**Ý tưởng:**

**Selection Sort** : Giả sử sắp xếp mảng tăng dần. Mảng được chia thành 2 mảng con là mảng đã và chưa được sắp xếp. Selection Sort tìm phần tử nhỏ nhất trong mảng chưa được sắp xếp và đổi chỗ với phần tử ở đầu mảng chưa sắp xếp. Tức là mỗi vòng lặp, phần tử nhỏ nhất của mảng chưa sắp xếp sẽ được chuyển đến mảng đã sắp xếp.

**Insertion Sort:** Giả sử sắp xếp mảng tăng dần. Bắt đầu duyệt từ phần tử thứ 2 trở đi, so sánh phần tử hiện tại với phần tử trước nó. Nếu phần tử hiện tại lớn hơn thì đổi chỗ 2 phần tử với nhau, sau đó tiếp tục so sánh và đổi chỗ như vậy ở các vòng lặp tiếp theo sao cho đảm bảo tính tăng dần của mảng cho đến khi hoàn thành sắp xếp.

**Binary Insertion Sort:** Giả sử sắp xếp mảng tăng dần. Mảng được chia thành 2 mảng con đã và chưa được sắp xếp. Duyệt từ phần tử thứ 2 đến phần tử cuối cùng, phần tử hiện tại là key. Tiếp đó dùng Binary Search vào mảng đã được sắp xếp để tìm vị trí “pos” có giá trị lớn hơn key. Dịch chuyển toàn bộ phần tử từ “pos” về 1 sang phải để tạo vị trí trống cho key.

**Bubble Sort:** Giả sử sắp xếp mảng tăng dần. Ở mỗi vòng lặp, so sánh 2 phần tử liền kề và đổi chỗ nếu chúng xếp sai thứ tự tăng dần. Lặp lại cho đến khi hoàn thành sắp xếp mảng.

**Radix Sort:** Giả sử sắp xếp mảng tăng dần. Radix Sort so sánh từng chữ số với nhau từ chữ số bé nhất (hàng đơn vị) đến chữ số lớn nhất.

**Flash Sort:** Giả sử sắp xếp mảng tăng dần. Gồm có 3 bước. Bước 1 là Phân lớp dữ liệu dựa trên giả thiết, bước 2 là Hoán vị toàn cục ( đưa các phần tử về lớp của chúng), bước 3 là Sắp xếp bố cục ( sắp xếp trong phạm vi từng lớp).

**Counting Sort:** Giả sử sắp xếp mảng tăng dần. Đầu tiên đếm số lần xuất hiện của các phần tử trong mảng A và lưu vào mảng C, sau đó thay đổi chỉ số giới hạn mảng C. Cuối cùng duyệt từng phần tử của A vào mảng B chứa kết quả sắp xếp thông qua mảng C.

**Heap Sort:** Giả sử sắp xếp mảng tăng dần. Heap Sort sử dụng Binary Heap. Bước 1 là xây dựng heap từ dữ liệu đầu vào. Vào lúc này, node lớn nhất nằm ở gốc heap, thay thế nó bằng node cuối cùng của heap và giảm kích thước của heap đi 1. Tiếp theo lặp lại bước 2 miễn là kích thước của heap lớn hơn 1.

**Nhận xét:**

Selection Sort :

* Các trường hợp tốt nhất, trung bình, tệ nhất đều có độ phức tạp là O(n2 ).
* Một vòng lặp để tìm kiếm phần tử là O(n), một vòng lặp để so sánh các phần tử là O(n). Vậy độ phức tạp là O(n) x O(n) = O(n2).

Insertion Sort:

* Trường hợp tốt nhất : O(n).
* Trường hợp trung bình : O(n2).
* Trường hợp tệ nhất : O(n2). Sử dụng 2 vòng lặp, một vòng lặp để so sánh phần tử là O(n), một để đổi chỗ các phần tử với nhau cũng là O(n).

Binary Insertion Sort:

* Trường hợp tốt nhất : O(n).
* Trường hợp trung bình : O(n2).
* Trường hợp tệ nhất : O(n2). Sử dụng 2 vòng lặp, một vòng lặp để so sánh phần tử là O(n), một để đổi chỗ các phần tử với nhau là O(n).

Bubble Sort:

* Trường hợp tốt nhất : O(n).
* Trường hợp trung bình : O(n2).
* Trường hợp tệ nhất : O(n2). Sử dụng 2 vòng lặp, một vòng lặp để tìm kiếm phần tử là O(n), một vòng lặp để so sánh các phần tử với nhau là O(n).

Radix Sort:

* Trường hợp tốt nhất : O(nlogbn).
* Trường hợp trung bình : O((n+b) \* logb(k)) với k là giá trị lớn nhất có thể có, b là hệ cơ số( với hệ thập phân thì b=10).
* Trường hợp tệ nhất : O(n), với giá trị của b lớn và b=n. Lúc này việc sắp xếp cần nhiều thời gian hơn cho từng chữ số. Điều này làm cho không gian dùng cho Radix Sort nên ít được dùng trong các thư viện phần mềm.

Flash Sort:

* Tất cả các trường hợp đều có độ phức tạp là O(n).

Counting Sort:

* Tất cả trường hợp đều có độ phức tạp là O(n) vì chỉ sử dụng những vòng lặp đơn để duyệt và đếm phần tử.

Heap Sort:

* Độ phức tạp tổng thế của tất cả các trường hợp là O(nlogn). Độ phức tạp về thời gian của heap là O(logn) và độ phức tạp về thời gian của heapSort là O(n).

Nguồn :

<https://www.geeksforgeeks.org/>

<https://www.programiz.com/dsa>

<https://www.w3resource.com/javascript-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-12.php>

<https://www.neubert.net/Flacodes/FLACodes.html>