Sorting Algorithms

February 21, 2025

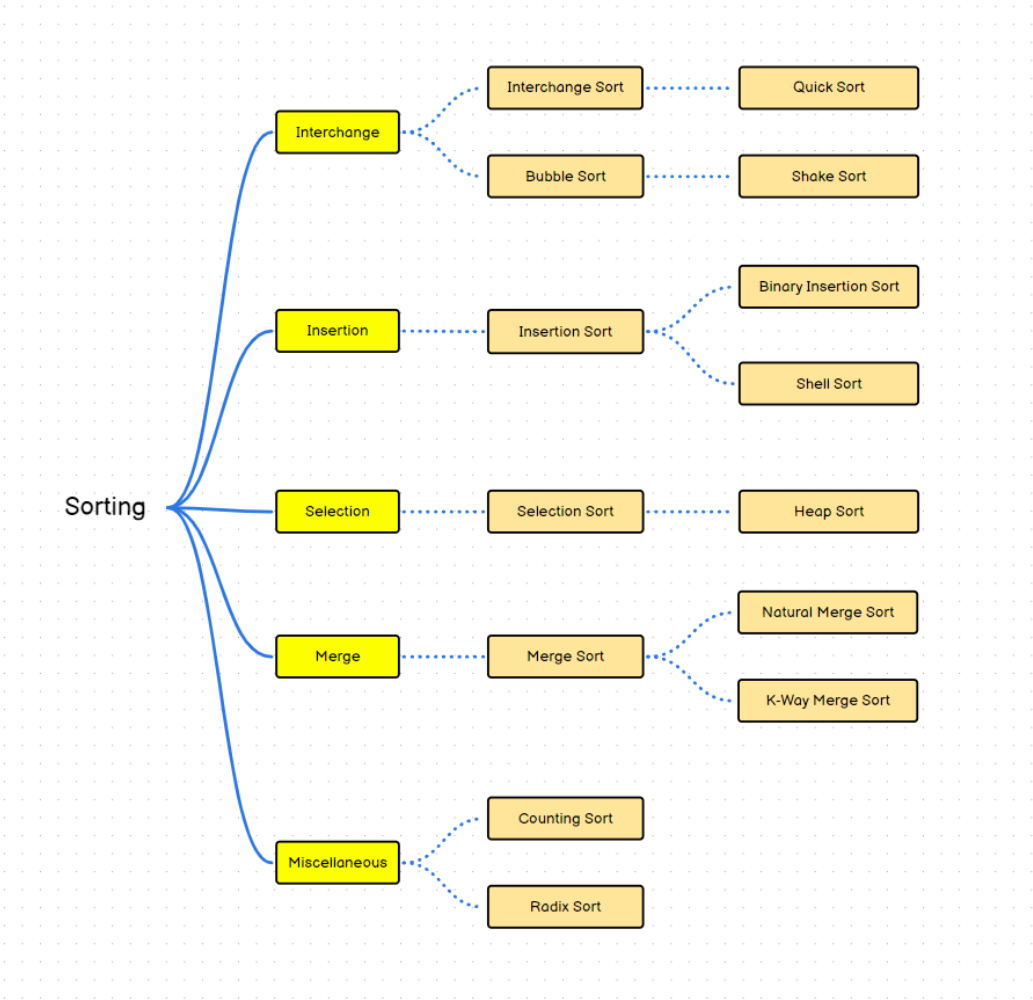
Mục lục

1 Mở đầu

Trong lập trình, bài toán sắp xếp và tìm kiếm là một vấn đề thú vị và không hề đơn giản vì thế nên ta sẽ cùng đi tìm hiểu và những thuật toán sắp xếp căn bản và thuật toán nâng cao hơn nhé.

Thứ tự các thuật toán sắp xếp sẽ được liệt kê như sơ đồ sau:

-



Trong sơ đồ trên, có 4 phân loại là **Interchange**, **Insertion**, **Selection** , **Merge** là 4 kiểu thao tác trên mảng, từ đó ta sẽ được ra được các thuật toán căn bản nhất và tối ưu lên (những thuật toán ở cột cuối cùng).

2 Interchange - Hoán đổi trực tiếp

Đúng như tên của nó, phân loại Interchange là phân loại dành cho thuật toán sắp xếp thông qua việc hoán đổi trực tiếp phần tử trong mảng.

2.1 Interchange Sort - Sắp xếp hoán đổi

Interchange Sort là thuật toán sắp xếp thông qua các cặp nghịch thế. Cụ thể là ta sẽ duyệt qua mọi cặp (i,j), kiểm tra có phải cặp nghịch thế không, nếu có thì thực hiện hoán đổi.

Mô hình thuật toán như sau, Pseudo-code :

```
for(i = 1 to n)
    for(j = i + 1 to n)
        if(a[i] > a[j])
            swap(a[i] , a[j])
```

Độ phức tạp thuật toán trên là $O(n^2)$.

Đây là 1 thuật toán sắp xếp **đúng** nhưng **chưa đủ nhanh** vì thuật toán đang còn rất thô sơ. Để cải tiến, ta sẽ dựa vào nhận xét, khi sử dụng **Interchange Sort** sẽ có những cặp nghịch thế **không đáng để swap**.

Lấy ví dụ 1 trường hợp như sau $A[5] = \{1,3,5,4,2\}$

Interchange Sort sẽ thực hiện hoán đổi những cặp sau: (2,5) , (3,4) , (3,5) , (4,5).
Tuy nhiên, thực tế ta chỉ cần hoán đổi cặp (2,5) và (3,5) là đã có thể sort lại mảng.

2.2 Quick Sort - Cải tiến của Interchange Sort

Để cải tiến thuật toán Interchange Sort, Quick Sort đã được áp dụng kĩ thuật chia để trị để cải tiến nhưng vẫn giữ được chính bản chất là **hoán đổi vị trí**.

Nói thêm về thuật toán của Quick Sort. Thuật toán sắp xếp này dựa vào **kỹ thuật chia để trị**, chủ yếu sẽ có những bước như sau:

1. Chọn 1 phần tử từ mảng làm $pivot$.
2. Hoán đổi vị trị sao cho mọi phần tử nhỏ hơn $pivot$ đều nằm bên trái của nó.
3. Áp dụng đệ quy và làm việc tương tự với phần bên trái và bên phải của $pivot$.

Để minh họa, ta có Pseudo-code như sau:

```
QuickSort(l , r):
    if(l >= r) return

    pivot = a[r]
    j = l

    for(i = l to r):
        if(a[i] < pivot):
            swap(a[j] , a[i])
            j++

    swap(a[j] , pivot)
    QuickSort(l , j - 1)
    QuickSort(j + 1, r);
```

Để phân tích độ phức tạp của Quick Sort, ta thấy độ phức tạp của bài toán sẽ hoàn toàn dựa vào việc chọn *pivot* sao cho phù hợp. Cụ thể hơn, độ phức tạp của kiểu sắp xếp này sẽ như sau:

- Best Case: $O(N)$
- Average Case: $O(NlogN)$
- Worst Case: $O(N^2)$

Tham khảo code C++ tại đây: Link
Để hiểu hơn về thuật toán, ta có thể thử theo dõi quá trình Quick Sort qua trang web VisualAlgo

2.3 Bubble Sort - Sắp xếp nổi bọt

Bubble Sort là thuật toán sắp xếp dạng sơ cấp tiếp theo được nhắc đến. Đúng như cái tên, thuật toán của nó được mô tả tựa như việc nổi bong bóng.

Cụ thể hơn, ta sẽ sắp xếp mảng bằng nhiều lần duyệt. Lần đầu, phần tử lớn nhất sẽ ở vị trí cuối cùng của mảng. Tương tự như vậy, lần thứ 2,

phần tử lớn thứ 2 sẽ kết thúc tại vị trí thứ 2 từ cuối của mảng. Tiếp tục thì ta sẽ được một mảng đã được sắp xếp.

Mô hình thuật toán như sau, Pseudo-code:

```
for(i = 1 to n)
    for(j = 1 to n - i)
        if(a[i] > a[i + 1])
            swap(a[i] , a[i + 1])
```

Thuật toán này rất đơn giản nhưng lại gặp phải cùng vấn đề với Interchange Sort đó chính là độ phức tạp quá lớn, lên tới $O(n^2)$.

2.4 Shake Sort - Cải tiến của Bubble Sort

Thuật toán sắp xếp Shake Sort là một phiên bản cải tiến của Bubble Sort. Tuy nhiên, độ phức tạp vẫn có phần không thay đổi vì về mặt bản chất, số lần so sánh của 2 thuật toán không quá chênh lệnh.

Thuật toán Shake Sort được thực hiện bằng cách duyệt nhiều lần để sắp xếp mảng, mỗi lần duyệt ta sẽ làm công việc như sau:

1. Duyệt từ trái sang phải và đưa phần tử lớn nhất về cuối dãy, gần tương tự Bubble Sort.
2. Duyệt lại từ phải sang trái và đưa phần tử nhỏ nhất về cuối mảng, làm ngược lại với bước trên.

Mô hình thuật toán như sau, Pseudo-code:

```
L = 1 , R = n
while(L < R)
    for(i = L to R - 1)
        if(a[i] > a[i + 1])
            swap(a[i] , a[i + 1]);

    for(i = R downto L + 1)
        if(a[i] < a[i - 1])
            swap(a[i] , a[i - 1])
```

Để đánh giá độ phức tạp, đoạn code trên có độ phức tạp trung bình ngang với Bubble Sort, tức là bằng $O(n^2)$.

3 Insertion - Chèn

Đây là phân loại dành cho những thuật toán sắp xếp được đặc trưng bởi việc sử dụng phép chèn. Khác với Interchange, Insertion chỉ cần duy nhất 1 phần tử để làm việc, so với Interchange thì phải hoán đổi giá trị của 2 phần tử riêng biệt.

3.1 Insertion Sort - Sắp xếp chèn

Insertion Sort là thuật toán tận dụng việc chèn để sắp xếp dãy. Thuật toán này được mô tả như sau:

Trước hết ta sẽ chia dãy ra làm 2 phần, A - phần chưa được sort và B - phần đã được sort, ban đầu toàn bộ mảng trong phần A. Tiếp tục, ta sẽ duyệt qua mọi phần tử của phần A và thêm nó vào phần B.

Tuy nhiên vì B là phần đã được sort nên ta không thể nào thêm tùy tiện 1 phần tử vào vị trí bất kì mà phải tìm vị trí thích hợp và chèn phần tử vào. Để tìm vị trí và chèn vào phần B, ta hãy theo dõi đoạn Pseudo-code sau:

```
for(i = 1 to n)
    value = a[i]
    j = i

    while(j > 0 && value < a[j - 1]){
        a[j] = a[j - 1];
        j--;
    }
    a[j] = value
```

Trong đoạn code trên, ta không thấy rõ phần A và phần B là vì phần A được xem như phần sau của mảng và phần B là i phần tử đầu với mỗi

i.

Đánh giá độ phức tạp của code trên:

- Best Case: $O(N)$
- Average Case: $O(N^2)$
- Worst Case: $O(N^2)$

3.2 Binary Insertion Sort - Sắp xếp chèn và tìm kiếm nhị phân

Thuật toán sắp xếp này là cải tiến của Insertion Sort. Nhìn thấy ràng việc tìm kiếm vị trí thích hợp trong để chèn phần tử bằng tìm kiếm tuyến tính là rất lâu nên ta sẽ sử dụng tìm kiếm nhị phân trên phần B của mình để tối ưu thuật toán.

Mô hình thuật toán như sau, Pseudo-code:

```
for(i = 1 to n)
    value = a[i]

    pos = i , L = 1 , R = i - 1
    while(L <= R)
        mid = (L + R) / 2
        if(a[mid] <= value)
            pos = mid
            L = mid + 1
        else R = mid - 1

    for(j = i down to pos + 1)
        a[j] = a[j - 1]
    a[pos] = value
```

Đánh giá code trên, ta thấy độ phức tạp của code trên không những không giảm mà còn tăng lên đôi chút. Độ phức tạp như sau:

- Best Case: $O(NlogN)$
- Average Case: $O(N^2)$
- Worst Case: $O(N^2logN)$

Lý do là vì ta chỉ tối ưu được phần tìm kiếm vị trí nhưng phần chèn vẫn chưa thay đổi. Vì vậy nên độ phức tạp của đoạn code trên không thay đổi vì thao tác chèn.

3.3 Shell Sort - Sắp xếp cách khoảng

Tiếp nối với Binary Insertion Sort, Shell Sort cũng là 1 thuật toán sắp xếp được phát triển từ Insertion Sort. Cải tiến của Shell Sort dựa trên nhận xét rằng thao tác chèn của ta mất rất nhiều lần dịch chuyển phần tử cách nó 1 đơn vị.

Vì vậy, Shell Sort sẽ dịch chuyển những phần tử có khoảng cách là k thay vì 1 và làm tương tự với nhiều giá trị k giảm dần cho tới khi $k = 1$.

Thuật toán của Shell Sort được mô tả như sau:

- Đặt ra giá trị cho k.
- Chia ra thành những dãy nhỏ hơn
- Thực hiện Insertion Sort trên dãy những phần tử có khoảng cách là k.
- Lặp lại 3 bước trên tới khi $k = 1$

Pseudo-code:

```
k = n / 2
while(k >= 1)
    for(i = k to n - 1)
        value = a[i]
        j = i
        while(j >= k and a[j - k] > value)
            a[j] = a[j - k];
            j -= k
        a[j] = value
    k = k / 2
```

Đánh giá độ phức tạp của code trên:

- Best Case: $O(NlogN)$
- Average Case: $O(NlogN)$
- Worst Case: $O(N^2)$

4 Selection - Chọn

Tiếp nối với 2 phân loại đã nhắc, Selection - Chọn là 1 phân loại thuật toán sắp xếp được dựa trên thao tác chọn.

4.1 Selection Sort - Sắp xếp chọn

Selection Sort là thuật toán sắp xếp mảng bằng các chọn ra phần tử **nhỏ nhất** và hoán đổi vị trí với số đầu tiên trong dãy. Lần lượt làm tương tự với các phần tử lớn thứ 2 , lớn thứ 3,... cho tới khi dãy đã được sắp xếp.

Mô hình thuật toán như sau, Pseudo-code:

```
for(i = 1 to n)
    min_index = i
    for(j = i + 1 to n)
        if(a[j] < a[min_index])
            min_index = j
```

```
        swap(a[i] , a[min_index])
```

Đoạn code này rất đơn giản nhưng độ phức tạp cũng rất cao, lên tới $O(n^2)$.

4.2 Heap Sort - Sắp xếp chọn và CTDL Heap

Heap Sort là thuật toán được cải tiến từ Selection Sort. Quan sát Selction Sort, ta thấy việc tìm số nhỏ nhất trong dãy bằng Linear Search là rất chậm, từ đó ta sẽ dùng **CTDL Heap** để cải tiến việc tìm số nhỏ nhất này.

Dành cho những ai không biết, trong C++ có CTDL là priority-queue, ta có thể hiểu nó như Heap.

Mô hình thuật toán của Heap Sort như sau:

```
    for(i = 1 to N)
        min_index = Heap.get_min_index()
        Heap.remove_min_index()
        swap(a[i] , a[min_index])
```

Đoạn code trên rất đơn giản nếu như không nhắc về việc code CTDL Heap. Ta có thể tham khảo code C++ như sau: Link
Độ phức tạp của đoạn code trên đa phần sẽ dựa vào Heap. Và vì Heap có độ phức tạp của thao tác thêm và xóa đều là $O(logN)$ nên độ phức tạp của ta sẽ như sau:

- Best Case: $O(NlogN)$
- Average Case: $O(NlogN)$
- Worst Case: $O(NlogN)$

5 Merge - Phép trộn

Tới phân loại này, phép Merge là phép trộn hai dãy và được thể hiện qua bài toán sau:

Cho dãy A bao gồm n số nguyên dương và dãy B bao gồm m số nguyên dương, dãy A và B đều đã được sắp xếp theo thứ tự không giảm. Hãy đưa ra dãy C sao cho C là dãy được kết hợp từ A, B và được sắp xếp theo thứ tự không giảm.

Đối với bài toán trên, ta phải thực hiện được với độ phức tạp tuyến tính $O(A+B)$ bằng cách dùng kĩ thuật 2 con trỏ.

Pseudo-Code:

```
A[n] , B[m]
i = 1 , j = 1

for(i = 1 to n)
    while(j <= m && B[j] <= A[i])
        C.insert(B[j])
        j++
    C.insert(A[i])

while(j <= m)
    C.insert(B[j])
    j++
```

5.1 Merge Sort - Sắp xếp trộn

Merge Sort là thuật toán sắp xếp dựa vào phép Merge bên trên. Cụ thể hơn, áp dụng kĩ thuật chia để trị thì ta có thuật toán như sau:

1. Gọi $[L,R]$ là dãy hiện tại, chia đôi dãy hiện tại thành 2 nửa.
2. Gọi đệ quy tới phần bên trái và phải
3. Áp dụng phép Merge với dãy bên trái và bên phải để tạo ra dãy $[L,R]$ hiện tại.

Pseudo-Code:

```
Merge(L1 , R1 , L2 , R2):
    ...

Merge_Sort(L , R):
    if(L >= R) return

    mid = (L + R) / 2

    Merge_Sort(L , mid)
    Merge_Sort(mid + 1 , R)

    Merge(L , mid , mid + 1 , R)
```

Hàm Merge($L1$, $R1$, $L2$, $R2$) có chức năng sử dụng phép Merge với đoạn [$L1$,$R1$] và [$L2$,$R2$].
Đoạn code trên có độ phức tạp rất ổn định và nhanh trong các thuật toán được liệt kê bên trên.
Độ phức tạp:

- Best Case: $O(Nlog_2N)$
- Average Case: $O(Nlog_2N)$
- Worse Case: $O(Nlog_2N)$

5.2 Natural Merge Sort - Cải tiến nhỏ của Merge Sort

Natural Merge Sort được phát triển dựa trên Merge Sort nhờ vào nhận xét sau. Thuật toán Merge Sort sẽ thực hiện viêc trộn kể cho khi dãy [L,R] hiện tại đã được sort, vì vậy nên ta sẽ chỉ thực hiện Merge Sort với những đoạn chưa được sắp xếp.

Pseudo-Code:

```
Merge(L1 , R1 , L2 , R2):
    ...
```

```
is_sorted(L , R):
    ...

Merge_Sort(L , R):
    if(L >= R) return
    if(is_isorted(L , R)) return

    mid = (L + R) / 2

    Merge_Sort(L , mid)
    Merge_Sort(mid + 1 , R)

    Merge(L , mid , mid + 1 , R)
```

Hàm is_sorted(L,R) được định nghĩa như một hàm kiểm tra từ vị trí L đến vị trí R trong dãy có được sắp xếp hay chưa.
Nhìn chung độ phức tạp của thuật toán này có chút thay đổi như sau:

- Best Case: $O(N)$
- Average Case: $O(Nlog_2N)$
- Worst Case: $O(Nlog_2N)$

Về mặt lý thuyết, độ phức tạp trong trường hợp trung bình không có gì thay đổi những sẽ cải thiện chút ít vì số lần sử dụng phép Merge sẽ ít đi so với Merge Sort thông thường.

5.3 K-way Merge Sort - Sắp xếp K mảng riêng biệt

Đối với thuật toán này, sẽ có đôi chút khác. Thuật toán này được sử dụng để giải quyết một vấn đề cụ thể là "Cho K dãy số đã được sắp xếp, hợp nhất K dãy số lại thành 1 dãy A bao gồm mọi phần tử trong K dãy và được sắp xếp tăng dần".

5.3.1 Hướng tiếp cận sử dụng phép Merge

Mượn ý tưởng của phép Merge, ta sẽ lấy giá trị nhỏ nhất trong những phần tử chưa được thêm vào dãy A của K dãy và sau đó thêm vào cuối dãy A.

Gọi $B[i][j]$ là phần tử thứ j của dãy số thứ i
$id[i]$ là phần tử nhỏ nhất của dãy số thứ i chưa được thêm vào A
N là tổng số lượng phần tử trong K dãy.
$size[i]$ là tổng số lượng phần tử trong dãy thứ i.
Ta sẽ mô tả thuật toán rõ hơn như sau:

- Duyệt qua mọi $B[i][id[i]]$ tìm giá trị nhỏ nhất, gọi u dãy mà giá trị đó thuộc về.
- Thêm $B[u][id[u]]$ vào A, tăng $id[u]$ lên 1 đơn vị.

Pseudo-code:

```
A[]
B[][] , id[]
K_Way_Merge():
    u = -1

    for(i = 1 to N)
        for(j = 1 to k)
            if(id[j] <= size[j]
            and (u == -1 or B[j][id[j]] < B[u][id[u]]))
                u = j

    A.insert(B[u][id[u]])
    id[u]++
```

Vì có K dãy số và mỗi lần tìm giá trị nhỏ nhất ta phải thực hiện K phép so sánh và ta phải thực hiện như vậy tổng cộng N lần nên độ phức tạp của ta là $O(N * K)$.

5.3.2 Phép tối ưu sử dụng Heap

Lại một lần nữa Heap được sử dụng để tối ưu một thuật toán sắp xếp. Trong lần này, tương tự với Heap Sort, CTDL Heap sẽ đóng vai trò tối ưu cộng việc tìm giá trị nhỏ nhất một cách triệt để.

Như đã nhắc tới, CTDL Heap có khả năng tìm giá trị nhỏ nhất trong độ phức tạp $O(logK)$ thay vì $O(K)$ trong như trong hướng tiếp cận trước. Vì vậy, độ phức tạp lần này của ta là $O(N * logK)$.

Pseudo-Code:

```
K_Way_Merge():
    for(int i = 1 to N)
        u = Heap.get_min_index()
        Heap.remove_min_index()

        A.insert(B[u][id[u]])

        id[u]++
        Heap.add(B[u][id[u]])
```

Ở đây, các thao tác với Heap chỉ là tượng trưng, theo dõi code C++ tại đây để rõ hơn: Link

6 Các thuật toán sắp xếp khác

Ở phân loại này, sẽ là những thuật toán không thuộc các phần trên và có độc đáo riêng của chính nó.

6.1 Counting Sort - Sắp xếp đếm

Ở thuật toàn này, ta sẽ đếm các lần lượt các giá trị và liệt kê lại chúng theo thứ tự tăng dần. Ví dụ như sau:

Giả sử ta có dãy $A = 1,3,2,2,5,1,3$, ta thấy:

- Giá trị 1 xuất hiện 2 lần
- Giá trị 2 xuất hiện 2 lần
- Giá trị 3 xuất hiện 3 lần
- Giá trị 5 xuất hiện 1 lần

Từ đây, ta sẽ liệt kê lại dần, gọi B là dãy được sắp xếp:

- Vì giá trị 1 xuất hiện 2 lần nên ta thêm 2 số 1 vào cuối dãy B, lúc này $B = 1,1$
- Tương tự giá trị 2 xuất hiện 2 lần nên ta thêm 2 số 2 vào cuối dãy B, lúc này $B = 1,1,2,2$
- Giá trị 3 xuất hiện 2 lần nên ta thêm 2 số 3 vào cuối dãy B, lúc này $B = 1,1,2,2,3,3$
- Giá trị 5 xuất hiện 1 lần nên ta thêm 1 số 5 vào cuối dãy B, lúc này $B = 1,1,2,2,3,3,5$

Từ đó, ta có được 1 dãy được sắp xếp từ A.
Pseudo-code:

```
Cnt[]

Counting_Sort(){
    for(i = 1 to n)
        Cnt[A[i]]++

    for(i = 1 to MAX_VALUE)
        for(int j = 1 to Cnt[i])
            B.insert(i)
}
```

Với $Cnt[]$ là mảng dùng để đếm. Lưu ý, trong một số trường hợp giá trị của A_i quá lớn, ta có thể sử dụng kĩ thuật nén số để sử dụng Counting

Sort.

Độ phức tạp của Counting Sort thực sự rất nhanh. Trong trường hợp lý tưởng, $A_i \leq n \ \forall i$ độ phức tạp như sau:

- Best Case: $O(N)$
- Average Case: $O(N)$
- Worst Case: $O(N)$

Tuy nhiên, độ phức tạp của Counting Sort sẽ bị ảnh hưởng bởi độ lớn của giá trị A_i khi đó, độ phức tạp sẽ phải cộng thêm $O(NlogN)$ - độ phức tạp của việc nén số.

6.2 Radix Sort - Sắp xếp cơ số

Đúng như cái tên, Radix Sort là thuật toán sắp xếp dựa vào cơ số, ở đây cụ thể là từng chữ số của 1 số. Đối với hệ thập phân, Radix Sort sẽ có mô hình như sau:

- Lần duyệt thứ 1, chia dãy A cần sắp xếp thành những nhóm có chữ số hàng đơn vị 0 , 1 , ... , 9. Và sắp xếp lại dãy A theo những nhóm với giá trị hàng đơn vị tăng dần.
- Làm tương tự với hàng chục, hàng trăm, ...

Pseudo-Code:

```
Groups[10][]
Group_size[10]

Radix_Sort()
    max_value = A[1]
    for(i = 2 to N)
        max_value = max(max_value , A[i])

    power_10 = 1
    while(max_value > 0)
        for(i = 1 to N)
            digit = (A[i] / power_10) % 10
            Groups[digit].insert(A[i])
            Group_size[digit0]++

        int cur_index = 1
        for(i = 0 to 9)
            for(j = 1 to Group_size[i])
                A[cur_index] = Groups[i][j]
                cur_index++

        power_10 = power_10 * 10
```

Phân tích độ phức tạp của code trên, ta thấy trong 1 lần phân chia dãy theo chữ số, ta thực hiện đúng trong $O(N)$. Bên cạnh đó, gọi K là số chữ số của số lớn nhất - max_value. Độ phức tạp của ta như sau:

- Best Case: $O(N * K)$
- Average Case: $O(N * K)$
- Wors Case: $O(N * K)$