

# Thuật toán sắp xếp

---

## Thuật toán sắp xếp

### Giới thiệu

Ứng dụng về sắp xếp có ở khắp mọi nơi:

- Một danh sách lớp với các học sinh được sắp xếp theo thứ tự bảng chữ cái.
- Một danh bạ điện thoại.
- Danh sách các truy vấn được tìm kiếm nhiều nhất trên Google.

Thuật toán sắp xếp cũng được dùng kết hợp với những thuật toán khác, như tìm kiếm nhị phân, thuật toán Kruskal để tìm cây khung nhỏ nhất của đồ thị.

Vì sao chúng ta phải học nhiều thuật toán sắp xếp? Khi code, bạn chỉ cần biết cài một thuật toán sắp xếp là đủ. Hoặc nếu bạn code C++ hay Java, bạn chỉ cần biết cách gọi thư viện. Tuy nhiên, các thuật toán sắp xếp khác nhau cho ta nhiều ý tưởng hay và độc đáo - điều này vô cùng hữu ích khi các bạn học các thuật toán khác.

### Những điểm cần chú ý

Hãy thử tưởng tượng bạn có một bộ bài đã được xáo, và bạn muốn sắp xếp lại các lá bài theo thứ tự tăng dần. Bạn sẽ làm như nào? Có rất nhiều cách tiếp cận khác nhau:

- Chia bộ bài theo giá trị: 2, 3, 4... Rồi gộp lại.
- Trãi tất cả các lá bài ra, rồi lần lượt lấy lá bài nhỏ nhất.
- Chia bộ bài ra thành nhiều nhóm nhỏ. Với mỗi nhóm, sắp xếp lại, sau đó gộp các nhóm lại với nhau theo thứ tự tăng dần.

Bạn sẽ thấy những cách tiếp cận khác nhau sẽ có thời gian nhanh chậm khác nhau. Các thuật toán sắp xếp cũng vậy. Có rất nhiều cách tiếp cận, với ưu, nhược điểm khác nhau.

Khi so sánh giữa các thuật toán này với nhau, có nhiều vấn đề phải quan tâm.

1. **Thời gian** chạy. Đối với các dữ liệu rất lớn, các thuật toán không hiệu quả sẽ chạy rất chậm và không thể ứng dụng trong thực tế.

2. **Bộ nhớ.** Các thuật toán nhanh đòi hỏi đệ quy sẽ có thể phải tạo ra các bản copy của dữ liệu đang xử lý. Với những hệ thống mà bộ nhớ có giới hạn (ví dụ embedded system), một vài thuật toán sẽ không thể chạy được.
3. **Độ ổn định (stability).** Độ ổn định được định nghĩa dựa trên điều gì sẽ xảy ra với các phần tử có giá trị giống nhau.
  - ▶ Đối với thuật toán sắp xếp ổn định, các phần tử bằng với giá trị bằng nhau sẽ giữ nguyên thứ tự trong mảng trước khi sắp xếp.
  - ▶ Đối với thuật toán sắp xếp không ổn định, các phần tử có giá trị bằng nhau sẽ có thể có thứ tự bất kỳ.

Trong bài viết này, ta giả sử cần sắp xếp tăng dần các phần tử. Để sắp xếp giảm dần, ta có nhiều cách:

- ▶ Sửa đổi thuật toán một cách phù hợp.
- ▶ Sắp xếp, sau đó đảo ngược thứ tự các phần tử.
- ▶ Định nghĩa lại việc so sánh nhỏ hơn.

## Sắp xếp nổi bọt (Bubble sort)

Đây là thuật toán cơ bản nhất cho việc sắp xếp.

### Ý tưởng

- ▶ Xét lần lượt các cặp 2 phần tử liên tiếp. Nếu phần tử đứng sau nhỏ hơn phần tử đứng trước, ta đổi chỗ 2 phần tử. Nói cách khác, phần tử nhỏ nhất sẽ **nổi** lên trên.
- ▶ Lặp lại đến khi không còn 2 phần tử nào thỏa mãn. Có thể chứng minh được số lần lặp không quá  $N - 1$ , do một phần tử chỉ có thể **nổi** lên trên không quá  $N - 1$  lần.

### Ưu điểm

- ▶ Code đơn giản, dễ hiểu
- ▶ Không tốn thêm bộ nhớ


### Nhược điểm

- ▶ Độ phức tạp  $\mathcal{O}(N^2)$ , không đủ nhanh với dữ liệu lớn.

### Code

```
1  for (int i = 0; i < n; i++)
2      for (int j = 0; j < n - 1; j++)
3          if (a[j] > a[j+1]) {
4              swap(a[j], a[j+1]);
5          }
```

## Minh họa

Bạn có thể vào [VisuAlgo](#) .

- Chọn **Bubble** ở thanh menu bên trên.
- Ấn vào nút **Create** ở phía dưới trang để tạo một dãy mới
- Ấn vào **Sort**, rồi **Go** để chạy thuật toán.

## Sắp xếp chèn (Insertion Sort)

### Ý tưởng

Ý tưởng chính của thuật toán là ta sẽ sắp xếp lần lượt từng đoạn gồm 1 phần tử đầu tiên, 2 phần tử đầu tiên, ...,  $N$  phần tử.

Giả sử ta đã sắp xếp xong  $i$  phần tử của mảng. Để sắp xếp  $i + 1$  phần tử đầu tiên, ta tìm vị trí phù hợp của phần tử thứ  $i + 1$  và "chèn" nó vào đó.

### Ưu điểm

- Nếu danh sách đã gần đúng thứ tự, Insertion Sort sẽ chạy rất nhanh. Ví dụ bạn cần sắp xếp Highscore trong game.


### Nhược điểm

- Độ phức tạp  $\mathcal{O}(N^2)$ , không đủ nhanh với dữ liệu lớn.

## Code

```
1  for (int i = 1; i < n; i++) {
2      // Tìm vị trí phù hợp cho i
3      int j = i;
4      while (j > 0 && data[i] < data[j-1]) --j;
5
6      // Đưa i về đúng vị trí
7      int tmp = data[i];
8      for (int k = i; k > j; k--)
9          data[k] = data[k-1];
10     data[j] = tmp;
11 }
```

## Minh họa

Bạn có thể vào [VisuAlgo](#) .

- Chọn **Insert** ở thanh menu bên trên.
- Ấn vào nút **Create** ở phía dưới trang để tạo một dãy mới
- Ấn vào **Sort**, rồi **Go** để chạy thuật toán.

## Sắp xếp trộn (Merge sort)

### Ý tưởng

Sắp xếp trộn hoạt động kiểu đệ quy:

- Đầu tiên chia dữ liệu thành 2 phần, và sắp xếp từng phần.
- Sau đó gộp 2 phần lại với nhau. Để gộp 2 phần, ta làm như sau:
  - Tạo một dãy  $A$  mới để chứa các phần tử đã sắp xếp.
  - So sánh 2 phần tử đầu tiên của 2 phần. Phần tử nhỏ hơn ta cho vào  $A$  và xóa khỏi phần tương ứng.
  - Tiếp tục như vậy đến khi ta cho hết các phần tử vào dãy  $A$ .

### Ưu điểm

- Chạy nhanh, độ phức tạp  $\mathcal{O}(N * \log N)$ .
- Ổn định

### Nhược điểm

- Cần dùng thêm bộ nhớ để lưu mảng  $A$ .

### Code


```
1 | int a[MAXN]; // mảng trung gian cho việc sắp xếp
2 |
3 | // Sắp xếp các phần tử có chỉ số từ left đến right của mảng data.
4 | void mergeSort(int data[MAXN], int left, int right) {
5 |     if (data.length == 1) {
6 |         // Dãy chỉ gồm 1 phần tử, ta không cần sắp xếp.
7 |         return ;
8 |     }
9 |     int mid = (left + right) / 2;
10 |    // Sắp xếp 2 phần
11 |    mergeSort(data, left, mid);
12 |    mergeSort(data, mid+1, right);
13 | }
```

```

14
15 // Trộn 2 phần đã sắp xếp lại
16 int i = left, j = mid + 1; // phần tử đang xét của mỗi nửa
17 int cur = 0; // chỉ số trên mảng a
18
19 while (i <= mid || j <= right) { // chừng nào còn 1 phần chưa hết phần tử.
20     if (i > mid) {
21         // bên trái không còn phần tử nào
22         a[cur++] = data[j++];
23     } else if (j > right) {
24         // bên phải không còn phần tử nào
25         a[cur++] = data[i++];
26     } else if (data[i] < data[j]) {
27         // phần tử bên trái nhỏ hơn
28         a[cur++] = data[i++];
29     } else {
30         a[cur++] = data[j++];
31     }
32 }
33
34 // copy mảng a về mảng data
35 for (int i = 0; i < cur; i++)
36     data[left + i] = a[i];
37 }

```

## Minh họa

Bạn có thể vào [VisuAlgo](#) .

- Chọn **Merge** ở thanh menu bên trên.
- Ấn vào nút **Create** ở phía dưới trang để tạo một dãy mới
- Ấn vào **Sort**, rồi **Go** để chạy thuật toán.

## Sắp xếp vun đống (HeapSort)

### Ý tưởng

Ta lưu mảng vào CTDL **Heap**.

Ở mỗi bước, ta lấy ra phần tử nhỏ nhất trong heap, cho vào mảng đã sắp xếp.

### Ưu điểm

- Cài đặt đơn giản nếu đã có sẵn thư viện Heap.

- Chạy nhanh, độ phức tạp  $\mathcal{O}(N * \log N)$ .

## Nhược điểm

- Không ổn định

## Code

```
1 | Heap h = Heap();
2 | for (int i = 0; i < n; i++) {
3 |     // thêm phần tử vào heap
4 |     h.push(data[i]);
5 | }
6 | int a[MAXN];
7 | for (int i = 0; i < n; i++) {
8 |     // lấy phần tử nhỏ nhất và cho vào mảng đã sắp xếp
9 |     a[i] = h.pop();
10 | }
```

## Sắp xếp nhanh (QuickSort)

### Ý tưởng

- Chia dãy thành 2 phần, một phần "lớn" và một phần "nhỏ".
  - Chọn một khóa **pivot**
  - Những phần tử lớn hơn **pivot** chia vào phần lớn
  - Những phần tử nhỏ hơn hoặc bằng **pivot** chia vào phần nhỏ.
- Gọi đệ quy để sắp xếp 2 phần.

### Ưu điểm

- Chạy nhanh (nhanh nhất trong các thuật toán sắp xếp dựa trên việc so sánh các phần tử). Do đó quicksort được sử dụng trong nhiều thư viện của các ngôn ngữ như Java, C++ (hàm `sort` của C++ dùng Intro sort, là kết hợp của Quicksort và Insertion Sort).


## Nhược điểm

- Tùy thuộc vào cách chia thành 2 phần, nếu chia không tốt, độ phức tạp trong trường hợp xấu nhất có thể là  $\mathcal{O}(N^2)$ . Nếu ta chọn pivot ngẫu nhiên, thuật toán chạy với độ phức tạp trung bình là  $\mathcal{O}(N * \log N)$  (trong trường hợp xấu nhất vẫn là  $\mathcal{O}(N^2)$ , nhưng ta sẽ không bao giờ gặp phải trường hợp đó).
- Không ổn định.

## Code

```
1 void quickSort(int a[], int left, int right) {
2     int i = left, j = right;
3     int pivot = a[left + rand() % (right - left)];
4     // chia dãy thành 2 phần
5     while (i <= j) {
6         while (a[i] < pivot) ++i;
7         while (a[j] > pivot) --j;
8
9         if (i <= j) {
10            swap(a[i], a[j]);
11            ++i;
12            --j;
13        }
14    }
15    // Gọi đệ quy để sắp xếp các nửa
16    if (left < j) quickSort(a, left, j);
17    if (i < right) quickSort(a, i, right);
18 }
```

## Minh họa

Bạn có thể vào [VisuAlgo](#) .

- Chọn **Quick** ở thanh menu bên trên.
- Ấn vào nút **Create** ở phía dưới trang để tạo một dãy mới
- Ấn vào **Sort**, rồi **Go** để chạy thuật toán.

## Sắp xếp cơ số (RadixSort)

### Ý tưởng

Khác với tất cả các thuật toán nêu trên, RadixSort không sử dụng việc so sánh 2 phần tử.

- Đầu tiên, thuật toán sẽ chia các phần tử thành các nhóm, dựa trên chữ số cuối cùng (hoặc dựa theo bit cuối cùng, hoặc vài bit cuối cùng).
- Sau đó ta đưa các nhóm lại với nhau, và được danh sách sắp xếp theo chữ số cuối của các phần tử. Quá trình này lặp đi lặp lại với chữ số ở cuối cho tới khi tất cả vị trí chữ số đã sắp xếp.


### Ưu điểm

- Có thể chạy nhanh hơn các thuật toán sắp xếp sử dụng so sánh. Ví dụ nếu ta sắp xếp các số nguyên 32 bit, và chia nhóm theo 1 bit, thì độ phức tạp là  $\mathcal{O}(N)$ . Trong trường hợp tổng quát, độ phức tạp là  $\mathcal{O}(N * \log(\max(a_i)))$

## Nhược điểm




- Không thể sắp xếp số thực.

## Minh họa

Bạn có thể vào [VisuAlgo](#) .

- Chọn **Radix** ở thanh menu bên trên.
- Ấn vào nút **Create** ở phía dưới trang để tạo một dãy mới
- Ấn vào **Sort**, rồi **Go** để chạy thuật toán.

## Nguồn tham khảo

- [Topcoder](#) 
- [VisuAlgo](#) 
- [Wikipedia](#) 

Được cung cấp bởi [Wiki.js](#)