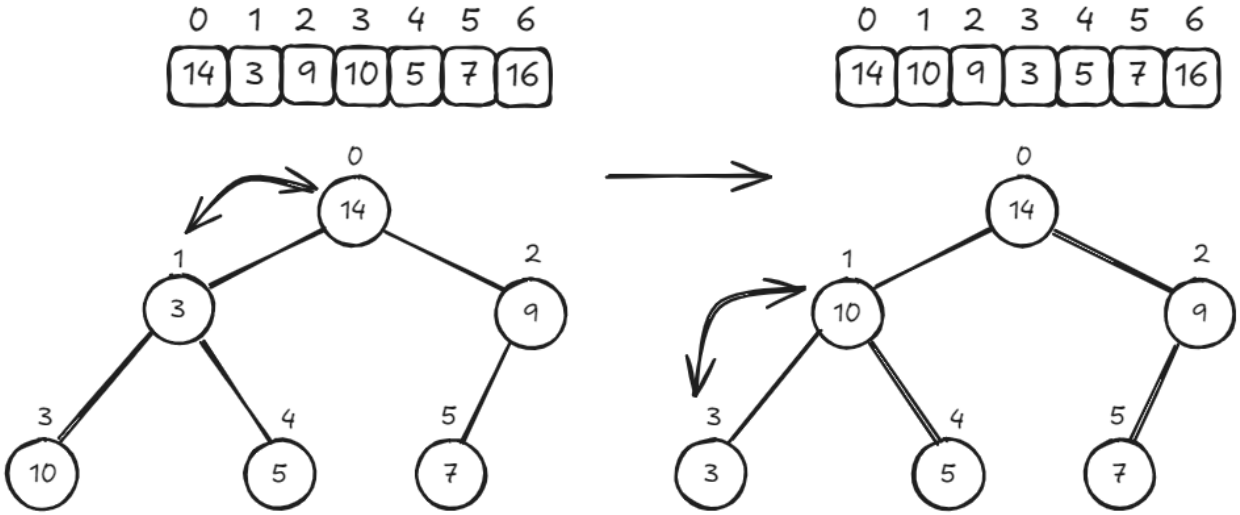
Giai đoạn 2: Trích xuất phần tử lớn nhất

Bước 1: Hoán đổi phần tử gốc với phần tử cuối cùng của heap, giảm kích thước của heap đi 1 và heapify từ gốc.



Bước 2: Hiệu chỉnh phần còn lại thành heap

Heapify từ gốc: so sánh a[0] = 3 với con: a[1] = 14, a[2] = 9. Ta thấy a[1] lớn nhất ⟶ Hoán vị a[0], a[1]. Sau đó kiểm tra heap ở a[1]: so sánh a[1] = 3 với con: a[3] = 10, a[4] = 5. Ta thấy a[3] lớn nhất ⟶ Hoán vị a[1], a[3]. Sau đó, kiểm tra heap ở a[3], mà a[3] không có con nên kết thúc



Heap đã hiệu chỉnh xong

Bước 3: Lặp lại 2 bước trên đến khi heap chỉ còn 1 phần tử

3.6.3 Complexity Analysis - Phân tích mức độ phức tạp

Time complexity:

Độ phức tạp khi xây dựng heap (Giai đoạn 1):

Để xây dựng một heap, ta cần heapify các nút. Ta thấy rằng running time của heapify liên quan chiều cao của cây và cây con là $log_2 n$ (với n là số lượng nút của cây/cây con)

Trong một heap có n nút, số lượng nút có chiều cao h là $\frac{n}{2^{h+1}}$

Ta có thể tính độ phức tập thời gian: $T(n) = \sum_{h=0}^{log_2 n}(\frac{n}{2^{h+1}}).O(h)$

Tổng trên có thể được ước tính: $O(n)$[12]

Vậy độ phức tạp khi xây dựng heap là $O(n)$

Độ phức tạp khi trích xuất phần tử và hiệu chỉnh lại heap (Giai đoạn 2):

Ta cần trích xuất n phần tử, mỗi lần trích xuất thì ta phải heapify nút gốc để hiệu chỉnh lại heap, mà độ phức tạp của mỗi lần heapify nút gốc là $O(log_2 n)$.

Vì thế độ phức tạp khi trích xuất phần tử và hiệu chỉnh lại heap là: $O(n.log_2 n)$

So sánh độ phức tạp khi xây dựng heap ($O(n)$) và độ phức tạp khi trích xuất phần tử và hiệu chỉnh lại heap ($O(n.log_2 n)$)

$\Longrightarrow$ Độ phức tạp thời gian của heap sort là $O(n.log_2 n)$

Độ phức tạp ở mọi trường hợp (tốt nhất, trung bình, tệ nhất) là $O(n.log_2 n)$. Vì ở giai đoạn 1, bất kể trường hợp thì đều cần được hiệu chỉnh thành heap với thời gian là $O(n)$; ở giai đoạn 2, thời gian cần để trích xuất và hiệu chỉnh lại heap là $O(n.log_2 n)$. Vì độ phức tạp của giai đoạn 2 luôn lớn hơn giai đoạn 1 nên độ phức tạp ở mọi trường hợp là $O(n.log_2 n)$

Space complexity:

Vì heap sort chỉ cần một lượng bộ nhớ cố định cho các biến tạm thời trong quá trình heapify và trích xuất nên độ phức tạp về không gian là $O(1)$

3.6.4 Variants and Optimizations - Các biến thể và tối ưu thuật toán

Ternary Heapsort: là một biến thể của Heapsort sử dụng heap ba ngã (ternary heap) thay vì heap nhị phân (binary heap). Trong cấu trúc ternary heap, mỗi phần tử trong heap có ba con thay vì hai.

Thuật toán này phức tạp hơn để lập trình, nhưng lại thực hiện ít hơn một số lượng hằng số các phép hoán đổi và so sánh. Điều này là do trong mỗi bước "sift-down"của một heap ba ngã, cần thực hiện ba phép so

sánh và một phép hoán đổi, trong khi với heap nhị phân chỉ cần hai phép so sánh và một phép hoán đổi. Một lợi thế khác là hai mức trong heap ba ngã bao phủ $3^2 = 9$ phần tử, làm được nhiều công việc hơn với số lần so sánh tương tự như ba mức trong heap nhị phân, vốn chỉ bao phủ $2^3 = 8$ phần tử.
Tuy nhiên, thuật toán này chủ yếu mang tính chất học thuật hoặc được sử dụng như một bài tập thực hành cho sinh viên, vì sự phức tạp tăng thêm không xứng đáng với lợi ích nhỏ thu được.[3]

Smoothsort: là một biến thể của Heapsort, được phát triển bởi Edsger W. Dijkstra vào năm 1981. Tương tự Heapsort, giới hạn trên của Smoothsort là O(n log n).
Tuy nhiên, điểm mạnh của Smoothsort nằm ở khả năng tiến gần đến O(n) khi dữ liệu đầu vào đã được sắp xếp một phần hoặc gần như hoàn toàn. Ngược lại, Heapsort luôn có độ phức tạp thời gian trung bình là O(n log n), bất kể trạng thái sắp xếp ban đầu của dữ liệu.
Mặc dù có ưu điểm này, do sự phức tạp trong triển khai, Smoothsort hiếm khi được sử dụng trên thực tế.[3]

3.7 Merge sort

3.7.1 Core concept - Ý tưởng

Sử dụng thuật toán đệ quy chia để trị (divide and conquer). **Chia (divide) :** Chia đôi mảng thành các phần nhỏ hơn và tiếp tục chia đến khi chỉ còn 1 phần tử. **Trị (conquer):** Sắp xếp các phần đã chia, ta sẽ hợp nhất (merge) các phần đã chia lại và sắp xếp chúng

3.7.2 Step-by-step Explanations - Giải thích từng bước

Bước 1:

Chia đôi mảng ban đầu thành 2 mảng con. Tiếp tục chia đệ quy mỗi mảng con thành các mảng nhỏ hơn cho đến khi mỗi mảng chỉ còn 1 phần tử.

Bước 2:

Sau khi kết thúc quá trình chia đệ quy, hợp nhất 2 mảng con thành 1 mảng lớn hơn đã được sắp xếp;

Trong quá trình hợp nhất:

- Gọi 2 mảng con lần lượt là a, b
- i = 0, j = 0;

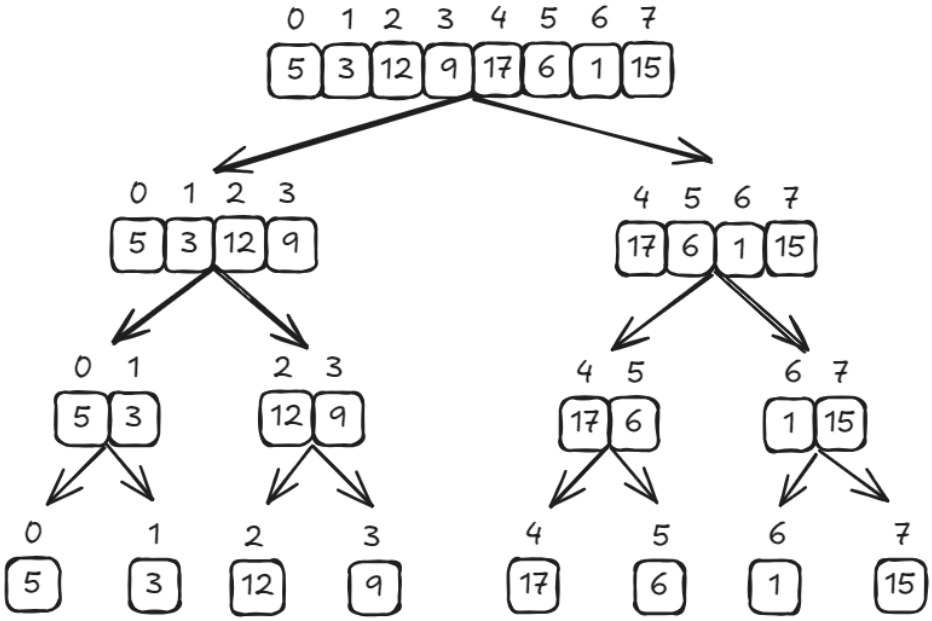
- So sánh 2 a[i] với b[j]: Nếu a[i] < b[j], thêm a[i] vào mảng lớn hơn và i=i+1. Ngược lại, thêm b[j] vào mảng lớn hơn và j=j+1.
- Tiếp tục so sánh và thêm phần tử cho đến khi một trong hai mảng con hết phần tử.
- Khi một mảng con đã được xử lý xong, thêm toàn bộ các phần tử còn lại của mảng kia vào mảng lớn hơn.

Tiếp tục quá trình hợp nhất các mảng, cho đến khi hợp nhất thành mảng có kích thước ban đầu

Ví dụ:

Giả sử ta có mảng a [5, 3, 12, 9, 17, 6, 1, 15]

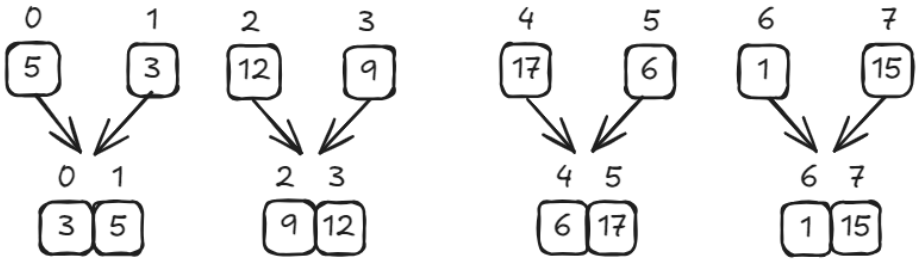
Bước 1: Chia đôi mảng thành các phần nhỏ hơn đến khi không thể chia được nữa



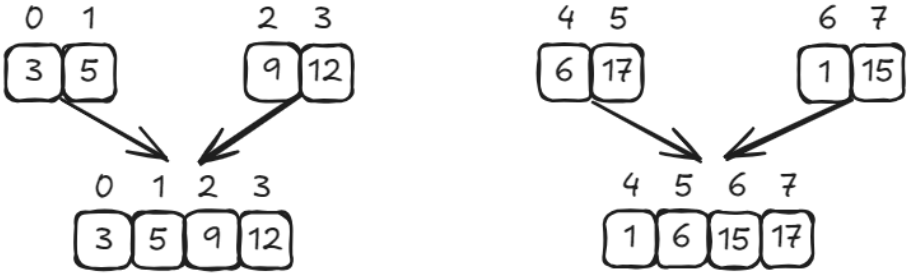
Bước 2: hợp nhất các phần đã chia và sắp xếp

Hợp nhất [5] và [3]: Trong quá trình hợp nhất, tạo mảng phụ trái chứa [5] và mảng phụ phải chứa [3], sau đó so sánh các phần tử giữa 2 mảng phụ để sắp xếp thành [3, 5]

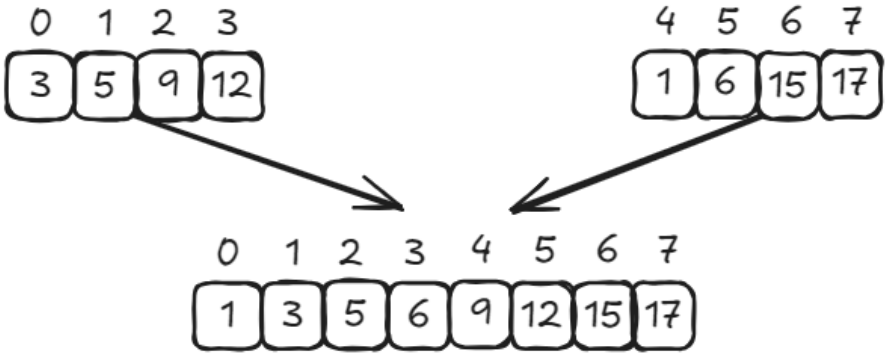
Tương tự, hợp nhất [12] và [9], [17] và [6], [1] và [15] $\longrightarrow$ [9, 12], [6, 17], [1, 15]



Tiếp tục, hợp nhất [3, 5] và [9,12]: Trong quá trình hợp nhất, tạo mảng phụ trái chứa [3, 5] và mảng phụ phải chứa [9, 12], sau đó so sánh các phần tử giữa 2 mảng phụ để sắp xếp thành [3, 5, 9, 12]
Tương tự, hợp nhất [6, 17] và [1,15] $\longrightarrow$ [1, 6, 15, 17]



Tiếp tục, hợp nhất [3, 5, 9, 12] và [1, 6, 15,17]: Trong quá trình hợp nhất, tạo mảng phụ trái chứa [3, 5, 9,12] và mảng phụ phải chứa [1, 6, 15, 17], sau đó so sánh các phần tử giữa 2 mảng phụ để sắp xếp thành [1, 3, 5, 6, 9, 12, 15, 17]



Như vậy ta đã sắp xếp xong

3.7.3 Complexity Analysis - Phân tích mức độ phức tạp

Time complexity

Gọi: T(k) = Thời gian cần để sort k phần tử; M(k) = Thời gian cần để hợp nhất k phần tử

Ta có công thức để tính độ phức tạp: $T(n) = 2.T(\frac{n}{2}) + M(n) = 2.T(\frac{n}{2}) + n.const$;
Vì $\frac{n}{2}$ phần tử này tiếp tục được chia đôi, nên:

- $T(n) = 2.T(\frac{n}{2}) + n.const = 4.T(\frac{n}{4}) + 2.n.const = \cdots = 2^k.T(\frac{n}{2^k}) + k.n.const$
- Tiếp tục chia đôi cho đến khi: $\frac{n}{2^k} = 1 \Longrightarrow k = log_2 n$
- Thay $k = log_2 n$ vào: $T(n) = n.T(1) + n.log_2 n.const = n + n.log_2 n$

Vậy Độ phức tạp thời gian là $O(n.log_2 n)$ [9]
Độ phức tạp ở mọi trường hợp (tốt nhất, trung bình, tệ nhất) đều là $O(n.log_2 n)$, vì dù mảng đã được sắp xếp hay chưa thì thuật toán đều sẽ chia để trị

Space complexity

Bởi vì thuật toán sử dụng mảng phụ có kích thước là n để sắp xếp các phần đã chia đôi
Vì vậy độ phức tạp không gian là $O(n)$ [9]

3.7.4 Variants and Optimizations - Các biến thể và tối ưu thuật toán

k-way merge sort: Chia đệ quy mảng thành k phần thay vì chia đôi như merge sort thông thường. Vì thế, độ phức tạp thời gian sẽ giảm xuống còn $O(n.log_k n)$ nhưng cách thức để hợp nhất các phần lại càng phức tạp khi k càng lớn

Natural merge sort: Là biến thể cải tiến merge sort, thuật toán ban đầu sẽ tìm và chia những đoạn đã được sắp xếp thành các phần riêng biệt, sau đó hợp nhất chúng. Cách này sẽ giúp thuật toán tránh việc chia để trị những đoạn đã sắp xếp. Từ đó ở trường hợp tốt nhất (best case), độ phức tạp của thuật toán này là $O(n)$. [15]
Ví dụ: Mảng [4, 8, 10, 3, 1, 12, 9]
Thuật toán sẽ tìm và chia đoạn đã sắp xếp thành các phần: [4,8,10], [3], [1, 12], [9].
Sau đó tiến hành hợp nhất: [4,8,10], [3], [1, 12], [9] $\longrightarrow$ [3, 4,8,10], [1, 9,12] $\longrightarrow$ [1, 3, 4, 8, 9, 10,12]

3.8 Quick sort

3.8.1 Core concept - Ý tưởng

Thuật toán Quick sort hoạt động trên nguyên lý chia để trị (Devide and Conquer). Ý tưởng của thuật toán lựa chọn phần tử chốt từ mảng cần được sắp xếp (pivot element), phân hoạch mảng thành 2 mảng

con xung quanh phần tử chốt với mảng con chứa các giá trị nhỏ hơn đứng trước, mảng con chứa các giá trị lớn hơn đứng sau.

3.8.2 Step-by-step Explanations - Giải thích từng bước

Thuật toán Quick sort tuân theo các bước:

- **Bước 1** Phân hoạch danh sách (Partition the list)
  - Chọn phần tử ở giữa làm phần tử chốt
  - Tiến hành phân hoạch: chia danh sách thành hai phần. Các phần tử nhỏ hơn phần tử chốt được đưa vào một mảng con ở phía trước, và các phần tử lớn hơn phần tử chốt được đưa vào mảng con ở phía sau.
- **Bước 2** Đệ quy (Recursion):
  - Sau khi phân hoạch, thuật toán sẽ tiếp tục áp dụng Quick Sort đệ quy cho các mảng con (phần trước và phần sau phần tử chốt).
  - Quá trình đệ quy dừng lại khi các mảng con có ít hơn hoặc bằng 1 phần tử, tức là đã đạt được điều kiện sắp xếp.
- **Bước 3** Gộp các mảng con lại với nhau

Xét ví dụ minh họa cụ thể để rõ hơn về phân hoạch (Partition)

Cho mảng ban đầu: $a = [4, 8, 9, 6, 3, 2, 9]$

Chọn phần tử **pivot** là phần tử ở giữa: 6.

Mục tiêu là sắp xếp các phần tử sao cho các phần tử **nhỏ hơn hoặc bằng pivot** nằm bên trái và các phần tử **lớn hơn pivot** nằm bên phải.

Đầu tiên, con trỏ trái left được đặt tại phần tử đầu tiên (4), và con trỏ phải right được đặt tại phần tử cuối cùng (9). Con trỏ trái di chuyển qua phải, lần lượt kiểm tra các phần tử. Phần tử 4 nhỏ hơn pivot nên tiếp tục, đến phần tử 8 lớn hơn pivot thì dừng lại. Tương tự, con trỏ phải di chuyển từ cuối mảng về phía trái, bỏ qua phần tử 9 (lớn hơn pivot) và dừng lại tại phần tử 2 (nhỏ hơn pivot). Lúc này, hoán đổi hai phần tử tại con trỏ trái và phải: hoán đổi 8 và 2, mảng sau khi hoán đổi là $[4, 2, 9, 6, 3, 8, 9]$.

Sau khi hoán đổi, cập nhật con trỏ: con trỏ trái tăng lên left $= 2$ (tại phần tử 9) và con trỏ phải giảm xuống right $= 4$ (tại phần tử 3). Tiếp tục quá trình, con trỏ trái dừng tại 9 (lớn hơn pivot), và con trỏ phải dừng tại 3 (nhỏ hơn pivot). Hoán đổi 9 và 3, mảng trở thành $[4, 2, 3, 6, 9, 8, 9]$. Cập nhật lại vị trí con trỏ: left $= 3$, right $= 3$. Khi left $\geq$ right, quá trình phân hoạch dừng lại.

Kết quả phân hoạch: pivot 6 đã nằm ở đúng vị trí, với các phần tử nhỏ hơn hoặc bằng pivot ở bên trái: $[4, 2, 3]$, và các phần tử lớn hơn pivot ở bên phải: $[9, 8, 9]$.

```
QuickSort(arr[], left, right)
  i = left, j = right
  //pivot la phan tu giua mang
  pivot = arr[(left + right) / 2];
  //Partition
  while (i <= j)
        while (arr[i] < pivot)
              i = i + 1
        while (arr[j] > pivot)
              j = j - 1
        swap(a[i], a[j])
        i = i + 1
        j = j - 1

  //Recursion
  if (left < j) //stop condition
        quickSort(arr, left, j)
  if (i < right) //stop condition
        quickSort(arr, i, right)
```

3.8.3 Complexity Analysis - Phân tích mức độ phức tạp

Độ phức tạp thời gian

Công thức tính thời gian của thuật toán Quick Sort được viết như sau:

$$T(n) = T(k) + T(n - k - 1) + O(n)[8]$$

Trong đó, $T(k)$ và $T(n - k - 1)$ là thời gian chạy hai lời gọi đệ quy đối với 2 mảng con, $O(n)$ là thời gian phân hoạch, và k là vị trí phần tử chốt sau khi phân hoạch.

Trong một lần phân hoạch, $T(x)$ sẽ đưa 1 phần tử vào đúng vị trí của nó trong mảng sắp xếp và gọi lại đệ quy với các mảng con nếu số lượng phần tử của mảng con lớn hơn 1. Thời gian thực thi của thuật toán phụ thuộc vào thời gian thực hiện 2 lời gọi đệ quy $T(k)$ và $T(n-k-1)$.

Dựa trên điều này, có thể suy ra trường hợp tốt phân hoạch chia 2 mảng con có kích thước gần tương đương nhau còn trường hợp xấu là mảng phân hoạch tạo ra 2 mảng con với kích thước chênh lệch lớn hoặc chỉ gọi được 1 lần đệ quy nếu 1 mảng con chỉ có 1 phần tử và mảng còn lại chứa n - 2 phần tử.
Ta cùng xem xét các trường hợp

- **Best Case:**

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Trong trường hợp tốt nhất, mỗi lần phân hoạch sẽ phân ra 2 mảng con có kích thước gần bằng $\frac{n}{2}$. Phân tích kĩ hơn về độ phức tạp thuật toán:

  - Mỗi lần $T(n)$ gọi 2 lời gọi đệ quy.
  - Tại mỗi mức đệ quy, tổng thời gian phân hoạch là $O(n)$.
  - Có $\log_2 n$ cấp gọi như thế.

Do đó, tổng độ phức tạp thời gian được tính như sau:

$$T(n) = O(n) + 2 \cdot O\left(\frac{n}{2}\right) + 4 \cdot O\left(\frac{n}{4}\right) + \dots$$

(với $\log_2 n$ mức). Vì tổng thời gian tại mỗi mức là $O(n)$, và có $\log_2 n$ mức:

$$T(n) = O(n \log_2 n)$$

Do đó, độ phức tạp thời gian trong trường hợp Best Case là:

$$T(n) = O(n \log n)$$

- **Average Case:** Trong trường hợp trung bình, mảng phân hoạch thành 2 mảng con với các phần tử không tuân theo quy luật nào cả.
Phân tích thời gian thực thi:

  - Tổng thời gian thực thi ở mỗi cấp là $O(n)$
  - Mức gọi đệ quy không chênh lệch nhiều so với $\log_2 n$ lần

Tổng thời gian suy ra vẫn tỉ lệ với

$$T(n) = O(n \log_2 n)$$

- **Worst Case:** Trong trường hợp xấu nhất, phần tử chốt chia mảng thành một mảng con có kích thước bằng 0 và một mảng con gần bằng kích thước n - 1 và gọi 1 lời đệ quy. Phương trình đệ quy là:

$$T(n) = T(n-1) + O(n)$$

Giải phương trình đệ quy:

  - Ở mỗi mức đệ quy, cần $O(n)$ để thực hiện bước phân hoạch.
  - Số cấp của cây đệ quy trong trường hợp này là n, vì mỗi lần chỉ xử lý một phần tử.

Tổng thời gian là:

$$T(n) = O(n) + O(n-1) + O(n-2) + \ldots + O(1) = O(n^2)$$

Do đó, độ phức tạp xấu nhất là:

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

Độ phức tạp không gian

Trong thuật toán Quick Sort, bước phân hoạch thực hiện trên chính mảng nên không yêu cầucầu chi phí bộ nhớ thêm. Mỗi lần gọi lời đệ quy Quick Sort đối với mảng con sẽ sử dụng ngăn xếp hàm để lưu trạng thái hiện tại của hàm. Vì thế sau $log_2 n$ cấp gọi đệ quy, với mỗi cấp tốn $O(n)$ chi phí để lưu trữ trạng thái của hàm, thì độ phức tạp không gian là $O(nlogn)$ đối với trường hợp trung bình và tốt nhất.

Xét trường hợp tệ nhất, mỗi lần chỉ gọi 1 lời đệ quy vậy nên sẽ có n - 1 lời gọi đệ quy. Vậy nên độ phức tạp không gian ở trong trường hợp này là $O(n^2)$.

Độ phức tạp không gian theo các trường hợp là:

- **Best Case:** $O(nlogn)$
- **Average Case:** $O(nlogn)$
- **Worst Case:** $O(n^2)$

3.8.4 Variants and Optimizations - Các biến thể và tối ưu thuật toán

Phần tử chốt (pivot element) đóng vai trò quan trong thuật toán Quick Sort. Vì thế việc chọn phần tử chốt có thể ảnh hưởng tới hiệu suất của thuật toán. Ở trên đã giới thiệu về cách phần tử chốt ở giữa mảng, dưới đây sẽ nói về một số biến thể khác của việc chọn phần tử chốt.

- **Vị trí đầu hoặc cuối của mảng**

  Việc chọn phần tử chốt ở đầu hoặc cuối mảng tương tự với chọn vị trí giữa mảng đối với mảng ngẫu nhiên, nhưng đối với mảng đã sắp xếp, hiệu suất sẽ nằm trong Worst Case với độ phức tạp thời gian là

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

- **Vị trí ngẫu nhiên**

  Việc chọn phần tử ngẫu nhiên trong mảng làm giảm khả năng gặp phải trường hợp xấu nhất nhất nhưng nhược điểm của biến thể này là cần thêm bước sinh số ngẫu nhiên và trong một số trường hợp, vị trí được chọn không thật sự phù hợp.

Kết luận:

- Việc lựa chọn phần tử chốt rất quan trọng đối với hiệu suất của thuật toán Quick Sort.
- Chọn phần tử chốt ở giữa mảng thường mang lại hiệu quả tốt và ít gặp trường hợp xấu nhất.
- Chọn phần tử chốt ở đầu hoặc cuối mảng có thể gây ra độ phức tạp `O(n²)` trong trường hợp mảng đã được sắp xếp, do đó không phải là lựa chọn tối ưu.
- Chọn phần tử **ngẫu nhiên** giúp giảm thiểu khả năng gặp phải trường hợp xấu nhất, mặc dù nó yêu cầu bước sinh số ngẫu nhiên, nhưng vẫn là một sự cải tiến hiệu quả cho thuật toán.

3.9 Radix sort

3.9.1 Core concept - Ý tưởng

Radix Sort là một thuật toán sắp xếp nhưng không dựa vào so sánh. Phương hướng tiếp cận của Radix Sort là phân nhóm các phần tử dựa theo vị trí các kí tự trong tất cả phần tử từ trái sang phải, bắt đầu từ chữ số hàng đơn vị, sau đó là hàng chục, hàng trăm...

3.9.2 Step-by-step Explanations - Giải thích từng bước

Thuật toán Radix Sort tuân theo việc phân nhóm các kí tự của các phần tử từ trái sang phải tương đường với các chữ số hàng đơn vị, hàng chục, hàng trăm, hàng nghìn, ... theo thứ tự

Các bước thực hiện thuật toán Radix Sort

- **Bước 1** Tìm phần tử có giá trị lớn nhất
- **Bước 2** Phân nhóm các phần tử dựa theo kí tự cuối tức là chữ số hàng đơn vị và sắp xếp lại. Ví dụ 31, 41 thì sẽ được phân nhóm vào nhóm 1 vì chữ số hàng đơn vị là 1.
- **Bước 3** Thực hiện lặp lại bước 2 với kí tự kế tiếp từ phải sang trái theo thứ tự là hàng chục, hàng trăm... cho tới kí tự trái nhất của phần tử lớn nhất.

Xét ví dụ minh họa cụ thể để rõ hơn về thuật toán Radix Sort Cho mảng ban đầu a = [2, 98, 34, 1, 4, 1313, 643, 236, 241, 3512, 56]
Mục tiêu là sắp xếp các phần tử của mảng theo thứ tự không giảm từ trái sang phải
Số lớn nhất trong mảng là 3512, có 4 chữ số. Vì vậy, thuật toán Radix Sort sẽ thực hiện 4 vòng lặp (tương ứng với các chữ số từ phải sang trái).

Vòng lặp 1: Phân nhóm theo chữ số đơn vị Phân nhóm các phần tử theo chữ số đơn vị (chữ số cuối cùng của mỗi phần tử):

$$1 : [1, 241]$$

$$2 : [2, 3512]$$

$$3 : [1313, 643]$$

$$4 : [4, 34]$$

$$6 : [236, 56]$$

$$8 : [98]$$

Sau khi phân nhóm xong, mảng được sắp xếp lại:

$$[1, 241, 2, 3512, 1313, 643, 4, 34, 236, 56, 98]$$

Vòng lặp 2: Phân nhóm theo chữ số hàng chục

Phân nhóm các phần tử theo chữ số hàng chục:

$$0 : [1, 2, 4]$$

$$1 : [3512, 1313]$$

$$3 : [34, 236]$$

$$4 : [241, 643]$$

$$5 : [56]$$

$$9 : [98]$$

Sau khi phân nhóm xong, mảng được sắp xếp lại:

$$[1, 2, 44, 3512, 1313, 34, 236, 241, 643, 56, 98]$$

Vòng lặp 3: Phân nhóm theo chữ số hàng trăm Phân nhóm các phần tử theo chữ số hàng trăm:

$$0 : [1, 2, 4, 34, 56, 98]$$

$$2 : [236, 241]$$

$$3 : [1313]$$

$$5 : [3512]$$

$$6 : [643]$$

Sau khi phân nhóm xong, mảng được sắp xếp lại:

$$[1, 2, 4, 34, 56, 98, 236, 241, 1313, 3512, 643]$$

Vòng lặp 4: Phân nhóm theo chữ số hàng nghìn Phân nhóm các phần tử theo chữ số hàng nghìn:

$$0 : [1, 2, 4, 34, 56, 98, 236, 241, 643]$$

$$1 : [1313]$$

$$3 : [3512]$$

Sau khi phân nhóm xong, mảng được sắp xếp lại:

$$[1, 2, 4, 34, 56, 98, 236, 241, 643, 1313, 3512]$$

3.9.3 Complexity Analysis - Phân tích mức độ phức tạp

Độ phức tạp thời gian

Công thức tính thời gian của thuật toán Radix Sort có thể được viết như sau:

$$T(n) = k * O(n)$$

Trong đó:

- k là vòng lặp của việc phân nhóm, tức là số chữ số của phần tử lớn nhất ở trong mảng
- $O(n)$ là thời gian thực hiện phân nhóm trong mỗi phần tử trong mỗi vòng lặp

Từ công thức độ phức tạp thời gian trên, có thể nhận xét độ phức tạp thời gian của Radix Sort là $O(n)$ với k không quá lớn.
Còn đối với mảng chứa các số cực lớn chứa nhiều chữ số thì độ phức tạp là $O(k * n)$.
Xét trong một số trường hợp nhất định, mảng tồn tại với phần lớn các phần tử có số chữ số tương đương nhau nhưng tồn tại một vài phần tử có giá trị cực lớn, chứa nhiều chữ số thì đây là một sự bất lợi đối với Radix Sort

Độ phức tạp không gian

Thuật toán sắp xếp Radix Sort sử dụng không gian để lưu trữ các phần tử sau khi phân nhóm, vậy nên độ phức tạp không gian là $O(n)$

3.9.4 Variants and Optimizations - Các biến thể và tối ưu thuật toán

Đối với thuật toán Radix Sort như đã giải thích ở trên, Radix Sort không thể áp dụng đối với mảng chứa phần tử âm. Để áp dụng thuật toán trên vào mảng chứa các phần tử âm, ta cần một số điều chỉnh:

- Tách thành 2 mảng con
- Áp dụng Radix Sort với mảng phần tử dương
- Đối với mảng âm, chuyển dấu tất cả phần tử thành phần tử dương, áp dụng Radix Sort với mảng con tách dấu, và trả lại dấu cho mảng và đảo nghịch mảng

- Gộp 2 mảng con lại với nhau

Tính toán về độ phức tạp thời gian, việc thực hiện Radix Sort trên 2 mảng con sẽ tốn thời gian lần lượt có công thức là $T(n-x)$ và $T(x)$ với công thức như đã đề cập ở trên nên tổng thời gian sẽ là $T(n-x) + T(x)$.
Vì thế, với hầu hết các trường hợp với số lượng chữ số của phần tử có trị tuyệt đối lớn nhất trong 2 mảng con không chênh lệch quá lớn, độ phức tạp thời gian của thuật toán sẽ gần tương đương với $O(n)$ đối với số lượng vòng lặp xử lí không quá lớn.
Còn khi số lượng vòng lặp xử lí đối với mảng chứa phần tử có giá trị cực kì lớn thì sẽ có độ phức tạp thời gian là $O(k * n)$ với k là số lượng vòng lặp cần tính toán.

3.10 Counting sort

3.10.1 Core concept - Ý tưởng

Counting Sort là thuật toán sắp xếp không dựa trên so sánh. Thuật toán này đặc biệt hiệu quả khi phạm vi giá trị đầu vào nhỏ so với số lượng phần tử cần sắp xếp. Ý tưởng cơ bản đằng sau Counting Sort là đếm tần suất của từng phần tử riêng biệt trong mảng đầu vào và sử dụng thông tin đó để đặt các phần tử vào đúng vị trí đã sắp xếp của chúng.[2]

3.10.2 Step-by-step Explanations - Giải thích từng bước

Thuật toán Counting Sort tuân theo các bước:

- **Bước 1:** Tìm phần tử lớn nhất (`maxElement`) trong mảng.
  - *Mục đích:* Xác định kích thước của mảng đếm `CountIndex`.
- **Bước 2:** Khởi tạo mảng `CountIndex` có kích thước `maxElement + 1`, ban đầu tất cả các giá trị đều bằng 0.
  - *Mục đích:* Lưu số lần xuất hiện của mỗi giá trị trong mảng ban đầu.
- **Bước 3:** Duyệt qua từng phần tử trong mảng và cập nhật giá trị trong `CountIndex`:

  countIndex[a[i]] = countIndex[a[i]] + 1

  - *Mục đích:* Đếm số lần xuất hiện của từng giá trị.
- **Bước 4:** Sử dụng mảng `CountIndex` để sắp xếp lại các phần tử trong mảng ban đầu:

- Khởi tạo `j = 0` để làm chỉ số của mảng gốc.
- Duyệt qua từng giá trị `i` trong `CountIndex` từ 0 đến `maxElement`:
  - Khi `countIndex[i] > 0`:
    - Gán `a[j] = i`.
    - Tăng `j` lên 1.
    - Giảm `countIndex[i]` đi 1.
- *Mục đích:* Đưa các giá trị trở về đúng vị trí trong mảng đã sắp xếp.

Xét ví dụ minh họa về Counting Sort

Cho mảng đầu vào:

$$\{1, 3, 6, 3, 13, 1, 4\}$$

- **Bước 1: Tìm phần tử lớn nhất**

$$\text{maxElement} = 13$$

- **Bước 2: Khởi tạo mảng đếm** `CountIndex` Khởi tạo mảng `CountIndex` có kích thước maxElement+ $1 = 14$, ban đầu tất cả các giá trị bằng 0:

$$\texttt{CountIndex} = \{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$$

- **Bước 3: Đếm số lần xuất hiện của các phần tử trong mảng đầu vào** Duyệt qua mảng đầu vào và cập nhật `CountIndex`:

$$\texttt{CountIndex}[1] = 2,$$
$$\texttt{CountIndex}[3] = 2,$$
$$\texttt{CountIndex}[4] = 1,$$
$$\texttt{CountIndex}[6] = 1,$$
$$\texttt{CountIndex}[13] = 1$$

  Mảng `CountIndex` sau khi đếm:

$$\texttt{CountIndex} = \{0, 2, 0, 2, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1\}$$

- **Bước 4: Xây dựng mảng đã sắp xếp từ** `CountIndex`

– Khởi tạo `j = 0`.

– Duyệt qua `CountIndex` từ 0 đến maxElement:

1. Với $i = 0$: `CountIndex`[0] = 0, không thêm phần tử nào.
2. Với $i = 1$: `CountIndex`[1] = 2, thêm 1 vào mảng 2 lần:

$$\text{Mảng hiện tại: } [1, 1, \_, \_, \_, \_, \_]$$

`j` tăng từ 0 lên 2.

3. Với $i = 2$: `CountIndex`[2] = 0, không thêm phần tử nào.
4. Với $i = 3$: `CountIndex`[3] = 2, thêm 3 vào mảng 2 lần:

$$\text{Mảng hiện tại: } [1, 1, 3, 3, \_, \_, \_]$$

`j` tăng từ 2 lên 4.

5. Với $i = 4$: `CountIndex`[4] = 1, thêm 4 vào mảng 1 lần:

$$\text{Mảng hiện tại: } [1, 1, 3, 3, 4, \_, \_]$$

`j` tăng từ 4 lên 5.

6. Với $i = 5$: `CountIndex`[5] = 0, không thêm phần tử nào.
7. Với $i = 6$: `CountIndex`[6] = 1, thêm 6 vào mảng 1 lần:

$$\text{Mảng hiện tại: } [1, 1, 3, 3, 4, 6, \_]$$

`j` tăng từ 5 lên 6.

8. Với $i = 7$ đến $i = 12$: `CountIndex`[i] = 0, không thêm phần tử nào.
9. Với $i = 13$: `CountIndex`[13] = 1, thêm 13 vào mảng 1 lần:

$$\text{Mảng hiện tại: } [1, 1, 3, 3, 4, 6, 13]$$

`j` tăng từ 6 lên 7.

– Mảng đã sắp xếp cuối cùng là:

$$\{1, 1, 3, 3, 4, 6, 13\}$$

3.10.3 Complexity Analysis - Phân tích mức độ phức tạp

Độ phức tạp thời gian

Thuật toán Counting Sort duyệt qua tất cả các phần tử quả mảng để đếm số lần xuất hiện của các phần tử nên tốn chi phí thời gian là $O(n)$. Sau đó duyệt từ 0 đến maxElement của mảng để thực hiện sắp xếp sẽ tốn chi phí thời gian là $O(m)$ với m là giá trị phần tử lớn nhất trong mảng.

Vậy nên chi phí thời gian là $O(n+m)$ với n là số lượng phần tử, m là giá trị lớn nhất của các phần tử trong mảng.

Từ đó ta có thể phân tích ra độ phức tạp thời gian trong các trường hợp:

Best case: $O(n+m)$

Average case: $O(n+m)$

Worst case: $O(n+m)$

Trong 3 trường hợp, độ phức tạp thời gian là như nhau

Độ phức tạp không gian

Với việc tạo mảng để đếm số lần xuất hiện của các phần tử trong mảng thì chi phí bộ nhớ là $O(m+1)$ với m là giá trị phần tử lớn nhất trong mảng.

3.10.4 Variants and Optimizations - Các biến thể và tối ưu thuật toán

Thuật toán Counting Sort không thể thực hiện đối với mảng số nguyên chứa giá trị âm. Thuật toán cần vài sự điều chỉnh để áp dụng được với mảng trên.

1. **Chia mảng thành hai mảng con**:
   - Mảng thứ nhất chứa các giá trị dương (bao gồm cả 0).
   - Mảng thứ hai chứa các giá trị âm.
2. **Áp dụng Counting Sort cho mảng chứa giá trị dương**:
   - Sử dụng thuật toán Counting Sort thông thường vì Counting Sort được thiết kế để hoạt động với các số nguyên không âm.
3. **Xử lý mảng chứa giá trị âm**:
   - Đổi dấu tất cả các giá trị âm để biến chúng thành số dương.

- Áp dụng Counting Sort để sắp xếp các giá trị này theo thứ tự tăng dần.
- Đổi dấu lại kết quả sau khi sắp xếp để trả về các giá trị âm theo thứ tự giảm dần.

4. **Gộp kết quả**:
   - Ghép mảng con chứa các giá trị âm (sắp xếp giảm dần) với mảng con chứa các giá trị dương (sắp xếp tăng dần) để tạo thành mảng đã sắp xếp hoàn chỉnh.

3.11 Binary Insertion sort

3.11.1 Core concept - Ý tưởng

Như đã đề cập ở mục Các biến thể và tối ưu thuật toán của Insertion sort, thay vì dùng tìm kiếm tuần tự, thì Binary Insertion sort dùng thuật toán Binary Search - tìm kiếm nhị phân, nhằm tìm vị trí cần thiết để chèn phần tử mới (từ dãy chưa được sắp xếp) vào dãy giá trị đã được sắp xếp trước đó.

3.11.2 Step-by-step Explanations - Giải thích từng bước

Bước 1: $i = 1$; // giả sử có đoạn $a[0]$ đã được sắp

Bước 2: Gán $x = a[i]$; Tìm vị trí pos thích hợp trong đoạn $a[1]$ đến $a[i-1]$ để chèn $a[i]$ bằng thuật toán **Binary Search**.

Bước 3: Dời chỗ các phần tử từ $x = a[pos]$ đến $x = a[i-1]$ sang phải một vị trí để dành chỗ cho $x = a[i]$.

Bước 4: $a[pos] = x$; // có đoạn $a[1]..a[i]$ đã được sắp.

Bước 5: $i = i + 1$;

Nếu $i < n$: Lặp lại Bước 2.

Ngược lại: Dừng.

Mã giả : [14]

Hàm Binay Insertion sort.

Input: mảng a có n phần tử chưa có thứ tự không giảm

Output: toàn bộ phần tử của mảng a đã được sắp xếp theo thứ tự không giảm.

```
for i = 1 to n - 1 do
    x = a[i]
```

pos = **BinarySearch** (a, 0, i - 1, x) //trả về vị trí cần chèn phần tử
j = i - 1
while $j >= pos$
    $a[j + 1] = a[j]$ // Dịch chuyển các phần tử sang phải.
    j = j - 1
$a[j + 1] = x$ // có đoạn $a[1]..a[i]$ đã được sắp

Hàm Binay Search.

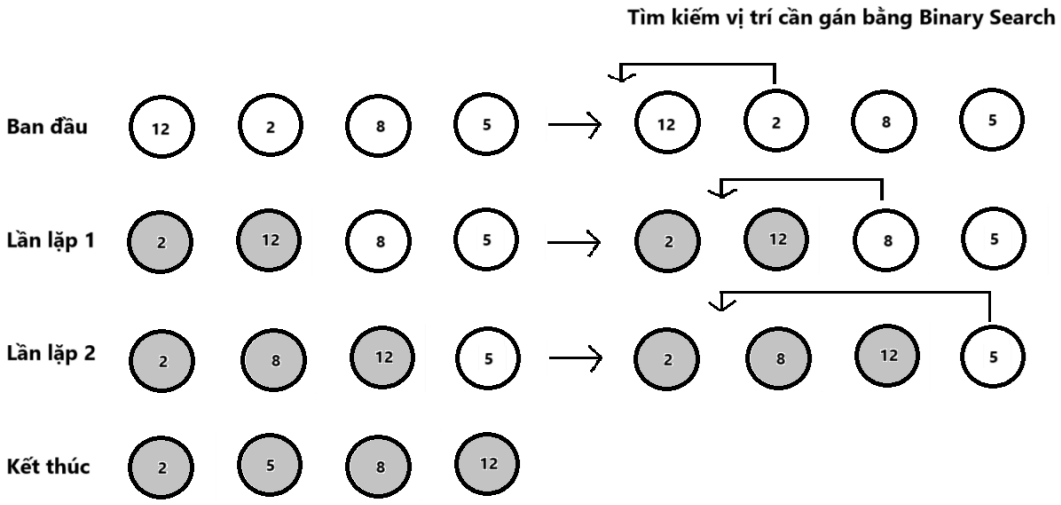
Input: mảng a, khoảng cần tìm kiếm trong mảng a, giá trị dựa vào để tìm kiếm.

Output: Vị trị phù hợp để chèn phần tử.

while left <= right do
    mid = (left + right) / 2
    if x < a[mid] then
        right = mid - 1
    else
        left = mid + 1
return left // Vị trí cần chèn (sau vòng lặp thì left là vị trí chính xác)

Ví dụ : Cho dãy số a:

$$12, 2, 8, 5$$



Hình 12: Mô tả quá trình sắp xếp chèn nhị phân

.

3.11.3 Complexity Analysis - Phân tích mức độ phức tạp

- **Độ phức tạp thời gian**

Tại mỗi vòng lặp ngoài, để chèn phần tử thứ i ($1 \leq i \leq n-1$) vào vị trí phù hợp trong dãy đã được sắp xếp trước đó thì giải thuật Binary Insertion sort cần khoảng $O(log_2 i)$ lần lặp để có thể tìm ra vị trí phù hợp. Tuy nhiên, khi chèn phần tử thứ i, thì cần phải dời tất cả phần tử có vị trí pos đến i-1 sang phải một ô. Hệ quả là thuật toán này sẽ mất nhiều nhất i lần lặp trong trường hợp tệ nhất (khi mà pos là vị trí đầu tiên trong dãy đã sắp xếp).[10]

- **Trường hợp tệ nhất (worst case)**

Đối với mỗi phần tử thứ i (với $i = 1, 2, \ldots, N-1$), thuật toán sử dụng tìm kiếm nhị phân trên dãy con đã sắp xếp gồm i phần tử có độ phức tạp $O(\log_2 i)$.

Sau khi tìm được vị trí cần chèn, ta cần dịch chuyển các phần tử từ vị trí chèn (pos) đến cuối dãy con $i-1$ để tạo khoảng trống. Trong trường hợp xấu nhất, giống như Insertion sort khi mà dãy có thứ tự không tăng thì ta phải di chuyển i phần tử.

Vậy độ phức tạp thời gian cho mỗi lần chèn là:

$$T_i worst = O(\log i) + O(i)$$

Tổng thời gian cho các lần chèn sẽ là:

$$T_{worst} = \sum_{i=1}^{n-1} T_i = O(n.log_2 n) + O(n.(n-1/2) \in O(n^2)$$

Vậy chúng tôi tìm ra được độ phức tạp thời gian tại trường hợp tệ nhất của Binary Insertion sort là $O(n^2)$

- **Trường hợp tốt nhất (best case)**

Dễ thấy ở trường hợp tốt nhất của Binary Insertion sort cũng tương tự như Insertion sort. Khi dãy đã có thứ tự không giảm thì chi phí để chèn phần tử thứ i sẽ là $O(1)$. Khi đó độ phức tạp thời gian cho mỗi lần chèn là:

$$T_i best = O(\log i) + O(i)$$

Tổng thời gian cho các lần chèn sẽ là:

$$T_{best} = \sum_{i=1}^{n-1} T_i best = O(n.log_2 n) + O(n-1) \in O(n.log_2 n)$$

Ở trường hợp tốt nhất, chúng tôi dễ dàng so sánh được độ phức tạp thời gian của Binary Insertion sort không như Insertion sort là $O(n)$ do một phần độ phức tạp thời gian của Binary Insertion sort có ảnh hưởng bởi thuật toán tìm kiếm nhị phân dẫn đến nó có độ phức tạp lớn hơn bản gốc là $O(n.log_2 n)$

- **Trường hợp trung bình (average case)**

Trong trường hợp trung bình, giả sử rằng các phần tử trong mảng ban đầu được sắp xếp ngẫu nhiên, không có thứ tự cụ thể. Khi đó vị trí cần chèn pos thường nằm ở giữa dãy đã được sắp xếp, tức là trung bình sẽ cần dời sang phải khoảng $i/2$ phần tử để tạo ra khoảng trống. Có nghĩa độ phức tạp thời gian cho mỗi lần chèn sẽ là [10]:

$$T_i avr = O(\log i) + O(i/2)$$

Tổng thời gian cho các lần chèn sẽ là:

$$T_{avr} = \sum_{i=1}^{n-1} T_i avr = O(n.log_2 n) + O(n.(n-1)/4) \in O(n^2)$$

Trong trường hợp trung bình, độ phức tạp thời gian để chèn mỗi phần tử giảm so với trường hợp xấu nhất. Mặc dù đã sử dụng tìm kiếm nhị phân để giảm số lần so sánh và thời gian thực thi của thuật toán nhưng độ phức tạp thời gian tổng thể của Binary Insertion sort vẫn là $O(n^2)$ do ảnh hưởng bởi chi phí dịch chuyển khi số phần tử tăng. Chúng tôi đi đến kết luận độ phức tạp thời gian trung bình của giải thuật này là: $O(n^2)$

• **Độ phức tạp không gian**

Tương tự như bản gốc của nó là Insertion sort, Binary Insertion sort cũng là một thuật toán sắp xếp tại chỗ (in-place), nó thao tác trực tiếp trên dãy các giá trị mà không cần sử dụng bộ nhớ ngoài hay bất kỳ cấu trúc dữ liệu bổ sung nào. Nhờ vậy, độ phức tạp không gian của nó là $O(1)$. Chi phí không gian sẽ là $O(n.log_2 n)$ khi sử dụng dạng đệ quy của Binary Search, dẫn đến thuật toán này cần khoảng $O(n.log_2 n)$ lời gọi đệ quy. [10]

3.11.4 Variants and Optimizations - Các biến thể và tối ưu thuật toán

Ngoài biến thể sử dụng tìm kiếm nhị phân bằng đệ quy cho Binary Insertion sort kể trên. Thì còn có một biến thể khác sử dụng thuật toán Shell sort (kết hợp với việc chèn nhị phân) để giảm dần khoảng cách h giữa các phần tử cần sắp xếp. Sau khi chọn dãy số theo khoảng cách, ta dùng Binary Bnsertion sort để sắp xếp các phần tử trên dãy đó.

3.12 Flash sort

Flashsort là một thuật toán sắp xếp phân phối cho thấy độ phức tạp tính toán tuyến tính O(n) cho các tập dữ liệu phân phối đều và yêu cầu bộ nhớ bổ sung tương đối ít. Công trình gốc được xuất bản vào năm 1998 bởi Karl-Dietrich Neubert.[1]

3.12.1 Core concept - Ý tưởng

Thuật toán Flash Sort hoạt động dựa trên nguyên lý chia tập dữ liệu thành nhiều lớp nhất định dựa trên công thức tính chỉ số lớp. Sau khi phân lớp, các phần trong mỗi lớp được sắp xếp tại chỗ để đảm bảo chúng đã nằm đúng vị trí.

3.12.2 Step-by-step Explanations - Giải thích từng bước

Trong báo cáo này, tôi lựa chọn triển khai thuật toán Flash Sort để sắp xếp mảng dữ liệu tăng dần dựa trên phân lớp và hoán đổi vị trí trực tiếp.

Bước 1: Xác định giá trị nhỏ nhất và lớn nhất của dữ liệu.

Bước 2: Chia dữ liệu đầu vào thành k lớp bằng công thức chỉ số lớp: index = $\left\lfloor \frac{(a[i]-\text{minValue})\cdot(k-1)}{\text{maxValue}-\text{minValue}} \right\rfloor$.

Bước 3: Sắp xếp lại các phần tử trong mảng bằng cách thực hiện hoán đổi trực tiếp các phần tử về đúng lớp của nó.

Bước 4: Sử dụng thuật toán sắp xếp khác để hoàn tất thứ tự cho từng lớp.

3.12.3 Complexity Analysis - Phân tích mức độ phức tạp

Thuật toán Flash Sort thực hiện nhiều công việc riêng biệt nhau nên khi phân tích độ phức tạp của thuật toán này ta cần chia thành nhiều phần nhằm giúp người đọc dễ dàng hình dung và hiểu rõ về cách hoạt động của nó.

Việc xác định giá trị lớn nhất và giá trị nhỏ nhất cần duyệt qua mảng dữ liệu một lần nên có độ phức tạp là O(n).

Việc tính chỉ số lớp cho từng phần tử và cập nhật mảng đếm số lượng phần tử trong mỗi lớp cũng tốn O(n) vì mỗi phần tử được tính toán và gán vào lớp duy nhất 1 lần.

Việc hoán đổi phần tử để đưa nó vào đúng vị trí lớp sẽ có độ phức tạp là O(n) nếu dữ liệu phân bố đều. Tuy nhiên, nếu dữ liệu phân bố không đồng đều thì số lần hoán đổi tăng lên đáng kể. Cụ thể, trong trường hợp nhiều phần tử rơi vào cùng một lớp thì việc hoán đổi trực tiếp sẽ trở nên phức tạp hơn. Do đó, độ phức tạp có thể lên đến $O(n^2)$.

Sau khi hoán đổi các phần tử vào đúng phân lớp thì tại mỗi lớp này ta dùng thuật toán Insertion Sort. Trung bình mỗi lớp chứa $\frac{n}{k}$ phần tử nên đối với mỗi phân lớp độ phức tạp là $O(\frac{n}{k}log(\frac{n}{k}))$. Khi k đủ lớn thì $log(\frac{n}{k})$ rất nhỏ nên ta hoàn toàn có thể xấp xỉ độ phức tạp lúc này thành O(n). Vì khi k là số lớp nên khi k tăng thì kích thước trung bình của mỗi lớp giảm (lý tưởng nhất là 1 phần tử trong một lớp), điều này làm giảm thời gian cần để sắp xếp các phần tử trong lớp.

Tóm lại, Flash Sort là một thuật toán có tính ứng dụng cao khi độ phức tạp của nó chỉ là O(n) cho trường hợp tốt nhất và trung bình còn trường hợp xấu nhất là $O(n^2)$.

3.12.4 Variants and Optimizations - Các biến thể và tối ưu thuật toán

Flash Sort là một thuật toán được tối ưu hóa cho dữ liệu phân bố đồng đều vậy nên nó cũng không có nhiều biến thể như các thuật toán khác tuy nhiên trong thực tế vẫn có một số cải tiến để thuật toán hoạt động tốt hơn.

Flash Sort sủ dụng phân lớp động: đối với Flash Sort thông thường thì số lớp k thường được xác định trước khi chạy thuật toán, đó cũng chính là điểm yếu khiến số lớp không được linh động. Trong biến

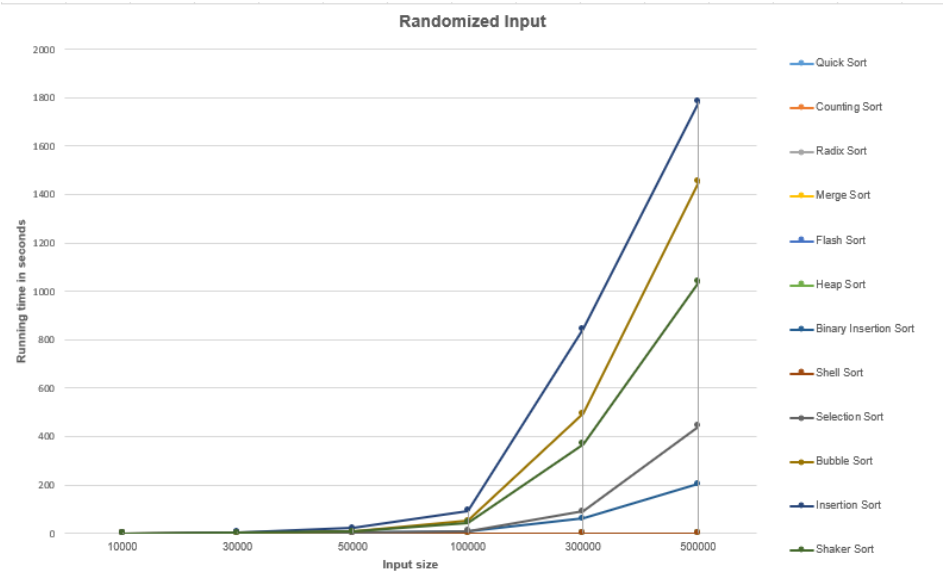
thể này, k được điều chỉnh dựa trên đặc điểm phân bố của dữ liệu trong quá trình phân lớp. Cải tiến này giúp giảm số lần hoán đổi trong trường hợp dữ liệu không đều ngoài ra còn giúp tăng hiệu quả trong việc phân loại các phần tử về đúng lớp.

4 Kết quả thí nghiệm và nhận xét

Nhóm đã tiến hành thực nghiệm chạy 12 thuật toán trên các dữ liệu đầu vào với 4 kiểu dữ liệu là: ngẫu nhiên, đã được sắp xếp, đảo ngược và gần được sắp xếp trên các kích thước: 10000, 30000, 50000, 100000, 300000, 500000 Dưới đây là kết quả của thực nghiệm

Dữ liệu đầu vào ngẫu nhiên (Random Input)

Random Input												
Datasize	**10,000**		**30,000**		**50,000**		**100,000**		**300,000**		**500,000**	
Resulting statics	**Running time**	**Comparision**	**Running time**	**Comparision**	**Running time**	**Comparision**	**Running time**	**Comparision**	**Running time**	**Comparision**	**Running time**	**Comparision**
Quick Sort	0.0021	172099	0.0024	565196	0.013	973260	0.0191	2052616	0.0552	6776345	0.0877	11896723
Counting Sort	0.0003	0	0.0011	0	0.0007	0	0.0012	0	0.011	0	0.0084	0
Radix Sort	0.001	9999	0.0011	29999	0.004	49999	0.0093	99999	0.0231	299999	0.0521	499999
Merge Sort	0.0032	578959	0.0056	1923407	0.0193	3360645	0.0391	7120793	0.1231	23243145	0.4833	40137819
Flash Sort	0.0002	6064	0.0007	18174	0.0011	30591	0.0025	68781	0.0097	267234	0.0183	467232
Heap Sort	0.0009	638218	0.0055	2151058	0.0095	3773290	0.0212	8044279	0.0738	26490814	0.2617	45972691
Binary Insertion Sort	0.0345	118992400	0.4047	404706	1.185	711346	6.5695	1522713	62.469	6776345	202.43	11896723
Shell Sort	0.0019	275842	0.0047	957238	0.0174	1941102	0.0918	4088379	0.1382	14403543	0.2486	27783730
Selection Sort	0.0873	49995000	0.8211	449985000	2.2747	1249975000	8.8948	4999950000	91.613	44999850000	442.41	124999750000
Bubble Sort	0.2394	75013265	2.6414	676665616	9.5329	1873514473	50.656	7504234040	493.98	67468531646	1453.2	187518832593
Insertion Sort	0.7012	24967187	5.6246	225571984	21.5478	623527878	78.645	2509044573	673.52	22526665610	1783.3	62515537260
Shaker Sort	0.1796	66932268	1.9799	605013483	8.0317	1663551461	43.291	6681076593	369.61	59948700648	1038.9	166775879065



Đồ thị thể hiện thời gian thực thi đối với dữ liệu ngẫu nhiên (randomized input) cho thấy sự khác biệt rõ rệt trong hiệu năng của 12 thuật toán sắp xếp. Counting sort và Radix sort nổi bật với

hiệu suất vượt trội, đặc biệt khi kích thước dữ liệu tăng lên. Là các thuật toán không dựa trên so sánh (non-comparison-based), chúng tận dụng các phương pháp như đếm tần suất hoặc xử lý theo chữ số, giúp giảm đáng kể thời gian xử lý xuống gần mức tuyến tính. Điều này giải thích tại sao chúng duy trì hiệu suất ổn định ngay cả với bộ dữ liệu lớn.

Flash sort cũng là một thuật toán có hiệu năng đáng chú ý. Với chiến lược phân vùng dữ liệu và sắp xếp cục bộ, nó đạt hiệu quả cao khi dữ liệu được phân bố đồng đều. Trên đồ thị, Flash sort nằm trong nhóm các thuật toán có thời gian chạy thấp nhất, chỉ chậm hơn một chút so với Counting sort và Radix sort. Tuy nhiên, hiệu năng của nó có thể bị ảnh hưởng nếu dữ liệu không được phân bố đều, do các bucket có thể mất cân bằng.

Quick sort cho thấy hiệu năng ấn tượng, nằm trong nhóm các thuật toán nhanh nhất trên đồ thị. Với độ phức tạp trung bình $O(n.log_2 n)$, thuật toán này phát huy hiệu quả nhờ chiến lược chia để trị. Tuy nhiên, vì dữ liệu ngẫu nhiên có thể dẫn đến các trường hợp phân chia không cân bằng, Quick sort không hoàn toàn vượt trội so với Counting sort hay Radix sort trong trường hợp này.

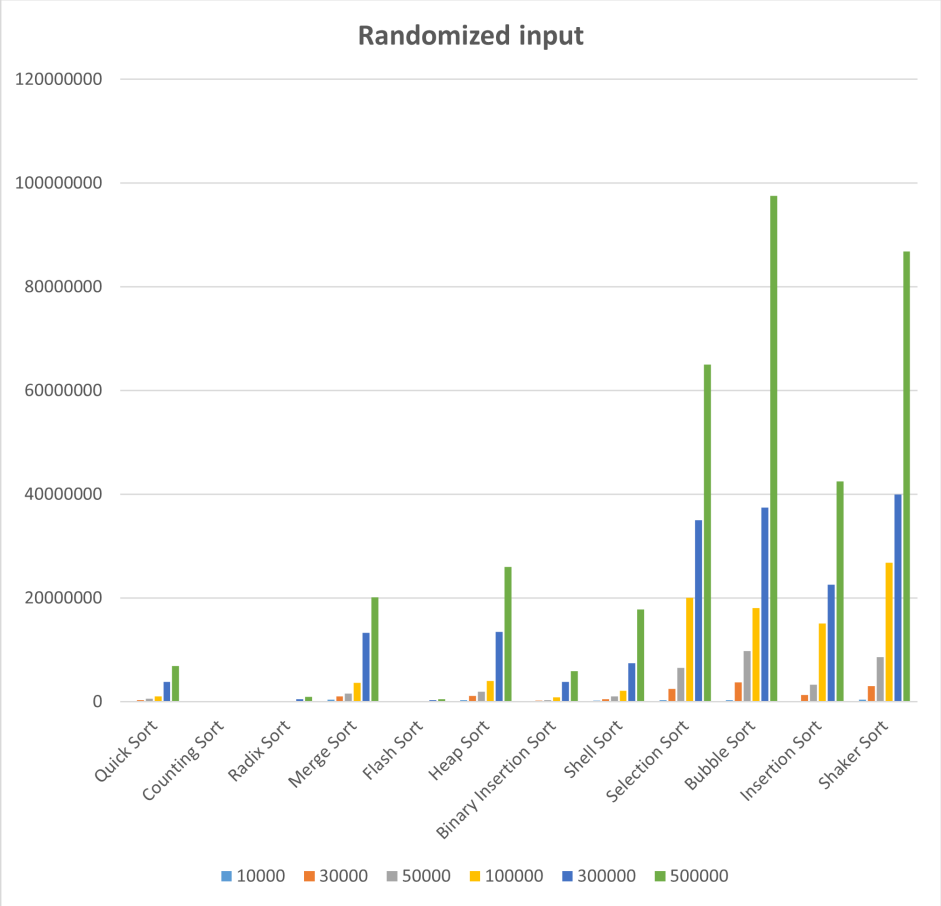
Merge sort và Heap sort thể hiện sự ổn định về hiệu năng, mặc dù chúng không nhanh bằng các thuật toán dẫn đầu. Merge sort, với chiến lược chia để trị sau đó hợp nhất, duy trì độ phức tạp $O(n.log_2 n)$ nhưng đòi hỏi thêm bộ nhớ phụ trợ. Trong khi đó, Heap sort tỏ ra hiệu quả nhờ cấu trúc heap, tuy nhiên việc duy trì tính chất của heap có thể làm tăng chi phí xử lý, khiến nó chậm hơn Quick sort.

Shell sort đạt hiệu năng trung bình (với khoảng cách gap chúng tôi đang dùng để thực nghiệm là $h_i = \left\lfloor \frac{2^i}{n} \right\rfloor, \in O(n^{1.5})$ tốt hơn đáng kể so với các thuật toán cơ bản như Bubble sort và Insertion sort, nhưng vẫn không thể cạnh tranh với các thuật toán như Quick sort hay Flash sort. Nhờ vào việc sử dụng các dãy khoảng cách trong sắp xếp, Shell sort giảm được số lượng so sánh, nhưng hiệu quả của nó giảm dần khi kích thước dữ liệu tăng lên.

Ngược lại, các thuật toán cơ bản như Bubble sort, Insertion sort, và Shaker sort nằm trong nhóm chậm nhất, với thời gian chạy tăng đột biến khi kích thước dữ liệu lớn. Điều này xuất phát từ độ phức tạp $O(n^2)$, khiến chúng phải thực hiện một lượng lớn các phép so sánh và hoán đổi. Trong số này, Shaker sort, mặc dù là phiên bản cải tiến hai chiều của Bubble sort, nhưng vẫn mất khoảng 1000s cho 500.000 phần tử so với khoảng 1400s của Bubble sort, có nghĩa là shaker sort không mang lại cải thiện đáng kể về hiệu năng. Cuối cùng, Binary Insertion sort nằm giữa nhóm trung bình và nhóm chậm nhất. Mặc dù có cải tiến trong việc tìm kiếm vị trí chèn nhờ tìm kiếm nhị phân, nhưng độ phức tạp $O(n^2)$ vẫn là rào cản lớn khiến nó không phù hợp với dữ liệu quy mô lớn.

Nhìn chung, Đồ thị nhấn mạnh ưu thế của các thuật toán không dựa trên so sánh như Counting sort và Radix sort, cùng với Flash sort, trong việc xử lý dữ liệu ngẫu nhiên với kích thước lớn. Các thuật toán như Quick sort, Merge sort, và Heap sort cũng đạt hiệu năng tốt, nhờ vào độ phức tạp O(nlogn).

Tuy nhiên, các thuật toán cơ bản như Bubble sort, Insertion sort, và Shaker sort cho thấy những hạn chế rõ ràng khi xử lý dữ liệu lớn, với thời gian chạy tăng lên đáng kể. Điều này cho thấy rằng, việc lựa chọn thuật toán phù hợp không chỉ phụ thuộc vào kích thước dữ liệu mà còn vào đặc điểm phân bố của nó.



Biểu đồ cột về số lần so sánh của các thuật toán sắp xếp trên dữ liệu ngẫu nhiên làm nổi bật những khác biệt lớn trong cách mỗi thuật toán xử lý dữ liệu khi kích thước tăng lên. Counting Sort và Radix Sort, hai thuật toán không dựa trên so sánh (non-comparison-based), gần như không phát sinh bất kỳ phép so sánh nào trong quá trình xử lý. Điều này lý giải tại sao cột của chúng hầu như không thay đổi trên toàn bộ các mức kích thước dữ liệu, dù số phần tử tăng đến 500.000. Đây là một trong những lý do chính giúp chúng đạt hiệu năng vượt trội so với các thuật toán khác.

Flash Sort tiếp tục thể hiện tính hiệu quả, với số lần so sánh thấp và tăng chậm hơn khi kích thước dữ liệu tăng lên. Điều này là nhờ vào chiến lược phân vùng và sắp xếp cục bộ hiệu quả của nó. Đây cũng là một ví dụ tiêu biểu về cách tối ưu hoá các phép so sánh trong thuật toán dựa trên phân phối.

Ngược lại, Bubble Sort, Insertion Sort, và Shaker Sort lại thể hiện số lần so sánh cao nhất, đặc biệt

khi kích thước dữ liệu lớn. Với dữ liệu 500.000 phần tử, các thuật toán này ghi nhận số lần so sánh vượt ngưỡng 600 triệu và gần chạm tới ngưỡng 1 tỷ lần so sánh đối với Bubble sort, trong khi các thuật toán khác dừng ở mức thấp hơn đáng kể. Điều này phản ánh độ phức tạp $O(n^2)$ vốn có, do chúng phải thực hiện hàng loạt các phép so sánh cho mỗi cặp phần tử để đảm bảo tính chính xác.

Binary Insertion Sort, mặc dù cải thiện so với Insertion Sort cơ bản bằng cách sử dụng tìm kiếm nhị phân để giảm phần lớn số lần so sánh, nhưng vẫn chịu ảnh hưởng lớn từ độ phức tạp $O(n^2)$. Điều này khiến nó không thể xử lý tốt khi số lượng phần tử tăng lên, và cột của nó trên biểu đồ phản ánh mức tăng đáng kể, đặc biệt với dữ liệu 300.000 và 500.000 phần tử.

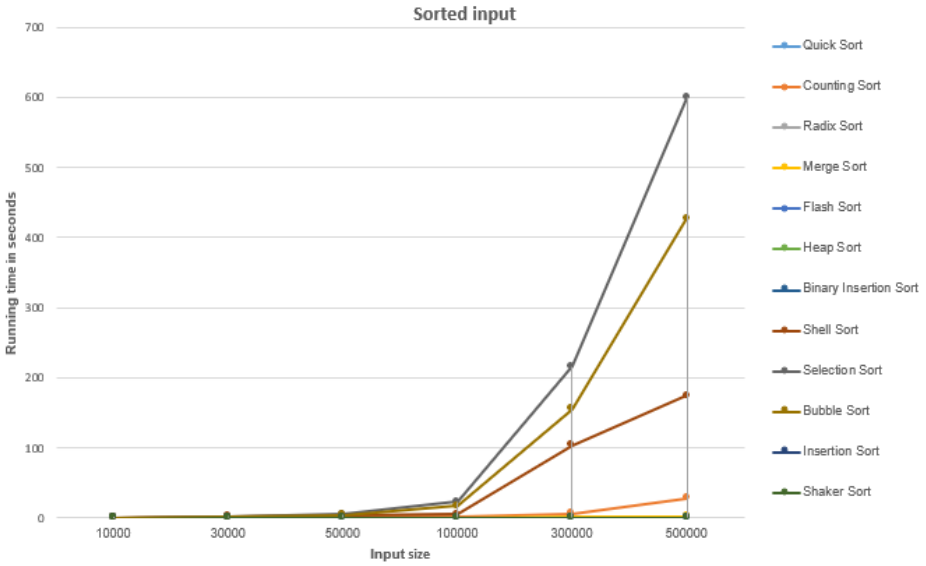
Quick Sort và Merge Sort nằm trong nhóm các thuật toán hiệu quả hơn, với số lần so sánh tăng lên tuyến tính hơn so với các thuật toán kém hiệu quả. Đặc biệt, Quick Sort duy trì mức số lần so sánh tương đối thấp trong nhóm các thuật toán dựa trên so sánh. Điều này có được nhờ chiến lược chia để trị, chia nhỏ bài toán sắp xếp thành các phân đoạn nhỏ hơn. Tuy nhiên, nếu dữ liệu không được phân phối đồng đều, số lần so sánh có thể tăng lên đáng kể.

Heap Sort và Shell Sort cũng nằm ở mức trung bình, với số lần so sánh cao hơn Quick Sort nhưng thấp hơn các thuật toán cơ bản như Bubble Sort. Đối với Heap Sort, cấu trúc heap đòi hỏi phải thực hiện nhiều phép so sánh để duy trì tính chất của heap trong mỗi thao tác chèn hoặc xoá. Shell Sort, mặc dù có cải tiến bằng việc giảm khoảng cách so sánh giữa các phần tử, vẫn không thể sánh được với các thuật toán dựa trên cấu trúc mạnh mẽ hơn như Merge Sort.

Tổng quát, biểu đồ nhấn mạnh rằng các thuật toán không dựa trên so sánh như Counting Sort và Radix Sort vượt trội về hiệu quả nhờ tránh được các phép so sánh trực tiếp. Trong khi đó, Flash Sort và Quick Sort thể hiện khả năng tối ưu hoá các phép so sánh trong nhóm các thuật toán dựa trên so sánh. Ngược lại, các thuật toán cơ bản như Bubble Sort, Insertion Sort, và Shaker Sort phải thực hiện lượng lớn phép so sánh, khiến chúng không phù hợp để xử lý dữ liệu quy mô lớn. Điều này cho thấy rằng số lần so sánh đóng vai trò quan trọng trong việc đánh giá hiệu năng của thuật toán, và các chiến lược tối ưu hoá phép so sánh có thể mang lại sự khác biệt rõ ràng khi dữ liệu tăng lên.

Dữ liệu đầu vào đã được sắp xếp (Sorted Input)

Sorted Input												
Datasize	**10,000**		**30,000**		**50,000**		**100,000**		**300,000**		**500,000**	
Resulting statics	**Running time**	**Comparision**	**Running time**	**Comparision**	**Running time**	**Comparision**	**Running time**	**Comparision**	**Running time**	**Comparision**	**Running time**	**Comparision**
Quick Sort	0.0003	41328	0.0007	114681	0.0041	229369	0.0063	458745	0.0082	1182496	0.0123	1835001
Counting Sort	0.15	0	1.0005	0	2.857	0	2.2681	0	5.9695	0	28.5548	0
Radix Sort	0.0006	9999	0.0044	29999	0.0061	49999	0.0102	99999	0.0376	299999	0.2951	499999
Merge Sort	0.0091	69008	0.0036	227728	0.0057	401952	0.1012	853904	1.2143	2797264	2.1424	4783216
Flash Sort	0.0005	73500	0.0012	220500	0.0031	367500	0.0037	735000	0.0228	2205000	0.0352	3675000
Heap Sort	0.0031	381416	0.0142	1281447	0.0182	2253742	0.0532	4813371	0.3231	15840909	0.5421	27443485
Binary Insertion Sort	0.6619	123617	4.0356	417233	6.9337	734465	12.3672	1568929	126.634	5175713	153.238	8975713
Shell Sort	0.3591	120005	2.7635	390007	3.9362	700006	5.6399	1500006	103.658	5100008	174.748	8500007
Selection Sort	0.3001	49995000	1.7862	449985000	5.3379	1249975000	23.558	4999950000	216.05	44999850000	598.96	124999750000
Bubble Sort	0.2052	50004999	1.5041	450014999	4.3439	1250024999	17.248	5000049999	155.45	45000149999	427.16	125000249999
Insertion Sort	0.00002	9999	0.00006	29999	0.00012	49999	0.00018	99999	0.00052	299999	0.00082	499999
Shaker Sort	0.00007	20002	0.00012	60002	0.00022	100002	0.00056	200002	0.0012	600002	0.00189	1000002



Đồ thị thể hiện thời gian thực thi của các thuật toán với dữ liệu đã được sắp xếp (sorted input) cho thấy sự phân hóa rõ rệt trong hiệu năng của 12 thuật toán sắp xếp. Trong trường hợp này, Insertion sort và Shaker sort nổi bật với thời gian chạy nhanh nhất, hầu như không đáng kể ngay cả khi kích thước dữ liệu đạt 500,000 phần tử. Đây là một kết quả đáng chú ý, bởi các thuật toán này thường được xem là kém hiệu quả trên dữ liệu ngẫu nhiên và dữ liệu đảo ngược (reversed input), nhưng lại tận dụng được tính chất đã sắp xếp của dữ liệu để giảm thiểu số lần so sánh và hoán đổi, dẫn đến thời gian chạy gần như tối ưu.

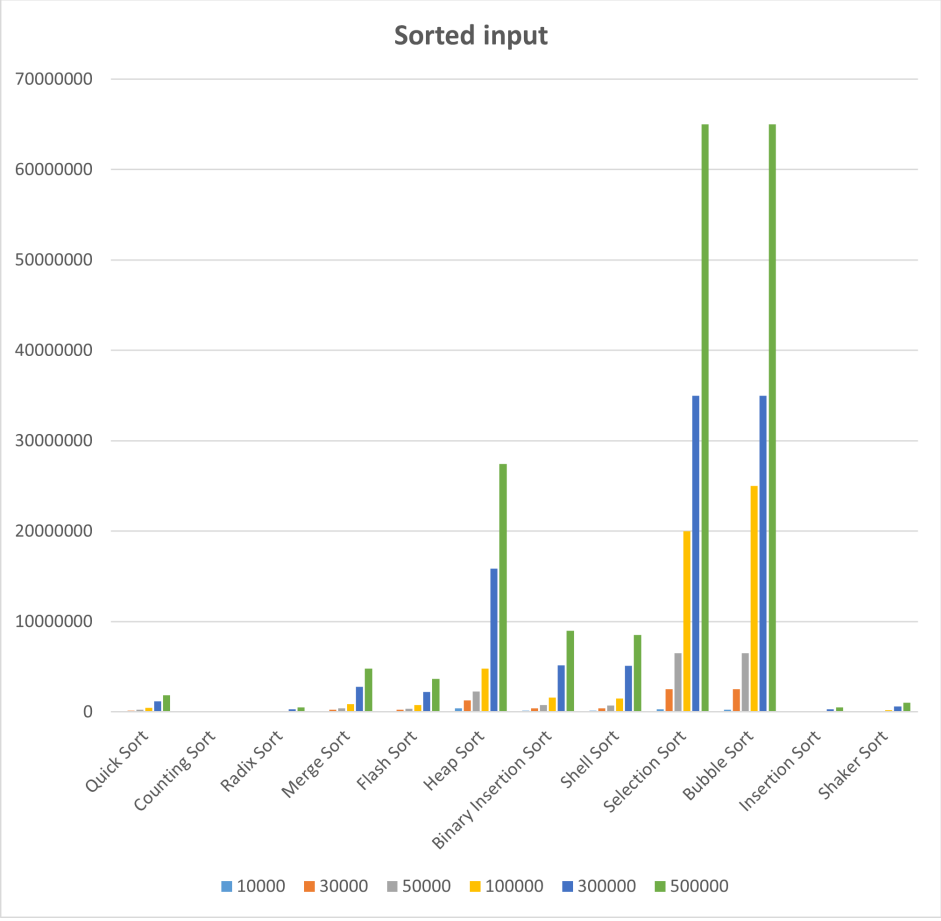
Quick sort, Flash sort và Radix sort cũng đạt hiệu năng xuất sắc, với thời gian chạy chỉ ở mức thấp, không đáng kể so với phần lớn các thuật toán khác. Điều này khẳng định sự ổn định của chúng trong các trường hợp dữ liệu đã sắp xếp. Quick sort, mặc dù phụ thuộc vào cách phân chia dữ liệu, vẫn giữ được ưu thế nhờ chiến lược chọn pivot hiệu quả.

Counting sort, mặc dù thường được đánh giá cao ở dữ liệu ngẫu nhiên, lại thể hiện thời gian chạy chậm hơn đáng kể trên dữ liệu đã sắp xếp. Điều này có thể liên quan đến chi phí xử lý và quản lý cấu trúc dữ liệu phụ trợ, đặc biệt khi kích thước dữ liệu lớn.

Nhóm thuật toán có hiệu năng trung bình bao gồm Merge sort, Heap sort và Binary Insertion sort, với thời gian chạy tăng dần nhưng vẫn duy trì ở mức hợp lý. Merge sort tiếp tục khẳng định sự ổn định nhờ chiến lược chia để trị sau đó hợp nhất, trong khi Binary Insertion sort tối ưu hóa so với Insertion sort thông thường, tuy nhiên không thể so sánh với các thuật toán vượt trội hơn.

Ở nhóm các thuật toán kém hiệu quả, Shell sort, Selection sort và Bubble sort thể hiện thời gian chạy lâu nhất. Bubble sort và Selection sort mất gần như toàn bộ thời gian của đồ thị để hoàn tất việc xử lý tất cả phần tử, với thời gian vượt ngưỡng cao nhất trong số 12 thuật toán. Điều này minh họa rõ ràng sự bất lợi của các thuật toán có độ phức tạp $O(n^2)$ khi xử lý dữ liệu lớn, bất kể dữ liệu đã sắp xếp.

Nhìn chung, đồ thị nhấn mạnh sự ưu việt của các thuật toán được tối ưu hóa hoặc thiết kế để khai thác tính chất của dữ liệu, chẳng hạn như Insertion sort, Shaker sort, và Radix sort. Đồng thời, kết quả cũng nhấn mạnh sự hạn chế của các thuật toán cơ bản như Bubble sort và Selection sort khi đối mặt với dữ liệu kích thước lớn, bất kể thứ tự ban đầu.



Insertion sort và Shaker sort nổi bật với số phép so sánh thấp nhất, đặc biệt khi kích thước dữ liệu đạt 500,000 phần tử. Điều này có thể lý giải bởi khả năng tận dụng tính chất đã sắp xếp của dữ liệu,

giúp giảm đáng kể số lần so sánh và hoán đổi, dẫn đến hiệu suất vượt trội trong trường hợp này.

Quick sort, Flash sort, và Radix sort cũng đạt hiệu năng xuất sắc với số phép so sánh chỉ ở mức thấp, không đáng kể so với phần lớn các thuật toán khác. Quick sort giữ được hiệu quả nhờ chiến lược chọn phần tử pivot tốt, cụ thể là chọn pivot có khả năng cân bằng dữ liệu ở hai bên, giúp giảm độ sâu của cây phân vùng và số phép so sánh cần thực hiện. Flash sort tận dụng phương pháp phân vùng hiệu quả để xử lý dữ liệu phân bố đều. Radix sort, mặc dù không dựa trên so sánh, vẫn duy trì hiệu năng ổn định nhờ xử lý theo từng chữ số.

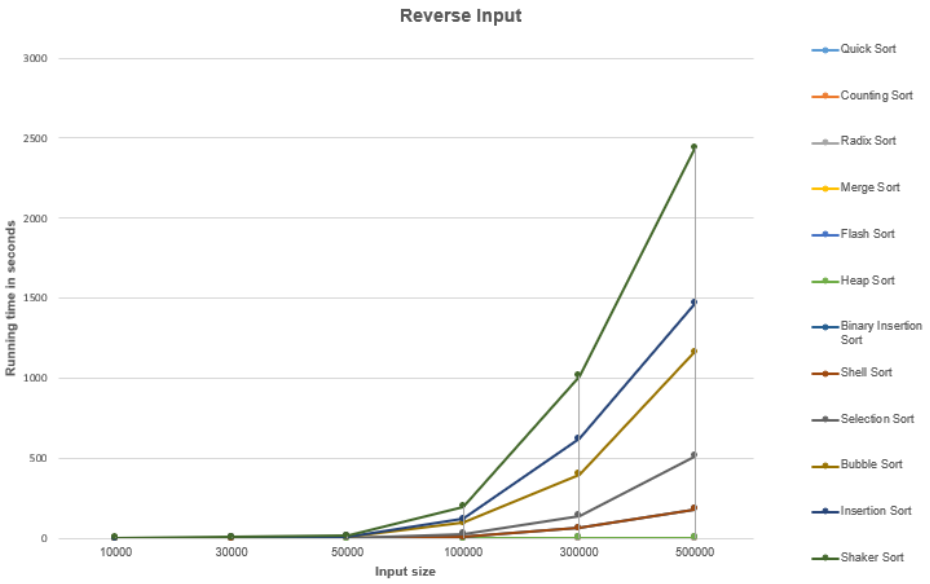
Merge sort, Heap sort, và Binary Insertion sort thể hiện hiệu năng trung bình. Merge sort khẳng định tính ổn định nhờ chiến lược chia để trị, trong khi Binary Insertion sort cải thiện so với Insertion sort thông thường bằng cách tận dụng thuật toán tìm kiếm nhị phân từ đó giảm phần lớn số phép so sánh. Heap sort tuy thuộc nhóm O(n log n) nhưng mất thời gian duy trì cấu trúc heap, dẫn đến số phép so sánh lớn hơn so với các thuật toán hiệu quả cao.

Bubble sort, Selection sort, và Shell sort có số phép so sánh rất cao, đặc biệt khi kích thước dữ liệu lớn. Bubble sort và Selection sort đều thuộc nhóm $O(n^2)$, với số phép so sánh vượt trội ở mức cao nhất trong số các thuật toán. Shell sort, mặc dù cải tiến hơn, nhưng vẫn không thể đạt hiệu quả tương đương các thuật toán dựa trên phân vùng hoặc không so sánh.

Nhìn chung, biểu đồ nhấn mạnh sự vượt trội của các thuật toán như Insertion sort và Shaker sort trong trường hợp dữ liệu đã sắp xếp, nhờ khả năng giảm thiểu số lần so sánh. Các thuật toán như Quick sort, Flash sort, và Radix sort duy trì hiệu suất ổn định và xuất sắc, trong khi Bubble sort, Selection sort, và Shell sort tiếp tục thể hiện sự hạn chế khi xử lý dữ liệu lớn.

Dữ liệu đầu vào đảo ngược (Reverse Input)

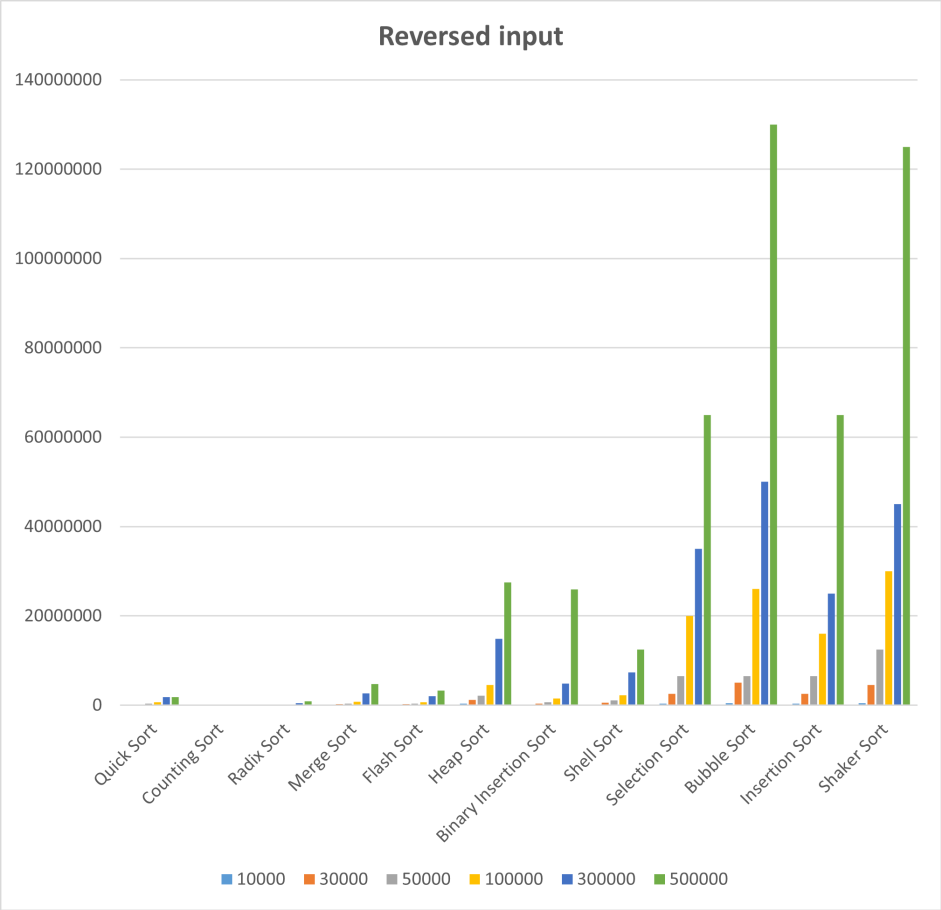
Reverse Input												
Datasize	**10,000**		**30,000**		**50,000**		**100,000**		**300,000**		**500,000**	
Resulting statics	**Running time**	**Comparision**	**Running time**	**Comparision**	**Running time**	**Comparision**	**Running time**	**Comparision**	**Running time**	**Comparision**	**Running time**	**Comparision**
Quick Sort	0.0003	41328	0.0007	114681	0.0041	229369	0.0063	458745	0.0082	1182496	0.0123	1835001
Counting Sort	0.15	0	1.0005	0	2.857	0	2.2681	0	5.9695	0	28.5548	0
Radix Sort	0.0006	9999	0.0044	29999	0.0061	49999	0.0102	99999	0.0376	299999	0.2951	499999
Merge Sort	0.0091	69008	0.0036	227728	0.0057	401952	0.1012	853904	1.2143	2797264	2.1424	4783216
Flash Sort	0.0005	73500	0.0012	220500	0.0031	367500	0.0037	735000	0.0228	2205000	0.0352	3675000
Heap Sort	0.0031	381416	0.0142	1281447	0.0182	2253742	0.0532	4813371	0.3231	15840909	0.5421	27443485
Binary Insertion Sort	0.6619	123617	4.0356	417233	6.9337	734465	12.3672	1568929	64.284	5175713	181.124	8975713
Shell Sort	0.3591	120005	2.7635	390007	3.9362	700006	5.6399	1500006	103.658	5100008	174.748	8500007
Selection Sort	0.3001	49995000	1.7862	449985000	5.3379	1249975000	23.558	4999950000	216.05	44999850000	598.96	124999750000
Bubble Sort	0.2052	99999999	1.5041	899999999	4.3439	1250299513	70.126	9999999999	258.52	89999999999	651.86	249999999999
Insertion Sort	0.00002	49995000	0.00006	449985000	0.00012	1249975000	0.00018	4999950000	0.00052	44999850000	0.00082	124999750000
Shaker Sort	0.00007	100005001	0.00012	900015001	0.00022	2500025001	0.00056	10000050001	0.0012	90000150001	0.00189	250000250001



Dựa trên bảng dữ liệu và các biểu đồ, ta có thể nhận thấy rằng các thuật toán sắp xếp được chia thành hai nhóm rõ rệt về thời gian thực thi. Nhóm đầu tiên, bao gồm Quick Sort, Counting Sort, Radix Sort, Merge Sort, Flash Sort, Heap Sort, có thời gian thực thi khá tốt. Nhóm thứ hai, bao gồm Binary Insertion Sort, Shell Sort, Selection Sort, Bubble Sort, Insertion Sort, Shaker Sort, có thời gian thực thi chậm hơn. Sự khác biệt này có thể được giải thích thông qua độ phức tạp thời gian. Cụ thể, các thuật toán trong nhóm đầu chủ yếu có độ phức tạp thời gian thuộc $O(nlogn)$ hoặc $O(n)$, trong khi các thuật toán thuộc nhóm thứ hai thường có độ phức tạp thời gian $O(n^2)$.

Đối với kiểu dữ liệu đảo ngược (reverse input), thuật toán Counting Sort thể hiện sự vượt trội ở hầu hết các kích thước dữ liệu, từ 10.000 đến 500.000 phần tử, với thời gian thực thi chỉ 0.0041 giây cho dữ liệu 500.000 phần tử. Ngược lại, Shaker Sort có thời gian thực thi lâu nhất so với các thuật toán còn lại, mất đến 970.43 giây cho cùng kích thước dữ liệu.

Nguyên nhân chính của sự khác biệt này là đặc tính của dữ liệu đảo ngược. Với các thuật toán dựa trên so sánh, kiểu dữ liệu này yêu cầu nhiều phép kiểm tra và hoán đổi hơn, dẫn đến thời gian thực thi tăng lên đáng kể. Ngược lại, các thuật toán không dựa trên so sánh như Counting Sort không bị ảnh hưởng bởi thứ tự dữ liệu, giúp chúng duy trì hiệu suất cao bất kể đặc tính của dữ liệu đầu vào. Điều này cho phép các thuật toán không dựa trên so sánh hoạt động ổn định với thời gian thực thi không phụ thuộc vào đặc điểm của dữ liệu.



Về số lượng phép so sánh, Counting Sort là trường hợp nổi bật khi không thực hiện bất kỳ phép so sánh nào, bất kể kích thước hay kiểu dữ liệu. Đây là đặc trưng của các thuật toán không dựa trên sự so sánh (non-comparison based), giúp Counting Sort đạt được hiệu suất vượt trội so với các thuật toán khác trong điều kiện dữ liệu đảo ngược.

Ngược lại, Shaker Sort là thuật toán thực hiện nhiều phép so sánh nhất, lên tới 2.500.000.000.000 (2 nghìn 5 trăm tỷ) phép với dữ liệu kích thước 500.000. Điều này dẫn đến thời gian thực thi cực kỳ chậm, thể hiện sự kém hiệu quả của thuật toán này khi xử lý dữ liệu có đặc tính không thuận lợi.

Tương tự, Bubble Sort và Selection Sort cũng cho thấy hiệu suất kém với thời gian lần lượt là 651.86 giây và 329.25 giây ở kích thước dữ liệu 500.000. Các thuật toán này phải thực hiện nhiều vòng lặp kiểm tra và hoán đổi, làm gia tăng đáng kể thời gian thực thi khi kích thước dữ liệu lớn.

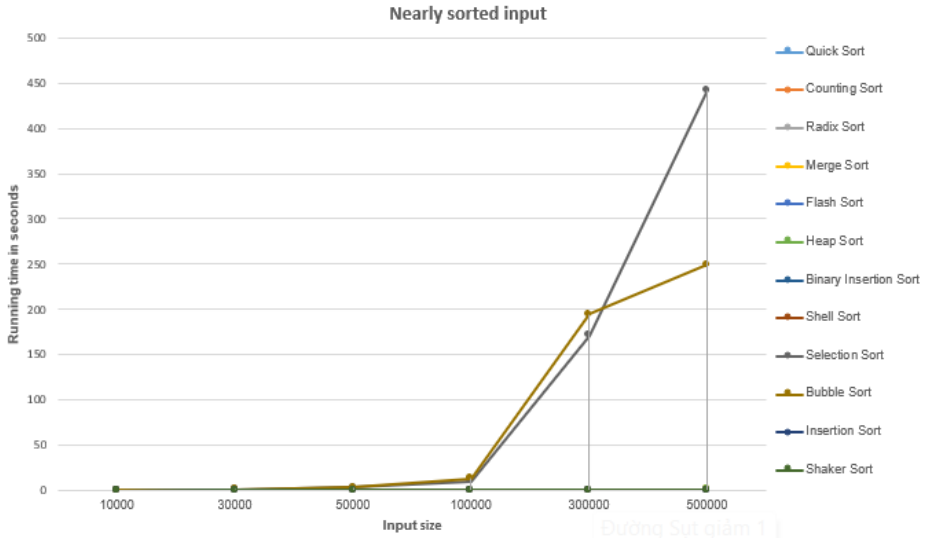
Insertion Sort và Binary Insertion Sort cũng gặp khó khăn trong việc xử lý dữ liệu đảo ngược. Insertion Sort mất 301.26 giây, trong khi Binary Insertion Sort, dù được cải tiến bằng cách sử dụng tìm kiếm nhị phân để giảm bớt số phép kiểm tra, vẫn mất 123.43 giây để xử lý cùng kích thước dữ liệu. Điều này cho thấy rằng các thuật toán thuộc nhóm có độ phức tạp $O(n^2)$ đều không phù hợp với kiểu dữ liệu đảo ngược khi kích thước tăng cao.

Tóm lại, kiểu dữ liệu đảo ngược (reverse input) đã làm rõ sự khác biệt giữa các thuật toán dựa trên so sánh và không dựa trên so sánh. Với các thuật toán không dựa trên so sánh như Counting Sort và Radix Sort, hiệu suất gần như không bị ảnh hưởng bởi đặc tính của dữ liệu đầu vào. Trong khi đó, các thuật toán dựa trên so sánh phải thực hiện nhiều phép kiểm tra hơn khi dữ liệu đã sắp xếp ngược, khiến thời gian thực thi tăng đột biến.

Trong trường hợp dữ liệu đảo ngược, Counting Sort là thuật toán hiệu quả nhất, nhờ khả năng xử lý không phụ thuộc vào thứ tự dữ liệu và độ phức tạp O(n). Ngược lại, Shaker Sort là thuật toán kém hiệu quả nhất do độ phức tạp $O(n^2)$ và sự phụ thuộc mạnh vào thứ tự dữ liệu, dẫn đến thời gian thực thi rất lớn. Nhìn chung, các thuật toán thuộc nhóm 1 với độ phức tạp O(n log n) hoặc O(n) luôn là lựa chọn tối ưu cho các tập dữ liệu lớn, đặc biệt là những kiểu dữ liệu không thuận lợi như dữ liệu đảo ngược.

Dữ liệu đầu vào gần được sắp xếp (Nearly Sorted Input)

Nearly Sorted Input												
Datasize	**10,000**		**30,000**		**50,000**		**100,000**		**300,000**		**500,000**	
Resulting statics	**Running time**	**Comparision**	**Running time**	**Comparision**	**Running time**	**Comparision**	**Running time**	**Comparision**	**Running time**	**Comparision**	**Running time**	**Comparision**
Quick Sort	0.0002	41408	0.0004	114753	0.0007	229457	0.0014	458825	0.0042	1182584	0.0214	1835089
Counting Sort	0.0001	0	0.0003	0	0.0006	0	0.0008	0	0.0022	0	0.0175	0
Radix Sort	0.0003	9999	0.0012	29999	0.0025	49999	0.0037	99999	0.0165	299999	0.1869	499999
Merge Sort	0.0075	80258	0.0273	269329	0.0335	446279	0.0651	905579	0.1968	2848103	0.6554	4839952
Flash Sort	0.0002	64272	0.0007	194884	0.0011	340653	0.0097	702996	0.0135	2173286	0.0203	3644133
Heap Sort	0.0015	381350	0.0054	1281423	0.0095	2253768	0.0168	4813269	0.053	15840887	0.2511	27443582
Binary Insertion Sort	0.0004	121901	0.0013	409481	0.0026	724506	0.0043	1557287	0.0144	5163801	0.0871	8965307
Shell Sort	0.0003	139816	0.0011	452598	0.0027	796235	0.003	1604696	0.0101	5171498	0.0978	8581726
Selection Sort	0.0914	49995000	0.9112	449985000	2.4766	1249975000	9.9723	4999950000	170.6	44999850000	442.418	124999750000
Bubble Sort	0.0904	50050691	0.8957	450166711	2.5476	1250299513	12.8098	5000354713	194.74	45000324487	249.53	124999750000
Insertion Sort	0.0001	55691	0.0004	181711	0.0006	324513	0.0023	404713	0.0064	474487	0.0073	729735
Shaker Sort	0.0003	131924	0.001	430133	0.0014	691097	0.0165	877423	0.0059	1062851	0.0101	1501108



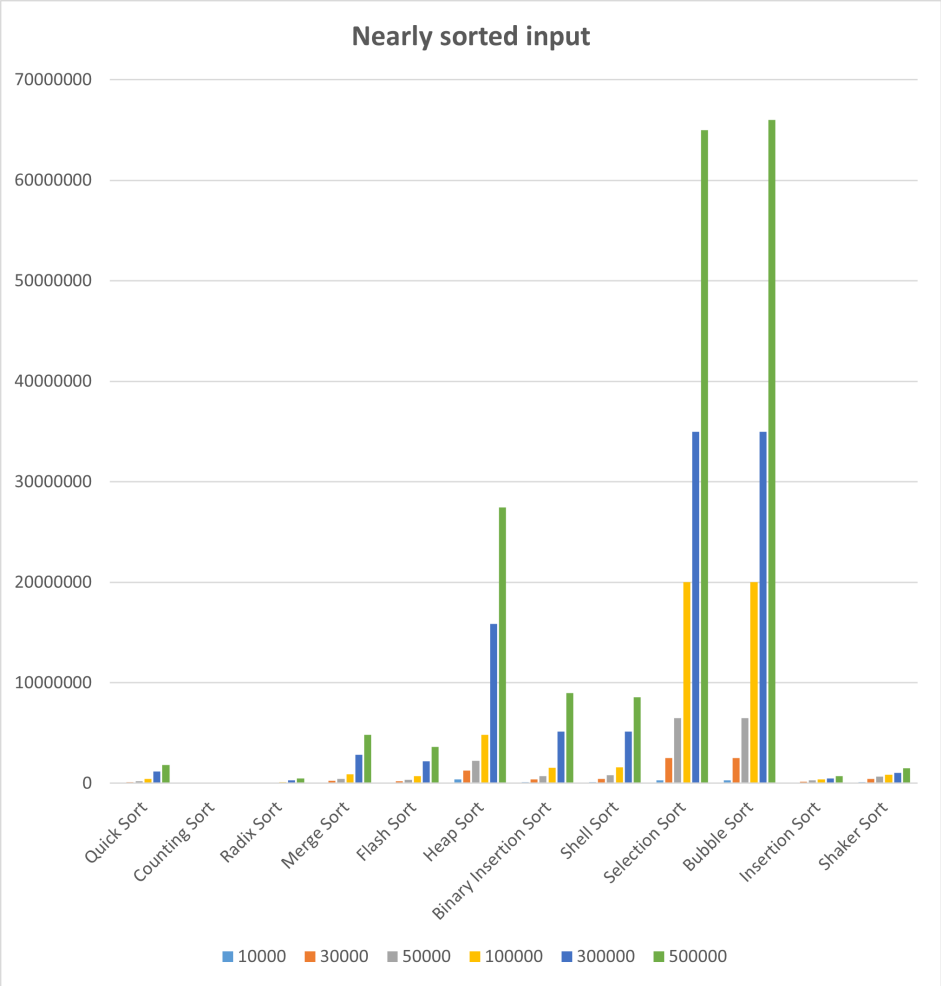
Dựa trên bảng dữ liệu và các biểu đồ với kiểu dữ liệu gần sắp xếp (nearly sorted), ta có thể nhận xét như sau Các thuật toán được phân thành hai nhóm rõ ràng về thời gian thực thi. Nhóm nhanh hơn

bao gồm Quick Sort, Counting Sort, Radix Sort, Merge Sort, Flash Sort, Heap Sort, và nhóm chậm hơn gồm Binary Insertion Sort, Shell Sort, Selection Sort, Bubble Sort, Insertion Sort, Shaker Sort. Sự khác biệt này được giải thích bởi độ phức tạp thời gian. Nhóm đầu tiên thường có độ phức tạp O(n log n) hoặc O(n), trong khi nhóm thứ hai có độ phức tạp $O(n^2)$.

Với dữ liệu gần như đã sắp xếp, Counting Sort duy trì hiệu suất vượt trội, không bị ảnh hưởng bởi đặc tính của dữ liệu và có thời gian thực thi thấp nhất ở tất cả các kích thước, chẳng hạn chỉ mất 0.0008 giây cho kích thước 100.000 và 0.0175 giây cho kích thước 500.000. Tương tự, Quick Sort cũng cho thấy hiệu suất cao, chỉ mất 0.0 214 giây cho kích thước 500.000 nhờ độ phức tạp O(n log n) và khả năng xử lý tốt dữ liệu gần như đã sắp xếp.

Ngược lại, Bubble Sort và Selection Sort thể hiện sự kém hiệu quả. Bubble Sort mất đến 249.53 giây và Selection Sort mất 442418.5 giây cho kích thước dữ liệu 500.000, do vẫn phải thực hiện rất nhiều phép so sánh và hoán đổi không cần thiết. Những thuật toán này gặp khó khăn lớn khi kích thước dữ liệu tăng cao, bất chấp dữ liệu đầu vào gần như đã sắp xếp.

Các thuật toán như Insertion Sort và Binary Insertion Sort được hưởng lợi từ đặc tính của dữ liệu gần như đã sắp xếp. Insertion Sort chỉ mất 0.0073 giây với kích thước 500.000, thấp hơn đáng kể so với dữ liệu đảo ngược, nhờ giảm số phép chèn và hoán đổi. Binary Insertion Sort cũng cho kết quả tốt hơn, với thời gian 0.0871 giây ở kích thước tương tự.



Về số lượng phép so sánh, Counting Sort vẫn không thực hiện bất kỳ phép so sánh nào nhờ đặc trưng của thuật toán không dựa trên sự so sánh. Ngược lại, Bubble Sort và Selection Sort thực hiện số lượng phép so sánh lớn nhất cùng là 1.25×10^{11} ở kích thước 500.000, làm tăng thời gian thực thi.

Nhận xét tổng quát về 12 thuật toán sắp xếp

Từ kết quả thực nghiệm và đối chiếu các số liệu với nhau, ta có thể thấy là flash sort có thời gian chạy là ngắn nhất trong số 12 sort. Và nhóm các thuật toán sắp xếp có thời gian chạy nhanh và ổn định với 4 loại dữ liệu đầu vào là: Quick Sort, Counting Sort, Radix Sort, Merge Sort, Heap Sort.Điều này cho thấy các thuật toán kể trên không chỉ có thời gian chạy nhanh mà còn ổn định với các loại dữ liệu đầu vào khác nhau. Trong đó, Quick Sort nổi bật nhờ khả năng thích ứng với nhiều phân bố dữ liệu, Counting Sort và Radix Sort lại hiệu quả trong trường hợp phạm vi dữ liệu đầu vào hẹp và xác định rõ. Merge Sort và Heap Sort, với tính ổn định và đảm bảo hiệu suất trong trường hợp xấu nhất, cũng cho thấy sự ổn định trong các tình huống khác nhau.

Ngược lại, các thuật toán sắp xếp còn lại trong so sánh có thể có thời gian chạy lâu hơn hoặc biến

đổi đáng kể tùy thuộc vào dữ liệu đầu vào, ví dụ như thuật toán shaker sort, thời gian chạy đối với kiểu dữ liệu đã được sắp xếp và gần được sắp xếp rất nhanh so với 2 kiểu dữ liệu còn lại là ngẫu nhiên và đảo ngược, điều này xảy ra do bản chất của thuật toán.

Từ những dữ liệu thí nghiệm và nghiên cứu của nhóm, thuật toán Quick Sort nhất bởi các yếu tố:

- Thời gian chạy nhanh và ổn định với các kiểu dữ liệu khác nhau.
- Phạm vi dữ liệu không bị giới hạn, một lợi thế khi so với Counting Sort.
- Dễ dàng cài đặt và sử dụng hơn so với Flash Sort.
- Khả năng sắp xếp đa dạng theo nhiều yêu cầu khác nhau, chẳng hạn như sắp xếp trị tuyệt đối lớn nhất, nhỏ nhất hoặc sắp xếp theo chữ, điều mà Radix Sort và Counting Sort không thể thực hiện được.

5 Tổ chức dự án và ghi chú lập trình

Cấu trúc thư mục của dự án được tổ chức như sau:

- **SOURCE folder**: Chứa các tệp mã nguồn
  - `02.cpp`: Chứa mã nguồn chính phục vụ cho phần Command Line.
  - `Experiment.cpp`: Chứa mã nguồn phục vụ cho phần thực nghiệm so sánh các thuật toán.
- **Report.pdf**: Báo cáo thực hành của dự án.
- **02.exe**: File thực thi.

5.1 Tổng quan về nội dung trong các file mã nguồn

Cả hai tệp `02.cpp` và `Experiment.cpp` có một số nội dung chung như sau:

- **12 thuật toán sắp xếp**: Bao gồm các thuật toán phổ biến được triển khai để so sánh hiệu suất.
- **Hàm sinh dữ liệu**: Được sử dụng để tạo ra các bộ dữ liệu đầu vào khác nhau.
- **Thư viện cmath**: Hỗ trợ trong thuật toán sắp xếp
- **Hàm tính toán thời gian chạy của thuật toán**: Sử dụng thư viện `<chrono>`, cụ thể là hàm `chrono::duration<double> countime()`, để đo thời gian với độ chính xác lên đến 8 chữ số thập phân.

- **Hàm xử lý tên thuật toán**: Hỗ trợ việc ánh xạ giữa tên thuật toán và chức năng tương ứng.
- **Hàm chạy thuật toán theo tên**: Đảm bảo khả năng thực thi các thuật toán dựa trên đầu vào tên thuật toán.

5.2 Chi tiết mã nguồn

5.2.1 File `02.cpp` (Command Line)

- **Hàm** `main`:
  - Nhận thông tin từ Command Prompt qua tham số `argc` và mảng `argv[]`.
  - Xử lý các tham số này để thực hiện các lệnh tương ứng.
- **Tổ chức Command Line thành 5 lệnh chính**:
  - Mỗi lệnh được viết trong một hàm riêng biệt để đảm bảo tính rõ ràng và dễ bảo trì.
  - Các hàm này sử dụng hàm `PrintCommand` hỗ trợ in hướng dẫn hoặc kết quả tương ứng.

5.2.2 File `Experiment.cpp` (Thực nghiệm)

- **Hàm** `runExperiment`:
  - Hàm này thực hiện so sánh hiệu năng của các thuật toán sắp xếp.
  - Các tham số chính bao gồm:
    - `sizeArr`: Kích thước mảng dữ liệu.
    - `dataType`: Loại dữ liệu (ngẫu nhiên, sắp xếp ngược, đã sắp xếp, v.v.).
    - `sort[]`: Danh sách các thuật toán sắp xếp cần chạy.
- **Hàm main**: chứa 2 vòng lặp để chạy các thuật toán trên các thứ tự dữ liệu và kích thước khác nhau.

Tài liệu

[1] Flash sort. https://en.wikipedia.org/wiki/Flashsort. Access date: 27/11/2024.

[2] Geeksforgeeks: Counting sort - hướng dẫn về cấu trúc dữ liệu và thuật toán. https://www.geeksforgeeks.org/counting-sort/. Access date: 01/12/2024.

[3] Heap sort. https://en.wikipedia.org/wiki/Heapsort#Other_variations. Access date: 01/12/2024.

[4] Insertion sort. https://www.geeksforgeeks.org/insertion-sort-algorithm/. Access date: 02/12/2024.

[5] Selection sort. https://en.wikipedia.org/wiki/Selection_sort#Variants. Access date: 25/11/2024.

[6] Shell sort. https://en.wikipedia.org/wiki/Shellsort/. Access date: 02/12/2024.

[7] Sắp xếp nổi bọt. https://vi.wikipedia.org/wiki/S%E1%BA%AFp_x%E1%BA%BFp_n%E1%BB%95i_b%E1%BB%8Dt. Access date: 25/11/2024.

[8] Teky.edu: Thuật toán quick sort là gì? giới thiệu lập trình chi tiết nhất. https://teky.edu.vn/blog/thuat-toan-quick-sort/. Access date: 01/12/2024.

[9] animeshdey. Geeksforgeeks: Time and space complexity analysis of merge sort. https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-merge-sort/. Access date: 23/11/2024.

[10] Swaminathan Iyer. Binary insertion sort. https://www.interviewkickstart.com/blogs/learn/binary-insertion-sort/. Access date: 06/12/2024.

[11] Anany Levitin. *Introduction to the Design and Analysis of Algorithm*. Pearson Education, 3rd edition, 2012.

[12] Chirag Manwani. Geeksforgeeks: Time complexity of building a heap. https://www.geeksforgeeks.org/time-complexity-of-building-a-heap/. Access date: 22/11/2024.

[13] Rezaul Chowdhury Pramod Ganapathi. *Parallel Divide-and-Conquer Algorithms for Bubble Sort, Selection Sort, and Insertion Sort*. State University of New York at Stony Brook, New York, USA, 2021.

[14] Th.S. Dương Anh Đức Th.S. Trần Hạnh Nhi. *Nhập môn CTDL và Thuật toán*. chịu trách nhiệm xuất bản: PGS. PTS Nguyễn Văn Đến, xưởng in trường Đại học Khoa học Tự nhiên TP. Hồ Chí Minh, 2003.

[15] Sven Woltmann. Happycoder: Merge sort – algorithm, source code, time complexity. https://www.happycoders.eu/algorithms/merge-sort/#Natural_Merge_Sort. Access date: 23/11/2024.