THUẬT TOÁN SẮP XẾP

Chu Quang Cường - 24520236

FEBRUARY 18, 2025

ZALO: 0398749556

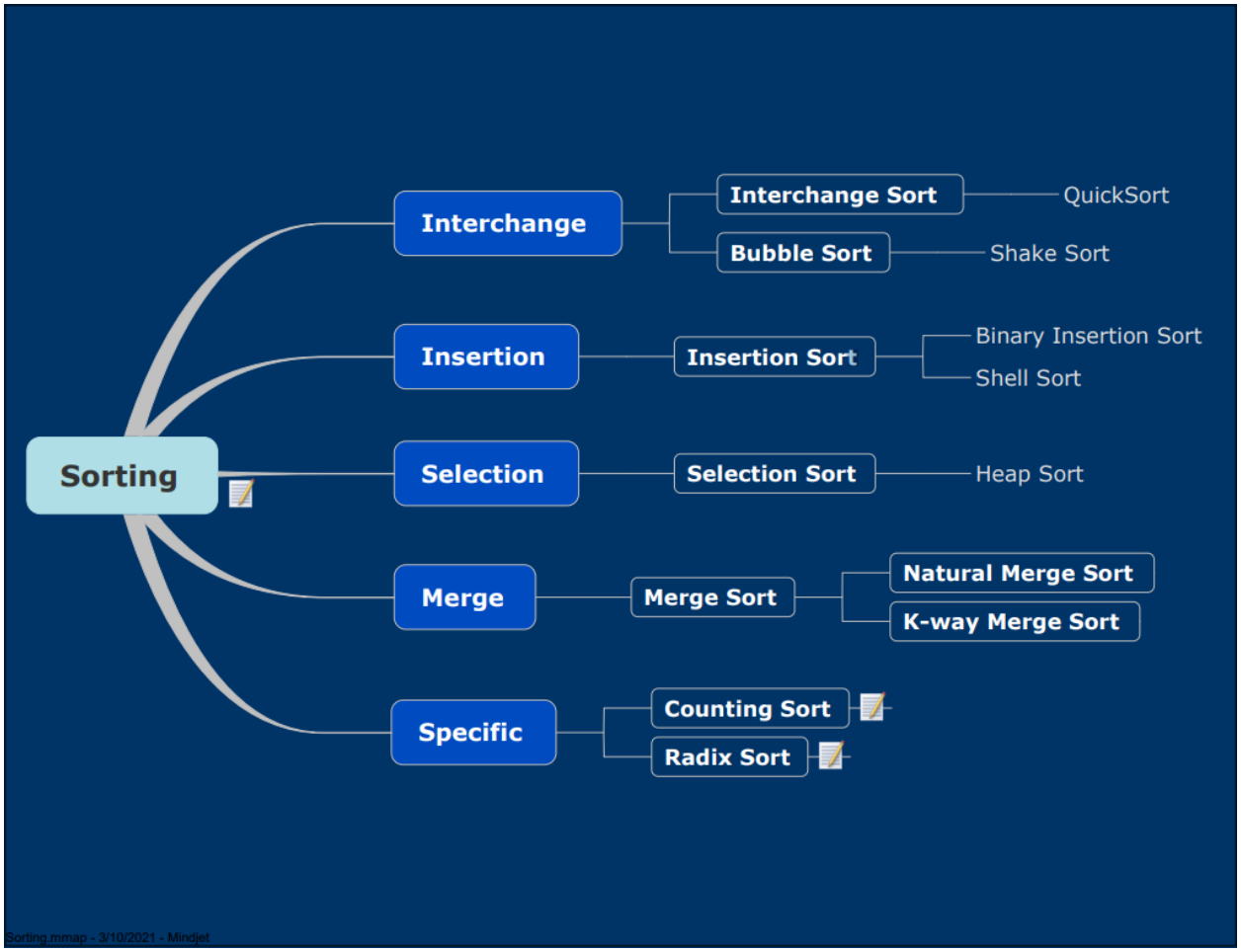
Email: cuongchu200@gmail.com

Mục Lục:

Lời mở đầu

Thuật toán sắp xếp được sử dụng để sắp xếp lại các phần tử của một mảng (hoặc một danh sách) theo thứ tự tăng (hoặc giảm).

Dưới đây là một số thuật toán sắp xếp mà mình sẽ đi tìm hiểu để làm rõ hơn những ưu, nhược điểm và điểm khác biệt của từng cái so với những thuật trước đó.



1. Interchange

Interchange là phương pháp đổi chỗ trực tiếp đối với các cặp phần tử trong mảng theo một thuật toán nào đó để hoàn thành việc sắp xếp theo thứ tự cần thiết.

1.1. Interchange Sort

Trước khi đề cập tới thuật toán này, chúng ta sẽ nhắc lại khái niệm nghịch thế:

Xét một mảng các số $a[0], a[1], \ldots a[n-1]$. Nếu có i < j và $a[i] > a[j]$, thì ta gọi đó cặp số (i, j) là một nghịch thế. Qua đó, ta rút ra nhận xét:

- Nếu mảng có tất cả cặp số ($\binom{n}{2}$ cặp) đều là nghịch thế thì đó là mảng giảm dần.
- Nếu mảng không có nghịch thế nào thì đó là mảng tăng dần.

Interchange Sort là thuật toán sắp xếp bằng cách xét tất cả các cặp để tìm các nghịch thế trong dãy và làm triệt tiêu dần chúng đi (hoặc ngược lại nếu muốn mảng giảm).

Cài đặt thuật toán:

```
void InterchangeSort(vector<int> &arr)
{
    int n = arr.size();
    // Xét tất cả các cặp trong mảng
    for (int i = 0; i < n - 1; i++)
        for (int j = i + 1; j < n; j++)
            // Triệt tiêu các cặp nghịch thế
            if(arr[i] > arr[j])
                swap(arr[i], arr[j]);
}
```

Độ phức tạp của thuật toán: $O(n^2)$

Ưu điểm: Đúng, thuật toán đơn giản.

Nhược điểm: Vì để tìm tất cả các cặp nghịch thế thì phải xét tất cả các cặp trong mảng nên nhiều khi bị triệt tiêu những cặp không lặp lại và làm chậm quá trình tính toán.

Do đó, để cải tiến thuật toán Interchange Sort, người ta đã phát minh ra thuật toán sau.

1.1.1. QuickSort

QuickSort là thuật toán cải tiến của Interchange Sort vì nó cũng triệt tiêu các cặp nghịch thế và được cải tiến bằng cách sử dụng nguyên lý chia để trị.

Cụ thể, thuật toán này có ba bước:

1. Chọn *Pivot* từ mảng (có thể là 1 phần tử bất kỳ trong mảng)
2. Sắp xếp lại vị trí sao cho tất cả các phần tử nhỏ hơn *pivot* nằm bên trái của nó.
3. Đệ quy quy trình tương tự cho hai mảng con (trái và phải của *pivot*) và dừng lại khi chỉ còn lại một phần tử trong mảng con.

Cài đặt thuật toán:

```
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[low];
    int i = low - 1, j = high + 1;

    while (true) {

        // Tìm phần tử bên trái ngoài cùng của dãy
        // mà lớn hơn hoặc bằng pivot
        do {
            i++;
        } while (arr[i] < pivot);

        // Tìm phần tử bên phải ngoài cùng của dãy
        // mà bé hơn hoặc bằng pivot
        do {
            j--;
        } while (arr[j] > pivot);

        /// Nếu hai phần tử đó gặp nhau
        if (i >= j)
            return j;

        // Nếu không thì sắp xếp cho đúng
        swap(arr[i], arr[j]);
    }
}
```

```
void QuickSort(vector<int>& arr, int low, int high) {

    if (low < high) {
        // pi là chỉ số pivot trước đã được
        // đặt ở đúng vị trí
        int pi = partition(arr, low, high);

        // Độc lập thực hiện sắp xếp cho hai phần
        // trái và phải của pivot như vậy
        QuickSort(arr, low, pi);
        QuickSort(arr, pi + 1, high);
    }
}
```

Độ phức tạp của thuật toán:

- Tốt nhất: $O(n)$
- Trung bình: $O(n \log n)$
- Tệ nhất: $O(n^2)$

1.2. Bubble Sort

Bubble Sort là thuật toán sắp xếp đơn giản hoạt động bằng cách hoán đổi liên tục các phần tử liền kề nếu chúng không đúng thứ tự.

Cài đặt thuật toán:

```cpp
void BubbleSort(vector<int>& arr) {
    int n = arr.size();
    // Xét tất cả các cặp liền kề trong mảng
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            // Sắp xếp các cặp này
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}
```

Độ phức tạp của thuật toán: $O(n^2)$

Ưu điểm: Thuật toán đơn giản và dễ hiểu.

Nhược điểm: Độ phức tạp lớn.

Để cải tiến thuật toán Bubble Sort, người ta đã phát minh ra thuật toán sau.

1.2.1. Shake Sort

Thuật toán Shake Sort là một phiên bản cải tiến của Bubble Sort. Shake Sort phá vỡ các rào cản của Bubble Sort khi làm việc không đủ hiệu quả trên các mảng lớn bằng cách không cho phép chúng trải qua các lần lặp không cần thiết trên một vùng cụ thể (hoặc cụm) trước khi chuyển sang phần khác của mảng.

Shake Sort duyệt qua một mảng đã cho theo cả hai hướng luân phiên.

Theo đó, có 3 bước tạo nên thuật toán này:

1. Lặp qua mảng từ trái sang phải và sắp xếp liên tục các cặp liền kề. Cuối lần lặp này, phần tử lớn nhất sẽ nằm ở cuối mảng.
2. Lặp qua mảng từ phải sang trái và sắp xếp liên tục các cặp liền kề. Cuối lần lặp này, phần tử nhỏ nhất sẽ nằm ở đầu mảng.
3. Tiếp tục lặp lại quy trình trên cho đến khi nhận được dãy đã được sắp xếp.

Cài đặt thuật toán:

```cpp
void ShakeSort(vector<int> &arr)
{
    int n = arr.size();
    int l = 0;
    int r = n - 1;
    int k = n - 1;
    while (l < r)
    {
        // Duyệt mảng từ trái sang phải
        for (int i = l; i < r; i++)
        {
            if (arr[i] > arr[i + 1])
            {
                swap(arr[i], arr[i + 1]);
                int k = i;
            }
        }
        // Thiết lập cận phải mới
        r = k;
        // Duyệt mảng từ phải sang trái
        for (int i = r; i > l; i--)
        {
            if (arr[i] < arr[i - 1])
            {
                swap(arr[i], arr[i - 1]);
                int k = i;
            }
        }
        // Thiết lập cận trái mới
        l = k;
    }
}
```

Độ phức tạp của thuật toán:

- Tốt nhất: $O(n)$
- Trung bình: $O(n^2)$
- Tệ nhất: $O(n^2)$

Dù đã được cải tiến, trong trường hợp mảng có ngẫu nhiên phần tử thì Bubble Sort và Shake Sort cho ra thời gian sắp xếp gần tương đương nhau. Do đó, thuật toán này vẫn còn kém hiệu quả hơn rất nhiều các thuật toán sắp xếp khác.

2. Insertion

Insertion là phương pháp sử dụng phép chèn đối với một phần tử theo một thuật toán nào đó để sắp xếp mảng đã cho.

2.1. Insertion Sort

Insertion Sort là một thuật toán sắp xếp hoạt động bằng cách lặp lại việc chèn từng phần tử của một danh sách chưa được sắp xếp vào đúng vị trí của nó trong một phần đã được sắp xếp của danh sách.

Cài đặt thuật toán:

```
void InsertionSort(int arr[], int n){
    for(int i = 1; i < n; ++i){
        int k = arr[i];
        int j = i - 1;
        /* Chuyển các phần tử lớn hơn
        phần tử đang xét sang phải
        1 vị trí so với vị trí hiện tại */
        while (j >= 0 && arr[j] > k){
            arr[j + 1] = arr[j];
            j--;
        }
        /* Chèn phần tử đang xét
        vào vị trí đúng */
        arr[j + 1] = k;
    }
}
```

Độ phức tạp của thuật toán:

- Tốt nhất: $O(n)$
- Trung bình: $O(n^2)$
- Tệ nhất: $O(n^2)$

Ưu điểm: Đơn giản, hiệu quả với danh sách nhỏ hoặc gần như được sắp xếp.

Nhược điểm: Không hiệu quả với danh sách lớn.

Để cải tiến thuật toán Insertion Sort, người ta đã phát minh ra 2 thuật toán sau.

2.1.1. Binary Insertion Sort

Binary Insertion Sort là thuật toán cải tiến của Insertion Sort. Thay vì sử dụng tìm kiếm tuyến tính để tìm kiếm vị trí chèn phần tử, thuật toán này sử dụng tìm kiếm nhị phân. Điều này làm giảm giá trị so sánh của việc chèn một phần tử duy nhất từ $O(n)$ xuống $O(\log n)$.

Cài đặt thuật toán:

```
void BinaryInsertionSort(int arr[], int n){
    // Bắt đầu xét từ phần tử thứ 2
    for(int i = 1; i < n; i++){
        int k = arr[i];
        /* Dùng tìm kiếm nhị phân
        để tìm vị trí đúng của
        phần tử đang xét */
        int low = 0;
        int high = i - 1;
        while (low < high){
            int mid = (low + high) / 2;
            if (arr[mid] <= k){
                low = mid + 1;
            }
            else{
                high = mid;
            }
        }
        /* Chuyển các phần tử lớn hơn
        phần tử đang xét sang phải
        1 lần so với vị trí hiện tại */
        for(int j = i - 1; j >= low; j--){
            arr[j + 1] = arr[j];
        }
        // Chèn vào đúng vị trí
        arr[low] = k;
    }
}
```

Độ phức tạp của thuật toán:

- Tốt nhất: $O(n \log n)$
- Trung bình: $O(n^2)$
- Tệ nhất: $O(n^2)$

Như vậy, thuật toán này lại làm tăng độ phức tạp bởi vì dù đã tối ưu được phần tìm kiếm vị trí nhưng phần dịch mảng thì vẫn giữ nguyên. Do đó, nó có giá trị học thuật nhưng không được sử dụng nhiều trong thực tế khi sắp xếp mảng lớn hoặc dữ liệu ngẫu nhiên.

2.1.2. Shell Sort

Shell Sort là một thuật toán sắp xếp cải tiến từ Insertion Sort. Thuật toán này khắc phục nhược điểm chỉ di chuyển phần tử lên một vị trí liền kề bằng cách so sánh và hoán đổi phần tử cách nhau một khoảng lớn trước, sau đó giảm dần khoảng này về 1.

Thuật toán này có 4 bước chính:

1. Chọn một giá trị gap lớn.
2. Thực hiện Insertion Sort trên dãy có các phần tử cách nhau bởi gap.
3. Giảm gap.
4. Lặp lại quá trình 3 bước trên tới khi gap bằng 1.

Cài đặt thuật toán:

```
void ShellSort(int arr[], int n){
    //Tạo gap và bước nhảy của nó
    for(int gap = n / 2; gap > 0; gap /= 2){
        // Chia thành các dãy cách nhau 1 khoảng gap
        for(int i = gap; i < n; i++){
            int temp = arr[i];
            int j;
            // Insertion Sort với các phần tử cách nhau bởi gap
            for(j = i; j >= gap && arr[j - gap] > temp; j -= gap){
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}
```

Độ phức tạp của thuật toán:

- Tốt nhất: $O(n \log n)$
- Trung bình: $O(n \log n)$ (hoặc $O\left(n^{\frac{4}{3}}\right)$ nếu chọn gap tối ưu)
- Tệ nhất: $O(n^2)$ (hoặc $O\left(n^{\frac{3}{2}}\right)$ nếu chọn gap tối ưu)

Khi so sánh với Insertion Sort, Shell Sort có hiệu suất tốt hơn trên các tập dữ liệu lớn bằng cách giảm khoảng cách hoán đổi dần dần để tối ưu bước dịch mảng.

3. Selection

Sau 2 phương pháp kể trên, Selection là phương pháp sử dụng thao tác chọn theo một thuật toán nào đó để sắp xếp mảng theo thứ tự đúng.

3.1. Selection Sort

Với Selection Sort cho dãy ban đầu có n phần tử, ý tưởng thuật toán là thực hiện n-1 lượt việc đưa phần tử nhỏ nhất trong dãy về vị trí đúng ở đầu dãy.

Cài đặt thuật toán:

```
void SelectionSort(int arr[], int n){
    // Xét từng vị trí trong mảng
    for(int i = 0; i < n; i++){
        int pos = i;
        /* Tìm phần tử nhỏ nhất
        từ vị trí i tới n */
        for(int j = i; j < n; j++){
            if(arr[j] < arr[i])
                pos = j;
        }
        /* Đưa phần tử thứ i
        về đúng vị trí */
        swap(arr[i], arr[pos]);
    }
}
```

Độ phức tạp của thuật toán: $O(n^2)$

Ưu điểm: Đơn giản, dễ hiểu.

Nhược điểm: Độ phức tạp cao.

Để cải tiến thuật toán Selection Sort, người ta đã phát minh ra thuật toán sau.

3.1.1. Heap Sort

Heap Sort cải tiến Selection Sort bằng cách sử dụng cấu trúc Heap để tìm phần tử lớn nhất (hoặc nhỏ nhất) một cách hiệu quả hơn thay vì duyệt hết mảng.

Thuật toán gồm 4 bước:

1. Xây dựng Max-Heap.
2. Hoán đổi phần tử gốc với phần tử cuối.
3. Giảm kích thước Heap bằng cách xóa phần tử cuối (không xóa trên mảng thực tế)
4. Lặp lại 3 bước trên cho đến khi kích thước Heap về 1.

Cài đặt thuật toán:

```
void Heapify(int arr[], int n, int i){
    // Coi i là lớn nhất và là gốc
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    // Nếu nút con trái lớn hơn gốc
    if(left < n && arr[left] > arr[largest])
        largest = left;
    // Nếu nút con phải lớn hơn gốc
    if(right < n && arr[right] > arr[largest])
        largest = right;
    // Nếu số lớn nhất không phải là gốc
    if(largest != i){
        swap(arr[i], arr[largest]);
        // Tiếp tục đệ quy với cây con
        Heapify(arr, n, largest);
    }
}
```

```
void HeapSort(int arr[], int n){
    // Xây dựng Max-Heap
    for(int i = n / 2 - 1; i >= 0; i--)
        Heapify(arr, n, i);
    // Trích xuất từng phần tử từ Heap
    for(int i = n - 1; i > 0; i--){
        // Đưa gốc về cuối
        swap(arr[0], arr[i]);
        // Heapify lại gốc
        Heapify(arr, i, 0);
    }
}
```

Độ phức tạp của thuật toán: $O(n \log n)$

Heap Sort có cùng một độ phức tạp thời gian trong mọi trường hợp. Điều này làm cho nó hiệu quả ngay cả khi sắp xếp với các tập dữ liệu lớn.

4. Merge

Merge là phương pháp sử dụng phép trộn đối với 2 mảng con được tách ra với mục đích sắp xếp mảng đã cho.

4.1. Merge Sort

Merge Sort là thuật toán sắp xếp theo phương pháp chia để trị. Thuật toán này bắt đầu bằng việc đệ quy chia mảng thành các mảng con nhỏ hơn, rồi sắp xếp các mảng con đó, hợp nhất chúng lại và cuối cùng thu được mảng cần tìm.

Cài đặt thuật toán:

```
void merge(int arr[], int left, int mid, int right){
    int n1 = mid - left + 1, n2 = right - mid;
    int L[100], R[100];
    // Chia mảng ban đầu thành hai mảng con
    for(int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for(int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0;
    int k = left;
    /* Hợp nhất hai mảng con này
    tạo thành mảng mới được sắp xếp*/
    while(i < n1 && j < n2){
        if(L[i] <= R[j]){
            arr[k] = L[i];
            i++;
        }
        else{
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    /* Sao chép những phần tử còn lại của
    mảng con bên trái nếu còn */
    while(i < n1){
        arr[k] = L[i];
        i++;
        k++;
    }
    /* Sao chép những phần tử còn lại của
    mảng con bên phải nếu còn */
    while(j < n2){
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

```
void MergeSort(int arr[], int left, int right){
    if(left >= right)
        return;
    /*Chia ra hai mảng từ left sang right
    rồi đệ quy sắp xếp hai mảng đó và hợp nhất*/
    int mid = left + (right - left) / 2;
    MergeSort(arr, left, mid);
    MergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}
```

Độ phức tạp của thuật toán: $O(n \log n)$

Thuật toán này có độ ổn định cao và nhanh hơn so với các thuật toán được liệt kê bên trên. Tuy dễ hiểu và đơn giản nhưng cũng còn một số phần hơi cứng nhắc nên cũng đã có nhiều thuật toán cải tiến làm nó trở nên tốt hơn như sau.

4.1.1. Natural Merge Sort

Thuật toán Natural Merge Sort khắc phục thuật toán Merge Sort ở chỗ thay vì trộn cho tất cả các dãy thì ta sẽ chỉ thực hiện trộn với dãy chưa được sắp xếp.

Cài đặt thuật toán:

```
bool isSorted(int arr[], int left, int right){
    for(int i = left; i < right; i++){
        if(arr[i] > arr[i + 1])
            return false;
    }
}

Tabnine | Edit | Test | Explain | Document
void MergeSort(int arr[], int left, int right){
    if(left >= right)
        return;
    if(isSorted(arr, left, right))
        return;
    /*Chia ra hai mảng từ left sang right
    rồi đệ quy sắp xếp hai mảng đó và hợp nhất*/
    int mid = left + (right - left) / 2;
    MergeSort(arr, left, mid);
    MergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}
```

Độ phức tạp của thuật toán:

- Tốt nhất: $O(n)$
- Trung bình: $O(n \log n)$
- Tệ nhất: $O(n \log n)$

Thuật toán này cải thiện được phần đã được sắp xếp nên có thể cho ra trường hợp tốt nhất nhanh hơn so với Merge Sort thông thường.

4.1.2. K-way Merge Sort

K-way Merge Sort là thuật toán giải quyết vấn đề chính là hợp nhất k dãy số đã được sắp xếp thành 1 dãy duy nhất cũng được sắp xếp.

Ở đây, chúng ta có hai hướng tiếp cận cho vấn đề này:

1. Sử dụng Merge Sort (Cho các mảng có kích thước bằng nhau)

Ý tưởng giống như Merge Sort, đây chủ yếu là sắp xếp hợp nhất. Ở đây thay vì sắp xếp các phần tử, chúng ta sắp xếp các mảng. Chia K mảng thành hai nửa chứa số lượng mảng bằng nhau cho đến khi có hai mảng trong một nhóm. Sau đó, hợp nhất các mảng theo cách từ dưới lên.

Cài đặt thuật toán:

```
void MergeArrays(vector<int>& a, vector<int>& b, vector<int>& c){
    int i = 0, j = 0, k = 0;
    int n1 = a.size();
    int n2 = b.size();
    c.resize(n1 + n2);
    // Duyệt qua cả hai mảng
    while(i < n1 && j < n2){
        if(a[i] < b[j]){
            c[k] = a[i];
            k++;
            i++;
        }
        else{
            c[k] = b[i];
            k++;
            i++;
        }
    }
    // Duyệt các phần tử còn lại của a
    while(i < n1){
        c[k] = a[i];
        k++;
        i++;
    }
    // Duyệt các phần tử còn lại của b
    while(j < n2){
        c[k] = b[i];
        k++;
        i++;
    }
}
```

```cpp
void MergeKArrays1(vector<vector<int>>& arr, int low, int high, vector<int>& res){
    // Nếu chỉ còn một mảng
    if (low == high) {
        res = arr[low];
        return;
    }
    // Nếu chỉ còn hai mảng, hợp nhất chúng
    if (high - low == 1) {
        MergeArrays(arr[low], arr[high], res);
        return;
    }
    int mid = (low + high) / 2;
    // Chia mảng thành hai mảng con
    vector<int> out1, out2;
    MergeKArrays1(arr, low, mid, out1);
    MergeKArrays1(arr, mid + 1, high, out2);
    // Hợp nhất hai mảng con lại
    MergeArrays(out1, out2, res);
}
```

Độ phức tạp của thuật toán: $O(n * \log k)$

2. Sử dụng Min-Heap (Cho các mảng có kích thước khác nhau)

Thuật toán bắt đầu bằng cách tạo MinHeap và chèn phần tử đầu tiên của tất cả k mảng. Xóa phần tử gốc của MinHeap và đặt nó vào mảng đầu ra rồi sau đó chèn phần tử tiếp theo từ mảng phần tử đã xóa. Tiếp tục cho đến khi không còn phần tử nào trong MinHeap thì ta nhận được mảng đã được sắp xếp.

Cài đặt thuật toán:

```cpp
vector<int> MergeKArrays2(vector<vector<int>> arr){
    vector<int> output;
    // Đây là MinHeap với mỗi node đều có phần tử đầu tiên của mảng
    priority_queue<pair<int , int> , vector<pair<int , int>> , greater<pair<int , int>>> heap;

    for (int i = 0; i < arr.size(); i++)
        heap.push({ arr[i][0], { i, 0 } });
    /* Lần lượt lấy từng phần tử nhỏ nhất từ MinHeap
    và thay thế với phần tử tiếp theo của mảng đó*/
    while(heap.empty() == false){
        pair<int, pair<int, int> > curr = heap.top();
        heap.pop();
        // i là giá trị
        // j là chỉ số của giá trị tương ứng
        int i = curr.second.first;
        int j = curr.second.second;

        output.push_back(curr.first);
        // Phần tử tiếp theo thuộc cùng mảng hiện tại
        if (j + 1 < arr[i].size())
            heap.push({ arr[i][j + 1], { i, j + 1 } });
    }
    return output;
}
```

Độ phức tạp của thuật toán: $O(n * \log k)$

5. Specific

Ở đây, các thuật toán sẽ khác biệt và không có kiểu mẫu như những thuật toán trên, vì vậy sẽ mang lại sự thú vị và độc đáo của riêng nó.

5.1. Counting Sort

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 là đếm tần suất của từng phần tử riêng biệt trong mảng và sử dụng nó để đặt các phần tử vào đúng vị trí.

Cài đặt thuật toán:

```
vector<int> CountSort(int arr[], int n){
    int m = 0;

    for (int i = 0; i < m; i++)
        m = max(m, arr[i]);
    // Tạo chuỗi đếm m+1 giá trị 0
    vector<int> count(m + 1, 0);
    /* Tương ứng tần suất của từng
    phần tử trong arr qua mảng ount */
    for (int i = 0; i < n; i++)
        count[arr[i]]++;
    /* Tính tổng tiền tố của từng
    chỉ số ở mảng count */
    for (int i = 1; i <= m; i++)
        count[i] += count[i - 1];

    // Tạo vector đầu ra dựa trên hàm count
    vector<int> output(n);

    for(int i = n - 1; i >= 0; i--){
        output[count[arr[i]] - 1]= arr[i];
        count[arr[i]]--;
    }
    return output;
}
```

Độ phức tạp của thuật toán: $O(n + m)$

Trong đó, n là kích thước của mảng đầu vào, m là kích thước của hàm đếm (count).

Counting Sort thường thực hiện nhanh hơn tất cả các thuật toán sắp xếp dựa trên so sánh nếu phạm vi đầu vào có thứ tự bằng số đầu vào và sẽ không hiệu quả nếu phạm vi giá trị cần sắp xếp quá lớn.

5.2. Radix Sort

Radix Sort là thuật toán sắp xếp tuyến tính các phần tử bằng cách xử lý chúng theo từng chữ số. Đây là một thuật toán sắp xếp hiệu quả cho các số nguyên hoặc chuỗi với khóa có kích thước cố định.

Thay vì so sánh trực tiếp các phần tử, Radix Sort phân phối chúng thành các nhóm dựa trên giá trị của từng chữ số. Bằng cách sắp xếp lặp lại các phần tử theo chữ số cùng cấp, từ đơn vị đến chục đến …, thuật toán có thể sắp xếp được mảng ban đầu.

Cài đặt thuật toán:

```
void countSort(int arr[], int n, int exp){
    int output[n];
    int i, count[10] = { 0 };
    // lưu trữ số lần xuất hiện trong mảng count
    for (i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;
    /* Thay đổi count[i] để count[i]
    chứa vị trí thực của chữ số trong hàm output */
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];
    // Xây dựng hàm output
    for (i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }
    // Chép hàm output qua arr
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}
```

```
void RadixSort(int arr[], int n){
    // Tìm giá trị lớn nhất trong arr
    int mx = arr[0];
    for(int i = 1; i < n; i++){
        mx = max(mx, arr[i]);
    }
    // Dùng sắp xếp đếm cho từng chữ số
    for (int exp = 1; mx / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}
```

Độ phức tạp của thuật toán: $O(n * k)$

Trong đó, n là kích thước mảng đầu vào, k là số chữ số của số lớn nhất.