Sorting Algorithms

Võ Lê Ngọc Thịnh

Ngày 22 tháng 2 năm 2025

Mục lục

1 Giới thiệu chung

1.1 Tổng quan

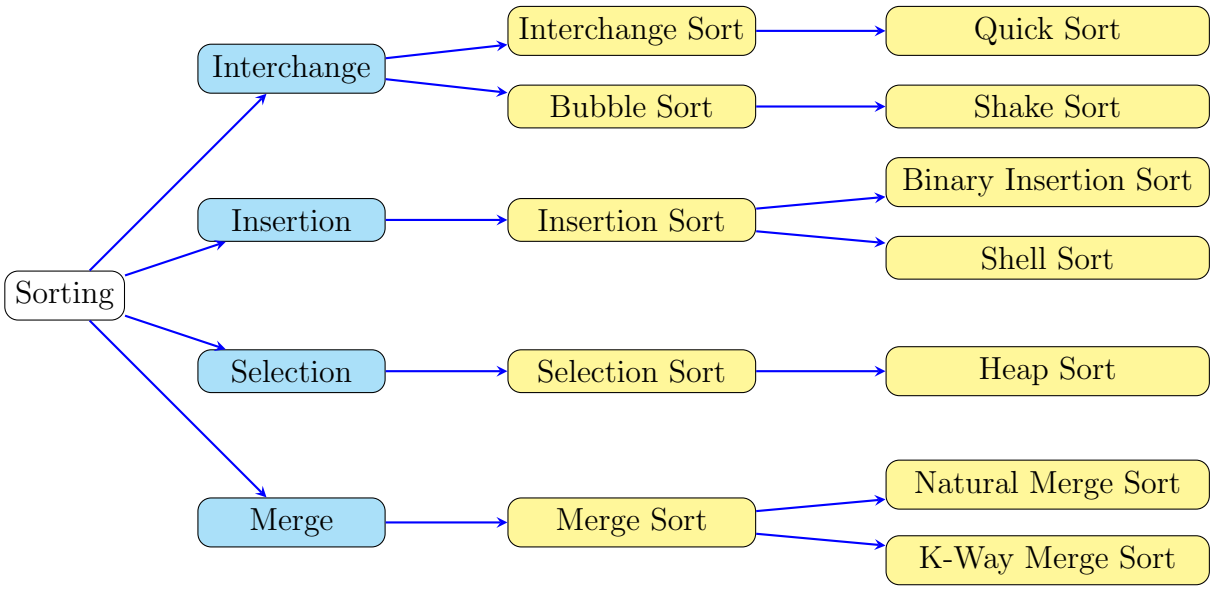
Sắp xếp (Sorting) là một trong những thao tác cơ bản nhất trong lĩnh vực Khoa học Máy tính và rất quan trọng trong nhiều ứng dụng thực tiễn. Mục tiêu của việc sắp xếp là đưa các phần tử của một dãy về trật tự đã định (thường là từ nhỏ đến lớn hoặc ngược lại).

Ý nghĩa của các thuật toán sắp xếp:

- Tiết kiệm thời gian trong quá trình tìm kiếm và truy xuất dữ liệu.
- Nền tảng cho các thuật toán phức tạp hơn.

1.2 Phân loại các thuật toán sắp xếp

Các thuật toán sắp xếp được phân loại dựa trên cách chúng thao tác và tái tổ chức lại dữ liệu. Mỗi thuật toán đều có những hướng tiếp cận khác nhau trong việc xử lí dữ liệu. Dưới đây là sơ đồ các nhóm thuật toán sắp xếp chính:



Trong sơ đồ trên, các thuật toán được chia làm bốn nhóm chính: **Interchange**, **Insertion**, **Selection** và **Merge** dựa trên ý tưởng về các thao tác cơ bản như đổi chỗ, chèn, lựa chọn và trộn.

2 Interchange

Interchange là phân loại dành cho những thuật toán thực hiện thao tác hoán đổi trực tiếp hai phần tử khi sắp xếp.

2.1 Interchange Sort

Ý tưởng của thuật toán này là xét tất cả các cặp phần tử trong dãy cần sắp xếp và thực hiện hoán đổi vị trí hai phần tử nếu chúng tạo thành một cặp nghịch thế.

Cách thực hiện:

- Xuất phát từ đầu dãy và tìm tất cả các cặp nghịch thế chứa phần tử này, triệt tiêu chúng bằng cách đổi chỗ phần tử này với phần tử tương ứng trong cặp nghịch thế.
- Lặp lại xử lý trên với các phần tử tiếp theo trong dãy.

Code minh họa:

```
1  void InterchangeSort(int arr[], int n) {
2      for (int i = 0; i < n; i++)
3          for (int j = i + 1; j < n; j++)
4              if (arr[i] > arr[j])
5                  swap(arr[i], arr[j]);
6  }
```

Độ phức tạp của thuật toán : O(n^2)

Đánh giá độ hiệu quả của thuật toán: Thuật toán đơn giản, dễ hiểu và cài đặt ngắn gọn nhưng khi đối mặt với bộ dữ liệu lớn thì thuật toán này chạy rất chậm vì phải xét qua tất cả các cặp phần tử có trong dãy, không quan tâm vào tình trạng ban đầu của các phần tử trong dãy.

Để nhìn rõ hơn điểm yếu của thuật toán, ta xét mảng $arr[5] = \{4, 2, 3, 1, 5\}$. Khi sử dụng thuật toán Interchange Sort để sắp xếp dãy, các cặp vị trí bị hoán đổi lần lượt là: (1, 2), (1, 4), (2, 3), (2, 4), (3, 4). Tuy nhiên, ta chỉ cần hoán đổi một cặp vị trí (1, 4) trên mảng ban đầu là ta đã có một mảng được sắp xếp tăng dần. Nhằm để khắc phục điểm yếu hoán đổi nhiều cặp nghịch thế vô ích, các nhà khoa học đã phát triển ra một thuật toán có tên là Quick Sort.

2.2 Quick Sort

Quick Sort được xây dựng với ý tưởng giảm số lần so sánh và hoán đổi của Interchange Sort bằng cách sử dụng khái niệm **pivot** và kĩ thuật **chia để trị**:

- Thay vì so sánh một phần tử với tất cả các phần tử khác, thuật toán chọn một phần tử gọi là **pivot**.
- Chia mảng làm hai phần:
  - Phần bên trái chứa các phần tử nhỏ hơn hoặc bằng **pivot**.
  - Phần bên phải chứa các phần tử lớn hơn **pivot**.
- Sau đó áp dụng đệ quy để sắp xếp riêng rẽ 2 phần tử này.

Ý tưởng này giúp thuật toán sử dụng ít phép biến đổi hơn so với Interchange Sort vì mỗi bước đều đưa **pivot** về đúng vị trí cuối cùng.

Code minh họa:

```
1   void QuickSort(int arr[], int l, int r){
2       int mid = (l + r)/2;
3       int pivot =  arr[mid];
4       int i = l, j = r;
5       while (i < j){
6           while (arr[i] < pivot)    i++;
7           while (arr[j] > pivot)    j--;
8           if (i <= j){
9               swap(arr[i], arr[j]);
10              i++, j--;
11          }
12      }
13      if (i < r)  QuickSort(arr, i, r);
14      if (l < j)  QuickSort(arr, l, j);
15  }
```

Độ phức tạp của thuật toán Quick Sort:

- Best case: $O(n)$
- Average case: $O(nlog(n))$
- Worse case: $O(n^2)$

2.3 Bubble Sort

Khác với thuật toán Interchange Sort, Bubble Sort chỉ thực hiện hoán đổi các cặp nghịch thế liền kề nhau, phương thức hoạt động nhìn giống như nổi bong bóng, các phần tử "nặng" sẽ chìm xuống dưới hoặc là các phần tử "nhẹ" sẽ nổi lên trên.

Cách thực hiện:

- Xuất phát từ cuối dãy, đổi chỗ các cặp phần tử kế cận để đưa phần tử nhỏ hơn trong cặp phần tử đó về vị trí đầu dãy hiện hành, sau đó sẽ không xét đến nó ở bước tiếp theo.
- Ở lần xử lý thứ i có vị trí đầu dãy là i.
- Lặp lại xử lý trên cho đến khi không còn cặp phần tử nào để xét.

Code minh họa:

```
1  void BubbleSort(int arr[], int n){
2      for (int i=0;i < n-1;i++ )
3          for (int j=n-1;j > i;j-- )
4              if (arr[j-1] > arr[j])
5                  swap(arr[j-1], arr[j]);
6  }
```

Độ phức tạp của thuật toán Bubble Sort: $O(n^2)$

Tuy rằng thuật toán Bubble Sort rất đơn giản và dễ hiểu, cài đặt ngắn gọn nhưng hiệu suất lại rất thấp do phải thực hiện nhiều phép so sánh và hoán đổi.

2.4 Shake Sort

Thuật toán Shake Sort được cải tiến với cách tiếp cận hai chiều (lượt xuôi và ngược) so với cách tiếp cận từ một chiều của thuật toán Bubble Sort:

- **Lượt duyệt xuôi:** Duyệt từ trái sang phải, đẩy phần tử lớn nhất về cuối mảng.
- **Lượt duyệt ngược:** Duyệt từ phải sang trái, đẩy phần tử nhỏ nhất về đầu mảng.

Code minh họa:

```
void ShakeSort(int arr[], int n){
    int l=0, r=n-1;
    while (l < r){
        for (int i=l;i < r;i++ )
            if (arr[i] > arr[i + 1])
                swap(arr[i], arr[i + 1]);
        r--;
        for (int j=r;j > l;j-- )
            if (arr[j - 1] > arr[j])
                swap(arr[j - 1], arr[j]);
        l++;
    }
}
```

Độ phức tạp của thuật toán: O(n^2)

Nhờ cải tiến tiếp cận từ cả hai đầu, thuật toán Shake Sort xử lý hiệu quả hơn Bubble Sort trong một số trường hợp, tuy nhiên độ phức tạp tổng thể trong trường hợp trung bình và xấu nhất của Shake Sort vẫn là O(n^2).

3 Insertion

Các thuật toán sắp xếp được phân loại vào nhóm **Insertion** đều có ý tưởng dựa trên "chèn phần tử vào đúng vị trí trong danh sách đã sắp xếp". Phần này bao gồm thuật toán Insertion Sort và hai thuật toán được cải tiến dựa trên Insertion Sort là Binary Insertion Sort và Shell Sort.

3.1 Insertion Sort

Insertion Sort là thuật toán cơ bản và phổ biến nhất trong nhóm sắp xếp chèn. Ý tưởng chính của thuật toán Insertion Sort là:

- Chia mảng thành 2 phần: Phần đã sắp xếp và phần chưa sắp xếp.
- Duyệt qua từng phần tử trong phần chưa sắp xếp, lấy từng phần tử đó và chèn vào đúng vị trí trong phần đã sắp xếp sao cho thứ tự sắp xếp (tăng dần hay giảm dần) được duy trì.
- Trong quá trình chèn, các phần tử lớn hơn (hoặc nhỏ hơn) trong phần đã sắp xếp phải được dịch chuyển để nhường chỗ cho phần tử đang xét.

Code minh họa:

```
1   void InsertionSort(int arr[], int n){
2       for (int i=1;i < n;i++ ){
3           int x = arr[i];
4           int pos = i;
5           while (pos > 0 && x < arr[pos - 1]){
6               arr[pos] = arr[pos - 1];
7               pos--;
8           }
9           arr[pos] = x;
10      }
11  }
```

Độ phức tạp của thuật toán:

- Best case: O(n)
- Average case: O(n^2)
- Worse case: O(n^2)

Thuật toán Insertion Sort hoạt động thực sự rất kém trên các mảng lớn vì:

- Tìm kiếm vị trí chèn chưa hiệu quả: Thuật toán sử dụng tìm kiếm tuyến tính để xác định vị trí cần chèn dẫn đến số phép so sánh rất lớn.

- Phép dịch chuyển chưa tối ưu: Khi chèn một phần tử mới, các phần tử trong dãy phải được dịch chuyển lên một vị trí để nhướng chỗ (đặc biệt khi dãy ban đầu là dãy giảm dần).

Các biến thể cải tiến hơn được ra đời như Binary Insertion Sort và Shell Sort để khắc phục các yếu điểm này.

3.2 Binary Insertion Sort

Thuật toán Binary Insertion Sort thay thế tìm kiếm tuyến tính trong Insertion Sort bằng **tìm kiếm nhị phân (binary search)** để tìm nhanh vị trí chèn. Điều này giảm số phép so sánh từ O(n) xuống O($log(n)$) cho mỗi phần tử.

Code minh họa:

```
    int BinarySearch(int arr[], int low, int high, int x){
        if (low >= high)
            return (arr[low] < x) ? (low + 1) : low;
        int mid = (left + right)/2;
        if (arr[mid] == x)  return (mid + 1);
        if (x > arr[mid])
            return BinarySearch(arr, mid + 1, high, x);
        return BinarySeach(arr, low, mid - 1, x);
    }

    void BinaryInsertionSort(int arr[], int n){
        for (int i=1;i < n;i++ ){
            int x = arr[i];
            int pos = BinarySearch(arr, 0, i - 1, x);
            for (int j=i;j>= pos;j-- )
                a[j] = a[j - 1];
            a[pos] = x;
        }
    }
```

Tuy giảm được số phép so sánh bằng tìm kiếm nhị phân nhưng số phép dịch chuyển phần tử vẫn giữ nguyên nên độ phức tạp thời gian về tổng thể vẫn là O(n^2).

3.3 Shell Sort

Không giống như Insertion Sort chỉ dịch chuyển các phần tử ở khoảng cách gần nhau dẫn đến hiệu suất thấp, thuật toán Shell Sort được cải tiến hơn nhờ:

- Sắp xếp các phần tử ở khoảng cách xa nhau trước (chia nhóm dựa theo khoảng cách).

- Trong mỗi nhóm con, các phần tử được sắp xếp nhằm đưa chúng "gần vị trí đúng" hơn. Kích thước của khoảng cách càng giảm (khi giảm gap), mảng trở nên ngày càng "gần đúng thứ tự".
- Khi gap = 1 (bước cuối cùng), Shell Sort thực hiện Insertion Sort trên mảng "gần sắp xếp", do đó giảm đáng kể số lần dịch chuyển.

Code minh họa:

```
void ShellSort(int arr[], int n){
    for (int k=n/2;k >= 1;k /= 2){
        for (int i=k;i < n;i++ ){
            int x = arr[i];
            int j = i;
            while (j >= k && arr[j - k] > x){
                arr[j] = a[j - k];
                j -= k;
            }
            arr[j] = x;
        }
    }
}
```

Độ phức tạp của thuật toán Shell Sort:

- Best case: O($nlog(n)$)
- Average case: O($nlog(n)$)
- Worse case: O(n^2)

4 Selection

Nhóm thuật toán **Selection** được gọi tên theo cách thức hoạt động chính của nó: lựa chọn phần tử phù hợp (nhỏ nhất hoặc lớn nhất) và đặt nó vào vị trí đúng trong danh sách qua mỗi lần lặp. Phần này bao gồm thuật toán Selection Sort và biến thể cải tiến là Heap Sort.

4.1 Selection Sort

Selection Sort là một thuật toán sắp xếp dựa trên ý tưởng đơn giản: tìm phần tử nhỏ nhất trong danh sách chưa được sắp xếp và hoán đổi nó với phần tử đầu tiên của danh sách đó. Sau đó, lặp lại quá trình này với phần còn lại của danh sách cho đến khi danh sách được sắp xếp hoàn toàn.

Code minh họa:

```
1    void SelectionSort(int arr[], int n){
2        for (int i=0;i < n;i++ ){
3            int minVal = i;
4            for (int j=i+1;j < n;j++ )
5                if (arr[minVal] > arr[j])
6                    minVal = j;
7            swap(arr[minVal], arr[i]);
8        }
9    }
```

Độ phức tạp của thuật toán Selection Sort: $O(n^2)$

Thuật toán này có một điểm yếu rất lớn là ở mỗi bước thứ i đều phải duyệt tuần tự trong dãy còn lại để tìm ra phần tử nhỏ nhất. Từ đó, một biến thể cải tiến của thuật toán Selection Sort có tên là Heap Sort được ra đời đã khắc phục được điểm này.

4.2 Heap Sort

Thuật toán Heap Sort về ý tưởng thì không khác gì so với thuật toán Selection Sort nhưng Heap Sort sử dụng cấu trúc dữ liệu Heap để thực hiện thao tác tìm phần tử nhỏ nhất của dãy còn lại. Việc sử dụng CTDL Heap làm giảm độ phức tạp tìm phần tử nhỏ nhất từ $O(n)$ về $O(log(n))$.

Code minh họa (Để dễ dàng cài đặt hơn, code ở đây sử dụng Max Heap và xây dựng mảng sắp xếp từ cuối về):

```
    void heapify(int arr[], int n, int i){
        int largest = i;
        int left = 2*i + 1, right = 2*i + 2;
        if (left < n && arr[left] > arr[largest])
            largest = left;
        if (right < n && arr[right] > arr[largest])
            largest = right;
        if (largest != i){
            swap(arr[largest], arr[i]);
            heapify(arr, n, largest);
        }
    }

    void HeapSort(int arr[], int n){
        for (int i=n/2 - 1;i>= 0;i-- )
            heapify(arr, n, i);

        for (int i=n - 1;i > 0;i-- ){
            swap(arr[i], arr[0]);
            heapify(arr, i, 0);
        }
    }
```

Độ phức tạp của thuật toán Heap Sort: O($nlog(n)$)

5 Merge

Thao tác **merge** là một kỹ thuật dùng để kết hợp hai mảng đã được sắp xếp trước thành một mảng mới sao cho mảng này cũng được sắp xếp.

Code minh họa thao tác merge hai dãy A và B đã được sắp xếp trước để tạo ra dãy C cũng được sắp xếp bằng phương pháp hai con trỏ:

```
int* merge(int a[], int n, int b[], int m){
    int* c = new int[n + m];
    int i = 0, j = 0, k = 0;
    while (i < n && j < m){
        if (a[i] <= b[j])
            c[k++] = a[i++];
        else c[k++] = b[j++];
    }
    while (i < n)
        c[k++] = a[i++];
    while (j < m)
        c[k++] = b[j++];
    return c;
}
```

Chúng ta có thể sử dụng hàm merge() được xây dựng sẵn trong thư viện algorithm thay vì cài đặt một cách thủ công.

Nhóm thuật toán sắp xếp Merge đều dựa trên thao tác merge cơ bản này và có những cải tiến riêng ở mỗi thuật toán.

5.1 Merge Sort

Merge Sort là một thuật toán sắp xếp sử dụng thao tác merge kết hợp phương pháp chia để trị. Nó chia dãy cần sắp xếp thành các phần nhỏ hơn, sắp xếp từng phần nhỏ, sau đó kết hợp chúng lại để tạo thành một dãy đã sắp xếp.

Code minh họa (Thao tác merge sử dụng hàm merge() trong thư viện algorithm):

```
void MergeSort(int arr[], int left, int right){
    if (left >= right)  return;
    int mid = (left + right)/2;
    MergeSort(arr, left, mid);
    MergeSort(arr, mid + 1, right);
    merge(arr + left, arr + mid + 1, arr + mid + 1,
          arr + right + 1, arr + left);
}
```

Độ phức tạp của thuật toán Merge Sort là O($nlog(n)$) và ổn định trong các trường hợp. Nhưng do đâu mà thuật toán này lại có độ ổn định trong tất cả các trường hợp (kể cả trường hợp tốt nhất) ?. Bởi vì ý tưởng cài đặt ban đầu của thuật toán này không quan tâm đến trạng thái mảng ban đầu, dù cho mảng ban đầu đã được sắp xếp sẵn nhưng thuật toán này liên tục thực hiện việc chia đôi mảng liên tục và thao tác merge. Và thuật toán Natural Merge Sort là một bản cải tiến nhỏ của thuật toán Merge Sort nhằm khắc phục yếu điểm này.

5.2 Natural Merge Sort

Natural Merge Sort là một cải tiến của thuật toán Merge Sort, tập trung vào việc tận dụng các đoạn liên tiếp đã được sắp xếp tự nhiên (natural runs) trong mảng đầu vào. Ý tưởng là thay vì luôn chia đôi mảng bất kể trạng thái của nó, thuật toán sẽ tự động nhận diện các đoạn liên tiếp trong mảng mà đã có thứ tự sắp xếp. Những đoạn con này sẽ được hợp lại theo từng bước cho đến khi toàn bộ mảng được sắp xếp hoàn chỉnh.

Code minh họa:

```
1  void NaturalMergeSort(int arr[], int n){
2      while (true){
3          vector<pair<int,int>> runs;
4          int start = 0;
5          while (start < n){
6              int end = start;
7              while (end + 1 < n && arr[end] <= arr[end + 1])
8                  end++;
9              runs.push_back({start, end});
10             start = end + 1;
11         }
12         if ((int)runs.size() <= 1)  break;
13         for (int i=0;i + 1 < (int)runs.size();i += 2){
14             int l = runs[i].first, mid = runs[i].second, r =
                   runs[i + 1].second;
15             merge(arr + l, arr + mid + 1, arr + mid + 1, arr
                   + r + 1, arr + l);
16         }
17     }
18 }
```

Độ phức tạp của thuật toán Natural Merge Sort:

- Best case: O(n)
- Average case: O($nlog(n)$)
- Worse case: O($nlog(n)$)

Mặc dù thuật toán Natural Merge Sort giảm được số lần thao tác merge nhờ tận dụng các đoạn đã sắp xếp tự nhiên trong mảng đầu vào, nhưng về bản chất, phương pháp này vẫn giữ nguyên cấu trúc và cơ chế của Merge Sort. Do đó, trong trường hợp trung bình, độ phức tạp thời gian của thuật toán vẫn là O($nlog(n)$).

5.3 K-Way Merge Sort

K-way Merge Sort là phiên bản cải tiến của thuật toán Merge Sort, thay vì chia mảng thành 2 phần như trong Merge Sort, ta chia mảng thành K phần gần bằng nhau. Mỗi phần được sắp xếp riêng lẻ trước khi được hợp nhất lại với nhau bằng cơ chế K-way merge – trong đó, các phần tử từ từng đoạn được gộp lại theo thứ tự tăng dần nhờ việc sử dụng cấu trúc dữ liệu như min-heap để liên tục chọn phần tử nhỏ nhất. Cách tiếp cận này giúp tối ưu hóa quá trình sắp xếp khi xử lý các đoạn nhỏ, từ đó cải thiện hiệu suất tổng thể của thuật toán trong các trường hợp chia nhỏ dữ liệu thành nhiều mảng con.

Tham khảo cách cài đặt của thuật toán K-Way Merge Sort.

Độ phức tạp trung bình của thuật toán K-Way Merge Sort là O($nlog_k(n)$).