ĐẠI HỌC KHOA HỌC TỰ NHIÊN, ĐHQG TP.HCM

KHOA CÔNG NGHỆ THÔNG TIN

- - - 🕮 - - -

BÁO CÁO

Môn học: Thực hành CTDL&GT

Học kỳ I (2020-2021)

Đề tài:

PHÂN TÍCH CÁC GIẢI THUẬT SẮP XẾP VÀ MINH HỌA THỜI GIAN CHẠY THỰC TẾ CỦA CÁC THUẬT TOÁN SẮP XẾP.

Sinh viên: Võ Chánh Hưng

MSSV: 19120523

Lớp: 19CTT1TN

Thành phố Hồ Chí Minh, tháng 11 năm 2020

PHÂN TÍCH CÁC GIẢI THUẬT SẮP XẾP VÀ MINH HOẠ THỜI GIAN CHẠY THỰC TẾ CỦA CÁC THUẬT TOÁN SẮP XẾP.

I. Phân tích các giải thuật sắp xếp:

1. Selection Sort:

_Ý tưởng:

Thuật toán sắp xếp chọn sẽ 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 (không phải đầu mảng). Thuật toán sẽ chia mảng làm 2 mảng con

1. Một mảng con đã được sắp xếp.
2. Một mảng con chưa được sắp xếp.

Tại mỗi bước lặp của thuật toán, phần tử nhỏ nhất ở mảng con chưa được sắp xếp sẽ được di chuyển về đoạn đã sắp xếp.

_Thuật toán:

+ Bước 1: Gán giá trị nhỏ nhất của phần mảng chưa được sắp xếp bằng giá trị đầu tiên của mảng đó.

+ Bước 2: Dùng vòng lặp chạy từ phần tử kế tiếp cho đến hết mảng, nếu giá trị tại một phần tử i là A[i] nhỏ hơn giá trị nhỏ nhất thì cập nhật lại giá trị nhỏ nhất và giá trị chỉ mục của phần tử nhỏ nhất hiện tại đó.

+ Bước 3: Hoán đổi phần tử đầu tiên của phần mảng chưa được sắp xếp với phần tử nhỏ nhất trong mảng chưa được sắp xếp đó.

+ Bước 4: Quay lại Bước 1.

_ Đánh giá giải thuật :

+ Độ phức tạp của thuật toán:

$$O(n) = (n-1) + (n-2) + ... + 1 = \frac{n(n-1)}{2} \in O(n^2)$$

Cả ba trường hợp xấu nhất, tốt nhất, và trung bình đều bắt buộc phải chạy hết hai vòng lặp nên đều có độ phức tạp như trên.

+ Bộ nhớ: O(1)

Chỉ sử dụng một số lượng cố định các biến tạm, sắp xếp trong phạm vi mảng, không cần tạo mảng phụ.

2. Insertion Sort.

_Ý tưởng:

Thuật toán sắp xếp chèn thực hiện sắp xếp dãy số theo cách duyệt từng phần tử và chèn từng phần tử đó vào đúng vị trí trong mảng con (dãy số từ đầu đến phần tử phía trước nó) đã sắp xếp sao cho dãy số trong mảng sắp đã xếp đó vẫn đảm bảo tính chất của một dãy số tăng dần.

_Thuật toán:

1. Khởi tạo mảng với dãy con đã sắp xếp có k = 1 phần tử (phần tử đầu tiên, phần tử có chỉ số 0)
2. Duyệt từng phần tử từ phần tử thứ 2, tại mỗi lần duyệt phần tử ở chỉ số i thì đặt phần tử đó vào một vị trí nào đó trong đoạn từ [0…i] sao cho dãy số từ [0…i] vẫn đảm bảo tính chất dãy số tăng dần. Sau mỗi lần duyệt, số phần tử đã được sắp xếp k trong mảng tăng thêm 1 phần tử.
3. Lặp cho tới khi duyệt hết tất cả các phần tử của mảng.

_Đánh giá giải thuật :

+ Độ phức tạp của thuật toán:

Trường hợp tốt nhất:

Mảng đã được sắp xếp tăng dần có độ phức tạp O(n).

Khi mảng đã được sắp xếp tăng dần thì thuật toán chỉ chạy duy nhất một vòng lặp ngoài bởi vì mọi phần tử bên trong mảng đã được xếp đúng vị trí của nó.

Trường hợp xấu nhất:

Mảng đã được sắp xếp giảm dần có độ phức tạp $O(n^2)$

Khi mảng đã được sắp xếp giảm dần thì mọi phần tử tại vị trí i sẽ phải dịch chuyển qua bên trái i lần.

$$O(n)=\sum_{i=1}^{n-1} i = 1+2+...+n-1 = \frac{n(n-1)}{2} \in O(n^2)$$

Trường hợp trung bình:

Trung bình mọi phần tử tại vị trí i sẽ dịch chuyển sang bên trái i/2 lần.

$$\sum_{i=1}^{n-1} \frac{i}{2} = \frac{1}{2}(1+2+...+n-1) = \frac{n(n-1)}{4} \in O(n^2)$$

+Bộ nhớ: O(1)

Chỉ sử dụng một số lượng cố định các biến tạm, sắp xếp trong phạm vi mảng, không cần tạo mảng phụ.

3. Binary Insertion Sort.

_Ý tưởng:

Ý tưởng bắt nguồn từ việc phần mảng từ đầu đến vị trí **i** cần được sắp đã được sắp xếp tăng nên có thể dùng tìm kiếm nhị phân để tìm ra vị trí phù hợp cho phần tử Arr[i] trong mảng sau khi đã được sắp xếp.

Điều này làm giảm các bước so sánh nhưng không làm giảm đi bước dịch chuyển của các phần tử liên quan.

_Thuật toán:

+ Bước 1:

Dùng tìm kiếm nhị phân tìm vị trí sắp xếp hợp lí cho phần tử A[i] trong mảng con từ 0 đến i-1 gọi là pos.

+ Bước 2: Dịch chuyển một mảng con từ vị trí pos đến i-1 sang phải một phần tử.

+ Bước 3: Chèn A[i] vào vị trí đã tìm được trước đó.

+ Bước 4: Tăng biến đếm i và trở lại bước 1.

_ Đánh giá giải thuật:

+ Độ phức tạp cho các trường hợp của sắp xếp chèn nhị phân và sắp xếp chèn là như nhau, nhưng sắp xếp chèn nhị phân giúp giảm các bước so sánh trong lúc cần tìm vị trí đứng phù hợp.

+ Bộ nhớ: O(1)

Chỉ sử dụng một số lượng cố định các biến tạm, sắp xếp trong phạm vi mảng, không cần tạo mảng phụ.

4. Bubble Sort.

_Ý tưởng:

Thuật toán sắp xếp nổi bọt thực hiện sắp xếp dãy số bằng cách lặp lại công việc đổi chỗ 2 số liên tiếp nhau nếu chúng đứng sai thứ tự (số sau bé hơn số trước với trường hợp sắp xếp tăng dần) cho đến khi dãy số được sắp xếp.

_Thuật toán:

+ Bước 1: i=0 là phần tử đầu tiên

+ Bước 2: Lần lượt so sánh và đổi chỗ (nếu cần) từ phải sang trái đối với các phần tử từ A[n-1] đến A[i] với biến gán j=n-i và lặp lại khi j>i.

+ Bước 3: i=i+1;

+ Bước 4: Nếu i<n, quay lại Bước 2.

Ngược lại, dừng, dãy đã cho đã sắp xếp đúng vị trí.

_Đánh giá giải thuật:

+ Độ phức tạp:

Với thuật toán sắp xếp nổi bọt thì độ phức tạp trong mọi trường hợp luôn là $O(n^2)$

Với thuật toán sắp xếp nổi bọt **đã được tối ưu** (dùng cờ hiệu để kiểm tra có lần đổi chỗ nào trong một lượt duyệt không) thì độ phức tạp tùy thuộc vào dữ liệu mảng đầu vào.

Trường hợp tốt nhất: O(n) khi mảng được sắp xếp tăng sẵn.

Trường hợp trung bình: $O(n^2)$

Trường hợp xấu nhất: $O(n^2)$ khi mảng được sắp xếp giảm sẵn.

+ Bộ nhớ: O(1)

Chỉ sử dụng một số lượng cố định các biến tạm, sắp xếp trong phạm vi mảng, không cần tạo mảng phụ.

5_Shaker Sort:

_Ý tưởng:

Shaker Sort là một cải tiến của Bubble Sort khi cách cài đặt tương tự với Bubble Sort chỉ thêm vào một vài yếu tố như cờ hiệu, biến lưu giữ vị trí hoán đổi cuối cùng khi đi từ đầu đến cuối mảng và ngược lại.

Dựa trên quan sát có được là nếu như cờ hiệu không thay đổi giá trị trong quá trình duyệt mảng có nghĩa là không có bất kì sự hoán đổi nào cần thiết phải thực hiện thì mảng đã được sắp xếp và không cần thực hiện thêm các lần lặp lại hao phí. Một kết quả nữa là sau mỗi lần duyệt mảng thì ta chỉ cần lưu vị trí hoán đổi cuối cùng để sau này khi đi ngược lại chỉ cần đi từ vị trí đó bởi vì phần mảng trước (sau) đã được sắp xếp rồi vì không có thêm hoán đổi nào.

_Thuật toán:

+ Bước 1: start=0, end=n-1, swapped=false;

+ Bước 2: Duyệt mảng từ vị trí start đến end. Nếu tại vị trí i có hoán đổi thì gán newEnd=i và đổi swapped=true;

+ Bước 3: Nếu swapped=false thì kết thúc do mảng đã được sắp xếp nếu không thì gán end=newEnd;

+ Bước 4: Đổi swapped=false;

Duyệt mảng từ vị trí end đến start. Nếu tại vị trí i có hoán đổi thì gán newStart=i và đổi swapped=true.

+ Bước 5: Nếu swapped=false thì kết thúc do mảng đã được sắp xếp nếu không thì gán start=newStart;

+ Bước 6: Quay lại bước 1.

_Đánh giá giải thuật:

+ Độ phức tạp: phụ thuộc khá nhiều vào mảng dữ liệu đầu vào.

Trường hợp trung bình: $O(n^2)$

Trường hợp tốt: O(n) khi mảng đã được sắp xếp tăng dần.

+ Bộ nhớ: O(1)

Chỉ sử dụng một số lượng cố định các biến tạm, sắp xếp trong phạm vi mảng, không cần tạo mảng phụ.

6_Shell Sort:

_Ý tưởng:

Thuật toán Shell Sort là một cải tiến của thuật toán Insertion Sort bằng cách chia nhỏ mảng ban đầu thành một số các danh sách con, mà mỗi danh sách được sắp xếp bằng thuật toán Insertion Sort.

Cách chia nhỏ mảng thành các danh sách con phụ thuộc vào "key" mảng. Thay vì chia nhỏ mảng thành các danh sách con cách 1 phần tử liên tiếp, thì thuật toán sử dụng bước tăng **i** để tạo một danh sách con bằng cách chọn mọi phần tử trong mảng cách nhau **i** phần tử.

_Thuật toán:

+ Bước 1: Khởi tạo bước nhảy bằng n/2.

+ Bước 2: Sử dụng Insertion Sort để sắp xếp các phần tử từ 0 đến n-1 mà cách nhau một khoảng bằng bước nhảy.

+ Bước 3: Giảm bước nhảy đi một nửa và lặp lại bước 1.

_Đánh giá giải thuật:

Độ phức tạp của giải thuật Shell Sort là $O(n^2)$. Tuy nhiên có nhiều cách để giảm bước nhảy mà có thể cải tiến giải thuật này lên.

7_Heap Sort:

_Ý tưởng: Dùng các đặc tính của cấu trúc heap để sắp xếp các phần tử trong mảng.

Trong một cấu trúc heap (max-heap) phần tử lớn nhất nằm trên đỉnh, tại mọi node có hai node con left child, right child của node đó có trường key đều nhỏ hơn trường key của node đó.

Ý tưởng của thuật toán này là đổi chỗ vị trí 0 và vị trí n-1 của mảng (biểu diễn cho heap) sau đó chuyển node tại vị trí 0 đi xuống cho đến khi cấu trúc heap được thỏa

mãn trở lại. Độ phức tạp của quá trình dịch chuyển xuống là O(h) trong đó h là chiều cao của heap. Do heap là một cây nhị phân hoàn toàn nên h tương đồng với logn trong đó n là số node có trong heap.

_Thuật toán:

+ Bước 1: Từ mảng ban đầu, ta dùng thao tác SiftDown để chuyển các phần tử từ vị trí $\frac{n}{2}-1$ đến 0 sao cho tạo ra được cấp trúc Max Heap.

Gán size=n;

+ Bước 2: Hoán đổi vị trí i với vị trí size-1 khi đó phần tử lớn nhất của mảng khi đó sẽ nằm ở cuối mảng.

+ Bước 3: Giảm size đi 1 đơn vị và SiftDown phần tử i=0 cho đến khi cấu trúc Heap được thỏa mãn.

+ Bước 4: Quay trở lại bước 2.

_Đánh giá giải thuật:

+ Độ phức tạp:

Trường hợp tốt nhất: các key của mọi node trong heap đều bằng nhau nên thao tác SiftDown chỉ mất O(1). Vậy nên độ phức tạp là O(n).

Trường hợp trung bình hoặc xấu nhất: Thao tá SiftDown mất O(h) trong đó h là chiều cao của heap mà heap là cây nhị phân hoàn toàn nên SiftDown có độ phức tạp là O(logn). Vậy để hoán đổi n phần tử thì cần O(nlogn) thao tác cần thiết.

+ Bộ nhớ: O(1)

Chỉ sử dụng một số lượng cố định các biến tạm, sắp xếp trong phạm vi mảng, không cần tạo mảng phụ.

8_Merge Sort:

_Ý tưởng:

Merge Sort là một thuật toán chia để trị. Thuật toán này chia mảng cần sắp xếp thành 2 nửa. Tiếp tục lặp lại việc này ở các nửa mảng đã chia. Sau cùng gộp các nửa đó thành mảng đã sắp xếp. Hàm merge() được sử dụng để gộp hai nửa mảng. Hàm merge(arr, l, m, r) là tiến trình quan trọng nhất sẽ gộp hai nửa mảng thành 1 mảng sắp xếp, các nửa mảng là arr[l…m] và arr[m+1…r] sau khi gộp sẽ thành một mảng duy nhất đã sắp xếp.

_ Thuật toán:

mergeSort(arr[], l, r)

If r > l

1. Tìm chỉ số nằm giữa mảng để chia mảng thành 2 nửa:
   middle m = (l+r)/2
2. Gọi đệ quy hàm mergeSort cho nửa đầu tiên:
   mergeSort(arr, l, m)
3. Gọi đệ quy hàm mergeSort cho nửa thứ hai:
   mergeSort(arr, m+1, r)
4. Gộp 2 nửa mảng đã sắp xếp ở (2) và (3):
   merge(arr, l, m, r)

_ Đánh giá giải thuật:

+ Độ phức tạp:

- Trường hợp tốt: O(nlog(n))
- Trung bình: O(nlog(n))
- Trường hợp xấu: O(nlog(n))

+ Không gian bộ nhớ: O(n): mảng phụ được tạo ra ở hàm Merge()

9_QuickSort.

_ Ý tưởng:

Quicksort là một thuật toán chia để trị. Thuật toán tạo ra hai mảng rỗng để lưu giữ các giá trị nhỏ hơn giá trị pivot và các giá trị lớn hơn giá trị pivot, sau đó gọi đệ quy để sắp xếp các mảng phụ.

Có hai thao tác chính của thuật toán này: hoán đổi các phần tử và phân hoạch mảng.

_ Thuật toán:

+ Bước 1: Chọn phần tử đầu tiên của mảng làm giá trị pivot.

+ Bước 2: Tạo 2 biến đếm i và j. Gán i và j vào phần tử đầu tiên và cuối cùng của mảng tương ứng.

+ Bước 3: Tăng i cho đến khi A[i] > pivot rồi dừng lại.

+ Bước 4: Dừng j cho đến khi A[j] < pivot rồi dừng lại.

+ Bước 5: Nếu i<j thì hoán đổi A[i] và A[j].

+ Bước 6: Lặp lại các bước 3,4,5 cho đến khi i>j.

+ Bước 7: Hoán đổi phần tử pivot với phần tử A[j].

_ Đánh giá giải thuật:

Thời gian chạy của QuickSort được viết là:

T(n)=T(n-k)+ T(n-k-1)+ O(n).

Độ phức tạp của QuickSort phụ thuộc vào mảng dữ liệu đầu vào và cách phân hoạch.

Trường hợp xấu nhất:

Khi phần tử pivot được chọn luôn là phần tử lớn nhất hoặc nhỏ nhất của mảng, có thể xảy ra khi mảng đã được sắp xếp tăng dần hoặc giảm dần.

Độ phức tạp trong trường hợp này là $\theta(n^2)$

Trường hợp tốt nhất:

Khi phần tử pivot là phần tử trung vị của mảng sau khi đã được xếp, tức phần tử đó chia mảng thành 2 phần bằng nhau.

T(n)=2T(n/2)+O(n).

Độ phức tạp trong trường hợp này là $\theta(n\log n)$

Bộ nhớ: O(1)

Chỉ sử dụng một số lượng cố định các biến tạm, sắp xếp trong phạm vi mảng, không cần tạo mảng phụ.

10_Counting Sort.

_Ý tưởng:

Thuật toán sắp xếp đếm hoạt động bằng cách duyệt qua toàn bộ mảng, đếm số lần một phần tử mảng xuất hiện và sử dụng các số liệu đó để tính toán vị trí đứng phù hợp cho phần tử đó trong mảng đã được sắp xếp sau cùng.

_Thuật toán (đã có cải tiến dành cho các số nguyên âm).

+ Bước 1: Tìm phần tử lớn nhất, nhỏ nhất max,min của mảng.

+ Bước 2:

Tạo một mảng L[max-min+1] có giá trị ban đầu của các phần tử là 0.

Duyệt toàn bộ mảng tại mỗi vị trí i tăng giá trị L[arr[i]-min] lên 1.

+ Bước 3:

Cộng dồn các phần tử trong mảng L từ trái qua phải.

Lúc này L[arr[i]-min]-1 chỉ giá trị đứng cuối cùng của phần tử arr[i] trong mảng đã được sắp xếp.

+ Bước 4:

Tạo một mảng phụ b[n] để lưu kết quả của mảng arr sau khi đã được sắp xếp.

Khi i=n-1 đến i=0, ta luôn có hệ thức b[L[arr[i]-min]-1]=arr[i];

Duyệt mảng theo chiều từ phải sang trái giữ cho thuật toán có tính ổn định.

Trường hợp một phần tử có thể lặp lại trong mảng ban đầu nên sau mỗi bước

L[arr[i]-min] <- L[arr[i]-min]-1;

Nghĩa là vị trí đứng cuối cùng của phần tử có giá trị bằng arr[i] (nếu có) trong mảng sau khi đã được sắp xếp giảm đi 1.

+ Bước 5:

Sao chép kết quả từ mảng b sang mảng arr. Vậy mảng arr bây giờ đã được sắp xếp.

_Đánh giá giải thuật:

Thời gian chạy cũng như bộ nhớ của thuật toán Counting Sort phụ thuộc khá lớn vào mảng dữ liệu đầu vào.

Trường hợp tốt nhất:

Chênh lệch giữa phần tử lớn nhất và phần tử nhỏ nhất trong mảng không quá lớn so chiều dài của mảng.

Độ phức tạp: $\theta(n)$

Bộ nhớ: $\theta(n)$

Thuật toán Counting Sort cần thêm mảng phụ để tính toán vị trí đứng phù hợp của các phần tử trong mảng và mảng phụ để lưu kết quả của mảng đã được sắp xếp.

Trường hợp xấu nhất:

Chênh lệch giữa phần tử lớn nhất và phần tử nhỏ nhất trong mảng rất lớn so với chiều dài của mảng.

Độ phức tạp: $\Omega(max - min)$

Bộ nhớ: $\Omega(max - min)$

Thuật toán Counting Sort cần thêm mảng phụ để tính toán vị trí đứng phù hợp của các phần tử trong mảng và mảng phụ để lưu kết quả của mảng đã được sắp xếp.

11_Radix Sort.

_Ý tưởng:

Radix sort là một thuật toán sắp xếp theo phương pháp cơ số không quan tâm đến việc so sánh giá trị của các phần tử như các thuật toán sắp xếp khác như Bubble sort, Selection sort, … Radix Sort sử dụng cách thức phân loại các con số trong dãy và thứ tự phân loại con con số này để tạo ra thứ tự cho các phần tử. Đây là cách tiếp cận khác so với các phương pháp sắp xếp khác.

_Thuật toán: (theo cơ số 10)

Để sắp xếp dãy a1,a2,…,an dùng giải thuật Radix Sort.

1. Trước tiên, ta giả sử mỗi phần tử ai trong dãy a1,a2,…,an là một số nguyên có tối đa m chữ số.

2. Ta phân loại các phần tử lần lượt theo các chữ số hàng đơn vị, hàng chục, hàng trăm,…

+ Bước 1: k cho biết chữ số dùng để phân loại hiện hành. (k=0: hàng đơn vị, k=1: hàng chục,…)

+ Bước 2: Tạo các lô chứa các loại phần tử khác nhau.

Khởi tạo 10 lô B0,B1,..,B9 rỗng.

+ Bước 3: Duyệt từ đầu đến cuối mảng, đặt ai vào lô Bt với t là chữ số thứ k của ai;

+ Bước 4: Nối B0,B1,..,B9 lại (theo đúng trình tự) thành a.

+ Bước 5: k=k+1;

Nếu k<m trở lại bước 2 ngược lại thì dừng do mảng đã được sắp xếp.

_Đánh giá giải thuật:

+ Độ phức tạp: $\theta(dn)$

Trong đó d là số lượng chữ số của phần tử có số lượng chữ số lớn nhất, n là số lượng phần tử có trong mảng.

Tại mỗi k=1 đến k<=d thì thuật toán có độ phức tạp $\theta(n)$ nên tổng thể thuật toán sẽ có độ phức tạp $\theta(dn)$.

+ Bộ nhớ: O(1)

Số lượng mảng phụ là hằng số và khi mà n hoặc d đủ lớn thì ảnh hưởng của số lượng mảng phụ này không còn đáng kể.

12_Flash Sort.

_Ý tưởng:

Thuật toán Flash Sort hoạt động dựa trên nguyên tắc: phân hoạch mảng ban đầu thành các phân vùng con mà các phần tử nằm trên cùng một phân vùng con có chung một đặc tính được tính toán bằng công thức và sau đó sử dụng các sắp xếp cơ bản như Insertion Sort, Bubble Sort để sắp xếp các phần tử trong các phân vùng đó.

_Thuật toán:

+ Bước 1: Tạo một mảng có m phần tử L[m] có giá trị khởi tạo bằng 0 đại diện cho m phân vùng con.

+ Bước 2: Tìm ra các phần tử lớn nhất, nhỏ nhất trong mảng max, min.

+ Bước 3: Xây dựng một công thức tính toán một phần tử arr[i] sẽ nằm trong phân lớp nào.

$$k = \left[\frac{(m-1)(arr[i]-\min)}{(\max-\min)} \right] + 1$$

+ Bước 4: Duyệt mảng từ đầu đến cuối, tìm k của từng phần tử arr[i] rồi tăng L[k]++;

+ Bước 5: Cộng dồn từ đầu đến cuối các phần tử trong mảng L.

Sau bước này, L[i] chỉ ra vị trí cuối cùng mà phần tử có k=i đứng trong mảng sau khi đã được phân hoạch.

+ Bước 6:

Nếu như phần tử i trong mảng thỏa $i \le L[k(arr[i])]$ thì phần tử đó chưa đứng đúng phân vùng của nó.

Đặt x=arr[i];

Thay phần tử tại vị trí $L[k(arr[i])]$ bởi x, giảm $L[k(arr[i])]$-- và tìm vị trí phù hợp cho phần tử được thay bằng cách tương tự cho đến khi $i > L[k(arr[i])]$ tức arr[i] đã đứng đúng phân vùng của nó.

+ Bước 7:

Sau khi các phần tử đã được phân hoạch thì gọi sắp xếp chèn Insertion Sort để sắp xếp các phần tử trong cũng một phân lớp.

_Đánh giá giải thuật.

+ Độ phức tạp:

Giai đoạn phân loại các phần tử có độ phức tạp $\theta(n)$.

Giai đoạn phân hoạch các phần tử đòi hỏi độ phức tạp O(n) (vì mỗi phần tử phải đổi chỗ đúng một lần, và n lần cho n phần tử).

Giai đoạn sắp xếp các phân vùng con (trong trường hợp tốt nhất là các phân vùng con có số phần tử bằng nhau) đòi hỏi độ phức tạp $mO\left(\left(\frac{n}{m}\right)^2\right) = O\left(\frac{n^2}{m}\right)$.

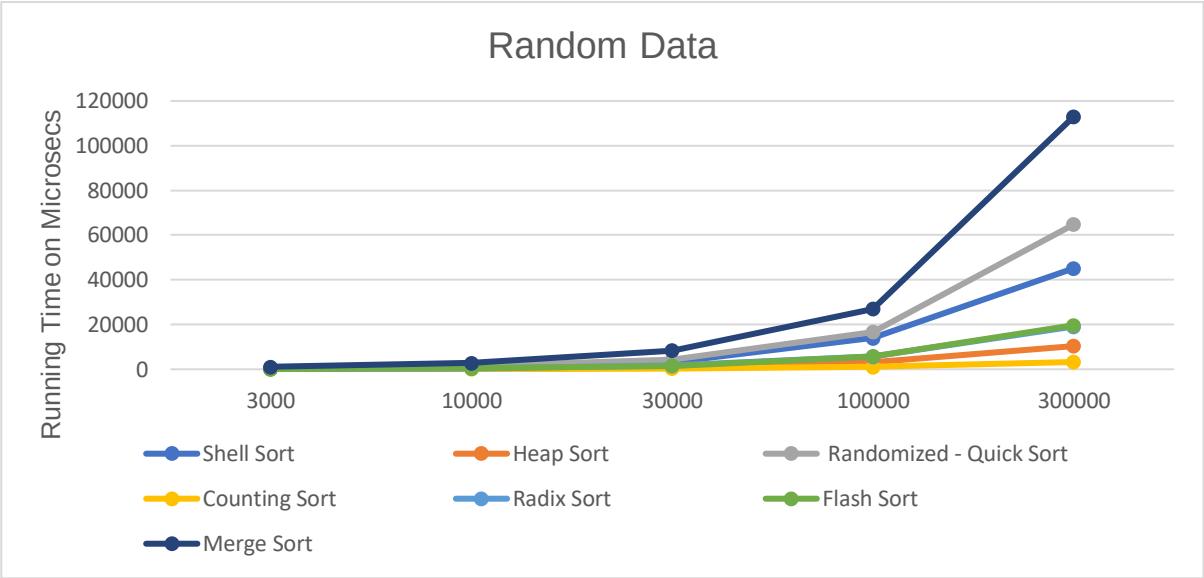
Rõ ràng thời gian chạy còn phụ thuộc vào cách chọn số phân lớp m của người cài đặt.

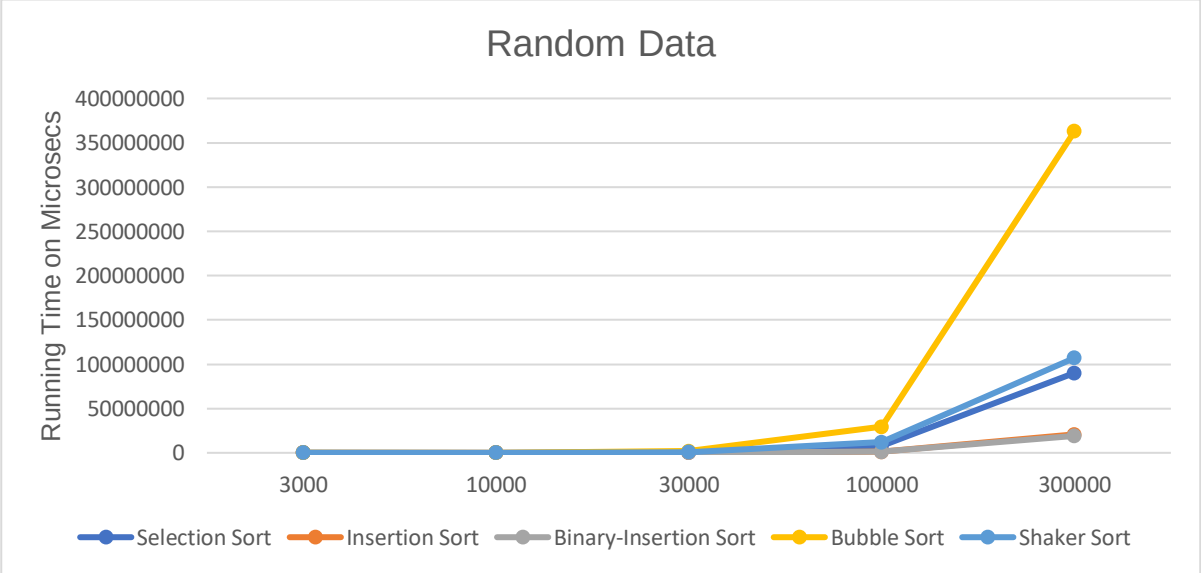
Theo thực nghiệm thì $m \approx 0.43n$ đạt kết quả tốt nhất.

+ Bộ nhớ: O(m) do chỉ cần thêm mảng phụ để tính toán số phần tử có trong các phân lớp.

II Minh họa thời gian chạy của các thuật toán sắp xếp:

1_Bộ dữ liệu ngẫu nhiên:





Với bộ dữ liệu ngẫu nhiên, độ lớn của mảng nhỏ hơn 30000 thì thời gian chạy của các thuật toán gần như bằng nhau không có sự khác biệt quá lớn.

Tuy nhiên khi độ lớn của mảng lớn hơn 30000 các thuật toán có độ phức tạp logarith và tuyến tính thể hiện ưu thế, với thời gian chạy nhanh hơn rõ rệt so với các thuật toán còn lại.

_Ở dữ liệu mảng có size 100000, Bubble Sort có thời gian chạy chậm nhất khi có độ phức tạp bình phương, tiếp theo là đến Selection Sort, Shaker Sort, Insertion Sort, Binay Insertion Sort,…

Counting Sort có thời gian chạy chậm nhất khi có độ phức tạp tuyến tính, tiếp theo là đến Heap Sort, Flash Sort, Radix Sort, Randomized QuickSort, Merge Sort.

_Ở dữ liệu mảng có size 300000, các thuật toán giữ nguyên thứ tự thời gian chạy.

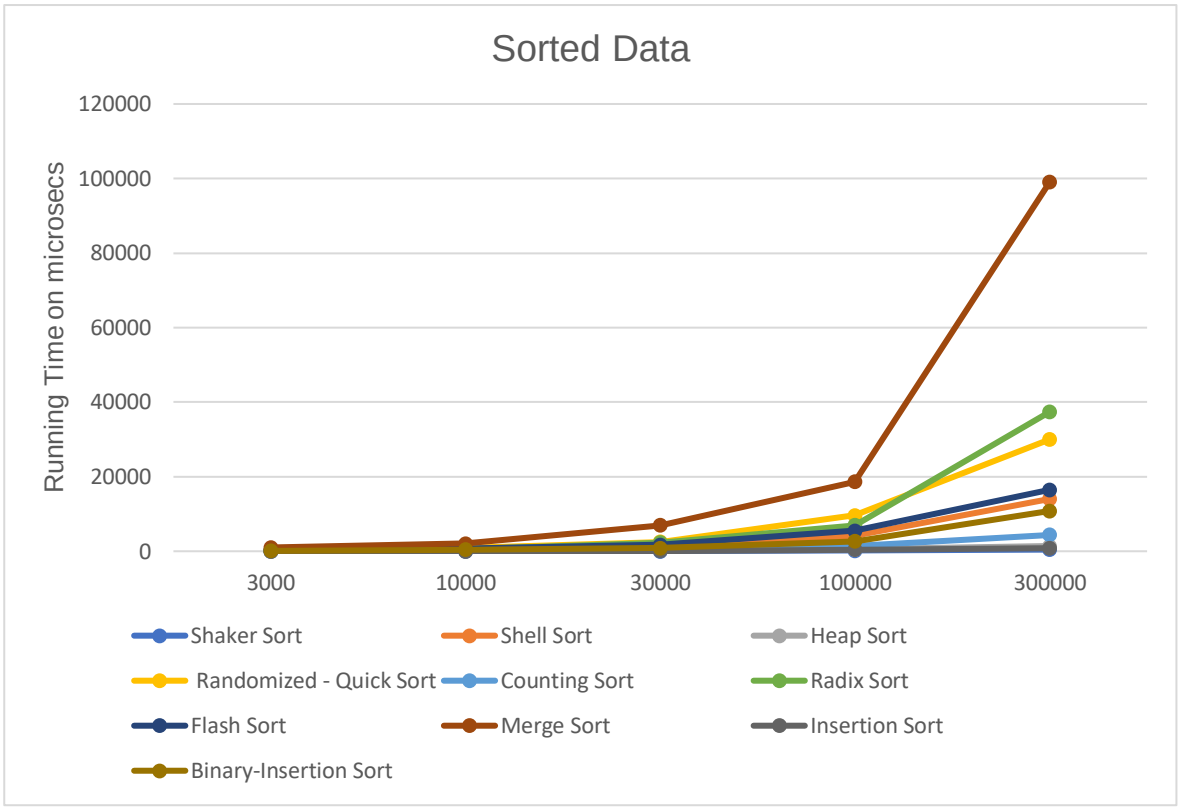
_Giải thích:

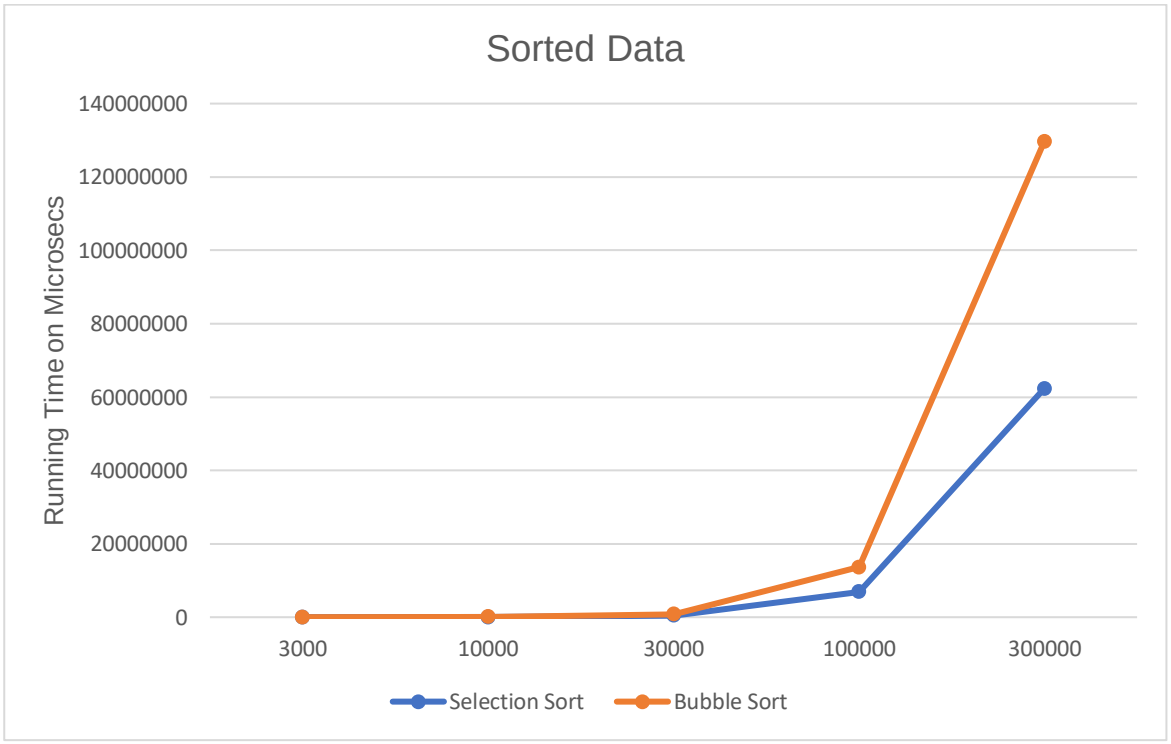
Các thuật toán Bubble Sort, Selection Sort luôn chạy đủ bình phương độ dài mảng các thao tác nên khi độ dài mảng tăng lên nhiều lần cũng làm thời gian chạy của các thuật toán này tăng cao.

Thuật toán Counting Sort với cách tạo ra data ngẫu nhiên đã cho thì độ chênh lệch dữ liệu giữa phần tử nhỏ nhất và phần tử nhỏ nhất trong mảng không vượt quá độ dài của mảng nên thuật toán này chạy với độ phức tạp tuyến tính cho ra thời gian chạy vượt trội so với thuật toán khác.

Với bộ dữ liệu ngẫu nhiên thì thuật toán phụ thuộc vào cờ hiệu như Shaker Sort không có khả năng hoàn thành sớm, các thuật toán Insertion Sort, Binary Insertion Sort vẫn có thể gặp các dữ liệu gây ra trường hợp xấu nên thời gian chạy vẫn còn lớn.

2_Dữ liệu đã được sắp xếp tăng sẵn:





Với bộ dữ liệu đã được sắp xếp tăng sẵn, khi độ dài của mảng nhỏ hơn 30000, các thuật toán có thời gian chạy gần như bằng nhau, không có sự chênh lệch quá nhiều.

Khi độ dài của mảng lớn hơn 30000, các thuật toán có thời gian chạy nhanh nhất là Shaker Sort, Insertion Sort, Heap Sort, Counting Sort,…các thuật toán có thời gian chạy chậm nhất vẫn là Bubble Sort, Selection Sort.

Giải thích:

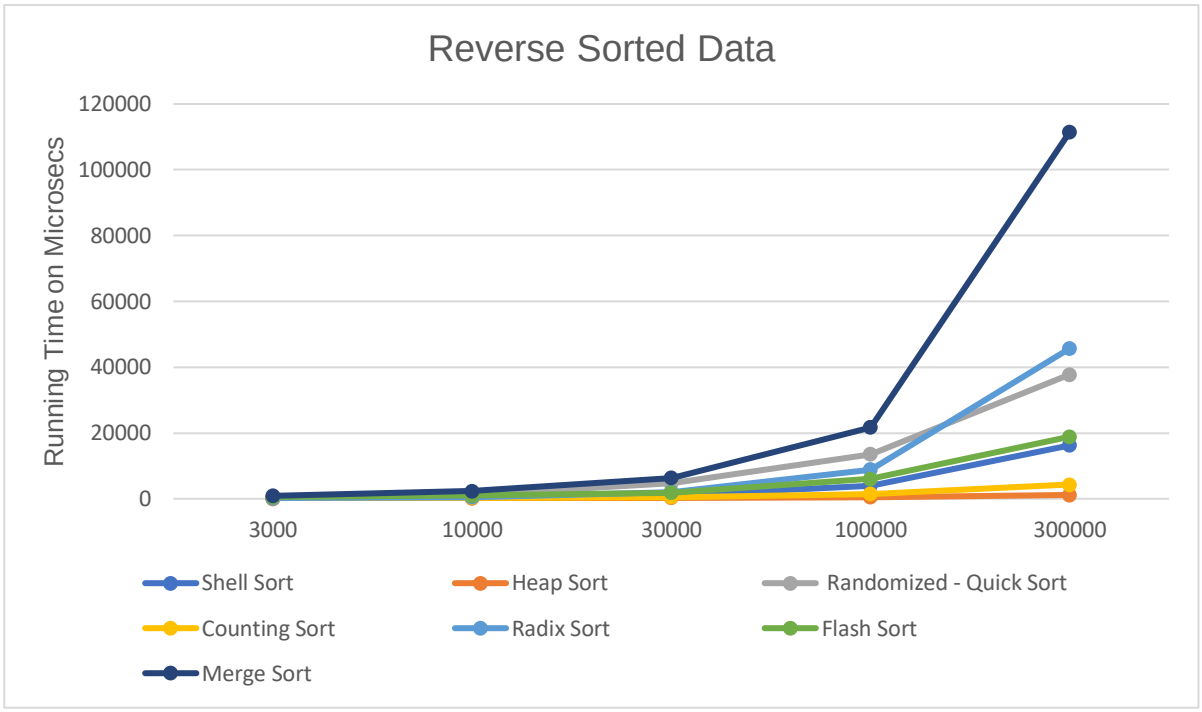
Với bộ dữ liệu đã được sắp xếp tăng sẵn, thuật toán sử dụng kết quả của cờ hiệu như Shaker Sort sẽ nhanh chóng kết thúc do nhận thấy rằng không có sự đổi chỗ nào của hai phần tử liền kề gần nhau.

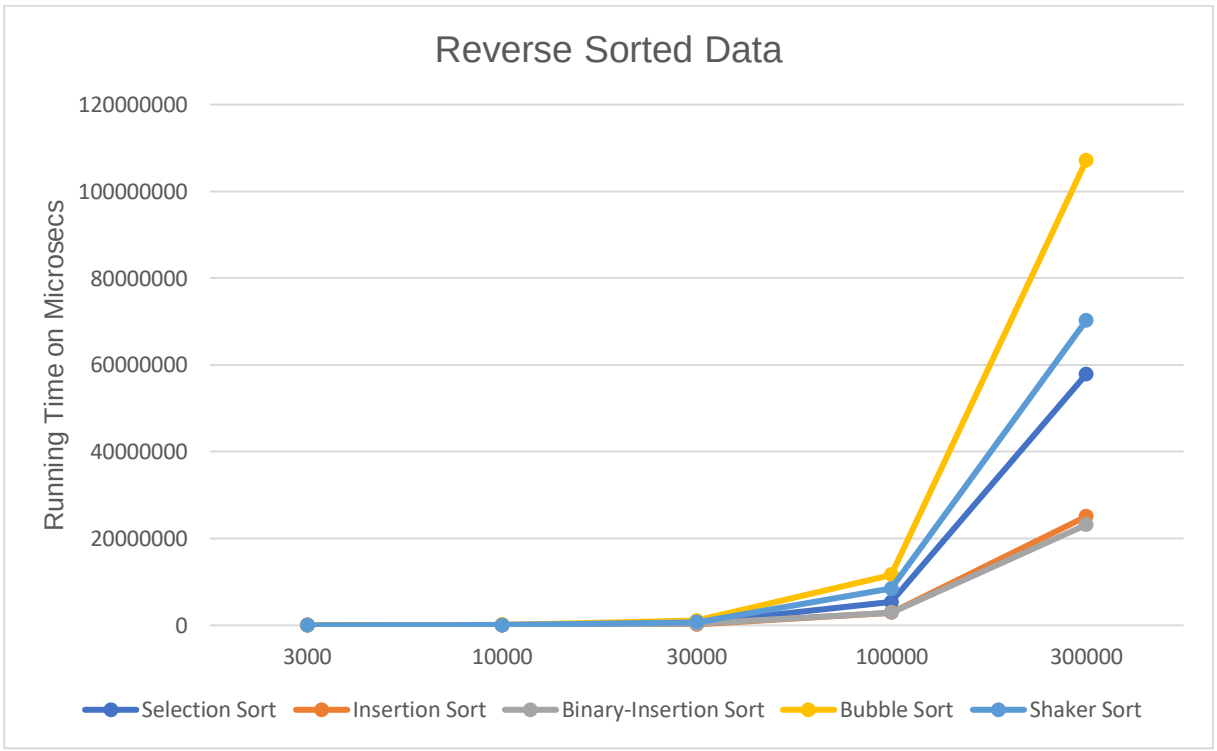
Thuật toán Insertion Sort khi muốn chèn phần tử arr[i] vào mảng con từ arr[0] đến arr[i-1] sẽ nhận thấy được không cần phải thực hiện gì nên tổng thể thuật toán chỉ cần chạy một vòng lặp bên ngoài duyệt toàn bộ mảng thôi, làm cho thời gian chạy của Insertion Sort rơi vào trường hợp tốt nhất khi chỉ ở mức tuyến tính. Thuật toán Binary Insertion Sort trong trường hợp này phải mất chi phí tìm kiếm vị trí đứng phù hợp cho từng phần tử mảng nên làm cho thời gian chạy không tốt bằng Insertion Sort nhưng vẫn khá tốt so với mặt bằng chung các thuật toán khác do không cần phải di chuyển các phần tử mảng đi đâu cả.

Với cách tạo data như đã cho thì cũng tạo điều kiện cho thuật toán Counting Sort có thời gian chạy nhanh, do phần tử mảng có có giá trị lớn nhất vẫn nhỏ hơn độ dài của mảng.

Các thuật toán Bubble Sort, Selection Sort có thời gian chạy không phụ thuộc nhiều phân bố của dữ liệu mà chỉ phụ thuộc phần lớn vào độ lớn dữ liệu của mảng nên thời gian chạy của những thuật toán vẫn lớn nhất khi độ lớn dữ liệu của mảng tăng.

3_Dữ liệu đã được sắp xếp giảm dần:





Với bộ dữ liệu đã được sắp xếp giảm sẵn, khi độ dài của mảng không vượt quá 30000, các thuật toán có thời gian chạy gần như bằng nhau, không có sự chênh lệch quá nhiều.

Khi độ dài của mảng lớn hơn 30000, các thuật toán có thời gian chạy nhanh nhất là Heap Sort, Counting Sort, Shell Sort, Flash Sort…các thuật toán có thời gian chạy chậm nhất là Bubble Sort, Shaker Sort, Selection Sort,…

Giải thích:

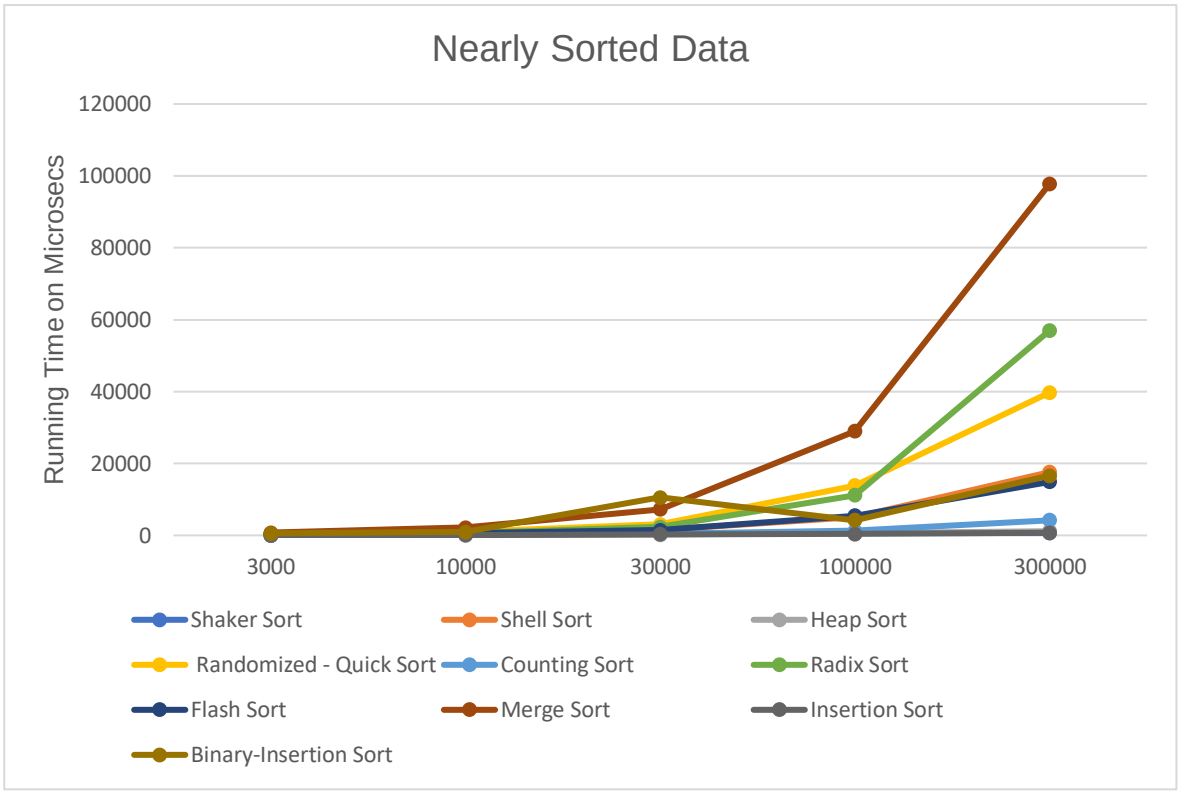
Với bộ dữ liệu đã được sắp xếp giảm sẵn, thuật toán Heap Sort có thời gian chạy nhanh rõ rệt là bởi vì cấu trúc Max Heap được biểu diễn qua mảng sẽ nhanh chóng được xây dựng ngay từ đầu bởi vì mảng đang được sắp xếp giảm dần.

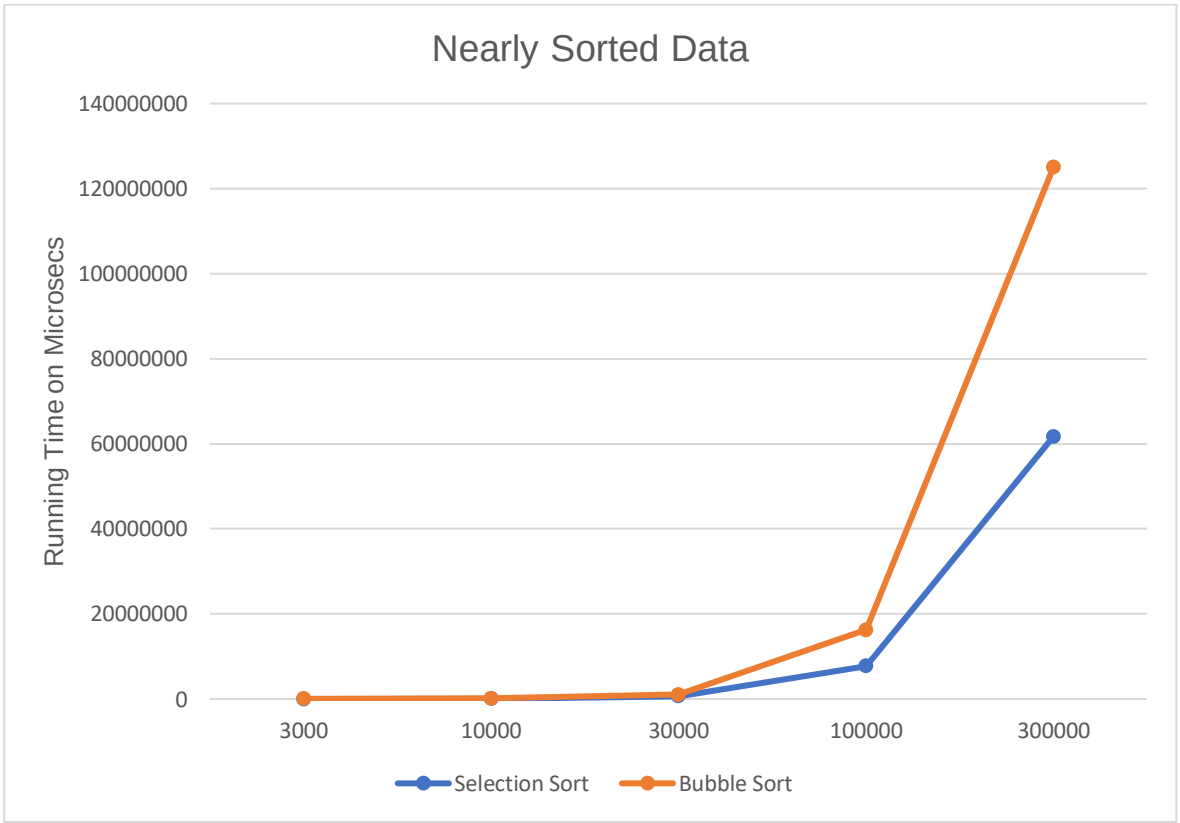
Với cách tạo data như đã cho thì cũng tạo điều kiện cho thuật toán Counting Sort có thời gian chạy nhanh, do phần tử mảng có có giá trị lớn nhất vẫn nhỏ hơn độ dài của mảng.

Thuật toán Shaker Sort trong trường hợp này các kĩ thuật cờ hiệu, lưu vị trí đổi chỗ cuối cùng trong cả hai chiều đi không còn tác dụng, nhưng dựa vào kết quả sau mỗi chiều đi thì các phần tử lớn nhất, nhỏ nhất của mảng được sắp xếp đúng vị trí thì thuật toán này có điểm tối ưu so với thuật toán Bubble Sort nên có thời gian chạy tốt hơn.

Với trường hợp dữ liệu được sắp xếp ngược lại như thế này, các thuật toán Insertion Sort, Binary Insertion Sort đã gặp phải trường hợp xấu nhất cho nên thời gian chạy của các thuật toán này khá chậm, có thể được xếp vào nhóm các thuật toán chậm nhất trong trường hợp này.

4_Dữ liệu gần như được sắp xếp tăng:





Với bộ dữ liệu đã được sắp xếp gần như tăng, khi độ dài của mảng không vượt quá 30000, các thuật toán có thời gian chạy gần như bằng nhau, không có sự chênh lệch quá nhiều.

Khi độ dài của mảng lớn hơn 30000, các thuật toán có thời gian chạy nhanh nhất là Shaker Sort, Insertion Sort, Heap Sort, Counting Sort,…các thuật toán có thời gian chạy chậm nhất là Bubble Sort, Selection Sort.

Giải thích:

Khi dữ liệu đã được sắp xếp gần như tăng, thì các thuật toán Insertion Sort, Shaker Sort thường rơi vào trường hợp gần với trường hợp tốt nhất nên cho ra thời gian chạy rất tốt.

Dữ liệu phân bố cũng tạo điều kiện cho Counting Sort thực thi nhanh nhưng thường lệ, các lợi thế từ cách phân bố dữ liệu cũng làm cho Binary Insertion Sort có thời gian chạy thấp hơn khi độ lớn dữ liệu tăng so với Merge Sort, Radix Sort, Randomized-QuickSort,…

Các thuật toán không có sự tối ưu nào như Bubble Sort, Selection Sort vẫn giữ thời gian chạy khá chậm như các phân bố dữ liệu khác.

III. Nhận xét chung:

_Về thời gian chạy:

Nhóm các thuật toán chậm nhất bao gồm Selection Sort, Bubble Sort khi mà hai thuật toán này không có chi tiết nào làm tối ưu thời gian chạy, tốc độ phụ thuộc không nhiều vào cách phân bố dữ liệu mà phụ thuộc phần lớn vào độ lớn của dữ liệu mảng đầu vào.

Các thuật toán như Merge Sort, Randomized Quicksort, Flash Sort,… cho thấy được sự ổn định của mình dù với bất kỳ độ lớn dữ liệu hay cách phân bố nào. Với sự ổn định đó, các thuật toán này thường được sử dụng ở bên ngoài thực tế với cách phân bố và loại giá trị đa dạng để cho ra thời gian chạy tốt nhất.

Thuật toán Heap Sort cũng cho thấy sự ổn định với nhiều kiểu phân bố dữ liệu và độ lớn dữ liệu và cũng khá tối ưu khi không dùng thêm bộ nhớ ngoài mảng.

Các thuật toán phụ thuộc nhiều vào phân bố dữ liệu như Counting Sort, Radix Sort,…có được ưu thế trong minh họa lần này vì cách tạo ra data cho việc thống kê thời gian chạy, điều này cũng cho thấy trong một số trường hợp dữ liệu được phân bố đặc biệt thì Counting Sort, Radix Sort nên được xem xét sử dụng còn các trường hợp khác thì không nên vì rất hao tốn tài nguyên của hệ thống và không còn giữ đươc ưu thế về độ phức tạp.

Các thuật toán Shaker Sort, Insertion Sort, Binary Insertion Sort cho thấy ưu thế khi làm việc với các dữ liệu gần trường hợp tốt nhất như mảng đã được xếp tăng hoặc là gần tăng, nhưng mà thời gian chạy vẫn còn lớn với các phân bố dữ liệu ngẫu nhiên thường xảy ra trong thực tế. Trong các thuật toán có độ phức tạp trung bình là bình phương thì các thuật toán này thường được sử dụng để sắp xếp các mảng có kích thước nhỏ vì thường chạy nhanh hơn các thuật toán có độ phức tạp Logarith do sai khác về hằng số trong biểu thức độ phức tạp. Việc thời gian chạy trên các mảng có kích thước nhỏ tốt hơn nên các thuật này thường được sử dụng các bước cơ sở của các thuật toán như Flash Sort, Merge Sort, QuickSort,…

_Về tính ổn định của các thuật toán:

Các thuật toán sắp xếp dựa trên sự so sánh của các phần tử như Insertion Sort, Binary Insertion Sort, Selection Sort, Bubble Sort, Shaker Sort, Merge Sort, Quick Sort giữ được tính ổn định.

Các thuật toán Radix Sort, Counting Sort với cách cài đặt đã nộp thì tính ổn định của thuật toán vẫn được giữ.

Randomized QuickSort, Shell Sort, HeapSort, Flash Sort không giữ được tính ổn định.

TƯ LIỆU THAM KHẢO:

1. https://www.geeksforgeeks.org/shellsort/
2. https://stackoverflow.com/questions/9798078/what-are-the-criteria-for-choosing-a-sorting-algorithm
3. https://medium.com/karuna-sehgal/a-quick-explanation-of-quick-sort-7d8e2563629b