Trường đại học Khoa học Tự nhiên

Khoa Công nghệ Thông tin

Báo cáo phân tích thời gian và đếm phương thức cơ sở của các thuật toán sắp xếp

Người thực hiện: Nguyễn Hoàng Lâm

Giảng viên: Nguyễn Thanh Phương

Trợ Giảng: Bùi Huy Thông

Môn học: Cấu trúc dữ liệu và giải thuật

11/2021

MỤC LỤC

I. Thông tin chung

Họ tên: Nguyễn Hoàng Lâm

MSSV: 20120316 – 20CTT1TN2

Cấu hình máy tính thực hiện quá trình đo và đếm:

- Acer Aspire 5

- Cpu: Intel Core i5-1135G7, 2.4 Ghz – 4.2Ghz, 8mb cache

- VGA: Intel® Iris® Xe Graphics, NVIDIA GeForce MX350

- RAM: 8gb

II. Giải thích thuật toán

1. Selection Sort

- Ý tưởng: chọn ra phần tử nhỏ nhất trong mảng bằng tìm kiếm tuyến tính rồi đưa lên đầu mảng.

- Giải thích từng bước: Cho mảng **arr** có n phần tử vị trí từ 1 - n

Vòng lặp biến i:1->n-1, vòng lặp 2 biến j:i+1 -> n. Chọn phần tử tại i (arr[i]) làm phần tử nhỏ nhất. Gán **min** bằng i (vị trí của phần tử nhỏ nhất). Lần lượt so sánh **arr[min]** với các phần tử kế tiếp, gặp phần tử nhỏ hơn thì lưu lại **vị trí** của phần tử đó vào **min**. Sau khi duyệt tới cuối mảng thì đổi chỗ phần tử tại i với phần tử nhỏ nhất (arr[min] và arr[i]). Nhận thấy sau mỗi vòng lặp thì phần tử nhỏ nhất đã được đưa lên đầu mảng. Tiếp tục lặp lại từ vị trí **i** kế tiếp.

- Phân tích độ phức tạp:

+ Thời gian: $O(n^2)$ ở mọi trường hợp vì có 2 vòng lặp lồng nhau, mỗi vòng lặp từ 1-n.

+ Không gian: O(1) vì sắp xếp trực tiếp trên mảng.

2. Insertion Sort

- Ý tưởng: kiểm tra một phần tử, so sánh lần lượt với phần tử bên trái nếu lớn hơn thì chèn phần tử hiện tại qua trái 1 ô.

- Giải thích từng bước: ban đầu chọn phần tử thứ 2 trong mảng để xét, lưu vào biến tạm. Các phần tử bên trái phần tử đang chọn được xem như đã xếp theo thứ tự. So sánh với phần tử bên trái, nếu phần tử bên trái lớn hơn thì gán dịch phần tử đó qua phải 1 bước, xong tiếp tục so sánh với phần tử bên tiếp theo. Lặp lại cho tới khi tới đầu mảng hoặc gặp phần tử nhỏ hơn bằng thì dừng lại. Chèn phần tử đã lưu trong biến tạm vào ô đó, tiếp tục chọn phần tử kế tiếp để thực hiện sắp xếp chèn.

- Phân tích độ phức tạp:

+ Thời gian: $O(n^2)$. Vì có 2 vòng lặp lồng nhau

+ Không gian: O(1) vì sắp xếp trực tiếp trên mảng.

- Biến thể: Binary Insertion Sort:

Ta thấy mảng bên trái tại vị trí đang chọn đã được sắp theo thứ tự tăng dần, vì vậy thuật toán này tìm kiếm được vị trí cần chèn bằng Binary Search, nhanh hơn so với tìm kiếm tuyến tính trong Insertion Sort.

Về độ phức tạp: thời gian: $O(n^2)$ vì thao tác chèn vẫn tốn thời gian O(n). Không gian: O(1)

- Biến thể khác: Shell sort: sẽ được phân tích ở dưới.

3. Bubble Sort

- Ý tưởng: tưởng tượng mảng dựng đứng lên, ta sẽ muốn phần tử lớn nhất nổi lên tới vị trị cuối mảng bằng cách so sánh lần lượt 1 cặp 2 phần tử cạnh nhau, phần tử nào lớn hơn sẽ được đổi chỗ ra sau.

- Giải thích từng bước: cho 2 vòng lặp, vòng lặp ngoài **i: 1->n -1**. Vòng thứ trong từ **j: 1 -> n - i**. Ta lần lượt so sánh cặp phần tử **arr[j]** và **arr[j+1]**, nếu phần tử lớn hơn đổi chỗ ra sau, sau đó dịch phải 1 ô, tiếp tục so sánh tới hết mảng. Nhận thấy sau vòng lặp đầu tiên thì phần tử lớn nhất đã "nổi bọt" tới cuối mảng (đúng vị trí). Vì thế vòng lặp kế tiếp, ta chỉ xét tới vị trí **n – 2**. Vậy nên vòng lặp trong chỉ chạy tới **n-i**.

- Phân tích độ phức tạp:

+ Thời gian: $O(n^2)$ dễ thấy vì có 2 vòng lặp.

+ Không gian: O(1). Sắp xếp trực tiếp trên mảng.

- Biến thể: họ đã cải tiến bubble sort bằng cách kiểm tra đổi chỗ. Nếu trong một lần lặp mà không có đổi chỗ thì xem như mảng đã được sắp và dừng vòng lặp. Ngoài ra còn một biến thể là Shaker Sort, sẽ được trình bày ở dưới.

4. Shaker Sort

- Shaker sort là một cải tiến của bubble sort. Ý tưởng của nó là sắp xếp nổi bọt nhưng mà ở cả 2 chiều lên và xuống.

- Giải thích từng bước: ban đầu đặt một biến kiểm tra việc hoán đổi, một biến left để xác định bên trái mảng chưa sắp, một biến right xác định vị trí bên phải. Cho thực hiện vòng lặp nổi bọt như bubble sort. Sau một lượt đi, kiểm tra hoán đổi xem, nếu không có hoán đổi xảy ra thì kết thúc chương trình. Nếu không tiếp tục thực hiện vòng lặp nổi bọt từ cuối về đầu mảng, kiểm tra hoán đổi. Lặp lại các bước trên tới khi mảng đã được sắp thì dừng.

- Phân tích độ phức tạp:

+ Thời gian: $O(n^2)$ dễ thấy vì có 2 vòng lặp.

+ Không gian: O(1). Sắp xếp trực tiếp trên mảng.

5. Shell Sort

- Ý tưởng: Shell Sort là biến thể của Insertion Sort, thay vì chèn lần lượt bên trái từng phần tử thì chèn một khoảng là amount được chọn.

- Giải thích từng bước: đầu tiên ta chọn một khoảng nhảy (amount) bằng **n/2**. Dùng vòng lặp để lần lượt chia đôi giảm khoảng này về 1. Trong vòng lặp, ta chọn phần tử ban đầu bằng đúng vị trí tại khoảng nhảy **i = amount + 1**, sau đó so sánh phần tử được chọn với phần tử **arr[i- amount].** Nếu **arr[i- amount]** nhỏ hơn **arr[i]** thì gán **arr[i] = arr[i- amount].** Ta thấy tương tự như Insertion Sort nhưng là chèn mỗi lần một khoảng bằng amount. Chọn phần tử kế tiếp và lặp lại công việc trên.

- Phân tích độ phức tap: tuy có cải tiến nhưng về mặt thời gian thì vẫn là $O(n^2)$, không gian là O(1).

6. Heap Sort

- Ý tưởng: Heap Sort là thuật toán sắp xếp dựa trên cấu trúc Heap, với sắp xếp tăng dần thì ta sẽ dùng Max-Heap.

- Giải thích từng bước: chúng ta có hàm **Heapify**: là hàm dùng để tạo ra một Max Heap từ một rễ của cây được truyền vào (root) và 2 nút con của nó. Nếu 1 trong 2 nút con lớn hơn nút cha và là lớn nhất thì đổi chỗ chúng, sau đó tiếp tục đệ quy **Heapify** với nút con đó. Thuật toán này dừng khi (root) là nút lớn nhất trong 3 nút.

Vì vậy, heap sort ban đầu sẽ xây dựng một max heap bằng cách dùng Heapify cho nút **(n+1/2)** và lặp dần về nút 1. Sau khi vòng lặp này hoàn thành, ta đã xây được một max heap.

Vì là Max Heap nên phần tử **arr[1]** là phần tử lớn nhất, ta hoán đổi **arr[1]** và **arr[n].** Khi đó vị trí từ n trở đi xem như đã được sắp. Lúc này tiếp tục dùng **Heapify** cho **arr[1]** để tạo ra Max Heap. Sau n lần lặp thì mảng đã được sắp.

- Phân tích độ phức tạp:

+ Thời gian: Heapify: O(logn), xây Max Heap tốn khoảng O(nlogn) và đổi chỗ phần tử và gọi Heapify tốn O(nlogn). Vậy độ phức tạp của thuật toán là O(nlogn).

+ Không gian: O(1)

7. Merge Sort

- Ý tưởng: Merge Sort dùng thuật toán chia để trị. Dùng đệ quy chia mảng làm đôi tới khi mỗi mảng con là 1 phần tử. Dùng thuật toán Merge và kĩ thuật 2 con trỏ để ghép chúng lại thành mảng được sắp.

- Giải thích từng bước:

Thuật toán Merge: cho 2 biến i và j chạy ở 2 mảng trái và phải. So sánh **arrLeft[i]** và **arrRight[j],** nếu phần tử nào nhỏ hơn thì thêm vào mảng chính cần ghép, tăng i hoặc j lên 1. Thực hiện tới khi một bên đã lấy hết các phần tử vào mảng chính. Dùng vòng lặp nạp toàn bộ phần tử còn lại của của mảng kia vào mảng chính.

Merge Sort: tính ra vị trí giữa **mid = l +(r-l)/2**. Thực hiện Merge Sort cho **L->mid** và từ **mid+1 -> R** (đệ quy). Sau khi chia nhỏ các đoạn ra thì dùng hàm Merge như đã giải thích ở trên.

- Phân tích độ phức tạp: O(nlog n). Quá trình tách nhỏ thành các mảng con 1 phần tử tốn logn, quá trình ghép 2 mảng con sẽ tốn n và quá trình đó lặp lại logn lần để thành mảng hoàn chỉnh. Suy ra độ phức tạp thuật toán của giải thuật là **O(nlog n)**.

- Không gian: O(n) vì phải lưu thêm các mảng phụ trong qua trình merge.

8. Quick Sort

- Ý tưởng: Quick Sort cũng dùng chia để trị, nó chọn một phần tử làm chốt và dùng thuật toán phân hoạch để đưa chốt về đúng vị trí của nó

- Giải thích từng bước: Trong bài này chọn chốt là phần tử ở chính giữa. Ban đầu chọn i ở đầu, j ở cuối, pivot ở giữa, 2 biến l, r là giá trị đầu cuối. Kiểm tra lần lượt **arr[i] < arr[pivot]** và **arr[j] > arr[pivot].** Nếu đúng thì **i++** hoặc **j--,** nếu sai thì dừng lại và đổi chỗ 2 phần tử cho nhau. Nhận thấy sau khi duyệt hết mảng thì phần tử Pivot đã được nằm đúng vị trí của mình. Sau đó thực hiện đệ quy Quick Sort với 1 nửa trái từ **l->j** (lúc này j < i) và nửa phải từ i tới r (cần kiểm tra l<j và i<r).

- Phân tích độ phức tạp:

+ Thời gian**: O(nlogn)** trong trường hợp tốt nhất và trung bình. Trường hợp xấu nhất là $O(n^2)$ khi chúng ta luôn chọn pivot là số lớn nhất hoặc nhỏ nhất.

+ Không gian: O(1)

9. Counting Sort

- Ý tưởng: với mỗi phần tử trong mảng, ta sẽ đếm số lần xuất hiện của phần tử đó vào 1 mảng đếm. Sau đó dùng một số biện pháp toán học để tính toán được ví trí của phần tử đó trong mảng kết quả.

- Giải thích từng bước: Tìm phần tử **max** trong input, tạo mảng **count** có **max** phần tử.

Dùng vòng **for** lặp toàn bộ mảng **input**, tại giá trị **i (1 $\leq$ i $\leq$ n)** ta tăng biến đếm lên 1 **(count[i]++).** Sau khi lặp xong ta có mảng **count** lưu số lần xuất hiện của mỗi phần tử trong input.

Sau đó ta dùng vòng lặp qua mảng count. Cho i chạy từ 1, tạo biến k = 1. Nếu **count[i] != 0** thì tiến hành đặt **output[k] = i, k++** và **count[i]--.** Lặp tới khi nào **count[i] == 0** thì tăng **i (i++).**

Pseudo code:

```
for i = 1 to n:
        While count[i] != 0:
                Arr[k] = i; k++; count[i]-- ;
```

- Phân tích độ phức tạp:

+ Thời gian: tìm max O(n), đếm phần tử O(n), vòng lặp tính output O(max). Vậy độ phức tạp thuật toán này là O(2n+max) ~ **O(n+max).**

Trong trường hợp xấu nhất là phần tử max = n^2 thì thời gian là O(n^2).

+ Không gian: O(max): tốn thêm vị trí lưu mảng phụ count.

- Biến thể, thông tin thêm:

Với thuật toán trên, counting sort không thể sắp xếp mảng có số âm hoặc số thực. Có một biến thể để sắp xếp được cả số âm, đó là thay vì tạo mảng đếm từ 1 – max thì ta sẽ tìm phần tử nhỏ nhất rồi tạo mảng đếm từ min – max. Ngoài ra các bước còn lại đều làm như counting sort bình thường.

Bên cạnh đó, ở thuật toán xuất ra output, thay vì duyệt lần lượt mảng count rồi đưa ra output thì ta sẽ cộng dồn từng phần tử của mảng count: **(count[i] += count[i-1]).** Sau khi cộng dồn mảng count, giá trị tại vị trí i trong mảng sẽ là vị trí thật sự của i trong output. Hay nói cách khác, phần tử i sẽ nằm ở vị trí count[i]. Ta dùng vòng lặp từ max->1 trong mảng count.

```
For i = n -> 1
        Output[count[arr[i]]] = arr[i];
        Count[arr[i]]--;
```

10. Radix Sort

- Ý tưởng: Radix Sort sẽ sắp xếp theo từng chữ số 1, sắp xếp từ hàng đơn vị, chục, … hoặc sắp xếp lần lượt từ hàng lớn nhất tới hàng đơn vị.

- Giải thích từng bước:

Ban đầu ta tìm phần tử lớn nhất trong mảng, để có thể biết được số lần lặp qua từng hàng đơn vị, chục, … Sau đó ta bắt đầu so sánh từ hàng đơn vị. Dùng vòng lặp cho k = 1, lặp tới khi **max/k=0** thì dừng lại. Sau mỗi vòng lặp thì k nhân thêm 10. Dùng k để thực hiện Counting Sort cho từng chữ số của phần tử trong mảng. Chúng ta sẽ so sánh và sắp mảng theo thứ tự tăng dần của hàng đơn vị, sau đó là chục, … hàng lớn nhất. Vì counting sort mang tính "Stable" nên mới có thể dùng radix sort. Ví dụ số 23 đứng trước 03 thì sau khi sắp xếp hàng đơn vị vẫn sẽ đứng trước.

Mã giả:

```
max = findMax(array)
For i=1, dừng khi max / i = 0
        Counting Sort(array, i)    //i truyền vô để có thể lần lượt so sánh theo từng chữ số từ hàng đơn vị
        i = i * 10
```

- Độ phức tạp:

+ Thời gian: tìm max tốn O(n), vòng lặp tốn $O(\log_{10}(max))$, counting sort tốn O(n+max). Vậy thuật toán tốn $O[(n+max)\log_{10}(max)]$ trong trường hợp max nhỏ. Có một cách có thể cải thiện thời gian là thay đổi hệ cơ số. Với hệ cơ số (base) lớn hơn ta có $\log_{base}(max)$ tiến về 1. Suy ra độ phức tạp tiến về O(n).

+ Không gian: $O(\log_{10}n * (n + 10))$, dùng để lưu mảng đếm và mảng kết quả.

- Biến thể, thông tin thêm: thay vì dùng Counting Sort để so sánh có thể dùng Bucket Sort. Bucket Sort dùng linked list để lưu các phần tử thay vì mảng đếm như bên Counting Sort.

11. Flash Sort

- Ý tưởng: Flash Sort sử dụng 3 bước để sắp xếp, dùng công thức chia ra các lớp, hoán vị các phần tử về đúng lớp của mình và hoán vị trong từng lớp.

- Giải thích từng bước:

+ Bước 1: chia ra các lớp. Ta tính phần tử lớn nhất và nhỏ nhất trong mảng bằng tìm kiếm tuyến tính. Sau đó tính số lớp cần chia bằng **0.43*n**.

+ Bước 2: Tạo mảng phụ có 0.43n phần tử. Thực hiện tính toán phân lớp của từng phần tử trong mảng ban đầu rồi cộng thêm 1 vào phân lớp đó:

Tính phân lớp **k = (m - 1) * (a[i] - min) / (max - min)**

Sau đó thêm vào mảng phụ: **sub[k]++**

Ta cộng dồn phần tử trong mảng phụ như trong Counting Sort:

For i=2->n:

    sub[i] += sub[i-1]

Sau khi cộng dồn thì ta đã có mảng sub mang vị trí kết thúc của từng phân lớp.

Giờ ta sẽ dùng hoán vị toàn cục để đưa từng phần tử về đúng phân lớp của mình.

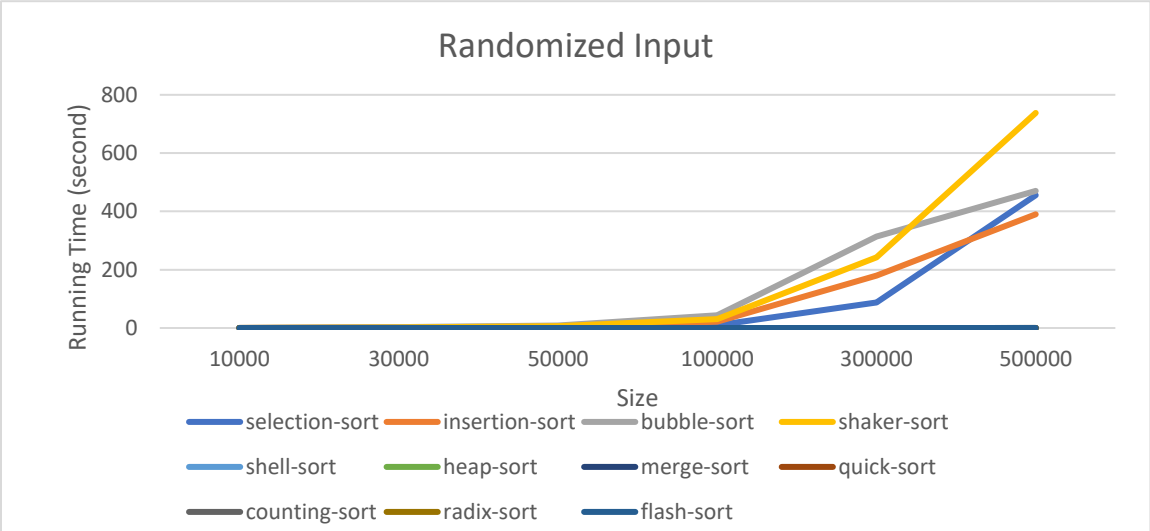
+ Bước 3: Ta hoán vị trong từng lớp ở trên. Ở bước 3 sẽ dùng sắp xếp chèn, tại vì khoảng cách di chuyển các phần tử trong mỗi phân lớp là không lớn nên Insertion Sort sẽ là thuật toán phù hợp trong trường hợp này.

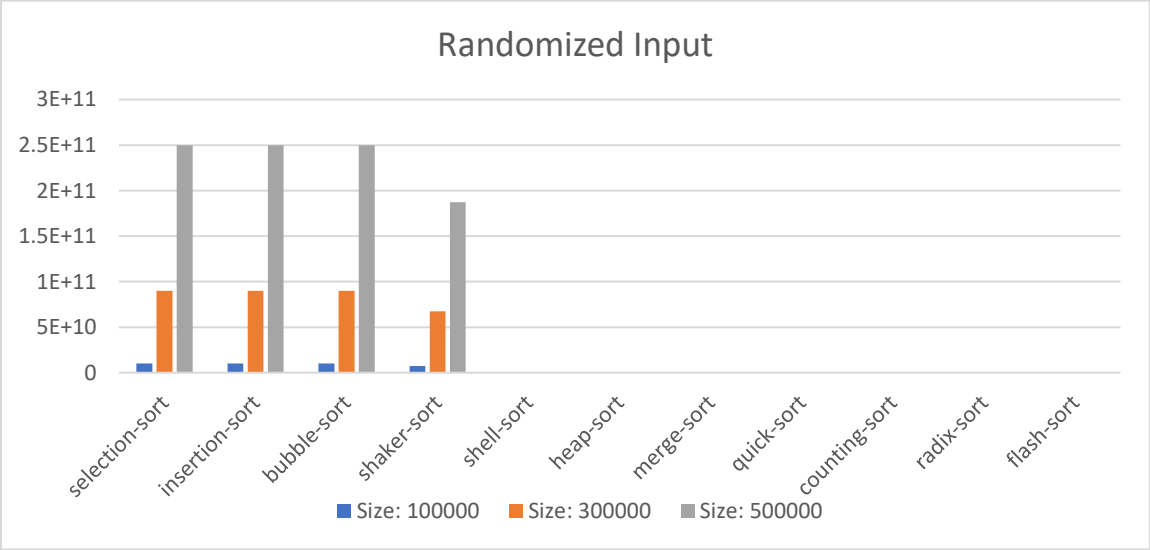
III. Kết quả thực nghiệm.

Trong các bảng dưới đây running time sẽ có đơn vị là **giây.**

1. Dữ liệu Random

Data order: RANDOM												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
selection-sort	0.1	100009999	0.88	900029999	2.31	2500049999	9.29	1E+10	87.58	9E+10	311.37	2.5E+11
insertion-sort	0.24	100009999	1.56	900029999	3.9	2500049999	22.19	1E+10	179.77	9E+10	278.85	2.5E+11
bubble-sort	0.28	100009999	2.92	900029999	8.4	2500049999	43.82	1E+10	314.16	9E+10	314.31	2.5E+11
shaker-sort	0.23	75259324	2.34	679655664	6.71	1866269724	30.19	7500999900	242.92	6.7549E+10	737.84	1.8736E+11
shell-sort	0	395772	0.01	1330986	0.01	2611438	0.02	6014544	0.08	20184174	0.14	38418293
heap-sort	0	273547	0.01	915185	0.01	1599899	0.03	3399923	0.09	11142161	0.15	19297225
merge-sort	0	457658	0	1513006	0.01	2637229	0.02	5574164	0.06	18136109	0.1	31288393
quick-sort	0	270636	0	928445	0.01	1589589	0.01	3341768	0.04	11219633	0.07	18881201
counting-sort	0	70004	0	209998	0	350004	0	700004	0.01	2100004	0.01	3500001
radix-sort	0	140058	0	510072	0	850072	0.01	1700072	0.04	6000086	0.04	10000086
flash-sort	0	101136	0	295672	0	486175	0	1016155	0.02	3051871	0.02	4998897

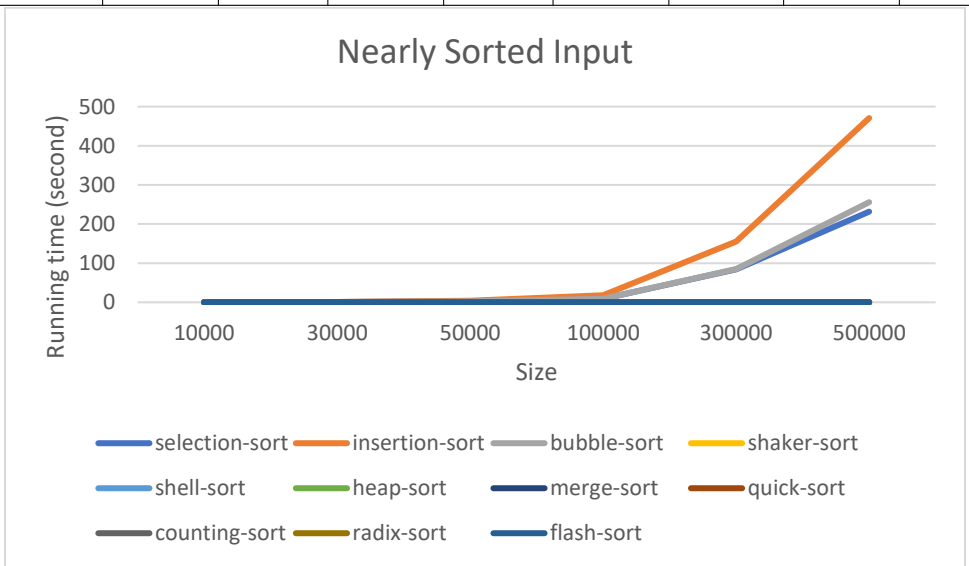


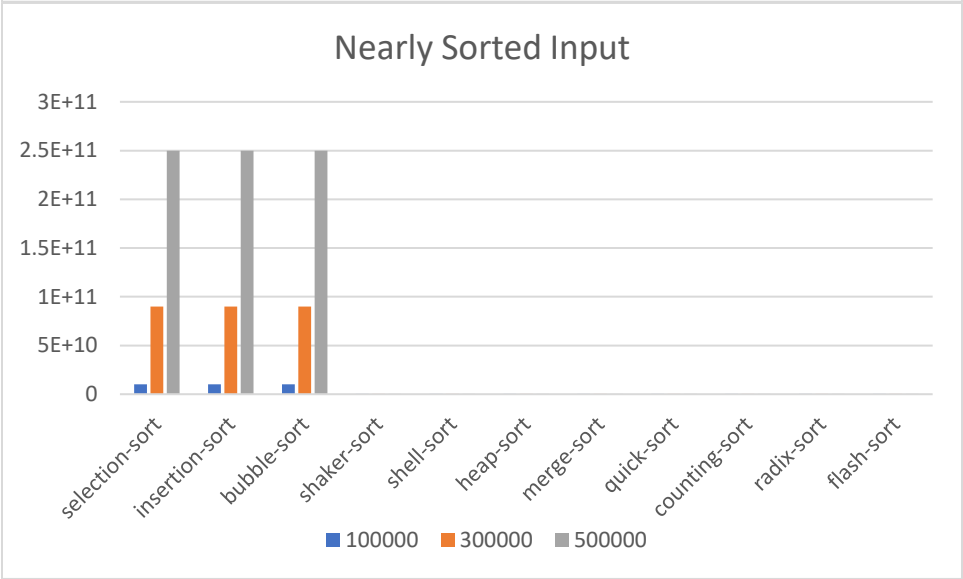


- Qua bảng so sánh tốc độ chạy, chúng ta thấy thời gian chạy có sự khác biệt rõ rệt giữa 4 thuật toán Selection Sort, Insertion Sort và Bubble Sort, Shaker Sort so với các thuật toán khác. Với đầu vào là RANDOM thì 4 thuật toán này đều có độ phức tạp thời gian là $O(n^2)$. Ở 3 thuật toán đầu ta thấy dù có thay đổi đầu vào bao nhiêu thì với cùng 1 INPUT SIZE đều sẽ cho ra số phép so sánh bằng nhau (điều này vẫn đúng với input là SORTED, NEARLY SORTED và REVERSE). Trong thuật toán Shaker Sort vì có đặt thêm biến kiểm tra mảng được sắp hay chưa nên số phép so sánh sẽ không cố định như 3 thuật toán trên. Tại 500,000 phần tử random thì Insertion Sort chạy chậm nhất, Counting Sort nhanh nhất.

2. Dữ liệu Nearly Sorted.

Data order: NEARLY SORTED												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
selection-sort	0.09	100009999	0.82	900029999	2.26	2500049999	9.04	10000099999	84.49	90000299999	231.66	2.5E+11
insertion-sort	0.1	100009999	0.93	900029999	4.13	2500049999	18.09	10000099999	155.68	90000299999	470.74	2.5E+11
bubble-sort	0.09	100009999	0.86	900029999	2.33	2500049999	9.39	10000099999	84.91	90000299999	255.86	2.5E+11
shaker-sort	0	219879	0	539919	0	1499775	0.01	2999775	0.01	5399919	0.05	14999775
shell-sort	0	261079	0	837501	0	1542721	0.01	3289583	0.02	10637745	0.08	18108451
heap-sort	0	288653	0.01	955165	0.01	1671705	0.02	3551521	0.07	11573641	0.25	19977665
merge-sort	0	420792	0	1375627	0.01	2389700	0.01	5036419	0.04	16323828	0.2	27985729
quick-sort	0	149099	0	485566	0	881127	0	1862200	0.01	5889332	0.11	10048626
counting-sort	0	70004	0	210004	0	350004	0	700004	0	2100004	0.02	3500004
radix-sort	0	140058	0	510072	0	850072	0.01	1700072	0.02	6000086	0.17	10000086
flash-sort	0	125763	0	377367	0	628969	0	1257965	0.01	3773965	0.03	6289971



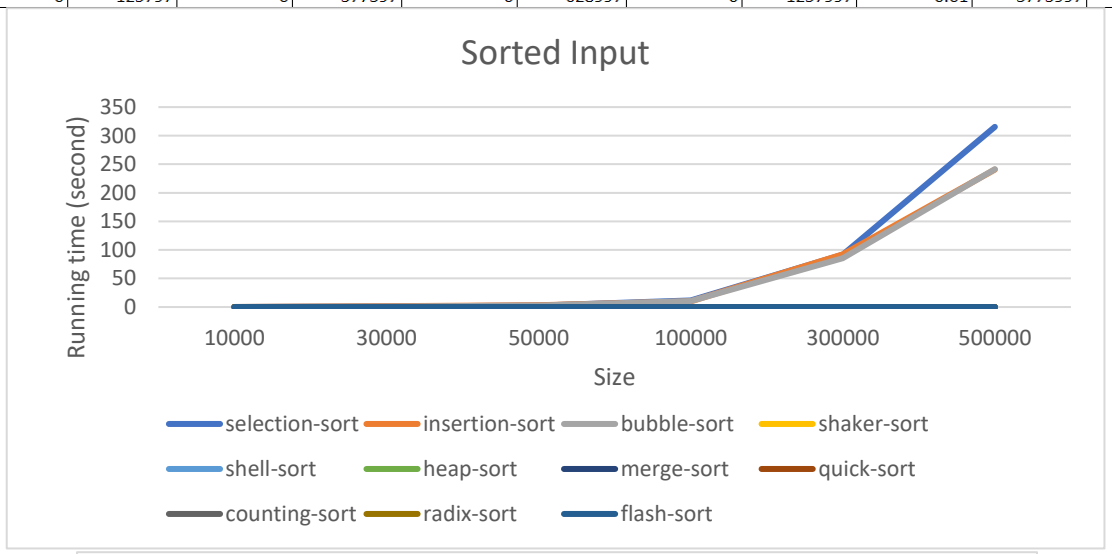


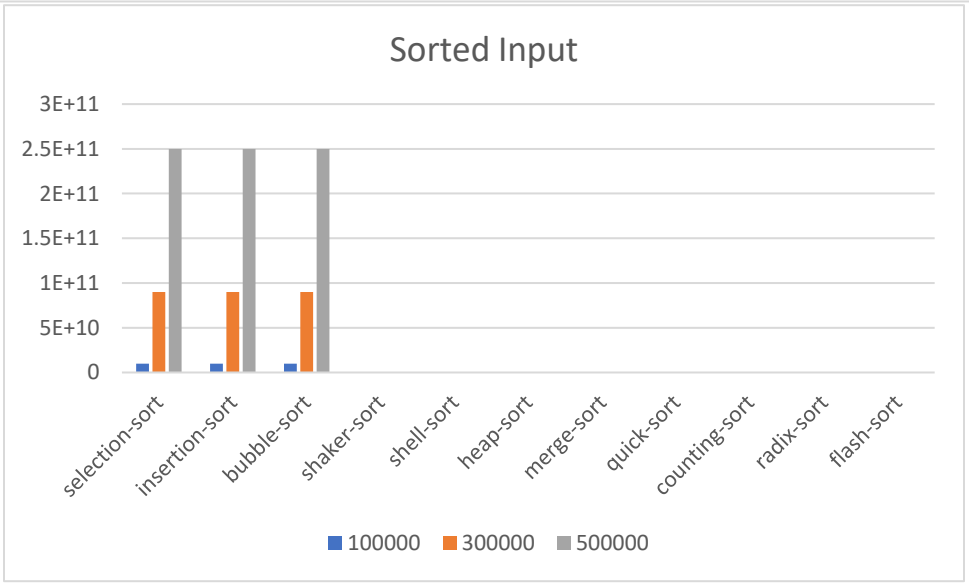
Tại kiểu dữ liệu gần như được xếp: NEARLY SORTED INPUT, Insertion Sort là thuật toán chạy chậm nhất và Counting Sort chạy nhanh nhất. Insertion Sort do phải chèn lần lượt từng phần tử nên tốc độ chậm

hơn so với Selection Sort và Bubble Sort. Shaker Sort ở kiểu dữ liệu này chạy nhanh hơn vì có biến kiểm tra mảng đã sắp hay chưa. Vì là kiểu dữ liệu gần như được sắp nên khi vừa sắp xếp xong thuật toán sẽ dừng ngay. Tuy đã cải thiện tốc độ nhưng về số lần sắp xếp cũng như tốc độ vẫn đứng sau Counting Sort.

3. Dữ liệu Sorted

Data order: SORTED												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
selection-sort	0.11	100009999	1.14	900029999	2.89	2500049999	11.51	10000099999	91.45	90000299999	315.39	2.5E+11
insertion-sort	0.11	100009999	1.25	900029999	3.45	2500049999	9.93	10000099999	92.13	90000299999	240.59	2.5E+11
bubble-sort	0.14	100009999	1.21	900029999	2.6	2500049999	10.59	10000099999	85.6	90000299999	241.35	2.5E+11
shaker-sort	0	19999	0	59999	0	99999	0	199999	0	599999	0	999999
shell-sort	0	240037	0	780043	0	1400043	0.01	3000045	0.01	10200053	0.02	17000051
heap-sort	0	288913	0.02	955201	0.01	1671609	0.02	3551709	0.07	11572321	0.11	19961401
merge-sort	0	406234	0	1332186	0.01	2320874	0.02	4891754	0.04	15848682	0.06	27234634
quick-sort	0	149055	0	485546	0	881083	0	1862156	0.01	5889300	0.02	10048590
counting-sort	0	70004	0	210004	0	350004	0	700004	0	2100004	0	3500004
radix-sort	0	140058	0	510072	0	850072	0.01	1700072	0.02	6000086	0.04	10000086
flash-sort	0	125797	0	377397	0	628997	0	1257997	0.01	3773997	0.01	6289997

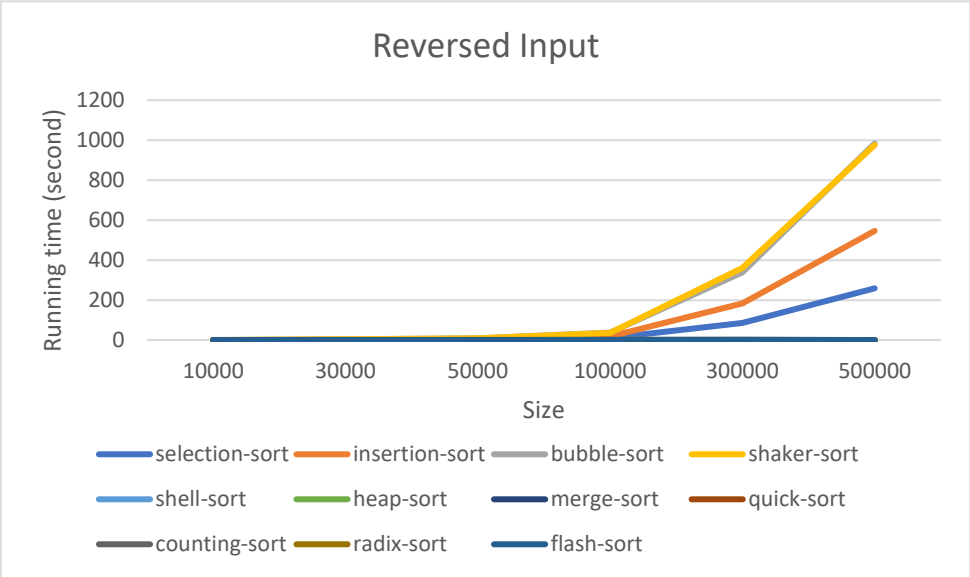


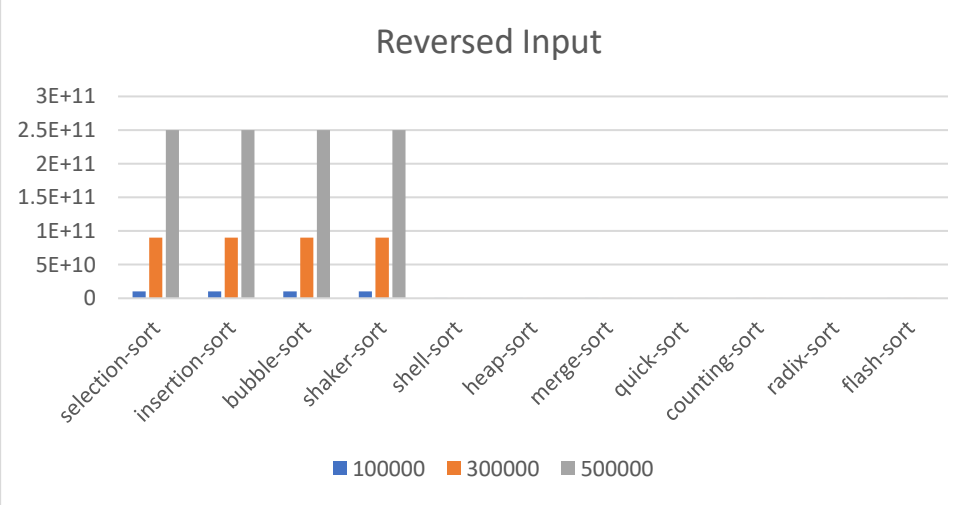


Ở kiểu dữ liệu đã được sắp: Hàm có tốc độ chạy nhanh nhất là Shaker Sort, vì nó chỉ cần duyệt qua mảng 1 lần là đã dừng. Hàm chậm nhất là Selection Sort. 2 hàm Insertion Sort và Bubble Sort có thời gian chạy chậm thứ 2. Ở mức 300,000 thì có vẻ 3 thuật toán chạy bằng nhau nhưng khi tới 500,000 thì đã có sự khác biệt. Về số lần so sánh thì Shaker Sort cũng đứng đầu. So sánh nhiều nhất vẫn là 3 thuật toán đầu.

4. Dữ liệu Reversed

Data order: REVERSED												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
selection-sort	0.09	100009999	0.85	900029999	2.35	2500049999	9.62	10000099999	86.17	90000299999	258.93	2.5E+11
insertion-sort	0.11	100009999	0.97	900029999	4.98	2500049999	18.74	10000099999	183.91	90000299999	546.17	2.5E+11
bubble-sort	0.32	100009999	3.33	900029999	9.05	2500049999	37.2	10000099999	338.34	90000299999	983.84	2.5E+11
shaker-sort	0.32	100000000	3.32	900000000	8.84	2500000000	35.48	10000000000	360.87	90000000000	976	2.5E+11
shell-sort	0	302597	0	987035	0	1797323	0.01	3844605	0.02	12700933	0.04	21428803
heap-sort	0	258393	0.01	873425	0.01	1522785	0.02	3244869	0.06	10702493	0.15	18586901
merge-sort	0	401834	0	1323962	0	2301434	0.01	4852874	0.04	15729866	0.23	27143914
quick-sort	0	159070	0	515556	0	931094	0.01	1962168	0.01	6189320	0.02	10548604
counting-sort	0	70004	0	210004	0	350004	0	700004	0	2100004	0.01	3500004
radix-sort	0	140058	0	510072	0.01	850072	0.01	1700072	0.02	6000086	0.13	10000086
flash-sort	0	108600	0	325800	0	543000	0	1086000	0.01	3258000	0.02	5430000





Ở kiểu dữ liệu REVERSED, 2 thuật toán Bubble Sort và Shaker sort đều chạy chậm nhất, vì với điều kiện arr[k] > arr[k+1] luôn đúng nên hàm swap sẽ được thực hiện khoảng n^2 lần dẫn đến 2 hàm này chạy rất chậm. Thứ 2 là Insertion Sort vì nó thực hiện chèn trên một quãng đường lớn. Như thường lệ thì hàm chạy nhanh nhất vẫn là Counting Sort, nhanh thứ 2 lầ Flash Sort. Về số lượng phép so sánh: Shaker Sort trong trường hợp này đã có số phép so sánh bằng với 3 thuật toán đầu tiên. Chỉ có 2 thuật toán có dưới 10,000,000 phép so sánh trong trường hợp này là Counting Sort và Flash Sort.

IV. Tổ chức chương trình

Chia file:

- constant.h: chưa mảng kiểu string gồm tên của cách lệnh sort, tên kiểu data, size.

- sort.h, sort.cpp: chứa thông tin về các hàm sort dùng nghiên cứu.

- sort_with_count.cpp/.h: chứa hàm sort có thêm biến đếm phép so sánh.

- func.cpp/.h: chứa các hàm hỗ trợ trong việc tính toán và xuất ra console.

- DataGenerator.cpp/.h: chứa các hàm tạo dữ liệu.

- main.cpp: hàm main.

Thư viện:

- <time.h> để tính thời gian

- <fstream> để xuất nhập file

Và một số thư viện phổ biến khác.

V. Nguồn tham khảo

- https://www.geeksforgeeks.org/selection-sort/

- https://www.geeksforgeeks.org/bubble-sort/

- https://www.geeksforgeeks.org/insertion-sort/

- https://www.geeksforgeeks.org/cocktail-sort/

- https://www.geeksforgeeks.org/shellsort/

- https://www.geeksforgeeks.org/heap-sort/?ref=lbp

- https://nguyenvanhieu.vn/thuat-toan-sap-xep-merge-sort/

- https://www.geeksforgeeks.org/radix-sort/

- Flash Sort: Video record cấu trúc dữ liệu và giải thuật P2 5/11/2021 – thầy Nguyễn Thanh Phương