VNUHCM - UNIVERSITY OF SCIENCE

SORTING ALGORITHM

Data Structure & Algorithms

—

Teacher's name

Dr. Chau Thanh Duc
Lecturer. Ngo Dinh Hy
Lecturer. Phan Thi Phuong Uyen

—

Student

Ngo Huy Anh
19127095
19CLC6

Contents

I. INSTALLED SORTING ALGORITHMS

1. Selection Sort

i. Ideas

Thuật toán Selection Sort là thuật toán sắp xếp một mảng bằng cách đi tìm phần tử có giá trị nhỏ nhất(Giả sử với sắp xếp mảng tăng dần) trong đoạn đoạn chưa được sắp xếp và đổi cho phần tử nhỏ nhất đó với phần tử ở đầu đoạn chưa được sắp xếp.

Thuật toán sẽ chia mảng thành 2 phần đã sắp xếp và chưa sắp xếp. Tìm phần tử nhỏ nhất trong mảng chưa được sắp xếp và đưa nó về đúng vị trí (đổi chỗ với phần tử đầu tiên của mảng chưa được sắp xếp). Sau mỗi lần chọn và sắp xếp, tăng số lượng phần tử của mảng sắp xếp lên và giảm số lượng phần tử của mảng chưa sắp xếp. Lặp lại cho đến khi mảng chưa sắp xếp còn 1 phần tử.

ii. Implement Algorithm (Step – by – step)

- Step 1 : Khai báo vị trí bắt đầu i = 0
- Step 2 : Triển khai vòng lặp j chạy từ i + 1 đến n - 1
  - Tìm phần tử nhỏ nhất a[min]
  - Đổi chỗ a[min] và a[i]
- Step 3 : So sánh i với n:
  - Nếu i < n, tăng i lên và quay lại bước 2
  - Ngược lại, dừng vòng lặp

iii. Time & Space Complexity

Time Complexity:

Selection Sort thực hiện 2 thao tác: Duyệt qua mảng và tìm ra phần tử nhỏ nhất trong dãy chưa được sắp xếp.

Duyệt qua mảng cần phải quét qua n phần tử, tìm phần tử thấp nhất cần quét qua n – 1 phần tử (n – 1 phép so sánh). Khi đó, tổng số phép so sánh là : 1 + 2 + … + n – 2 + n – 1 = $\frac{(n-1)*(n-1+1)}{2}$ = $\frac{n*(n-1)}{2}$. Do đó, độ phức tạp thuật toán là $O(n^2)$.

Bất kì trường hợp nào cũng phải duyệt qua từng phần tử của mảng, sau đó quét qua n – 1 phần tử để tìm phần tử nhỏ nhất. Do đó, độ phức tạp thuật toán trong tất cả trường hợp luôn là $O(n^2)$.

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

Space Complexity: O(1)

iv. Comments

Thuật toán ít phải đổi chỗ các phần tử nhất trong số các thuật toán sắp xếp(n lần hoán vị) nhưng có độ phức tạp so sánh là $O(n^2)$ ($n^2/2$ phép so sánh).

Thuật toán tốn thời gian gần như bằng nhau đối với mảng đã được sắp xếp cũng như mảng chưa được sắp xếp.

2. Insertion Sort

i. Ideas

Insertion Sort hoạt động giống như cách mà chúng ta sắp xếp quân bài.

Ta thống nhất với nhau rằng quân bài thứ 1 đã được sắp xếp. Ta bắt đầu xét từ quân bài thứ 2, nếu quân bài thứ 2 có giá trị nhỏ hơn quân bài thứ 1 thì ta sẽ đặt nó qua bên trái quân bài thứ 1 (Lúc này quân bài thứ 1 sẽ trở thành quân bài thứ 2), còn không thì đặt qua bên phải. Ta xét đến quân bài thứ 3, nếu quân bài thứ 3 nhỏ hơn quân bài thứ 2 thì ta so sánh tiếp với quân bài thứ 1, nếu quân bài thứ 3 nhỏ hơn quân bài thứ 1 thì ta sẽ đặt nó qua bên trái quân bài thứ 1. Tức là, ta sẽ kiếm quân bài nào nhỏ hơn quân bài ta đang xét và đặt qua bên phải nó ! Tương tự, các quân bài chưa được sắp xếp sẽ được xét và đặt vào đúng vị trí phù hợp.

ii. Implement Algorithm (Step – by – step)

Thống nhất rằng vị trí bắt đầu của phần tử là 0.

- Step 1 : Chạy vòng lặp từ phần tử thứ 1 đến n - 1
- Step 2 : Xét từ vị trí j = i – 1 cho đến 0
- Step 3 : So sánh với tất cả các phần tử của mảng con đã qua sắp xếp để tìm ra vị trí của phần tử nhỏ hơn phần tử đang xét
- Step 4 : Di chuyển tất cả các phần tử trong mảng con từ vị trí thoả mãn đến vị trí i - 1 qua bên phải 1 ô
- Step 5 : Chèn giá trị đó vào vị trí thoả mãn
- Step 6 : Lặp lại cho đến khi danh sách được sắp xếp.

iii. Time & Space Complexity

- Time Complexity

  Insertion Sort thực hiện 2 thao tác: Duyệt qua mảng và so sánh phần tử đang xét trong khoảng từ 0 đến i – 1, di chuyển các phẩn tử.

  Khi mảng đầu vào đã được sắp xếp theo thứ tự, thuật toán sẽ không thực hiện nhiệm vụ so sánh và di chuyển các phần tử (Vòng lặp bên trong không được kích hoạt do key > array[j]). Do đó, trường hợp tốt nhất, độ phức tạp thuật toán là O(n).

  Khi mảng đầu vào được sắp xếp theo thứ tự ngược lại. Khi i = 1, lúc này ta sẽ cần 1 phép so sánh và dịch chuyển. Khi i = 2, lúc này ta sẽ cần 2 phép so sánh và dịch chuyển. Tuần tự như vậy, khi i = n – 1, lúc này ta sẽ cần n – 1 phép so sánh và dịch chuyển. Khi đó, tổng chi phí thực hiện thuật toán là: 2 * (1 + 2 + … + n – 2 + n – 1) = $\frac{2 * (n-1) * (n - 1 + 1)}{2}$ = n * (n – 1). Khi đó, tổng chi phí mà ta sử dụng là : $O(n^2)$. Do đó, trường hợp tệ nhất, độ phức tạp thuật toán là $O(n^2)$.

  Trong trường hợp trung bình, giả sử ta có $\frac{n}{2}$ phép duyệt và $\frac{n}{2}$ phép so sánh, dịch chuyển. Khi đó, độ phức tạp thuật toán sẽ là : $\frac{2 * \frac{n}{2} * (\frac{n}{2} + 1)}{2} = \frac{n^2}{4} \approx n^2$.

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

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

- Space Complexity
  Space Complexity là O(1) bởi vì ta có sử dụng biến key để lưu tạm giá trị của phần tử đang xét.

iv. Comments

Insertion Sort thường được sử dụng khi :
- Số phần tử của một mảng là số nhỏ.
- Mảng gần như đã được sắp xếp.

3. Binary – Insertion Sort

i. Ideas

Tương tự Insertion Sort nhưng dùng tìm kiếm nhị phân thay cho tìm kiếm tuyến tính.

ii. Implement Algorithm (Step – by – step)

Thống nhất rằng vị trí bắt đầu của phần tử là 0.
- Step 1 : Chạy vòng lặp từ phần tử thứ 1 đến n - 1
- Step 2 : Áp dụng thuật toán Binary Search. So sánh với tất cả các phần tử của mảng con đã qua sắp xếp để tìm ra vị trí của phần tử nhỏ hơn phần tử đang xét
- Step 3 : Di chuyển tất cả các phần tử trong mảng con từ vị trí thoả mãn đến vị trí i - 1 qua bên phải 1 ô
- Step 4 : Chèn giá trị đó vào vị trí thoả mãn
- Step 5 : Lặp lại cho đến khi danh sách được sắp xếp.

iii. Time & Space Complexity

- Time Complexity
  Tương tự như Insertion Sort nhưng :

  Trong trường hợp tốt nhất – Mảng đã được sắp xếp. Khi đó, ta không thể kiếm ra phần tử thoả mãn. Điều này dẫn đến độ phức tạp thuật toán Binary Search sẽ xảy ra trường hợp xấu nhất là $O(\log_2 n)$. Khi đó độ phức tạp là O(n*logn)

Trong trường hợp trung bình và xấu nhất. Độ phức tạp thuật toán vẫn là O(n^2). Mặc dù ta giảm số lần so sánh xuống còn O(logn) nhưng max(O(logn), O(n - 1)) là O(n - 1) – với O(n – 1) là chi phí thực hiện phép dịch chuyển. Dẫn đến độ phức tạp thuật toán vẫn là O(n^2).

- Best case: O($n*\log n$)
- Average case: O(n^2).
- Worst case: O(n^2).

- Space Complexity
  Space Complexity là O(1).

iv. Comments

4. Bubble Sort

i. Ideas

Bubble Sort là một thuật toán sắp xếp đơn giản - như cái tên của nó, thuật toán sắp xếp bằng cách đẩy phần tử lớn nhất xuống cuối dãy, đồng thời những phần tử có giá trị nhỏ hơn sẽ dịch chuyển dần về đầu dãy. Tựa như sự nổi bọt vậy, những phần tử nhẹ hơn sẽ nổi lên trên và ngược lại, những phần tử lớn hơn sẽ chìm xuống dưới.

Đầu tiên, ta chạy 1 vòng lặp để ghi nhớ vị trí bắt đầu của những phần tử chưa được sắp xếp (duyệt qua mảng). Sau đó ta duyệt mảng từ phần tử cuối cùng đến vị trí đã ghi nhớ. Ta sẽ so sánh mỗi phần tử với phần tử liền trước nó, nếu chúng đứng sai vị trí, ta sẽ đổi chỗ chúng cho nhau. Quá trình này sẽ được dừng khi vòng lặp bên ngoài kết thúc.

ii. Implement Algorithm (Step – by – step)

- Step 1 : Chạy vòng lặp từ phần tử thứ 0 đến n – 1.
- Step 2 : Chạy vòng lặp từ phần tử thứ n – 1 đến phần tử thứ i.
- Step 3 : Nếu giá trị phần tử thứ j nhỏ hơn giá trị phần tử thứ j – 1 thì đổi chỗ chúng.

iii. Time & Space Complexity

- Time Complexity

Bubble Sort thực hiện 2 thao tác : Duyệt qua mảng và so sánh.

Khi i = 1, chương trình thực hiện n – 1 phép so sánh. Khi i = 2, chương trình thực hiện n - 2 phép so sánh. Khi i = 3, chương trình thực hiện n – 3 phép so sánh. Tuần tự như vậy, khi i = n – 1, chương trình thực hiện 1 phép so sánh. Khi đó, tổng chi phí thực hiện là :
1 + 2 + 3 + 4 + … + (n – 2) + (n – 1) = $\frac{(n-1)*(n-1+1)}{2}$ = $\frac{(n-1)*n}{2}$. Do đó độ phức tạp thuật toán là O(n^2).

Nếu một mảng đã có thứ tự được sắp xếp thì thuật toán Bubble Sort không tạo ra hoán đổi, thuật toán có thể kết thúc sau một lần chuyển. Do đó, nếu thuật toán Bubble Sort gặp một mảng đã được sắp xếp, nó sẽ kết thúc sau O(n) thời gian.

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

- Space Complexity

Bubble sort là một thuật toán sắp xếp ổn định với Space Complexity là O(1).

iv. Comments

Mặc dù thuật toán Bubble Sort rất dễ thực hiện, song nó không để giải quyết các vấn đề mang tính thực tế do thời gian trong trường hợp xấu nhất, trung bình là O(n^2).

Bubble Sort chỉ nên sử dụng trong các trường hợp sau :
- Độ phức tạp của code không quan trọng
- Ưu tiên code ngắn, dễ hiểu.

5. Shaker Sort

i. Ideas

Shaker Sort là một thuật toán cải tiến của Bubble Sort. Sau khi đưa phần tử nhỏ nhất về đầu dãy, thuật toán sẽ giúp chúng ta đưa phần tử

lớn nhất về cuối dãy. Do đưa các phần tử về đúng vị trí ở cả hai đầu nên thuật toán Shaker Sort sẽ giúp cải thiện thời gian sắp xếp dãy số.

Ta sẽ dùng 2 biến left, right để kiểm soát vị trí đầu & cuối của mảng. Ta bắt đầu với left = 0, right = n – 1. Khi ta đưa phần tử nhỏ nhất lên đầu dãy, thì ta sẽ tăng left lên 1 đơn vị – Lúc này, phần tử đầu dãy đã là phẩn tử nhỏ nhất, nên ta không cần xét nữa. Sau đó, ta đưa phần tử lớn nhất xuống cuối dãy, thì ta sẽ giảm right xuống 1 đơn vị - Lúc này phần tử cuối dãy đã là phần tử lớn nhất, nên ta không cần xét nữa.

ii. Implement Algorithm (Step – by – step)

- Step 1 : Khai báo biến left = 0 (kiểm soát vị trí đầu), right = n – 1 (kiểm soát vị trí cuối)
- Step 2 : Đặt vòng lặp while (left < right) để kiểm tra khi nào vị trí đầu vượt qua vị trí cuối
- Step 3 : Chạy vòng lặp từ left đến right
- Step 4 : Nếu giá trị phần tử thứ i lớn hơn giá trị phần tử thứ i + 1 thì đổi chỗ chúng.
- Step 5 : Giảm giá trị right xuống 1 đơn vị
- Step 6 : Chạy vòng lặp từ right xuống left
- Step 7 : Nếu giá trị phần tử thứ i nhỏ hơn giá trị phần tử thứ i + 1 thì đổi chỗ chúng.
- Step 8 : Tăng giá trị left lên 1 đơn vị.

iii. Time & Space Complexity

Time Complexity
Khi left = 0, right = n – 1: (*) có độ phức tạp là n
Khi đó:
(1) chạy từ 0 đến n – 2 : Có chi phí là n – 1
(2) chạy từ n – 3 xuống 0: Có chi phí là n – 2
Suy ra, chi phí của (1) và (2) là max((n – 1), (n – 2)) = (n – 1).

Khi left = 1, right = n – 2
Khi đó:
(1) Có chi phí là : n – 3
(2) Có chi phí là : n - 4
Suy ra, chi phí của (1) và (2) là max((n – 3), (n – 4)) = (n – 3).

Tuần tự như vậy, khi đó :
Tổng chi phí là : 1 + 3 + 5 + … + (n – 3) + (n – 1) = $\frac{(n-1)*(n-1+1)}{4}$ = $\frac{(n-1)*n}{4}$.
Do đó độ phức tạp thuật toán là O(n^2).

Nếu một mảng đã có thứ tự được sắp xếp thì thuật toán Shaker Sort không tạo ra hoán đổi. Do đó, nếu thuật toán Shaker Sort gặp một mảng đã được sắp xếp, nó sẽ kết thúc sau O(n) thời gian.

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

Space Complexity
Tương tự như Bubble sort, Shaker Sort là một thuật toán sắp xếp ổn định với Space Complexity là O(1).

iv. Comments

Trong trường hợp mảng có các phần tử là [2, 3, 4, 5, 1] thì đối với Thuật toán Shaker Sort chỉ cần 1 lần duyệt là đã đưa các phần tử của mảng về đúng vị trí, còn với Bubble Sort cần tới 4 lần duyệt để đưa các phần tử về đúng vị trí. Tuy nhiên, trong trường hợp mảng có ngẫu nhiên phần tử với thứ tự đảo lộn thì Bubble Sort và Thuật toán Shaker Sort cho thời gian sắp xếp gần tương đương nhau.

Vì vậy, ta có thể nói rằng Thuật toán sắp xếp cocktail ưu thế hơn Bubble Sort trong trường hợp các phần tử trong mảng đã gần có thứ tự như trong ví dụ trên là mảng [2, 3, 4, 5, 1].

6. Shell Sort

i. Ideas

Thuật toán Shell Sort là thuật toán so sánh tại chỗ. Nó có thể được xem như là một khái quát của sắp xếp bằng cách trao đổi hoặc sắp xếp bằng cách chèn.

Ta sẽ phân hoạch mảng thành các dãy con, sau đó sắp xếp các dãy con bằng phương pháp chèn trực tiếp. Cuối cùng sắp xếp lại cả dãy bằng phương pháp chèn trực tiếp.

ii. Implement Algorithm (Step – by – step)

- Step 1: Khởi tạo giá trị gap.
- Step 2: Chia dãy thành các mảng con có độ lớn tương ứng gap.
- Step 3: Sắp xếp các mảng con tương tự Insertion Sort.
- Step 4: Lặp lại cho đến khi dãy được sắp xếp có thứ tự hoàn chỉnh.

iii. Time & Space Complexity

Time Complexity

- Best case: O(n*log(n)).
- Average case: O(n).
- Worst case : O(n*log(n)^2).

Space Complexity : O(1)

- Worst case : O(n)

iv. Comments

7. Heap Sort

i. Ideas

Tương tự Selection Sort. Gọi lại Heap để lấy phần tử lớn nhất đưa về cuối mảng sau đó giảm số phần tử của Heap xuống cho đến khi Heap còn 1 phần tử.

ii. Implement Algorithm (Step – by – step)

Xây dựng Heap

- Step 1: Bắt đầu từ chính giữa mảng, khai báo start = (n – 1) / 2
- Step 2: Trong khi start >= 0
    - Xây dựng lại Heap (Heap Rebuild) ở vị trí start
    - Giảm start 1 đơn vị

Xây dựng lại Heap

- Step 1: Khai báo k = pos, v = a[k], isHeap = false
- Step 2: Trong khi isHeap sai và 2 * k + 1 < n
    - J = 2 * k + 1 // phần tử đầu tiên

- Nếu j < n – 1 // Có đủ 2 phần tử
- Nếu a[j] < a[j+ 1] thì j = j + 1
- Nếu v>=a[j] thì IsHeap = true, nếu không thì đổi chỗ a[k] và a[j]
- Gán k = j

Heap Sort

- Step 1: Xây dựng heap
- Step 2:
  - Lấy phần tử lớn nhất ở vị trí đầu đổi chỗ với phần tử cuối
  - Giảm size của Heap xuống 1
  - Xây dựng lại Heap tại vị trí đầu
  - Lặp lại bước 2 đến khi Heap còn 1 phần tử

iii. Time & Space Complexity

Time Complexity

- Best case: O(n).
- Average case: O(n*log(n)).
- Worst case : O(n*log(n)).

iv. Comments

8. Merge Sort

i. Ideas

Dùng phương pháp chia để trị. Chia mảng cần sắp xếp thành 2 nửa mảng con, tiếp tục chia đôi ở các mảng con, sau đó gộp các mảng con thành các mảng đã sắp xếp và cuối cùng là gộp thành 1 mảng đã sắp xếp hoàn chỉnh.

ii. Implement Algorithm (Step – by – step)

Merge Sort

- Step 1: So sánh left < right
  - Tìm phần tử chính giữa
  - Chia đôi mảng thành nửa 2 phần bằng nhau
- Step 2: Nối các mảng con lại với nhau bằng hàm Merge

Merge

- Step 1: Tạo 2 mảng con, 1 mảng để chưa các phần tử từ đầu đến phần tử chính giữa mảng và 1 mảng chứa phần còn lại
- Step 2: So sánh theo vị trí của từng phẩn tử trong 2 mảng (a[i] và b[j])
    - Nếu a[i] > b[j] thì đặt a[i] vào mảng kết quả và tăng I lên 1, nếu không thì đặt b[j] vào và tăng j lên 1
    - Kiểm tra mảng a và b xem đã hết phần tử chưa và quay lại bước 2. Nếu có 1 trong 2 mảng hết thì dừng lại.

iii. Time & Space Complexity

Time Complexity

- Best case: O(n*log(n)).
- Average case: O(n*log(n)).
- Worst case : O(n*log(n)).

iv. Comments

9. Quick Sort

i. Ideas

Quicksort là thuật toán sắp xếp dựa trên nền tảng của kỹ thuật chia để trị (Divided and Conquer) để chia nhỏ danh sách thành những danh sách nhỏ, rồi thực hiện sắp xếp trong từng danh sách con đó.

Đầu tiên, ta sẽ tìm vị trí làm vách ngăn của 2 danh sách con (partitionIndex). Ta phải chọn một phần tử làm chốt (pivot), sau đó chia danh sách ra làm 2 danh sách con – 1 bên gồm những phần tử có giá trị lớn hơn pivot, bên còn lại gồm những phần từ có giá trị bé hơn hoặc bằng pivot. Ta thực hiện như vậy trên những danh sách con vừa được chia ra. Thực hiện như vậy đến khi danh sách con có độ dài bằng 1.

ii. Implement Algorithm (Step – by – step)

iii. Time & Space Complexity

Time Complexity

- Best case: O(n*log(n)).
- Average case: O(n*log(n)).
- Worst case : O(n^2).

Space Complexity : O(log(n))

iv. Comments

Quicksort được triển khai khi :

- Ngôn ngữ lập trình tốt cho đệ quy.
- Yêu cầu phức tạp về vấn đề thời gian
- Yêu cầu phức tạp về vấn đề không gian

10. Counting Sort

i. Ideas

Sắp xếp đếm phân phối là một phương pháp sắp xếp có độ phức tạp tuyến tính. Nó dựa trên giả thiết rằng, các khoá cần sắp xếp là các số tự nhiên giới hạn trong một khoảng nào đó, chẳng hạn từ *1* đến *N*.

Ta sẽ đếm số lần xuất hiện của các giá trị khác nhau trong mảng ban đầu và gán giá trị vừa đếm vào vị trí tương ứng của 1 mảng có độ dài là giá trị lớn nhất của mảng.

ii. Implement Algorithm (Step – by – step)

- Step 1 : Tìm giá trị Maximum của mảng.
- Step 2 : Tạo mảng A có độ dài bằng giá trị Maximum và gán phần tử của mảng bằng 0.
- Step 3 : Đếm số lần xuất hiện của các giá trị khác nhau trong mảng ban đầu và gán giá trị vào vị trí tương ứng ở mảng A.
- Step 4: Sắp xếp lại mảng ban đầu bằng cách duyệt mảng A.

iii. Time & Space Complexity

Time Complexity

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

Space Complexity : O(Maximum)

iv. Comments

Counting được sử dụng khi :

- Mảng có các số nguyên nhỏ với nhiều số đếm
- Cần độ phức tạp tuyến tính

11. Radix Sort

i. Ideas

Radix Sort là thuật toán sắp xếp dựa trên cơ số.

Ta sẽ sắp xếp theo cơ số từ 0 đến 9. Lần lượt từ hàng đơn vị cho đến hàng lớn nhất, ta đưa chữ số của hàng đó về đúng vị trí từ 0 đến 9, sau đó lấy ra để được 1 dãy số mới. Đến hàng cuối cùng sau khi sắp xếp và lấy ra ta sẽ được 1 dãy số sắp xếp có thứ tự.

ii. Implement Algorithm (Step – by – step)

- Step 1 : Tìm giá trị Maximum của mảng.
- Step 2 : Khai báo k = 1.
- Step 3 : Kiểm tra Maximum / k có lớn hơn 0, nếu không thì dừng lại.
- Step 4: Tạo mảng a có 10 phần tử tương ứng cơ số 0 đến 9
- Step 5: Đếm số phần tử a[i] tại cơ số tương ứng
- Step 6: Phần tử ở vị trí a[i] sẽ được gán bằng giá trị a[i] + a[I – 1]
- Step 7: Lần lượt lưu từng số vào mảng kết quả ở các vị trí thích hợp dựa vào a[i] và k, đồng thời giảm số phần tử tại a[i] xuống 1.
- Step 8: Tăng k thêm 10 lần
- Step 9: Quay lại bước 3

iii. Time & Space Complexity

Time Complexity

- Best case: O(d(n + Maximum)).
- Average case: O(d(n + Maximum)).
- Worst case : O(d(n + Maximum)).

Space Complexity : O(d(n + Maximum)) + Maximum).

iv. Comments

Radix Sort được sử dụng trong :

- Thuật toán DC3.
- Những nơi có số lượng phạm vi lớn.

12. Flash Sort

i. Ideas

Flashsort là một thuật toán sắp xếp phân phối cho thấy độ phức tạp tính toán tuyến tính cho các tập dữ liệu được phân phối đồng đều và yêu cầu bộ nhớ bổ sung tương đối ít. Ta sẽ phân loại các phần tử, kế đến phân bố các phần tử vào đúng các phân lớp, sau đó sắp xếp các phần tử trong các phân lớp đó theo đúng thứ tự, ta được dãy sắp xếp có thứ tự hoàn chỉnh.

ii. Implement Algorithm (Step – by – step)

- Step 1 : Tìm min, max của mảng và lưu i là vị trí của phần tử max
- Step 2 : Khởi tạo mảng tạm t với độ dài là số phần lớp
- Step 3 : Chạy vòng lặp từ left đến right
- Step 4 : Nếu giá trị phần tử thứ i lớn hơn giá trị phần tử thứ i + 1 thì đổi chỗ chúng.
- Step 5 : Giảm giá trị right xuống 1 đơn vị
- Step 6 : Chạy vòng lặp từ right xuống left

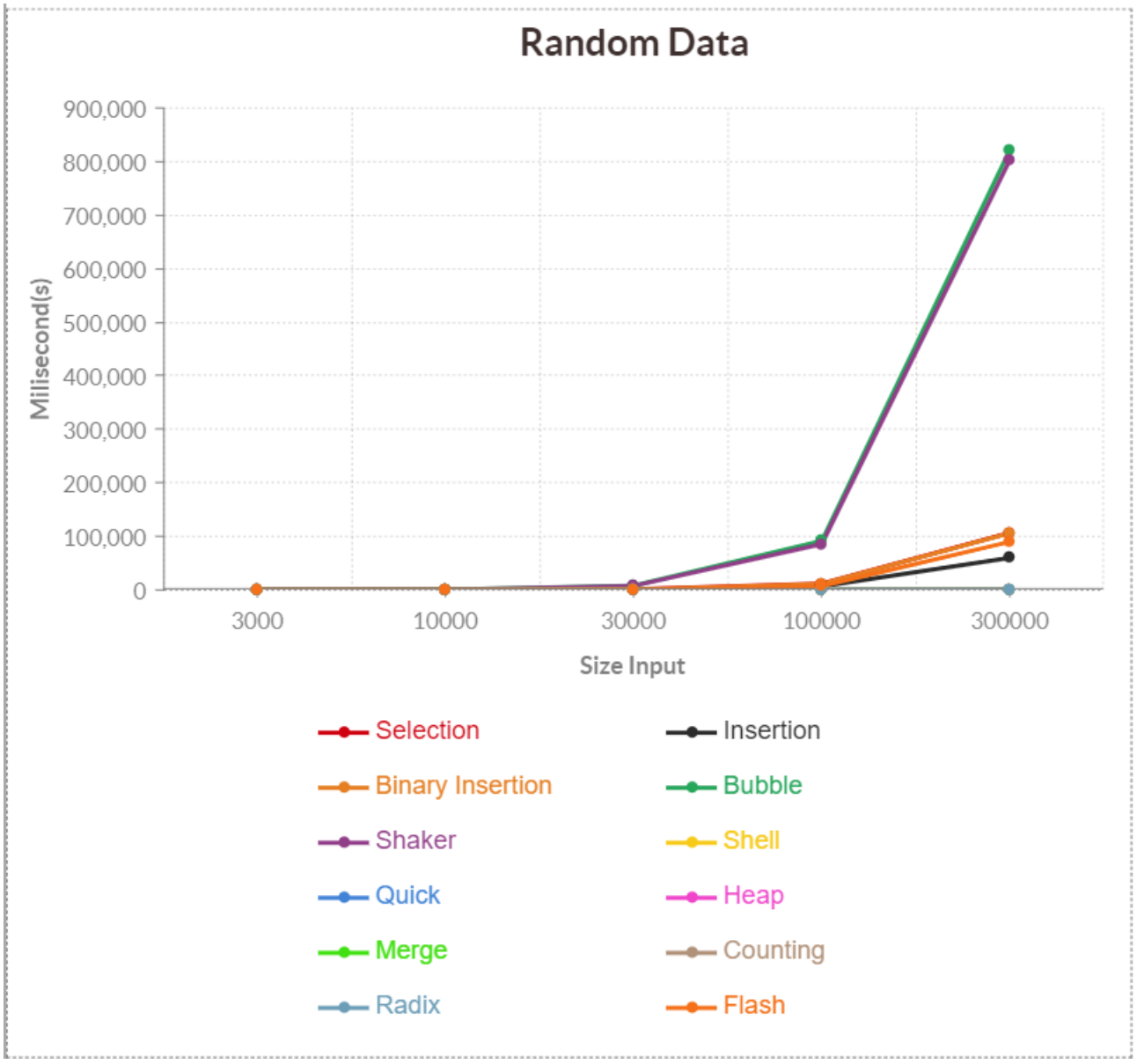
iii. Time & Space Complexity

- Best case: O(n).
- Average case: O(n + r).
- Worst case : O(n^2).

iv. Comments

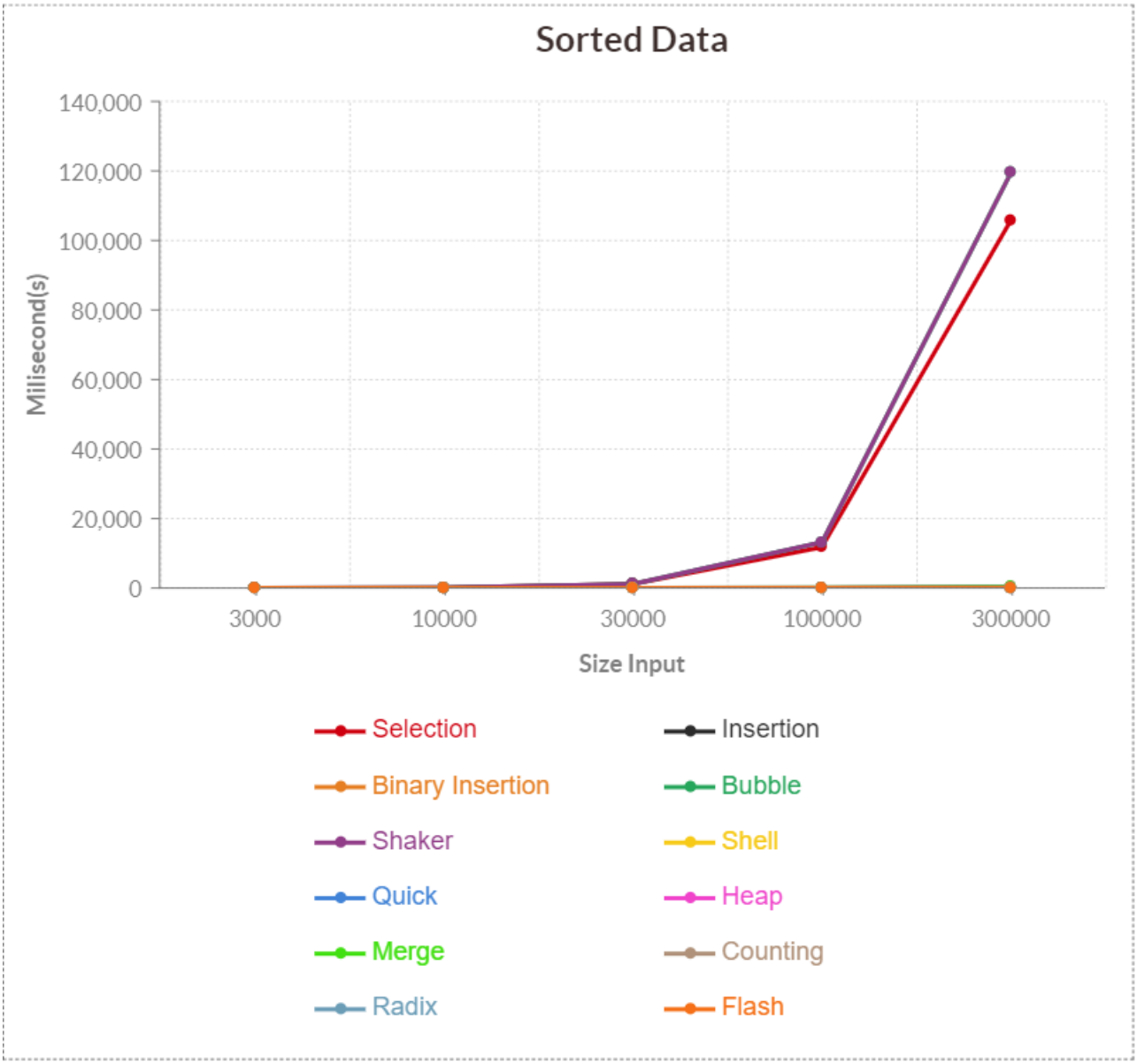
II. EXPERIMENTAL RESULTS AND COMMENTS

1. Random Data



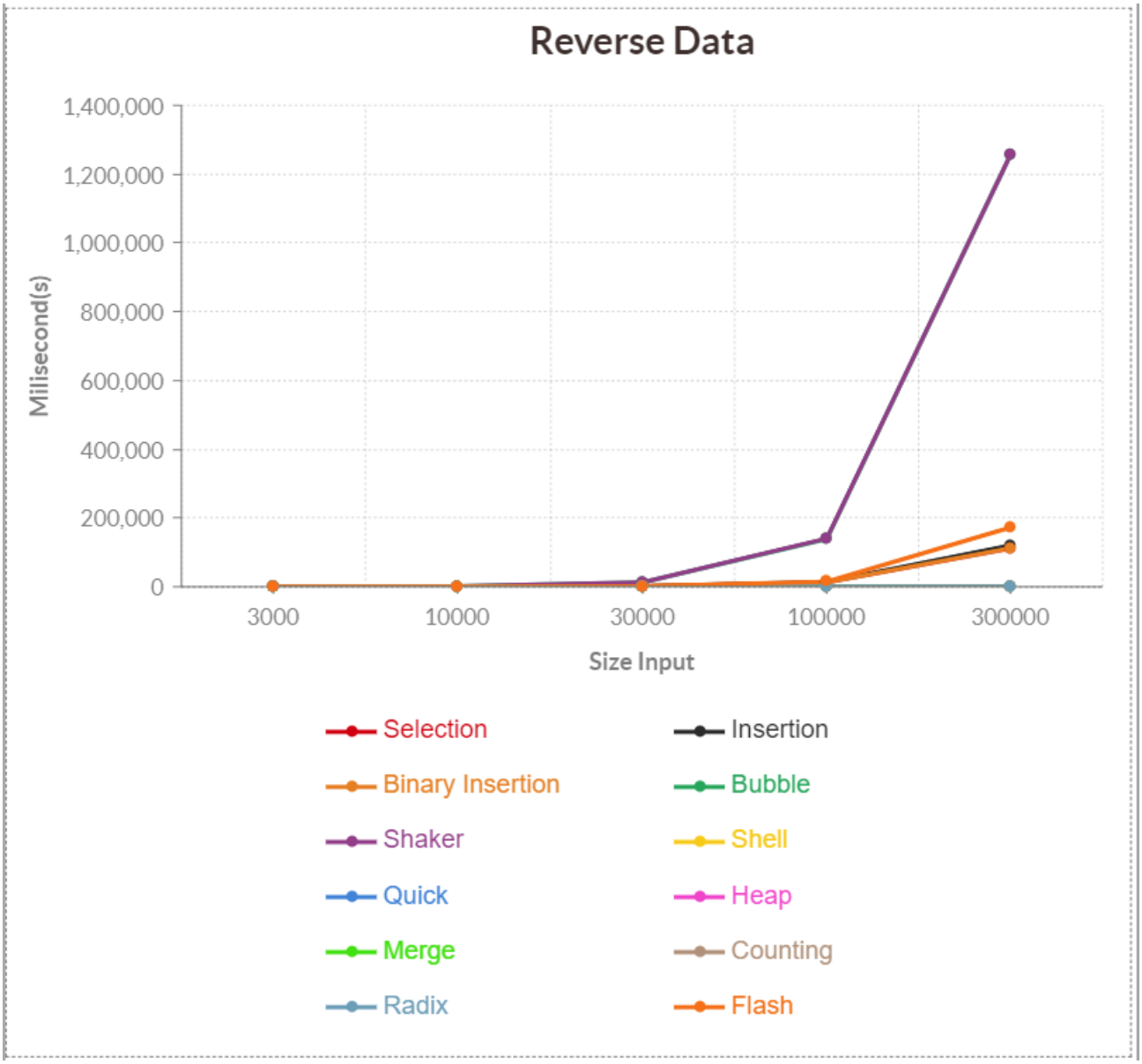
Với mảng dữ liệu rất lớn, Shaker Sort và Bubble Sort đã thể hiện điểm yếu của mình vì có O(n^2). Nó mất hơn 50s để sắp xếp trong khi các thuật toán còn lại mất chưa đến 15s. Radix Sort, Counting Sort, Quick Sort, Merge Sort là những thuật toán tối ưu nhất với mảng dữ liệu lớn và các phần tử chưa được sắp xếp vì chi phí chỉ là O(n) và O ($n*\log n$).

2. Sorted Data



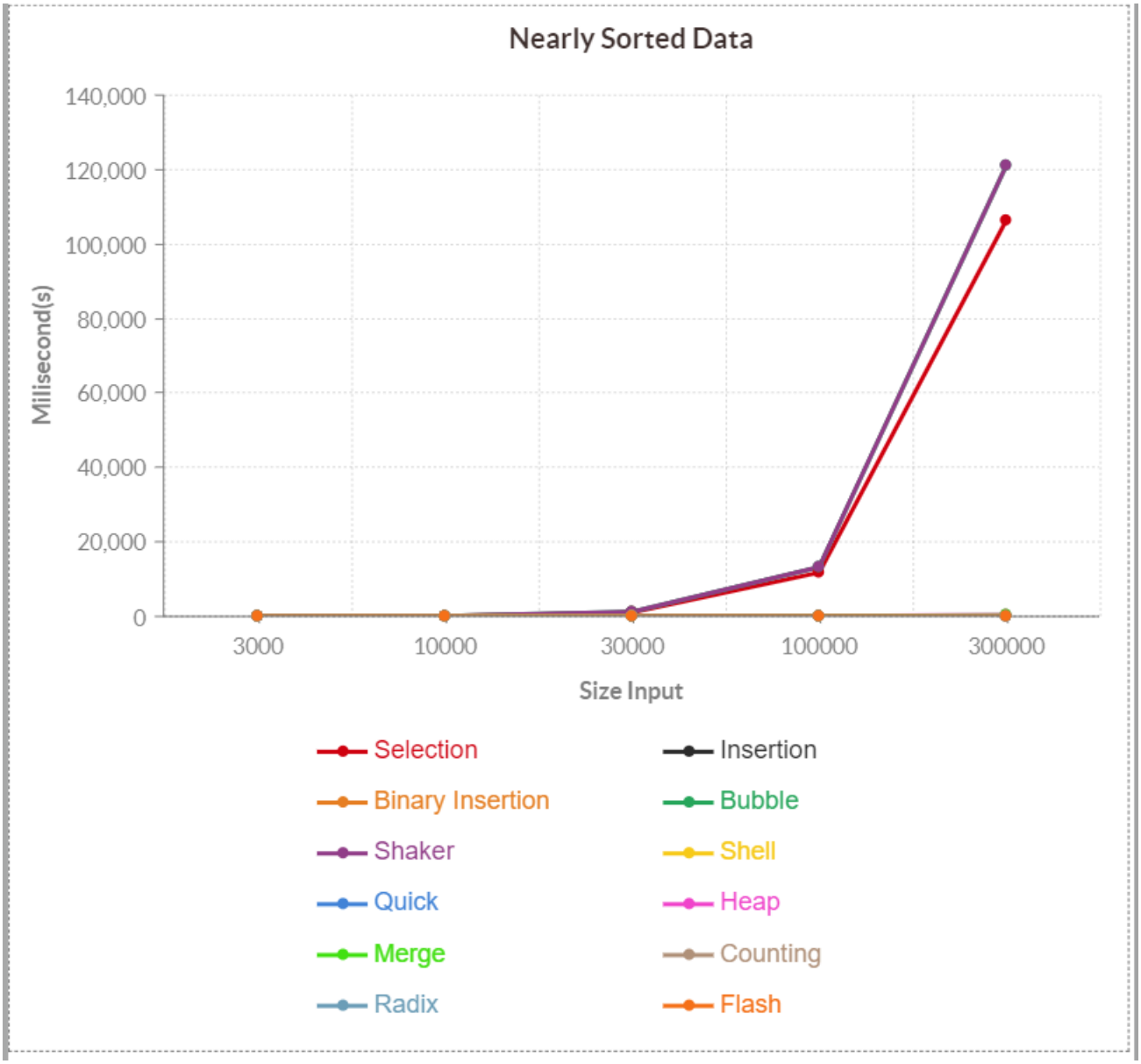
Với kiểu dữ liệu đã sắp xếp, các thuật toán đều chạy tốt trừ Bubble Sort và Selection Sort vì có độ phức tạp trong trường hợp trung bình là O (n^2).

3. Reverse Data



Với mảng sắp xếp ngược có dữ liệu rất lớn, Bubble Sort và Shaker Sort là 2 thuật toán tệ nhất vì nó rơi vào trường hợp xấu nhất của thuật toán là O(n^2). Các thuật toán Heap Sort, Merge Sort, Quick Sort hay Radix Sort có độ phức tạp ở trường hợp này là O(n) và O($n*\log n$).

4. Nearly Sorted Data



Đa số các thuật toán đều đạt được tốc độ nhanh, tuy nhiên Selection Sort và Buble Sort do có thời gian trung bình O ($n2$) và việc số lượng phần tử lớn dẫn đến thời gian chạy lâu.

III. REFERENCES

1. Geekforgeeks.org
2. Programiz.com
3. Wikipedia
4. Brilliant.org