Shaker sort

Ref: <https://en.wikipedia.org/wiki/Cocktail_shaker_sort>

Là một biến thể từ thuật toán Bubble sort. Nhưng thay vì đẩy các phần tử lớn nhất về sau như thuật toán Bubble sort, thuật toán Shaker sort đồng thời đẩy phần tử lớn nhất về sau và đẩy các phần tử nhỏ nhất về trước, từ đó giảm thiểu tối đa các vòng lặp. Hình ảnh thuật toán đẩy các phần tử lớn nhất về sau và phần tử nhỏ nhất về phía trước trông giống như đang lắc một bình nước (Shake: lắc).

Trong trường hợp xấu nhất là mảng sắp xếp giảm dần, thuật toán phải swap liên tục các phần kề nhau trong 2 vòng for nên độ phức tạp thuật toán lúc đó sẽ là O(n2).

Trong trường hợp tốt nhất là mảng đã sắp xếp đúng, thuật toán chỉ duyệt qua mảng 1 lần, nên độ phức tạp thuật toán lúc đó sẽ là O(n).

Trong trường hợp trung bình, thuật toán vẫn duyệt hết các phần tử trong mảng trong 2 vòng for nên độ phức tạp thuật toán lúc đó sẽ là O(n2).

Độ phức tạp không gian trong cả 3 trường hợp là O(1).

Pseudo Code:

**procedure** cocktailShakerSort(A **:** list of sortable items) **is**

**do**

swapped := false

**for each** i **in** 0 **to** length(A) − 2 **do:**

**if** A[i] > A[i + 1] **then** // kiểm tra 2 phần tử có nằm sai vị trí không

swap(A[i], A[i + 1]) // hoán vị 2 phần tử

swapped := true

**end if**

**end for**

**if not** swapped **then**

// Nếu không có hoán vị thì có thể kết thúc thuật toán

**break do-while loop**

**end if**

swapped := false

**for each** i **in** length(A) − 2 **to** 0 **do:**

**if** A[i] > A[i + 1] **then**

swap(A[i], A[i + 1])

swapped := true

**end if**

**end for**

**while** swapped // nếu không có phần tử nào được hoán vị thì mảng đã được sắp xếp

**end procedure**

Quick sort:

Ref: <https://en.wikipedia.org/wiki/Quicksort>

Dựa trên phương pháp Chia để trị.

Thuật toán bắt đầu bằng việc chọn pivot cho mảng, sau đó đưa các phần tử bé hơn hoặc bằng pivot vào bên trái pivot, đưa các phần tử lớn hơn pivot vào bên phải pivot, sau đó thực hiện tương tự cho mảng con bên trái pivot và mảng con bên phải pivot. Quá trình này được gọi là partition.

Có 3 cách chọn pivot: chọn phần tử đầu tiên, chọn phần tử cuối cùng, chọn phần tử trung vị. Tuy nhiên, cách chọn pivot có thể ảnh hưởng tới độ phức tạp thuật toán. Do trong trường hợp xấu nhất, mảng con bên phải pivot rỗng, mảng con bên trái của pivot có kích thước n-1 với n là độ dài của mảng dẫn đến thuật toán có độ phức tạp là O(n2). Để tránh trường hợp đó, còn có một cách chọn pivot khác là median-of-three, tức là lấy phần tử trung vị của phần tử đầu tiên, phần tử trung vị của mảng, phần tử cuối cùng.

Độ phức tạp thời gian:

* Trường hợp xấu nhất: O(n2).
* Trường hợp tốt nhất: do mỗi lần partition, mảng được chia làm 2 nên độ phức tạp sẽ là O(nlogn).
* Trường hợp trung bình: O(nlogn).

Pseudo code:

*// Sorts a (portion of an) array, divides it into partitions, then sorts those*

**algorithm** quicksort(A, lo, hi) **is**

**if** lo >= 0 && hi >= 0 && lo < hi **then**

p := partition(A, lo, hi)

quicksort(A, lo, p) // Note: the pivot is now included

quicksort(A, p + 1, hi)

*// Divides array into two partitions*

**algorithm** partition(A, lo, hi) **is**

*// Pivot value*

pivot := A[ floor((hi + lo) / 2) ] *// The value in the middle of the array*

*// Left index*

i := lo - 1

*// Right index*

j := hi + 1

**loop forever**

*// Move the left index to the right at least once and while the element at*

*// the left index is less than the pivot*

**do** i := i + 1 **while** A[i] < pivot

*// Move the right index to the left at least once and while the element at*

*// the right index is greater than the pivot*

**do** j := j - 1 **while** A[j] > pivot

*// If the indices crossed, return*

**if** i ≥ j **then** **return** j

*// Swap the elements at the left and right indices*

**swap** A[i] **with** A[j]

Radix sort:

Ref: <https://en.wikipedia.org/wiki/Radix_sort>

Thể loại: không sử dụng so sánh.

Thuật toán phân bố các phần tử vào các bucket dựa trên các kí tự của các phần tử đó. Với các phần tử có nhiều hơn 1 kí tự, thuật toán sẽ lặp lại trên tất cả các kí tự của phần tử đó. Tại vị trí đang duyệt, kí tự của phần tử nào “lớn” hơn sẽ nằm đằng sau và ngược lại.

Có 2 cách cài đặt cho radix sort, đó là bắt đầu từ trái sang (most significant digit: MSD) và bắt đầu từ phải sang (least significant digit: LSD) của các phần tử.

Ví dụ cho cách cài đặt bắt đầu từ phải sang:

Mảng đầu vào:

[170, 45, 75, 90, 2, 802, 2, 66]

Bắt đầu từ phải sang, sắp xếp các số vào các bucket:

[{170, 90}, {2, 802, 2}, {45, 75}, {66}]

Tiếp tục với chữ số tiếp theo:

[{*0*2, 802, *0*2}, {45}, {66}, {170, 75}, {90}]

Và chữ số cuối cùng:

[{*00*2, *00*2, *0*45, *0*66, *0*75, *0*90}, {170}, {802}]

Có thể sử dụng counting sort để tối ưu thời gian phân bố các số vào các bucket.

Độ phức tạp thuật toán:

Độ phức tạp thời gian: O(n\*w) với n là số phần tử, w là độ dài của phần tử có độ dài lớn nhất.

Độ phức tạp không gian: O(k) với k là phạm vi của kí tự của các phần tử trong mảng. Đối với mảng số nguyên, k nằm trong đoạn [0, 9].