TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

NGUYỄN NHẬT QUANG – 21127151

PHÙNG QUANG MINH HUY – 21127306

VŨ MINH PHÁT – 21127739

PROJECT

LAB 3: SORTING

OPTION: SET 2 (11 ALGORITHMS)

Tp.HCM, năm 2022

MỤC LỤC

A. THÔNG TIN VỀ THỬ NGHIỆM

Các thử nghiệm được tiến hành với kích thước dữ liệu đầu vào lần lượt là 10.000, 30.000, 50.000, 100.000, 300.000, 500.000 với các loại dữ liệu lần lượt là mảng có thứ tự ngẫu nhiên, mảng gần như đã có thứ tự tăng dần (sai ở 1 số vị trí), mảng có thứ tự tăng dần, mảng có thứ tự giảm dần. Mục đích là đưa về mảng có thứ tự tăng dần.

Cấu hình máy thực hiện thử nghiệm:

- CPU: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, 2803 Mhz.
- RAM: 16 GB LPDDR4X 4266MHz bus.
- VGA: Intel® Iris® Xe Graphics.
- OS Name: Microsoft Windows 11 Home Single Language.
- Compiler: Visual Studio Code Version 1.68.

B. GIỚI THIỆU THÀNH VIÊN

Đồ án được thực hiện bởi 3 thành viên, lần lượt là:

1. Nguyễn Nhật Quang – MSSV: 21127151.
2. Phùng Quang Minh Huy – MSSV: 21127306.
3. Vũ Minh Phát – MSSV: 21127739.

C. TRÌNH BÀY THUẬT TOÁN

1. Selection Sort

Ý tưởng: Tìm phần tử nhỏ nhất trong mảng sau đó đổi chỗ với phần tử đầu mảng để tạo ra một mảng tăng dần

Các bước:

- Bước 1: Chọn phần tử đầu mảng
- Bước 2: Tìm phần tử nhỏ nhất trong khoảng phần tử đã chọn cho tới cuối mảng
- Bước 3: Đổi chỗ phần tử đã chọn ở bước 1 với phần tử nhỏ nhất tìm được ở bước 2.
- Bước 4: Lặp lại bước 2 với phần từ tiếp theo của mảng.

Độ phức tạp thời gian:

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

Độ phức tạp không gian: O(1)

Biến thể:

- Heap Sort.
- Bingo Sort: Sau khi tìm được giá trị nhỏ nhất trong mảng thì ta đem tất cả các phần tử có cùng giá trị đó ra sau mảng. Thuật toán này sẽ chạy tốt hơn Selection Sort nếu có nhiều phần tử có giá trị trùng nhau trong mảng.
- Double Selection Sort: Như cái tên thì thuật toán này chạy như Selection Sort nhưng sẽ tìm giá trị nhỏ nhất và lớn nhất của mảng và đổi chỗ với phần tử đầu và cuối của mảng.

2. Insertion Sort

Ý tưởng: Ta sẽ lấy từng phần tử của mảng rồi tìm vị trí thích hợp để chèn vào

Các bước thực hiện:

B1: Lấy một phần tử từ mảng ra

B2: So sánh phần tử đó với các phần tử liền trước nó. Nếu phần tử hiện tại nhỏ hơn phần tử liền trước nó thì di chuyển phần tử liền trước lên một ô và tiếp tục so sánh phần tử hiện tại với phần tử liền trước tiếp theo trong mảng. Lặp lại cho tới khi gặp phần tử liền trước lớn hơn phần tử hiện tại hoặc chạy hết mảng thì ta chèn phần tử hiện tại vào ô trống đã chừa ra.

B3: Lặp lại bước một với phần tử tiếp theo trong mảng cho tới khi mảng kết thúc.

Độ phức tạp thời gian:

- Worst case: $O(n^2)$ khi mảng sắp xếp ngược
- Average case: $O(n^2)$
- Best case: O(n) khi mảng đã được sắp xếp sẵn

Độ phức tạp không gian: O(1)

Biến thể:

- Shell sort.
- Binary Insertion Sort: sử dụng Binary Search để tìm vị trí thích hợp để chèn phần tử vào sau đó đôn các phần tử lên và chèn vào.
- Library Sort: chừa ra các ô trống để việc chèn phần tử diễn ra với tốc độ nhanh hơn. Điều này dẫn tới việc giảm tốc độ chạy xuống còn O(nlogn) tuy nhiên tăng chi phí không gian lên O(n).

3. Bubble Sort

Ý tưởng: Là thuật toán sắp xếp đơn giản nhất. Ta chỉ việc so sánh các phần tử liền kề nhau xem đã đúng vị trí chưa nếu chưa thì đổi vị trí hai phần tử.

Các bước:

- Bước 1: Lặp lại từ i = 0 tới n.
- Bước 2: Lặp lại từ j = 0 tới n – i – 1.
- Bước 3: So sánh phần tử ở vị trí j với phần tử ở vị trí j + 1. Nếu sai vị trí thì ta đổi chỗ.

Độ phức tạp thời gian: bằng nhau do số lần lặp không thay đổi.

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

Độ phức tạp không gian: O(1)

Biến thể:

- Shaker Sort.
- Odd-even Sort: thay vì kiểm tra các phần tử liền kề nhau thì ta kiểm tra các phần tử chẵn-lẻ ở gần nhau. Nếu sai vị trí thì đổi chỗ hai phần tử đó.

4. Shaker Sort

Ý tưởng:

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 $O(n^2)$.

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

Mã giả:

```
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
```

5. Shell Sort

Ý tưởng:

Shell Sort là một biến thể của giải thuật sắp xếp chèn (Insertion Sort). Trong sắp xếp chèn, ta chỉ di chuyển các phần tử về phía trước một vị trí. Khi một phần tử cần được di chuyển xa về phía trước, ta phải tốn nhiều chi phí cho việc thay đổi đó. Ý tưởng của Shell Sort là cho phép trao đổi các các phần tử ở xa nhưng không mất nhiều chi phí, mang lại hiệu quả cao cho thuật toán sắp xếp.

Giải thuật này sử dụng giải thuật sắp xếp chèn trên các phần tử có khoảng cách xa nhau, sau đó sắp xếp các phần tử có khoảng cách hẹp hơn. Khoảng cách này thường được gọi là khoảng (interval) hoặc là trình tự, nó là số vị trí từ phần tử này tới phần tử khác. Hiệu suất của kiểu sắp xếp Shell phụ thuộc vào kiểu trình tự được sử dụng cho một mảng đầu vào nhất định.

Một số trình tự tối ưu được sử dụng là:

- Trình tự ban đầu của Shell: N/2, N/4, …, 1
- Tăng dần Knuth: 1, 4, 13, …, (3k-1)/2
- Số gia tăng của Sedgewick: 1, 8, 23, 77, 281, 1073, 4193, 16577…
- Số gia tăng của Hibbard: 1, 3, 7, 15, 31, 63, 127, 255, 511, …
- Số gia tăng của Papernov & Stasevich: 1, 3, 5, 9, 17, 33, 65, …
- Pratt: 1, 2, 3, 4, 6, 9, 8, 12, 18, 27, 16, 24, 36, 54, 81, …

Cách hoạt động:

Bước 1 – Bắt đầu thuật toán Shell Sort.

Bước 2 – Khởi tạo giá trị của khoảng cách (gap/interval…). Tạm gọi là h.

Bước 3 – Chia mảng ban đầu thành các mảng nhỏ. Mỗi mảng có kích thước bằng h.

Bước 4 – Sử dụng thuật toán Insertion Sort để sắp xếp các mảng con.

Bước 5 – Lặp lại bước 2 đến khi toàn bộ mảng đã được sắp xếp.

Bước 6 – Dừng thuật toán.

Mã giả:

```
PROCEDURE SHELL_SORT(ARRAY, N)
    // Khởi tạo giá trị của khoảng cách
    WHILE GAP < LENGTH(ARRAY)/3 :
        GAP = ( INTERVAL * 3 ) + 1
    END WHILE LOOP

    WHILE GAP > 0 :
        FOR (OUTER = GAP; OUTER < LENGTH(ARRAY); OUTER++):
            INSERTION_VALUE = ARRAY[OUTER]
            INNER = OUTER

            // Thực hiện Insertion Sort trên các mảng con có kích thước bằng GAP
            WHILE INNER > GAP-1 AND ARRAY[INNER – GAP] >= INSERTION_VALUE:
                ARRAY[INNER] = ARRAY[INNER – GAP]
                INNER = INNER – GAP
            END WHILE LOOP

            ARRAY[INNER] = INSERTION_VALUE
        END FOR LOOP

        // Cập nhật giá trị cho GAP
        GAP = (GAP - 1)/3
    END WHILE LOOP
END SHELL_SORT
```

Độ phức tạp của thuật toán:

- Độ phức tạp về thời gian:
    + Worst-case: $O(n^2)$ - trong trường hợp mảng sắp xếp theo thứ tự giảm dần.
        - Độ phức tạp của trường hợp xấu nhất của sắp xếp Shell Sort luôn nhỏ hơn hoặc bằng $O(n^2)$.

- Theo định lý Poonen, độ phức tạp trong trường hợp xấu nhất cho kiểu sắp xếp này là $\Theta(N(\log N)^2)$.

+ Best-case: $O(n \log n)$ - trong trường hợp mảng đã được sắp xếp.
    - Khi mảng đã được sắp xếp, tổng số phép so sánh cho mỗi khoảng bằng kích thước của mảng.
+ Average-case: $O(n \log n)$ - trong các trường hợp còn lại.
    - Nằm ở nằm trong khoảng xung quanh $O(n^{1,25})$.
    - Độ phức tạp phụ thuộc vào khoảng được chọn. Các độ phức tạp sẽ khác nhau đối các trình tự gia tăng khác nhau. Không có trình tự nào là tốt nhất cho mọi trường hợp.

- Độ phức tạp về không gian: $O(1)$.

Đặc điểm:

- Shell Sort là một thuật toán sắp xếp in-place, sẽ giúp tiết kiệm bộ nhớ và không gian lưu trữ. Nên phù hợp với các mảng dữ liệu có kích thước từ trung bình đến lớn.
- Nó mang bản chất của Insertion Sort nhưng tinh giảm bớt chi phí so sánh khóa. Có thể dùng thay thế cho Insertion Sort để thu được thời gian thực thi nhanh hơn.
- Shell Sort là một thuật toán sắp xếp không ổn định vì thuật toán này không kiểm tra các phần tử nằm giữa các khoảng. Nên có thể thay đổi thứ tự tương đối của các phần tử có giá trị khóa bằng nhau.
- Đây là một thuật toán "adaptive sorting" (tạm dịch: sắp xếp thích ứng). Thuật toán sẽ thực thi nhanh hơn khi đầu vào được sắp xếp một phần (partially sorted).

6. Heap Sort

Ý tưởng: Lấy ý tưởng từ cấu trúc Heap. Xem mảng như một cây nhị phân hoàn chỉnh sau đó hiểu chỉnh thành cấu trúc Heap. Phần tử lớn nhất hoặc nhỏ nhất của Heap chắc chắn sẽ là root của Heap. Do đó để sắp xếp ta chỉ cần đổi chỗ root với vị trí cuối cùng sau đó hiệu chỉnh mảng (trừ phần tử cuối cùng) lại thành Heap và lặp lại cho tới khi Heap còn 0 phần tử.

Các bước:

- Bước 1: Biến Heap thành Max Heap bắt các Heapify từ dưới lên
- Bước 2: Đổi chỗ root và phần tử cuối cùng của mảng sau đó Heapify lại mảng (trừ phần tử cuối cùng).
- Bước 2: Lặp lại bước 2 cho tới khi Heap còn 0 phần tử

Độ phức tạp thời gian:

- Worst case: O(nlogn)
- Average case: O(nlogn)
- Best case: O(nlogn)

Độ phức tạp không gian: O(1)

Biến thể:

- Ternary Heap Sort: sử dụng ternary heap (mỗi nhánh có 3 phần tử con).
- Out-of-place Heap Sort: thay vị thay root với vị trí cuối mảng thì ta sẽ thay với giá trị -∞ sau đó tiếp tục Heapify cả mảng thay vì mảng mất một phần tử cuối như trên cho tới khi heap chỉ có các giá trị -∞. Điều này làm giảm số lần so sánh của vì -∞ không thể di chuyển lên được nữa.

7. Merge Sort

Ý tưởng: Là thuật toán chia để trị. Ta liên tục chia mảng thành hai nửa mảng con và sau đó kết hợp các mảng con lại theo thứ tự cần sắp xếp để có được mảng sắp xếp hoàn chỉnh.

Các bước:

- Bước 1: Chia mảng thành hai nửa mảng con. Lặp lại cho tới khi không chia được nữa
- Bước 2: Lần lượt ghép các mảng con đã chia lại theo thứ tự cần sắp xếp cho tới khi mảng hoàn chỉnh.

Độ phức tạp thời gian:

- Worst case: O(nlogn)
- Average case: O(nlogn)
- Best case: O(nlogn)

Độ phức tạp không gian: O(n)

Biến thể:

- 3-Way Merge Sort: Chia mảng ra làm 3 thay vì 2.
- Bottom-up Variant: Chia mảng ra thành mảng có độ dài là 1-2-4-8-… cho tới khi độ dài mảng con bằng mảng chính.

8. Quick Sort

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 $O(n^2)$. 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: $O(n^2)$.
- 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]
```

9. Counting Sort

Ý tưởng:

Sắp xếp đếm hay Counting Sort là một thuật toán sắp xếp sắp xếp các phần tử của một mảng bằng cách đếm số lần xuất hiện của mỗi phần tử duy nhất trong mảng (loại băm). Số đếm được lưu trữ trong một mảng con và việc sắp xếp được thực hiện bằng cách ánh xạ số đếm làm một chỉ số của mảng con. Sau đó, thực hiện một số phép tính để tính toán vị trí của mỗi đối tượng trong mảng đầu ra.

Thuật toán này thường được dùng cho một mảng các phần tử mà mỗi phần tử là các số nguyên không âm; hoặc là một danh sách các kí tự được ánh xạ về dạng số để sort theo bảng chữ cái. Như vậy, Counting Sort là một thuật toán sắp xếp các số nguyên không âm, không dựa vào so sánh.

Cách hoạt động:

Bước 1 – Bắt đầu thuật toán Counting Sort.

Bước 2 – Tìm ra phần tử lớn nhất (giả sử là biến max) từ mảng đã cho.

Bước 3 – Khởi tạo một mảng có độ dài max+1 với tất cả các phần tử 0. Mảng này được sử dụng để lưu trữ số lượng các phần tử trong mảng. Tạm gọi là mảng count.

Bước 4 – Lưu trữ số đếm của từng phần tử tại chỉ số tương ứng của chúng trong mảng count.

Ví dụ, nếu mảng ban đầu có 2 phần tử cùng mang giá trị 4, thì giá trị 2 được lưu ở vị trí thứ 4 của mảng count. Nếu trong mảng không có giá trị 1 thì giá trị 0 được lưu ở vị trí thứ 1 của mảng count.

Bước 5 – Lưu trữ tổng tích lũy của các phần tử của mảng count. Nó giúp đặt các phần tử vào chỉ số chính xác của mảng đã sắp xếp.

Bước 6 – Tìm chỉ số của từng phần tử của mảng ban đầu trong mảng count. Sau khi đặt mỗi phần tử vào đúng vị trí của nó, ta sẽ giảm số đếm của nó đi một đơn vị.

Bước 7 – Dừng thuật toán.

Mã giả:

```
function CountingSort(input, k)
    // count là mảng lưu số lượng từng phần tử trong input
    // k là giá trị phần tử (không âm) lớn nhất của input
    count ← array of k + 1 (elements with value) zeros

    // mảng chứa kết quả sau khi quá trình sort
    output ← array of same length as input

    // lưu số lượng các phần tử ứng với mỗi khóa riêng biệt
    for i = 0 to length(input) - 1 do
        // key là giá trị của phần tử input[i]
        j = key(input[i])
        count[j] += 1

    // tìm ra tổng số tích lũy,
    // là vị trí của các phần tử trong output
    for i = 1 to k do
        count[i] += count[i - 1]

    // lưu lại kết quả vào trong mảng output
    // giảm biến đếm của mỗi phần tử được lưu đi 1 đơn vị
    for i = length(input) - 1 downto 0 do
        j = key(input[i])
        count[j] -= 1
        output[count[j]] = input[i]

    //  sao chép output --> input
    for i = 0 to length(input) - 1 do
         input[i] = output[i]

end function
```

Độ phức tạp của thuật toán:

- Độ phức tạp về thời gian: Có bốn vòng lặp chính (việc tìm giá trị lớn nhất có thể được thực hiện bên ngoài hàm)

For loop	Time Complexity
Lần 1	O(max)
Lần 2	O($size$)
Lần 3	O(max)
Lần 4	O($size$)

  ⇨ Suy ra độ phức tạp là: $O(\max) + O(size) + O(\max) + O(size) = O(\max + size)$.

  + Best-case: $O(n+k)$.
  + Average-case: $O(n+k)$.
  + Worst-case: $O(n+k)$.

  - Trong tất cả các trường hợp trên, độ phức tạp thuật toán là như nhau vì dù các phần tử được đặt trong mảng như thế nào thì thuật toán cũng phải trải qua n+k lần.
  - Khi k = $O(n)$ thì T(n) = $O(n) < O(n \log n)$ của các thuật toán sắp xếp dựa trên so sánh.
  - Ta thấy không có sự so sánh giữa bất kỳ phần tử nào với nhau, vì vậy nó tốt hơn so với kỹ thuật sắp xếp dựa trên so sánh. Tuy nhiên, nhận định trên sẽ là sai lầm nếu các số nguyên là rất lớn vì mảng phụ trợ có kích thước như vậy sẽ phải được tạo ra.

- Độ phức tạp về không gian: $O(\max)$. Phạm vi các phần tử càng lớn thì độ phức tạp về không gian càng lớn.

Đặc điểm:

- Nó sử dụng một mảng tạm để hỗ trợ quá trình sắp xếp nên đây không phải một thuật toán sắp xếp in-place.
- Counting Sort không phải là một thuật toán dựa trên so sánh, nó băm giá trị vào trong một mảng đếm tạm thời và sử dụng chúng để sắp xếp.
- Sắp xếp đếm sẽ hiệu quả nếu phạm vi dữ liệu đầu vào không lớn hơn đáng kể so với số lượng đối tượng được sắp xếp. Ví dụ với một mảng 5 phần tử là <1,2,10,10000,50000> thì thuật toán sẽ tỏ ra kém hiệu quả vì tốn quá nhiều tài nguyên.

- Thuật toán được trình bày ở trên không thể dùng cho số âm, tuy nhiên ta có thể nâng cấp thuật toán để nó hoạt động được với số âm.
- Counting Sort là một thuật toán sắp xếp ổn định (stable), có thể dùng để làm hàm phụ trợ cho thuật toán Radix Sort…

Cải tiến của Counting Sort để thực hiện với mảng có số âm:

Vấn đề của thuật toán bên trên là chỉ có thể thao tác với mảng không có số âm vì các chỉ số của mảng đều lớn hơn hoặc bằng 0. Nên ý tưởng của thuật toán cải tiến là tìm phần tử có giá trị nhỏ nhất và lưu nó ở vị trí đầu mảng. Ta có thể minh họa thuật toán bằng đoạn mã giả sau:

```
function CountingSortWithNegative(input)
    max ← find value of max element
    min ← find value of min element
    // kích thước của mảng phụ
    range = max - min + 1;

    // count là mảng lưu số lượng từng phần tử trong input
    count ← array of range (elements with value) zeros

    // mảng chứa kết quả sau khi quá trình sort
    output ← array of same length as input

    // lưu số lượng các phần tử ứng với mỗi khóa riêng biệt
    for i = 0 to length(input) - 1 do
        // key là giá trị của phần tử input[i]
        j = key(input[i]) – min // tức là cộng cho giá trị tuyệt đối của min để j >= 0
        count[j] += 1

    // tìm ra tổng số tích lũy,
    // là vị trí của các phần tử trong output
    for i = 1 to range - 1 do
        count[i] += count[i - 1]

    // lưu lại kết quả vào trong mảng output
```

```
    // giảm biến đếm của mỗi phần tử được lưu đi 1 đơn vị
    for i = length(input) - 1 downto 0 do
        j = key(input[i]) – min // tức là cộng cho giá trị tuyệt đối của min để j >= 0
        count[j] -= 1
        output[count[j]] = input[i]

    //  sao chép output → input
    for i = 0 to length(input) - 1 do
        input[i] = output[i]

end function
```

⇨ Ta thấy rằng điểm khác biệt nổi bật giữa thuật toán ban đầu và thuật toán cải tiến là ở việc có thêm thao tác <trừ min>. Thật chất, ở thuật toán ban đầu cũng có <trừ min> nhưng ta ngầm định min là 0 vì tất cả phần tử đều lớn hơn hoặc bằng 0. Nhưng khi mảng có số âm thì ta phải xác định tường minh giá trị của min và thêm thao tác <trừ min> khi thực hiện thao tác trên mảng count[].

10. 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 = 10 do phạm vi của kí tự nằm trong đoạn [0, 9].

11. Flash Sort

Ý tưởng:

Flash Sort ra đời năm 1998 bởi tác giả Karl-Dietrich Neubert, đây là một biến thể của giải thuật Bucket Sort (một biến thể khác của Insertion Sort). Khi mới ra mắt thì đây là một thuật toán nhanh nhất có thể (đạt đến tốc độ tới hạn).

Đối với các thuật toán dựa vào so sánh, ta sẽ sử dụng các câu lệnh if… else… để so sánh giá trị hai phần tử nhằm đưa ra quyết định có đổi chỗ chúng hay không. Việc so sánh khóa nhiều lần như vậy sẽ tốn rất nhiều thời gian vì trong các dự án thực tế thì giá trị được đem so sánh sẽ phức tạp hơn nhiều so với một số nguyên. Vì vậy thuật toán này được đề xuất nhằm hạn chế tối đa số lần so sánh khóa trong quá trình sort, bằng cách chia mảng thành các phân lớp riêng biệt mà không cần sử dụng mảng phụ. Nó gán mỗi phần tử đầu vào cho một trong số m nhóm (còn gọi là xô), sắp xếp lại đầu vào một cách hiệu quả để đặt các nhóm theo đúng thứ tự, sau đó thực hiện sắp xếp trên từng nhóm.

Cách hoạt động: gồm 3 bước:

Bước 1 – Phân lớp dữ liệu, tức là dựa trên giả thiết dữ liệu tuân theo một cách phân bố nào đó, chẳng hạn như phân bố đều, để tìm ra một công thức ước tính vị trí (lớp - "class") của phần tử sau khi sắp xếp.

Bước 2 – Hoán vị toàn cục, tức là dịch chuyển các phần tử trong mảng về đúng lớp của mình.

Bước 3 – Sắp xếp cục bộ, tức là sắp xếp lại các phần tử trong phạm vi của từng lớp.

Cụ thể hơn:

Ở bước phân lớp dữ liệu ta có thể chia nhỏ thêm các bước như sau (với a[] là mảng cần được sắp xếp có n phần tử):

- Bước 1: Tìm giá trị nhỏ nhất của các phần tử trong mảng (tạm gọi là: minVal) và chỉ số của phần tử lớn nhất trong mảng (tạm gọi là: max).
- Bước 2: Khởi tạo 1 vector L có m phần tử (ứng với m lớp, ta thường chọn $m = \lfloor 0.45n \rfloor$, với 0.45 là hằng số tối ưu được các nhà khoa học máy tính tìm ra).

- Bước 3: Đếm số lượng phần tử các lớp theo quy luật, phần tử a[i] sẽ thuộc lớp k, với: $k = int\left(\left(m-1\right)*\left(a\left[i\right]-minVal\right)/\left(a\left[max\right]-minVal\right)\right)$.
- Bước 4: Tính vị trí kết thúc của phân lớp thứ j theo công thức L[j] = L[j] + L[j - 1] (j tăng từ 1 đến m - 1).

Sau khi đã có được vị trí kết thúc của các phân lớp ta tiến hành bước hoán vị toàn cục. Việc này sẽ hình thành các chu trình hoán vị: mỗi khi ta đem một phần tử ở đâu đó đến một vị trí nào đó thì ta phải nhấc phần tử hiện tại đang chiếm chỗ ra, tiếp tục với phần tử bị nhấc ra và đưa đến chỗ khác cho đến khi quay lại vị trí ban đầu thì hoàn tất vòng lặp.

Cuối cùng, sau bước hoán vị toàn cục, mảng của chúng ta hiện tại sẽ được chia thành các lớp (thứ tự các phần tử trong lớp vẫn chưa đúng) do đó để đạt được trạng thái đúng thứ tự thì khoảng cách phải di chuyển của các phần tử là không lớn vì vậy Insertion Sort sẽ là thuật toán thích hợp nhất để sắp xếp lại mảng có trạng thái như vậy.

Phân tích:

- Giả sử khi ta chạy trên một mảng có 100 phần tử sẽ mất chi phí O(1).
- Với mảng có n phần tử, khi ta chia nó thành các mảng đủ nhỏ có chi phí O(1) thì chi phí duyệt mảng chỉ là O(n/100).
- Ngoài ra, ta còn tốn thêm:
    + Chi phí để tìm min/max là O(n).
    + Chi phí tạo mảng động là O(n).
    + Chi phí sắp xếp là O(n).

    ⇨ Tổng chi phí chỉ là O(n).

Độ phức tạp thuật toán:

- Độ phức tạp về thời gian:
    + Best-case: $O(n)$
        - Trong trường hợp mảng n phần tử được phân bố đều vào m nhóm.
        - Các phần tử trong mỗi nhóm đã được sắp xếp một phần (hoặc hoàn toàn).
    + Average-case: $O(n+r)$.

+ Worst-case: $O(n^2)$

    - Trong trường hợp mảng không phân bố đều vào m nhóm.
    - Các nhóm đều có thứ tự giảm dần.

- Độ phức tạp về không gian: $O(m)$ – với là kích thước của mãng L ($m = \lfloor 0.45n \rfloor$) .

Đặc điểm:

- Cách cài đặt tương đối phức tạp.
- Flash Sort là một thuật toán sắp xếp tại chỗ (in-situ, tức là không dùng thêm mảng phụ trợ khi sort – đặc điểm được kế thừa từ Insertion Sort).
- Tốc độ thực thi rất nhanh, xấp xỉ $O(n)$, thậm chí nhanh hơn cả Quick Sort…
- Tương đối tiết kiệm bộ nhớ nếu so với Merge Sort…
- Tuy nhiên, đây là một thuật toán không stable (vị trí tương đối của các phần từ cùng key có thể bị xáo trộn).

D. KẾT QUẢ THỬ NGHIỆM VÀ NHẬN XÉT

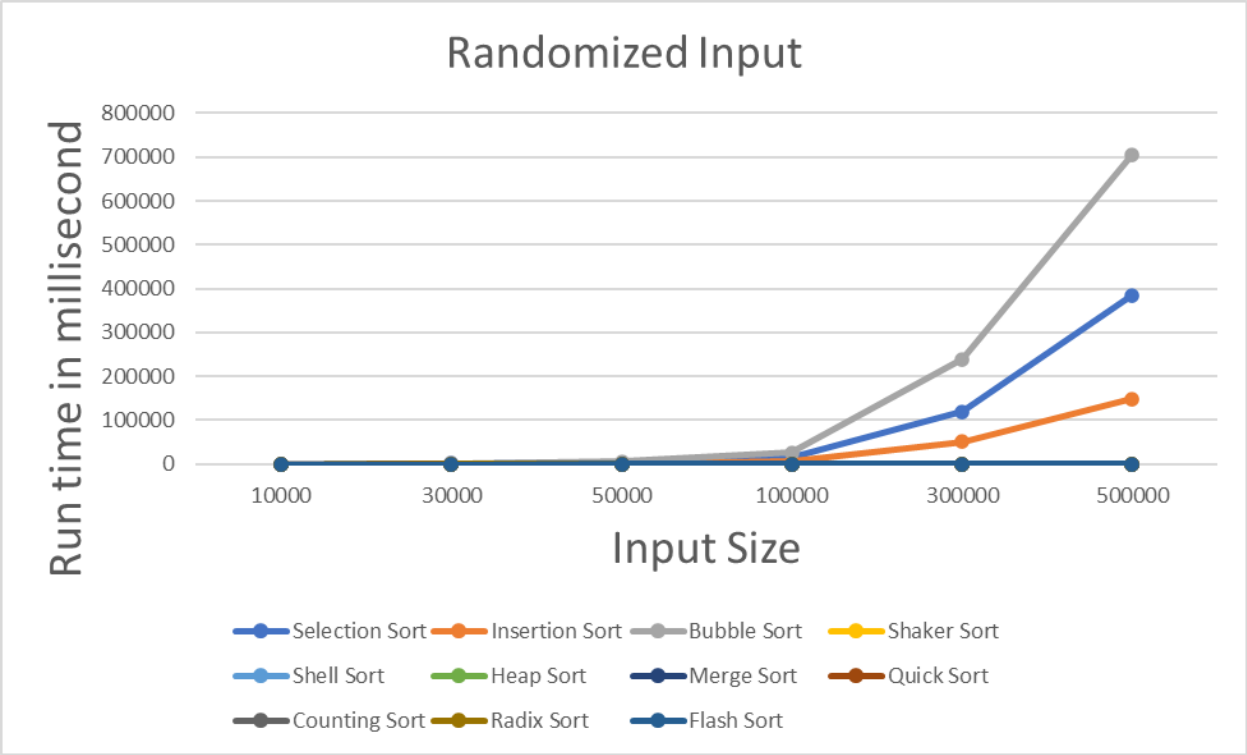
Data order: Randomized												
Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	154 ms	100,010,001	1182 ms	900,030,001	3279 ms	2,500,050,001	15399 ms	10,000,100,001	120283 ms	90,000,300,001	385329 ms	250,000,500,001
Insertion Sort	52 ms	25,211,380	457 ms	225,682,802	1263 ms	624,643,031	5217 ms	2,501,171,774	51803 ms	22,492,640,028	149156 ms	62,537,755,868
Bubble Sort	225 ms	100,009,999	2112 ms	900,029,999	5943 ms	2,500,049,999	26649 ms	10,000,099,999	239357 ms	90,000,299,999	705384 ms	250,000,499,999
Shaker Sort	1 ms	39,998	1 ms	119,998	1 ms	199,998	1 ms	399,998	2 ms	1,199,998	5 ms	1,999,998
Shell Sort	2 ms	624,254	6 ms	2,243,635	11 ms	4,388,198	30 ms	10,232,607	83 ms	33,914,122	151 ms	65,402,913
Heap Sort	1 ms	257,824	3 ms	825,637	5 ms	1,354,663	10 ms	2,939,704	33 ms	9,249,043	55 ms	15,483,742
Merge Sort	2 ms	452,844	8 ms	1,502,077	19 ms	2,616,656	26 ms	5,533,614	78 ms	18,003,904	130 ms	31,156,164
Quick Sort	1 ms	181,779	2 ms	602,767	4 ms	1,071,662	7 ms	2,267,616	25 ms	7,200,390	42 ms	12,397,150
Counting Sort	1 ms	30,003	1 ms	90,003	1 ms	150,003	1 ms	300,003	2 ms	900,003	7 ms	1,500,003
Radix Sort	1 ms	140,058	2 ms	510,072	2 ms	850,072	6 ms	1,700,072	21 ms	5,100,072	35 ms	8,500,072
Flash Sort	1 ms	94,164	1 ms	292,915	2 ms	476,986	3 ms	889,197	10 ms	2,538,097	19 ms	4,288,995

Data order: Nearly sorted												
Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	156 ms	100,010,001	1194 ms	900,030,001	3308 ms	2,500,050,001	16484 ms	10,000,100,001	130053 ms	90,000,300,001	396768 ms	250,000,500,001
Insertion Sort	1 ms	98,659	1 ms	262,191	1 ms	289,379	1 ms	365,471	2 ms	774,823	3 ms	1,206,891
Bubble Sort	133 ms	100,009,999	1298 ms	900,029,999	3718 ms	2,500,049,999	17815 ms	10,000,099,999	158561 ms	90,000,299,999	412863 ms	250,000,499,999
Shaker Sort	1 ms	36,468	1 ms	114,874	1 ms	165,176	1 ms	265,318	1 ms	660,026	1 ms	1,065,026
Shell Sort	1 ms	400,508	3 ms	1,329,549	4 ms	2,233,572	10 ms	4,720,284	24 ms	15,443,419	40 ms	25,683,292
Heap Sort	1 ms	96,247	1 ms	278,602	1 ms	442,786	3 ms	873,946	6 ms	2,603,278	9 ms	4,364,053
Merge Sort	2 ms	417,641	6 ms	1,375,286	15 ms	2,345,825	18 ms	4,905,110	53 ms	15,778,531	93 ms	27,188,534
Quick Sort	1 ms	154,988	1 ms	501,974	2 ms	913,899	5 ms	1,927,728	10 ms	6,058,281	18 ms	10,310,770
Counting Sort	1 ms	30,003	1 ms	90,003	1 ms	150,003	1 ms	300,003	2 ms	900,003	4 ms	1,500,003
Radix Sort	1 ms	140,058	2 ms	510,072	2 ms	850,072	7 ms	1,700,072	25 ms	6,000,086	40 ms	10,000,086
Flash Sort	1 ms	123,465	1 ms	370,461	1 ms	617,457	2 ms	1,234,961	8 ms	3,704,961	13 ms	6,174,959

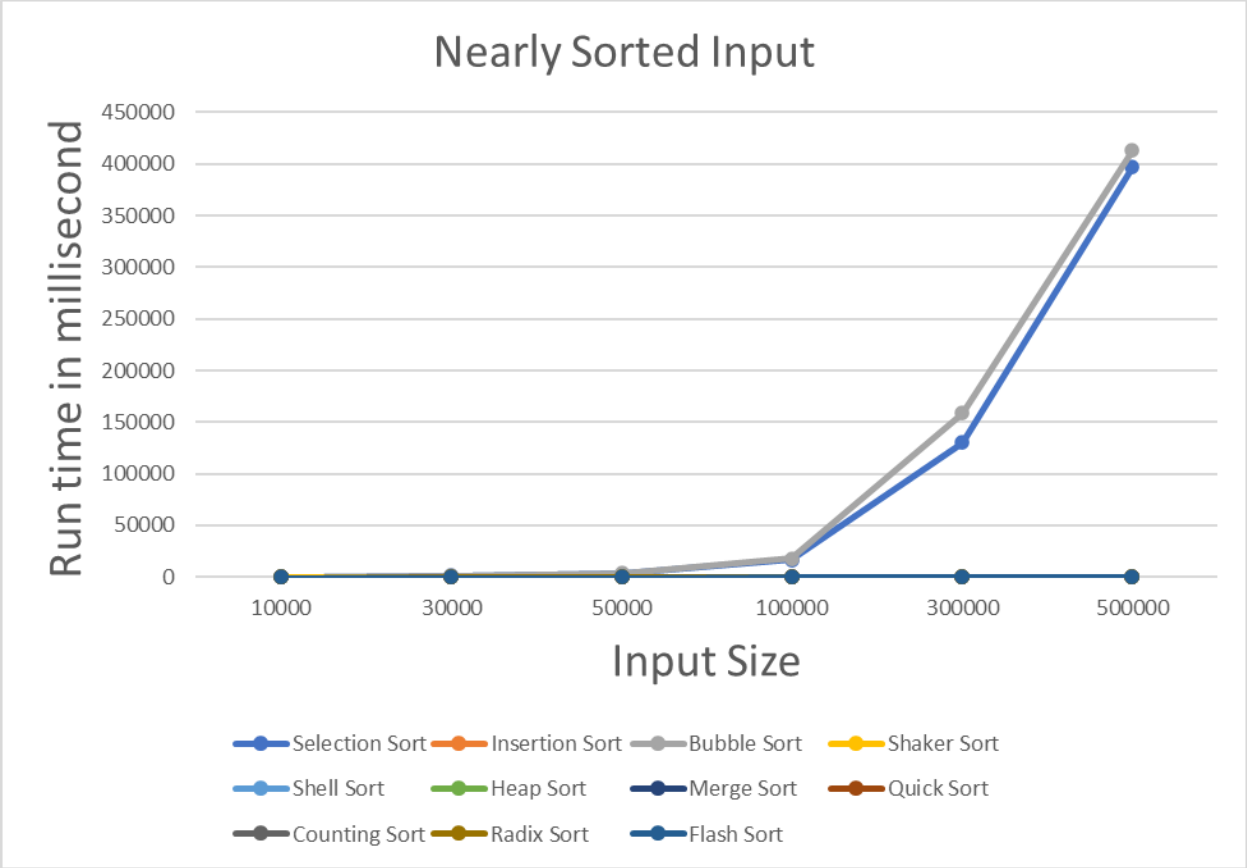
Data order: Sorted												
Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	164 ms	100,010,001	1231 ms	900,030,001	3336 ms	2,500,050,001	16822 ms	10,000,100,001	133228 ms	90,000,300,001	395768 ms	250,000,500,001
Insertion Sort	1 ms	20,001	1 ms	60,001	1 ms	100,001	1 ms	200,001	1 ms	600,001	3 ms	1,000,001
Bubble Sort	126 ms	100,009,999	1175 ms	900,029,999	3391 ms	2,500,049,999	16086 ms	10,000,099,999	141014 ms	90,000,299,999	410314 ms	250,000,499,999
Shaker Sort	1 ms	20,002	1 ms	60,002	1 ms	100,002	1 ms	200,002	1 ms	600,002	1 ms	1,000,002
Shell Sort	1 ms	360,042	1 ms	1,170,050	3 ms	2,100,049	8 ms	4,500,051	24 ms	15,300,061	40 ms	25,500,058
Heap Sort	1 ms	93,535	1 ms	277,030	1 ms	442,795	1 ms	873,967	7 ms	2,603,092	10 ms	4,363,573
Merge Sort	1 ms	401,834	6 ms	1,323,962	9 ms	2,301,434	18 ms	4,852,874	49 ms	15,729,866	89 ms	27,143,914
Quick Sort	1 ms	154,960	1 ms	501,930	2 ms	913,851	3 ms	1,927,692	11 ms	6,058,229	19 ms	10,310,734
Counting Sort	1 ms	30,003	1 ms	90,003	1 ms	150,003	1 ms	300,003	3 ms	900,003	4 ms	1,500,003
Radix Sort	1 ms	140,058	1 ms	510,072	3 ms	850,072	6 ms	1,700,072	27 ms	6,000,086	40 ms	10,000,086
Flash Sort	1 ms	123,491	1 ms	370,491	1 ms	617,491	2 ms	1,234,991	7 ms	3,704,991	13 ms	6,174,991

Data order: Reverse sorted												
Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	161 ms	100,010,001	1231 ms	900,030,001	3512 ms	2,500,050,001	14611 ms	10,000,100,001	143706 ms	90,000,300,001	415981 ms	250,000,500,001
Insertion Sort	106 ms	50,015,001	948 ms	450,045,001	2559 ms	1,250,075,001	10577 ms	5,000,150,001	105669 ms	45,000,450,001	311741 ms	125,000,750,001
Bubble Sort	161 ms	100,009,999	1500 ms	900,029,999	4309 ms	2,500,049,999	20626 ms	10,000,099,999	185045 ms	90,000,299,999	495842 ms	250,000,499,999
Shaker Sort	1 ms	39,998	1 ms	119,998	1 ms	199,998	1 ms	399,998	3 ms	1,199,998	4 ms	1,999,998
Shell Sort	1 ms	475,175	3 ms	1,554,051	4 ms	2,844,628	11 ms	6,089,190	34 ms	20,001,852	57 ms	33,857,581
Heap Sort	2 ms	385,648	4 ms	1,282,420	7 ms	2,285,065	15 ms	4,844,227	47 ms	15,966,193	82 ms	27,856,483
Merge Sort	1 ms	401,834	7 ms	1,323,962	11 ms	2,301,434	17 ms	4,852,874	51 ms	15,729,866	99 ms	27,143,914
Quick Sort	1 ms	164,957	2 ms	531,927	3 ms	963,848	4 ms	2,027,689	12 ms	6,358,226	19 ms	10,810,731
Counting Sort	1 ms	30,003	1 ms	90,003	1 ms	150,003	1 ms	300,003	3 ms	900,003	5 ms	1,500,003
Radix Sort	1 ms	140,058	1 ms	510,072	3 ms	850,072	6 ms	1,700,072	25 ms	6,000,086	39 ms	10,000,086
Flash Sort	1 ms	106,000	1 ms	318,000	1 ms	530,000	2 ms	1,060,000	6 ms	3,180,000	13 ms	5,300,000

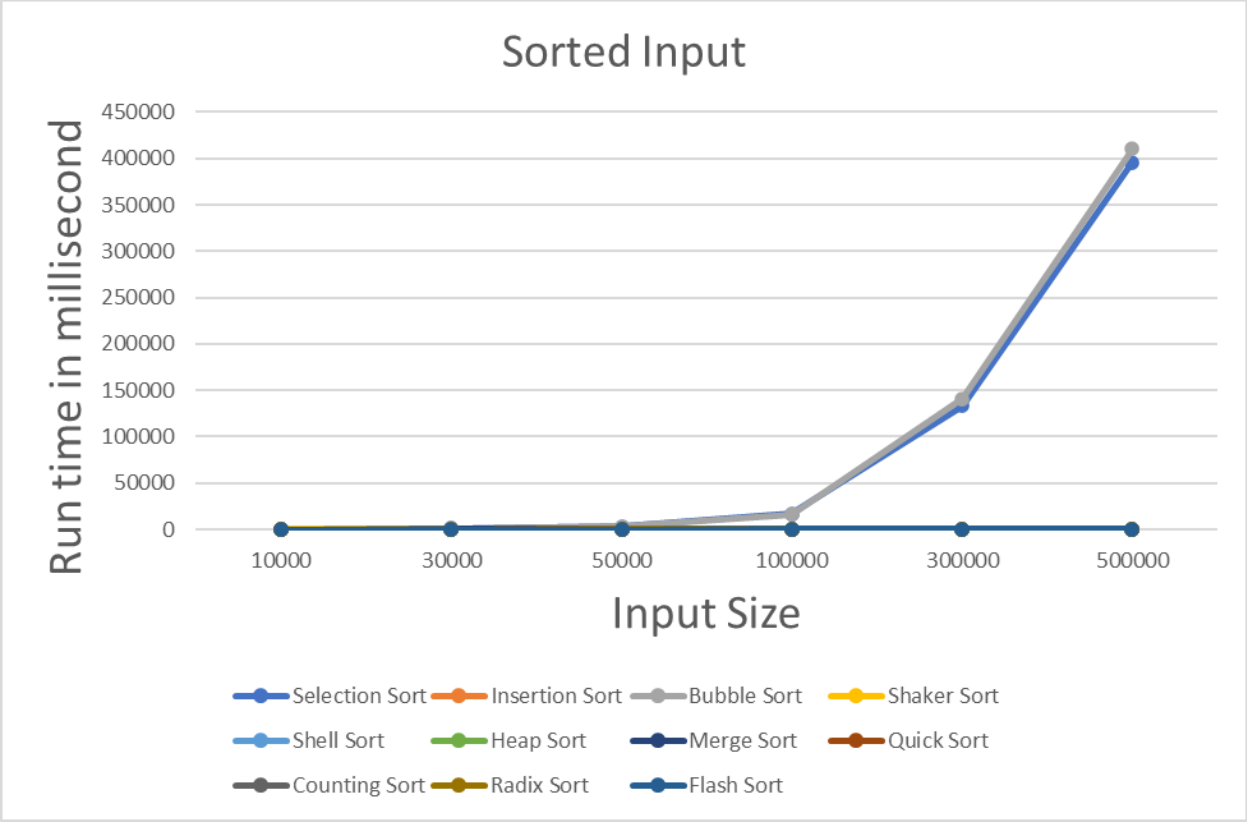
THỜI GIAN CHẠY



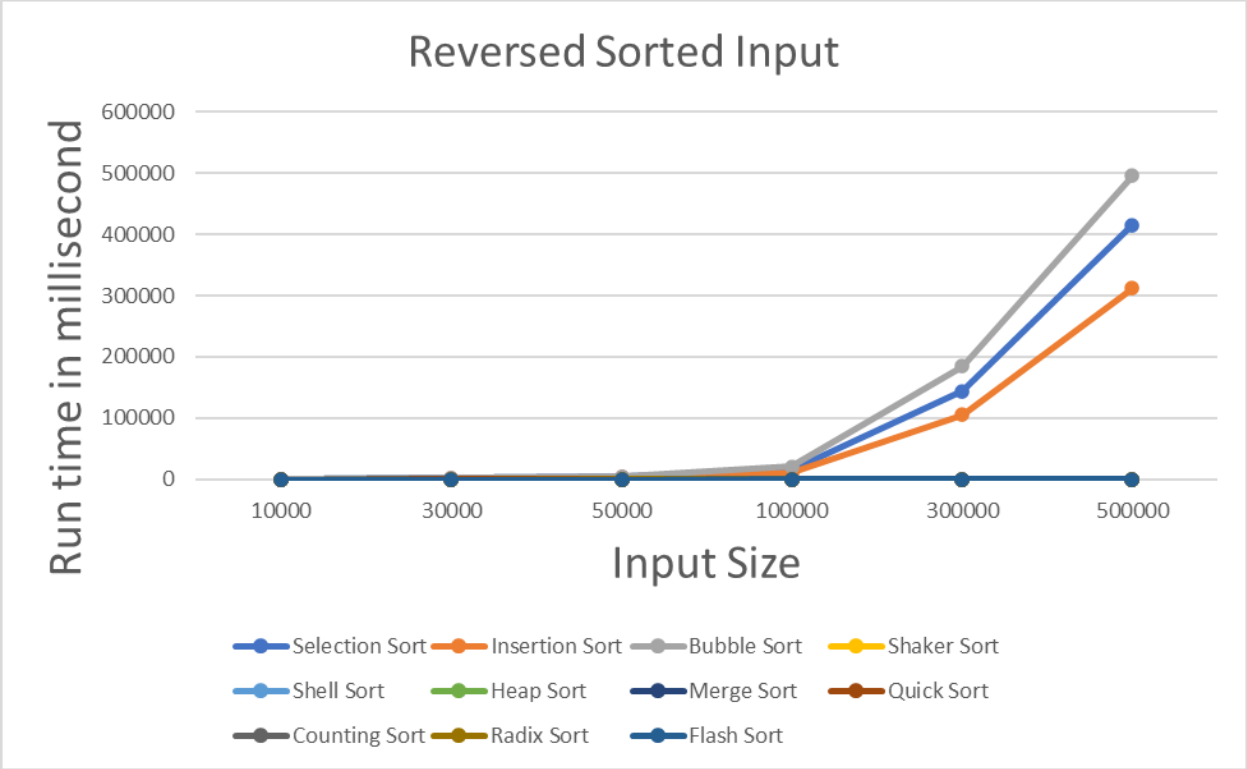
- Có thể thấy rằng, trong dữ liệu random, thuật toán Bubble Sort có thời gian chạy chậm nhất, lần lượt sau đó là thuật toán Selection Sort, Insertion Sort. Các thuật toán còn lại có thời gian chạy gần như như nhau.



- Trong trường hợp mảng xếp gần như tăng dần, cũng như là trường hợp gần như là tốt nhất của đa số các thuật toán sắp xếp nên thời gian chạy giảm đi rất nhiều so với trường hợp mảng xếp ngẫu nhiên. Tuy nhiên thì thuật toán Bubble Sort và Selection Sort vẫn có sự chênh lệch nhiều so với các thuật toán còn lại.

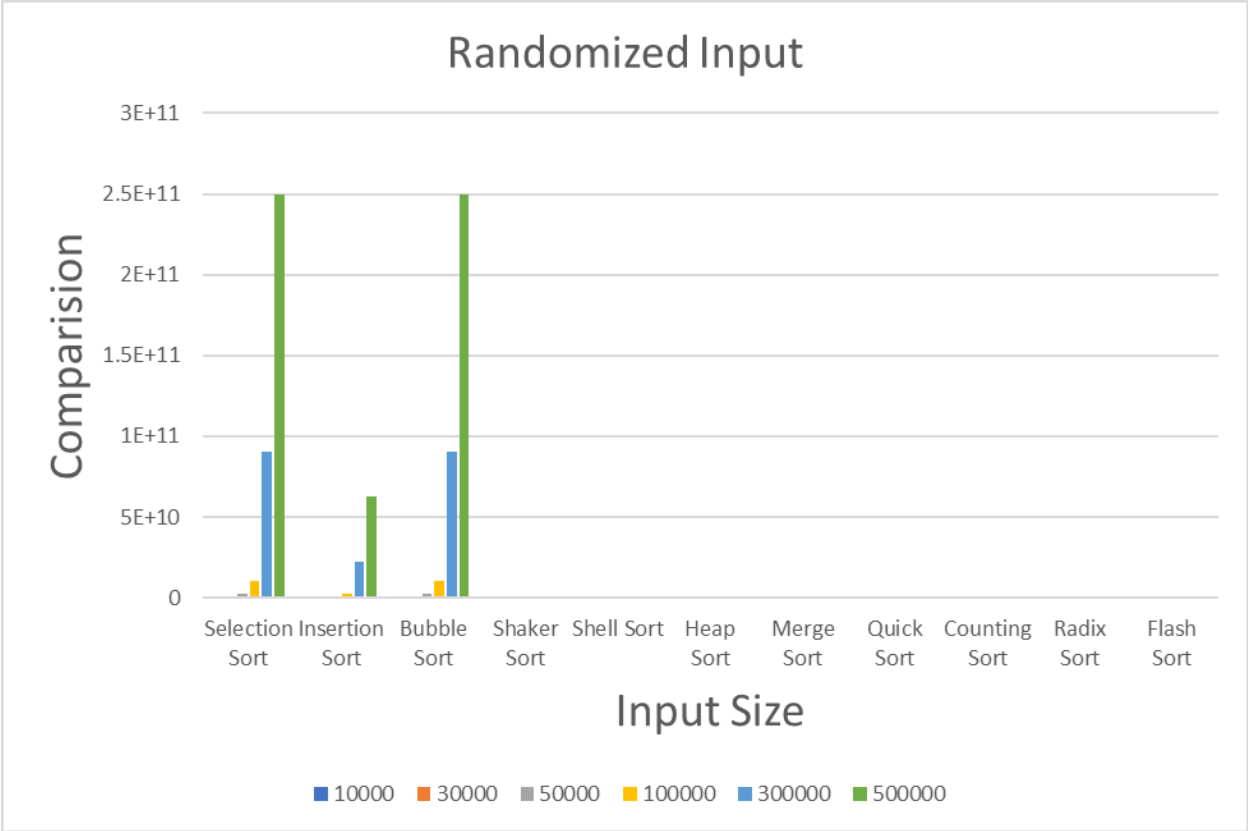


- Trong trường hợp mảng xếp tăng dần, đây là trường hợp tốt nhất của đa số thuật toán như: Bubble Sort, Insertion Sort,… nên biểu đồ có sự thay đổi nhẹ về mặt thời gian so với trường hợp mảng gần như xếp tăng dần. Bubble Sort và Selection Sort vẫn chạy lâu nhất.



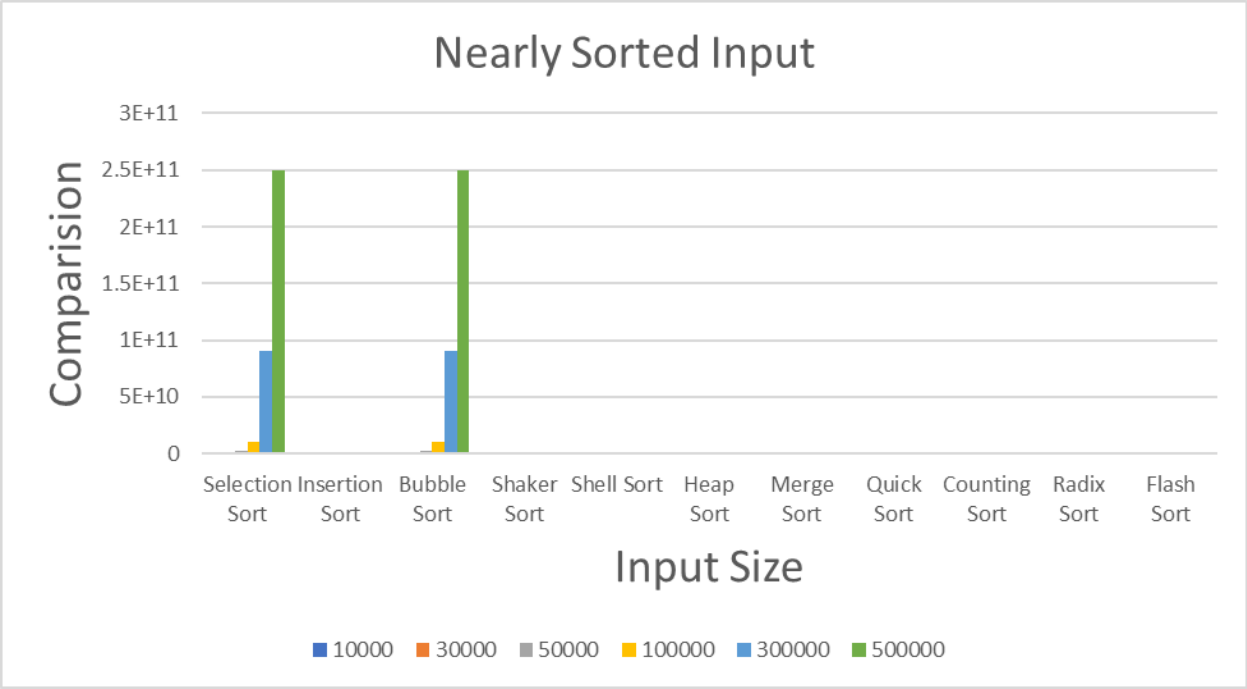
- Trường hợp mảng xếp ngược là trường hợp xấu nhất cho Bubble Sort, Selection Sort, Insertion Sort, tuy nhiên thời gian chạy lại nhanh hơn so với mảng xếp ngẫu nhiên.

SỐ PHÉP SO SÁNH

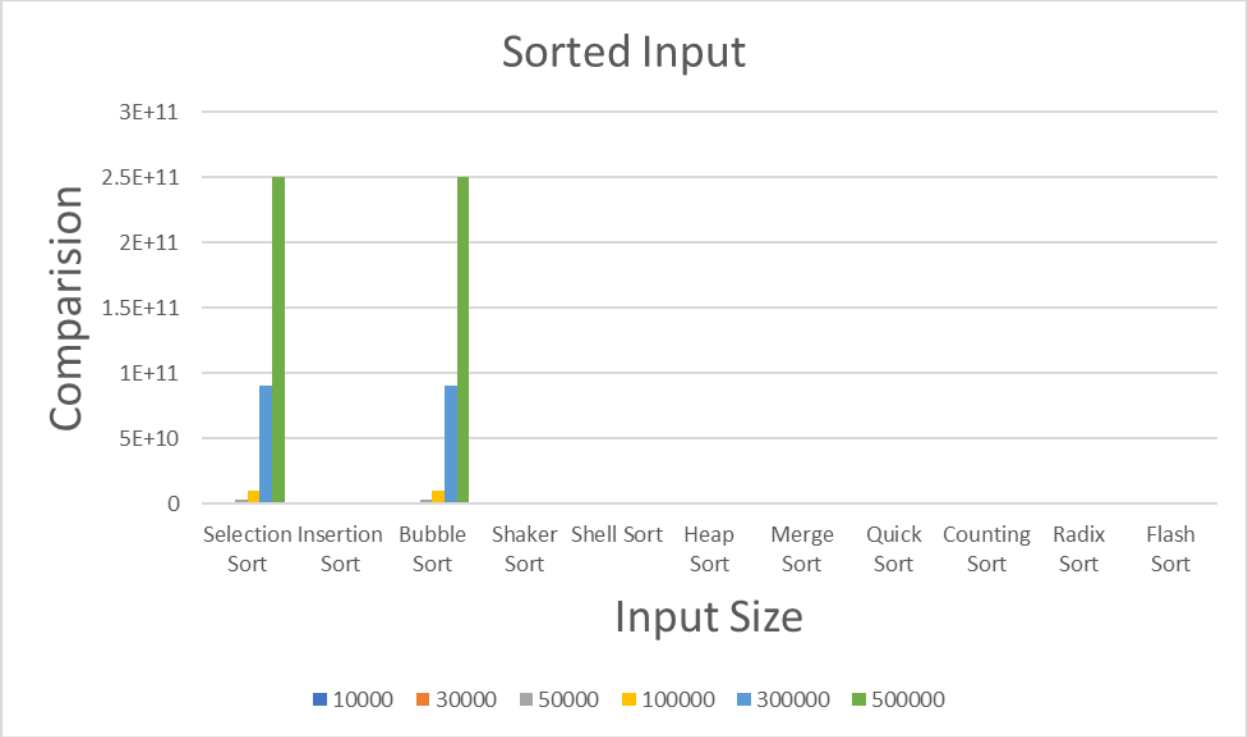


- Trong trường hợp mảng xếp ngẫu nhiên, vẫn là 3 thuật toán Selection Sort, Insertion Sort, Bubble Sort có sự chênh lệch nhiều nhất so với các thuật toán khác. Tuy nhiên thuật toán Insertion Sort sử dụng ít hơn đáng kể phép so sánh so với 2 thuật toán còn lại. 2 thuật toán Selection Sort và Bubble Sort sử dụng gần như bằng nhau các phép so sánh để sắp xếp mảng.

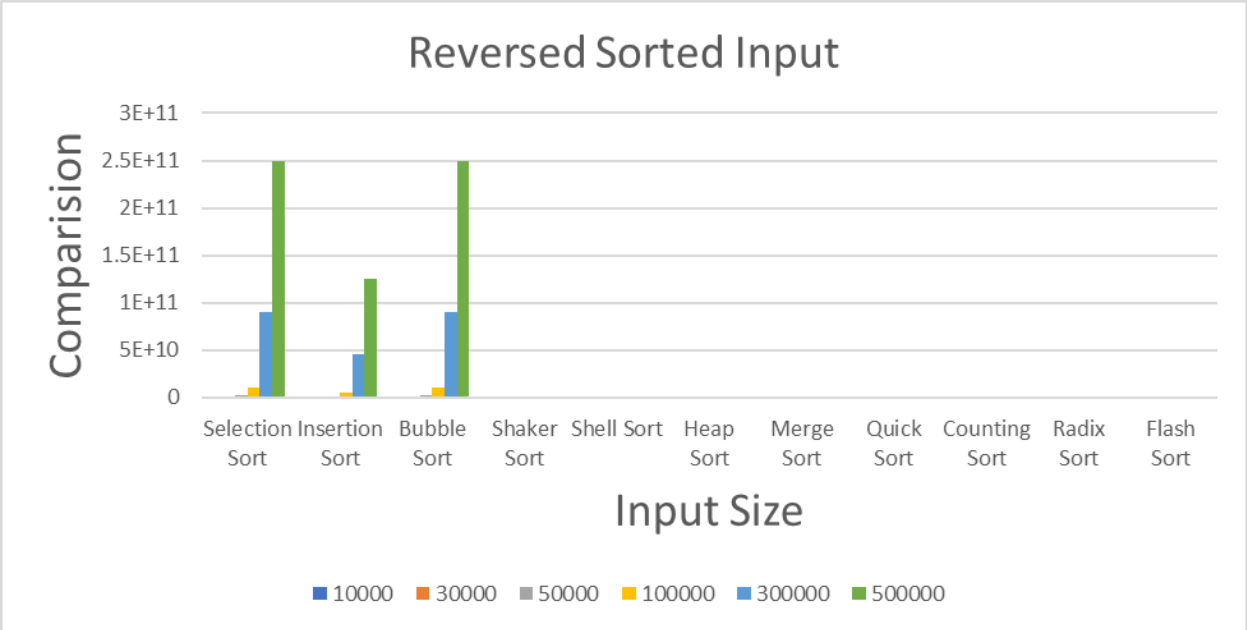
- Các thuật toán còn lại sử dụng rất ít phép so sánh, gần như không đáng kể.



- Trong trường hợp mảng xếp gần như tăng dần, các thuật toán khác ngoại trừ Selection Sort và Bubble Sort đều sử dụng rất ít phép so sánh. Do 2 thuật toán Selection Sort và Bubble Sort trong mọi trường hợp đều sử dụng như nhau các phép so sánh, số phép so sánh chỉ thay đổi khi kích thước mảng thay đổi.



- Tương tự với mảng xếp tăng dần thì Selection Sort và Bubble Sort vẫn giữ nguyên số phép so sánh.



- Trong trường hợp mảng xếp ngược, đây là trường hợp xấu nhất của Insertion Sort nên số phép so sánh của thuật toán này cũng tăng lên. Số phép so sánh của các thuật toán khác vẫn không đáng kể.

Tổng kết:

Khi so sánh thời gian chạy, Bubble Sort chạy chậm nhất, sau đó là Selection Sort và Insertion Sort. Điều ngạc nhiên ở đây là trong các trường hợp, theo lí thuyết Shaker Sort sẽ có độ phức tạp tương đương với 3 thuật toán trên nhưng ở đây Shaker Sort lại chạy rất nhanh, tương đương với các thuật toán có phân lớp O(NlogN) như Heap Sort và Merge Sort.

Khi so sánh số phép so sánh, Bubble Sort và Selection Sort có số phép so sánh như nhau trong mọi trường hợp, chỉ phụ thuộc vào kích thước mảng. Insertion Sort có số phép so sánh ít hơn so với 2 thuật toán trên. Các thuật toán còn lại có số phép so sánh rất ít, không đáng kể.

E. TỔ CHỨC DỰ ÁN VÀ GHI CHÚ

Project được tạo trên Github để các thành viên có thể dễ dàng chỉnh sửa code (https://github.com/LightKod/Lab3).

Source code sẽ gồm 4 phần chính:

- main.cpp: là phần chính của Project, chứa các hàm xử lý command line và hàm main().
- DataGenerator.cpp: chứa các hàm tạo mảng ngẫu nhiên, sắp xếp,… do giảng viên cung cấp
- Sort.cpp: chứa các thuật toán sort
- Utility.cpp: chứa các hàm hỗ trợ như nhập / xuất mảng từ file, kiểm tra số nguyên,…

Project chỉ sử dụng thư viện chuẩn của C / C++ ngoài ra không sử dụng thư viện bên ngoài nào.

F. DANH SÁCH TÀI LIỆU THAM KHẢO

https://nguyenvanhieu.vn/thuat-toan-sap-xep-selection-sort/ - tham khảo thuật toán Selection Sort.

https://xlinux.nist.gov/dads/HTML/bingosort.html- tham khảo thuật toán Bingo Sort.

https://en.wikipedia.org/wiki/Selection_sort#Variants - tham khảo biến thể Selection Sort.

https://brilliant.org/wiki/insertion - tham khảo thuật toán Insertion Sort.

https://ieeexplore.ieee.org/abstract/document/7443165/authors#authors- tham khảo thuật toán Library Sort.

https://en.wikipedia.org/wiki/Insertion_sort - tham khảo biến thể Insertion Sort.

https://www.geeksforgeeks.org/bubble-sort/ - tham khảo thuật toán Bubble Sort.

https://en.wikipedia.org/wiki/Odd–even_sort - tham khảo thuật toán Odd-even Sort.

https://en.wikipedia.org/wiki/Cocktail_shaker_sort - Tham khảo ý tưởng và mã giả Shaker Sort.

geeksforgeeks - tham khảo: ý tưởng, cách hoạt động, mã giả, độ phức tạp, đặc điểm Shell Sort.

tek4.vn - tham khảo: trình tự tối ưu, đặc điểm, độ phức tạp thuật toán Shell Sort.

https://duongdinh24.com/thuat-toan-heap-sort/ - tham khảo thuật toán Heap Sort

https://en.wikipedia.org/wiki/Heapsort#Variations- tham khảo biến thể thuật toán Heap Sort

https://nguyenvanhieu.vn/thuat-toan-sap-xep-merge-sort/ - tham khảo thuật toán Merge Sort

https://stackoverflow.com/questions/10342890/merge-sort-time-and-space-complexity - tham khảo độ phức tạp thuật toán Merge Sort

www.geeksforgeeks.org/3-way-merge-sort - tham khảo thuật toán 3-Way Merge Sort

https://stackoverflow.com/questions/35201963/differences-in-variations-of-mergesort - tham khảo biến thể thuật toán Merge Sort

https://en.wikipedia.org/wiki/Quicksort - Tham khảo ý tưởng và mã giả thuật toán Quick Sort.

tek4.vn – tham khảo: ý tưởng, cách hoạt động, độ phức tạp thuật toán Counting Sort.

wikipedia - tham khảo: ý tưởng, mã giả thuật toán Counting Sort.

geeksforgeeks - tham khảo: đặc điểm, độ phức tạp thuật toán và cải tiến để thuật toán có thể hoạt động với số âm thuật toán Counting Sort.

https://en.wikipedia.org/wiki/Radix_sort - Tham khảo ý tưởng và mã giả thuật toán Radix Sort

codelearn.io - tham khảo: ý tưởng, cách hoạt động, đặc điểm thuật toán Flash Sort

codelearn.io - tham khảo: độ phức tạp thuật toán Flash Sort

neubert.net - tham khảo: ý tưởng, đặc điểm thuật toán Flash Sort