Các ý tưởng của các thuật toán sắp xếp

I. Interchange

Interchange Sort và Quicksort

Interchange Sort, hay còn gọi là thuật toán sắp xếp đổi chỗ trực tiếp, là một phương pháp sắp xếp đơn giản. Thuật toán này hoạt động bằng cách duyệt qua từng cặp phần tử trong danh sách và hoán đổi vị trí của chúng nếu chúng không theo thứ tự mong muốn. Quá trình này được lặp lại cho đến khi toàn bộ danh sách được sắp xếp.

Nguuyên lý hoạt động:

1. Bắt đầu từ phần tử đầu tiên trong danh sách.
2. So sánh phần tử này với các phần tử đứng sau nó.
3. Nếu phát hiện cặp phần tử nào đang ở vị trí sai (nghịch thế), tiến hành hoán đổi chúng.
4. Tiếp tục thực hiện cho đến khi không còn cặp nghịch thế nào trong danh sách.

```
void InterchangeSort(int a[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (a[i] > a[j]) {
                int temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

Đánh giá thuật toán:

- Số phép so sánh: Không phụ thuộc vào tình trạng ban đầu. Với mỗi phần tử, thuật toán so sánh nó với các phần tử còn lại, dẫn đến tổng số phép so sánh là n * (n - 1) / 2.
- Số phép hoán vị: Phụ thuộc vào tình trạng ban đầu của danh sách. Trong trường hợp xấu nhất (danh sách sắp xếp ngược), số phép hoán vị là n * (n - 1) / 2. Trong trường hợp tốt nhất (danh sách đã sắp xếp), không cần thực hiện hoán vị nào.
- Độ phức tạp thời gian: $O(n^2)$, do phải thực hiện số lượng lớn phép so sánh và hoán vị khi kích thước danh sách tăng.
  Mặc dù Interchange Sort dễ hiểu và triển khai, nhưng do hiệu suất không cao với các danh sách lớn, nó thường được sử dụng cho các tập dữ liệu nhỏ hoặc khi đơn giản là ưu tiên hàng đầu.

Bubble Sort và Shake Sort

Sắp xếp nổi bọt (Bubble Sort) là một thuật toán sắp xếp đơn giản và dễ hiểu, thường được giới thiệu trong các khóa học nhập môn về lập trình và cấu trúc dữ liệu. Thuật toán này hoạt động bằng cách liên tục so sánh các cặp phần tử liền kề và hoán đổi chúng nếu chúng không theo thứ tự mong muốn. Quá trình này được lặp lại cho đến khi danh sách được sắp xếp hoàn toàn.

Nguyên lý hoạt động:

1. Bắt đầu từ đầu danh sách, so sánh cặp phần tử đầu tiên.
2. Nếu phần tử đứng trước lớn hơn phần tử đứng sau (đối với sắp xếp tăng dần), hoán đổi chúng.
3. Tiếp tục với cặp phần tử tiếp theo cho đến cuối danh sách.
4. Sau mỗi lần duyệt, phần tử lớn nhất sẽ "nổi" lên vị trí cuối cùng của danh sách.
5. Lặp lại quá trình cho phần còn lại của danh sách (không bao gồm phần đã được sắp xếp) cho đến khi không cần hoán đổi nào nữa.

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                std::swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```

Đánh giá thuật toán:

- Độ phức tạp thời gian:
  Trường hợp tốt nhất: O(n) – xảy ra khi danh sách đã được sắp xếp.
  Trường hợp trung bình và xấu nhất: $O(n^2)$ – xảy ra khi danh sách ngẫu nhiên hoặc sắp xếp ngược.
  Độ phức tạp không gian: O(1) – sử dụng bộ nhớ bổ sung không đáng kể.
- Tính ổn định: Bubble Sort là thuật toán sắp xếp ổn định, nghĩa là nó giữ nguyên thứ tự tương đối của các phần tử có giá trị bằng nhau.
- Ưu điểm:
  Đơn giản, dễ hiểu và dễ triển khai.
  Hiệu quả với danh sách nhỏ hoặc gần như đã được sắp xếp.

- Nhược điểm:
  Hiệu suất kém với danh sách lớn do độ phức tạp thời gian cao.
  Không phù hợp cho các ứng dụng yêu cầu hiệu suất cao.

=> **Shake Sort**, còn được gọi là **Cocktail Shaker Sort**, là một cải tiến của thuật toán **Bubble Sort**. Sự cải tiến này giúp tăng hiệu suất sắp xếp bằng cách duyệt mảng theo cả hai chiều, thay vì chỉ một chiều như trong Bubble Sort.

Ý tưởng chính của Shaker Sort:

1. Duyệt xuôi (từ trái sang phải): So sánh các cặp phần tử liền kề; nếu phần tử đứng trước lớn hơn phần tử đứng sau, hoán đổi chúng. Sau lần duyệt này, phần tử lớn nhất sẽ được đưa về vị trí cuối cùng của danh sách.
2. Duyệt ngược (từ phải sang trái): Tiếp tục so sánh các cặp phần tử liền kề; nếu phần tử đứng sau nhỏ hơn phần tử đứng trước, hoán đổi chúng. Sau lần duyệt này, phần tử nhỏ nhất sẽ được đưa về vị trí đầu tiên của danh sách.
3. Thu hẹp phạm vi sắp xếp: Sau mỗi lượt duyệt xuôi và ngược, giới hạn phạm vi cần sắp xếp bằng cách loại bỏ các phần tử đã được đặt đúng vị trí ở đầu và cuối danh sách.
4. Lặp lại quá trình: Tiếp tục lặp lại các bước trên cho đến khi không còn phần tử nào cần sắp xếp.

Đánh giá thuật toán

- Ưu điểm:
  Hiệu quả hơn trên danh sách gần như đã sắp xếp: Nhờ việc di chuyển các phần tử lớn nhất và nhỏ nhất về đúng vị trí trong mỗi lượt duyệt, Shaker Sort có thể giảm số lần duyệt cần thiết, đặc biệt hiệu quả với các danh sách mà phần tử lớn nhất hoặc nhỏ nhất nằm gần đầu hoặc cuối.
  Phát hiện sớm danh sách đã sắp xếp: Nếu trong một lượt duyệt không có sự hoán đổi nào, thuật toán có thể kết luận rằng danh sách đã được sắp xếp và dừng lại.
- Nhược điểm:
  Độ phức tạp thời gian: Trong trường hợp xấu nhất, độ phức tạp thời gian của Shaker Sort vẫn là $O(n^2)$, tương tự như Bubble Sort.

II. Insertion

1. Insertion Sort

Sắp xếp chèn **(Insertion Sort)** là một thuật toán sắp xếp đơn giản và hiệu quả cho các tập dữ liệu nhỏ hoặc gần như đã được sắp xếp. Thuật toán này hoạt động tương tự như cách bạn sắp xếp các quân bài trên tay: lấy từng quân bài và chèn vào vị trí thích hợp trong phần đã được sắp xếp.

Nguyên lý hoạt động:

1. Bắt đầu với phần tử thứ hai của mảng (vì phần tử đầu tiên được coi là đã sắp xếp).
2. Lấy giá trị của phần tử hiện tại làm "khóa" (key).
3. So sánh "khóa" với các phần tử trước đó trong mảng đã sắp xếp.
4. Dịch chuyển các phần tử lớn hơn "khóa" sang bên phải để tạo chỗ trống.
5. Chèn "khóa" vào vị trí thích hợp.
6. Lặp lại quá trình cho đến khi toàn bộ mảng được sắp xếp.

```
void insertionSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            --j;
        }
        arr[j + 1] = key;
    }
}
```

Đánh giá thuật toán:

- Độ phức tạp thời gian:
  Trường hợp tốt nhất: O(n) – khi mảng đã được sắp xếp.
  Trường hợp trung bình và xấu nhất: $O(n^2)$ – khi mảng sắp xếp ngược.
- Ưu điểm:
  Dễ hiểu và cài đặt.
  Hiệu quả với các mảng nhỏ hoặc gần như đã sắp xếp.
  Là thuật toán sắp xếp ổn định (không thay đổi thứ tự của các phần tử có giá trị bằng nhau).
- Nhược điểm:
  Không hiệu quả với các mảng lớn do độ phức tạp $O(n^2)$.

2. Binary Insertion Sort và Shell Sort

Binary Insertion Sort và **Shell Sort** là những cải tiến dựa trên thuật toán **Insertion Sort**, nhằm nâng cao hiệu suất sắp xếp trong các trường hợp cụ thể.

Sắp xếp chèn nhị phân **(Binary Insertion Sort)** là một biến thể của thuật toán Sắp xếp chèn **(Insertion Sort)**, trong đó sử dụng tìm kiếm nhị phân để xác định vị trí chèn thích hợp cho mỗi phần tử. Mục tiêu của việc này là giảm số lần so sánh cần thiết khi tìm vị trí chèn, từ đó cải thiện hiệu suất của thuật toán.

Nguyên lý hoạt động:

1. Khởi tạo: Giả sử phần đầu tiên của mảng (gồm phần tử đầu tiên) đã được sắp xếp.
2. Duyệt mảng: Bắt đầu từ phần tử thứ hai (vị trí thứ nhất), lần lượt lấy từng phần tử trong mảng.
3. Tìm vị trí chèn: Sử dụng tìm kiếm nhị phân trên phần đã sắp xếp để xác định vị trí thích hợp cho phần tử hiện tại.
4. Chèn phần tử: Dịch chuyển các phần tử lớn hơn vị trí chèn sang phải một vị trí để tạo khoảng trống, sau đó chèn phần tử vào vị trí đã xác định.
5. Lặp lại: Tiếp tục quá trình cho đến khi toàn bộ mảng được sắp xếp.

Đánh giá thuật toán

- Ưu điểm:
  Giảm số lần so sánh: Sử dụng tìm kiếm nhị phân giúp giảm số lần so sánh từ O(n) xuống O(log n) cho mỗi lần chèn.
  Hiệu quả với mảng nhỏ hoặc gần như đã sắp xếp: Trong các trường hợp này, thuật toán hoạt động nhanh và hiệu quả.
- Nhược điểm:
  Độ phức tạp thời gian trung bình và xấu nhất vẫn là $O(n^2)$: Mặc dù số lần so sánh giảm, nhưng số lần dịch chuyển phần tử vẫn có thể lên đến O(n) cho mỗi lần chèn, dẫn đến tổng thể độ phức tạp không thay đổi.
  Không phù hợp cho mảng lớn: Với các mảng có kích thước lớn, các thuật toán sắp xếp khác như Quick Sort hoặc Merge Sort thường hiệu quả hơn.

Shell Sort là một thuật toán sắp xếp được cải tiến từ **Insertion Sort**, nhằm tăng hiệu quả sắp xếp bằng cách so sánh và hoán đổi các phần tử ở khoảng cách xa nhau trước khi giảm dần khoảng cách này. Phương pháp này giúp giảm thiểu số lần dịch chuyển phần tử, đặc biệt hiệu quả khi xử lý các tập dữ liệu lớn.

Nguyên lý hoạt động của Shell Sort:

1. Chia mảng thành các phần tử cách nhau một khoảng (gap): Bắt đầu với một giá trị khoảng cách gap lớn, thường được chọn bằng một nửa kích thước mảng, sau đó giảm dần gap theo một quy tắc nhất định cho đến khi gap bằng 1.
2. Sắp xếp các phần tử trong mỗi khoảng: Với mỗi giá trị gap, sắp xếp các phần tử cách nhau gap vị trí bằng cách sử dụng Insertion Sort.
3. Giảm khoảng cách và lặp lại: Giảm gap và lặp lại quá trình cho đến khi gap bằng 1. Lần sắp xếp cuối cùng với gap = 1 sẽ đảm bảo mảng được sắp xếp hoàn toàn.

Đánh giá thuật toán:

- Độ phức tạp thời gian: Phụ thuộc vào dãy gap được chọn. Trong trường hợp tốt nhất, độ phức tạp có thể đạt O(n log n), nhưng trong trường hợp xấu nhất có thể lên đến $O(n^2)$.

- Ưu điểm:
  Cải thiện hiệu suất so với Insertion Sort, đặc biệt với các mảng lớn.
  Dễ cài đặt và không yêu cầu bộ nhớ bổ sung.
- Nhược điểm:
  Hiệu suất phụ thuộc vào việc lựa chọn dãy gap tối ưu, điều này có thể phức tạp.

III. Selection

Selection Sort

Selection Sort (Sắp xếp chọn) là một thuật toán sắp xếp đơn giản hoạt động bằng cách chia danh sách thành hai phần: phần đã sắp xếp và phần chưa sắp xếp. Thuật toán sẽ liên tục tìm phần tử nhỏ nhất (hoặc lớn nhất) trong phần chưa sắp xếp và đưa nó về đúng vị trí trong phần đã sắp xếp.

Nguyên lý hoạt động:

1. Duyệt qua danh sách, tìm phần tử nhỏ nhất trong phần chưa sắp xếp.
2. Hoán đổi phần tử nhỏ nhất đó với phần tử đầu tiên của phần chưa sắp xếp.
3. Di chuyển ranh giới giữa phần đã sắp xếp và phần chưa sắp xếp sang phải.
4. Lặp lại cho đến khi toàn bộ danh sách được sắp xếp.

Đánh giá thuật toán:

- Độ phức tạp:
  Best case: $O(n^2)$
  Worst case: $O(n^2)$
  Average case: $O(n^2)$
  Không yêu cầu bộ nhớ phụ
- Nhược điểm:
  Không hiệu quả khi làm việc với danh sách lớn vì độ phức tạp thời gian là $O(n^2)$.
- Ưu điểm:
  Dễ hiểu, dễ triển khai.
  Không tốn thêm bộ nhớ ngoài vì chỉ sử dụng phép hoán đổi.
  Hoạt động tốt với danh sách nhỏ.

Heap Sort

Heap Sort có thể được xem là một bản nâng cấp so với **Selection Sort** về hiệu suất, vì nó vẫn duy trì ý tưởng chọn phần tử lớn nhất/nhỏ nhất để đưa về đúng vị trí, nhưng thay vì quét toàn bộ mảng để tìm giá trị nhỏ nhất (như **Selection Sort**), nó sử dụng cấu trúc dữ liệu **Heap** để làm điều đó một cách hiệu quả hơn.

Nguyên lý hoạt động:

1. Xây dựng một max heap (hoặc min heap) từ danh sách.
2. Trích xuất phần tử lớn nhất (hoặc nhỏ nhất) từ heap và đưa nó về cuối danh sách.
3. Heapify lại phần còn lại của heap để tiếp tục sắp xếp.
4. Lặp lại quá trình đến khi mảng được sắp xếp hoàn toàn.

Heap là một cây nhị phân hoàn chỉnh thỏa mãn tính chất:

1. **Max Heap:** Mỗi node cha có giá trị lớn hơn hoặc bằng các node con.
2. **Min Heap:** Mỗi node cha có giá trị nhỏ hơn hoặc bằng các node con.
   -> Heap giúp tìm phần tử lớn nhất (hoặc nhỏ nhất) nhanh chóng chỉ trong O(logn)

Đánh giá thuật toán:

- Ưu điểm:
  Hiệu suất tốt: Luôn chạy trong O(nlogn).
  Sắp xếp tại chỗ (In-place Sort): Không cần bộ nhớ phụ trợ nhiều.
  Hoạt động tốt với dữ liệu lớn.
- Nhược điểm:
  Không ổn định (Stable Sort): Có thể thay đổi vị trí của các phần tử có giá trị bằng nhau.
  Cài đặt phức tạp hơn so với Bubble Sort, Insertion Sort.

IV. Merge

Merge Sort

Merge Sort (Sắp xếp trộn) là một thuật toán chia để trị (Divide and Conquer), hoạt động bằng cách chia nhỏ danh sách thành các phần nhỏ hơn, sắp xếp từng phần rồi trộn (merge) chúng lại theo thứ tự đúng.

Nguyên lý hoạt động:

1. Chia: Chia mảng thành hai nửa bằng nhau.
2. Gọi đệ quy: Tiếp tục chia nhỏ hai nửa đó cho đến khi mỗi phần chỉ còn một phần tử (vì một phần tử luôn được coi là đã sắp xếp).
3. Trộn: Ghép hai danh sách con đã sắp xếp lại thành một danh sách lớn hơn theo đúng thứ tự.

Natural và K-Way Merge Sort

Natural Merge Sort là một biến thể của **Merge Sort**, trong đó thay vì chia danh sách thành các phần cố định, thuật toán sẽ tìm các chuỗi con đã sắp xếp sẵn (runs) trong danh sách và chỉ trộn chúng lại.

Nguyên lý hoạt động:

1. Tìm các dãy con đã sắp xếp sẵn (runs) trong danh sách.
2. Ghép các dãy con đó lại với nhau bằng Merge Sort cho đến khi toàn bộ danh sách được sắp xếp.

Ưu điểm: Nếu danh sách ban đầu có nhiều dãy con đã sắp xếp, thuật toán sẽ nhanh hơn Merge Sort tiêu chuẩn vì nó tận dụng các đoạn sắp xếp sẵn.

K-Way Merge Sort là một phiên bản mở rộng của Merge Sort, trong đó thay vì chia thành 2 phần (2-way Merge Sort), ta chia thành K phần và trộn chúng lại.

Nguyên lý hoạt động:

1. Chia danh sách thành K phần nhỏ hơn.
2. Đệ quy sắp xếp từng phần bằng Merge Sort.
3. Sử dụng K-Way Merge để hợp nhất K phần đã sắp xếp lại thành một danh sách hoàn chỉnh.

- **Ưu điểm:**
  Nhanh hơn 2-Way Merge Sort khi làm việc với dữ liệu lớn (ví dụ: khi xử lý file lớn trên ổ đĩa).
  Giảm độ sâu đệ quy, giúp thuật toán chạy nhanh hơn.
- **Nhược điểm:**
  Tốn bộ nhớ hơn, vì phải theo dõi nhiều dãy con cùng lúc.
  Cần một thuật toán trộn K mảng (K-Way Merge Algorithm) tối ưu để không làm giảm hiệu suất.

V. Specific

Counting Sort

Counting Sort (Sắp xếp đếm) là một thuật toán sắp xếp không dựa trên so sánh (non-comparison sort), hoạt động bằng cách đếm số lần xuất hiện của mỗi phần tử rồi sử dụng thông tin này để sắp xếp mảng.

Nguyên tắc hoạt động:

1. Tìm giá trị lớn nhất và nhỏ nhất trong mảng để xác định phạm vi giá trị cần đếm.
2. Tạo mảng đếm (count array) để lưu số lần xuất hiện của mỗi phần tử.
3. Cập nhật mảng đếm thành mảng vị trí (prefix sum array) để biết vị trí chính xác của mỗi phần tử trong mảng kết quả.

4. Duyệt ngược mảng gốc và điền phần tử vào đúng vị trí của nó trong mảng kết quả.

Đánh giá thuật toán:

- Ưu điểm:
  Nhanh hơn các thuật toán so sánh thông thường (O(nlogn)) khi phạm vi giá trị nhỏ.
  Ổn định (Stable Sort) nếu được triển khai đúng cách.
  Tốt cho dữ liệu có phạm vi giá trị nhỏ, đặc biệt là số nguyên.
- Nhược điểm:
  Không hiệu quả nếu phạm vi giá trị k quá lớn, làm tốn bộ nhớ.
  Chỉ hoạt động với số nguyên hoặc số có phạm vi nhỏ, không dùng cho số thực hoặc chuỗi dài.
  Không phải là thuật toán sắp xếp tại chỗ (In-Place Sort) vì cần mảng phụ trợ.

Radix Sort

Radix Sort là một thuật toán sắp xếp không dựa trên so sánh (non-comparison sort), hoạt động bằng cách sắp xếp các chữ số của số từ chữ số ít quan trọng nhất (Least Significant Digit - LSD) đến chữ số quan trọng nhất (Most Significant Digit - MSD).

Thuật toán này thường kết hợp với **Counting Sort** hoặc **Bucket Sort** để sắp xếp các chữ số trong từng bước.

Nguyên lý hoạt động:

1. Xác định chữ số lớn nhất trong số lớn nhất để biết cần bao nhiêu vòng lặp.
2. Bắt đầu từ chữ số nhỏ nhất (LSD), dùng Counting Sort để sắp xếp theo chữ số đó.
3. Lặp lại bước 2 với các chữ số tiếp theo cho đến khi hết.
4. Khi hoàn tất, mảng sẽ được sắp xếp đúng thứ tự.

Đánh giá thuật toán:

- Ưu điểm:
  Nhanh hơn Quick Sort hoặc Merge Sort trong một số trường hợp.
  Ổn định (Stable Sort), giữ nguyên thứ tự của các phần tử có cùng giá trị.
  Hiệu quả khi sắp xếp số nguyên hoặc dữ liệu có độ dài cố định.
- Nhược điểm:
  Không hoạt động tốt với số thực hoặc dữ liệu dạng chuỗi dài.
  Cần bộ nhớ phụ trợ (mảng đếm + output), không phải thuật toán in-place như Quick Sort.
  Không hiệu quả nếu giá trị số lớn và có nhiều chữ số (k lớn).