TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ

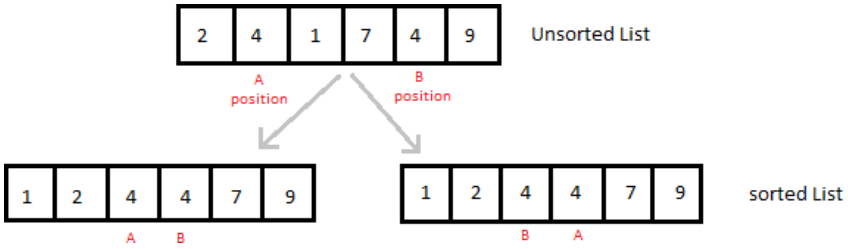
NHIÊN KHOA TOÁN – TIN

_____oOo _____

BÀI BÁO CÁO

CÁC THUẬT TOÁN SẮP XẾP VÀ TÌM KIẾM

Môn học: Thực hành cấu trúc dữ liệu và giải thuật



Giảng viên: Nguyễn Bảo Long

Sinh viên: Nguyễn Ngọc Thành – 21280108

10, 2022

Lời nói đầu

Đồ án thực hiện giải thuật 12 thuật toán sắp xếp (**theo thứ tự tăng dần**) và 2 thuật toán tìm kiếm dựa trên C++ và Python. Thử với các dãy dữ liệu [1000, 3000, 10000, 30000, 100000] được sắp xếp tăng dần, giảm dần và ngẫu nhiên. Sau đó biểu diễn thời gian chạy trên biểu đồ đường và xuất các file txt kết quả chạy của từng thuật toán.

Bài báo cáo sẽ trình bày cách xây dựng các thuật toán sắp xếp và tìm kiếm. Đánh giá độ phức tạp và bộ nhớ sử dụng, đồng thời phân tích giải thuật và cách thuật toán hoạt động.

Sau phần giải thích và đánh giá các thuật toán là phần cài đặt 12 thuật toán bằng python và chạy thử với các chuỗi dữ liệu đã nêu trong phần trước. Do máy yếu nên em chỉ chạy tới chuỗi 10000. Em sẽ đính kèm file report.ipynb để thầy có thể chạy nốt các chuỗi còn lại (uncomment phần em đã comment) cùng với các file ./result/*.txt và Runtime.csv. Ngoài ra phần quick sort do máy lỗi nên em không chạy được với các mảng lớn nên cũng đã comment lại. Các thuật toán tìm kiếm sẽ được trình bày sau phần sắp xếp.

MỤC LỤC

ĐÁNH GIÁ BÀI BÁO CÁO

Mặc dù có thể sẽ có vài thiếu sót nhưng em đánh giá sau chuỗi ngày bạc tóc đau lưng một mình làm chiếc đồ án này thì em tự chấm mình 11 điểm.

Về thuật toán:
Đã cài đặt thành công 12 thuật toán sắp xếp bằng code python và c++, 2 thuật toán tìm kiếm bằng c++. Vẽ đồ thị bằng matplotlib ba kiểu dữ liệu: ngẫu nhiên, sắp xếp từ bé đến lớn, sắp xếp từ lớn đến bé.
Tìm hiểu và trình bày ý tưởng, nêu ví dụ với mảng a = [86, 4, 6905, 43, 7, 659] và đưa ra nhận xét đánh giá từng thuật toán

Ngoài ra xin gửi lời cảm ơn đến bạn Nguyễn Đặng Anh Thư - 21280111 đã chạy giúp hai input size 30000 và 100000 hết 5 tiếng đồng hồ.

CÁC THUẬT TOÁN TÌM KIẾM

1. Linear search.

Ý tưởng:

Tìm kiếm tuần tự duyệt từ phần tử đầu mảng và so sánh với từng phần tử trong mảng cho đến khi có phần tử bằng thì trả về vị trí của phần tử đó trong mảng hoặc trả về -1 nếu duyệt hết mảng mà không có phần tử nào bằng.

Ví dụ:

Cho mảng a = [86, 4, 6905, 43, 7, 659]
Cần tìm phần tử 43
| 86 | 4 | 6905 | 43 | 7 | 659 | (86 != 43), i=0
| 86 | 4 | 6905 | 43 | 7 | 659 | (4 != 43), i=1
| 86 | 4 | 6905 | 43 | 7 | 659 | (6905 != 43), i=2
| 86 | 4 | 6905 | 43 | 7 | 659 | (43 == 43), i=3
⇒ trả về i = 3

Nhận xét:

Ta thấy, về ý tưởng thuật toán tìm kiếm tuần tự rất đơn giản nhưng chi phí thời gian lại có thể khá cao khi phần tử cần tìm nằm ở cuối mảng.

- Trường hợp tốt: O(0).
- Trung bình: O(n/2).
- Trường hợp xấu: O(n).
- Bộ nhớ: O(1).

2. Binary search.

Ý tưởng:

Tìm kiếm phần tử từ một mảng đã sắp xếp bằng cách duyệt mảng từ giữa và so sánh nếu phần tử đó lớn hơn thì duyệt từ giữa bên phải và ngược lại cho đến khi tìm được vị trí phần tử cần tìm.

Ví dụ:

Cho mảng a = [4, 7, 43, 86, 659, 6905]
Cần tìm phần tử 7
| 4 | 7 | 43 | 86 | 659 | 6905 | (86 != 7) & (86 > 7) ⇒ trái
| 4 | 7 | 43 | 86 | 659 | 6905 | (7 == 7) i = 1
⇒ trả về i = 1

Nhận xét:

Ta thấy, so với tìm kiếm tuần tự mỗi lần duyệt mảng ta đã có thể loại trừ được phân nửa độ dài của mảng điều này khiến thời gian duyệt trung bình của tìm kiếm nhị phân nhanh hơn gấp đôi tìm kiếm tuần tự. Tuy nhiên một nhược điểm của tìm kiếm nhị phân là mảng đầu vào là mảng đã sắp xếp tăng dần.

- Trường hợp tốt: O(0).
- Trung bình: O(n/4).
- Trường hợp xấu: O(n/2).
- Bộ nhớ: O(1).

CÁC THUẬT TOÁN SẮP XẾP

1. Insertion sort.

Ý tưởng:

Sắp xếp lấy ý tưởng từ việc chơi bài, dựa theo cách người chơi "chèn" thêm một quân bài mới vào một bộ bài đã được sắp xếp trên tay.

Thuật toán:

- Duyệt và tìm phần tử bé hơn phần tử đằng trước nó.
- Đổi chỗ cho phần tử được xác định là lớn hơn.
- Lặp lại bước này và có được một mảng đã được sắp xếp.

Ví dụ:

Cho mảng a = [86, 4, 6905, 43, 7, 659]

Duyệt mảng:

| 86 | 4 | 6905 | 43 | 7 | 659 |

| 86 | 4 | 6905 | 43 | 7 | 659 | ⇒ swap(86, 4)

| 4 | 86 | 6905 | 43 | 7 | 659 | ⇒ swap(86, 43)

| 4 | 43 | 86 | 6905 | 7 | 659 | ⇒ swap(43, 7)

| 4 | 7 | 43 | 86 | 6905 | 659 | ⇒ swap(6905, 659)

| 4 | 7 | 43 | 86 | 659 | 6905 |

⇒ mảng đã được sắp xếp.

Nhận xét:

Ta thấy, ở trường hợp tốt nhất thì dãy đã được sắp xếp thì ta cần tổng cộng O(n) phép so sánh và phép gán. Còn trong trường hợp xấu nhất là dãy sắp xếp từ lớn đến bé thì ta phải duyệt qua O(n^2) phép so sánh và gán. Như vậy ta có kết luận.

- Trường hợp tốt: O(n).
- Trung bình: O(n^2).
- Trường hợp xấu: O(n^2).
- Bộ nhớ: O(1).
- Tính ổn định: Có.

2. Bubble sort.

Ý tưởng:

Thuật toán sắp xếp nổi bọt sẽ đẩy phần tử lớn nhất xuống cuối mảng, đồng thời những phần tử nhỏ hơn sẽ dịch chuyển dần về đầu mảng.

Thuật toán:

- 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 cho đến khi dãy số được sắp xếp.
- Duyệt từ đầu mảng đến cuối mảng và quay lại.

Ví dụ:

Cho mảng a = [86, 4, 6905, 43, 7, 659]

Duyệt mảng:

| 86 | 4 | 6905 | 43 | 7 | 659 | ⇒ swap(86, 4)

| 4 | 86 | 6905 | 43 | 7 | 659 | ⇒ swap(6905, 43)

| 4 | 86 | 43 | 6905 | 7 | 659 | ⇒ swap(6905, 7)

| 4 | 86 | 43 | 7 | 6905 | 659 | ⇒ swap(6905, 659)

| 4 | 86 | 43 | 7 | 659 | 6905 | ⇒ swap(43, 86)

| 4 | 43 | 86 | 7 | 659 | 6905 | ⇒ swap(86, 7)

| 4 | 43 | 7 | 86 | 659 | 6905 | ⇒ swap(86, 7)

| 4 | 43 | 7 | 86 | 659 | 6905 | ⇒ swap(43, 7)

| 4 | 7 | 43 | 86 | 659 | 6905 | ⇒ loop()

| 4 | 7 | 43 | 86 | 659 | 6905 |

⇒ mảng đã được sắp xếp.

Nhận xét:

Ta thấy, số phép so sánh cần thực hiện ở lần lặp đầu tiên là n - 1 sau đó giảm đi thêm 1 lần ở mỗi lần lặp sau. Vì vậy trong trường hợp xấu nhất, số phép hoán vị sẽ bằng số phép so sánh. Còn trong trường hợp tốt nhất là không có phép hoán vị khi mảng đã được sắp xếp.Từ đó ta có kết luận.

- Trường hợp tốt: O(n).
- Trung bình: O(n^2).
- Trường hợp xấu: O(n^2).
- Bộ nhớ: O(1).
- Tính ổn định: Có.

3. Selection sort.

Ý tưởng:

Thuật toán sắp xếp chọn sắp xếp một mảng bằng cách đi tìm phần tử có giá trị nhỏ nhất trong mảng và đổi chỗ nó cho phần tử đầu tiên chưa đúng vị trí. Thuật toán sẽ chia làm 2 mảng con.

Thuật toán:

- Tìm phần tử nhỏ nhất trong đoạn chưa được sắp xếp.
- Đổi chỗ với phần tử đầu tiên chưa đúng vị trí.
- Lặp lại cho đến khi mảng đã được sắp xếp

Ví dụ:

Cho mảng a = [86, 4, 6905, 43, 7, 659]

Duyệt mảng:

| 86 | 4 | 6905 | 43 | 7 | 659 | ⇒ min = 4

| 86 | 4 | 6905 | 43 | 7 | 659 | ⇒ swap(86, 4)

| 4 | 86 | 6905 | 43 | 7 | 659 | ⇒ min = 7

| 4 | 86 | 6905 | 43 | 7 | 659 | ⇒ swap(86, 7)

| 4 | 7 | 6905 | 43 | 86 | 659 | ⇒ min = 43

| 4 | 7 | 43 | 6905 | 86 | 659 | ⇒ min = 86

| 4 | 7 | 43 | 6905 | 86 | 659 | ⇒ swap(86, 6905)

| 4 | 7 | 43 | 86 | 6925 | 659 | ⇒ min = 658

| 4 | 7 | 43 | 86 | 6925 | 659 | ⇒ swap(6925, 659)

| 4 | 7 | 43 | 86 | 659 | 6905 |

⇒ mảng đã được sắp xếp.

Nhận xét:

Ta thấy, sau mỗi lần swap, số phép so sánh giảm dần còn:

(n - 1) + (n - 2) + (n - 3) + ··· + 2 + 1 = [n * (n - 1)] / 2

⇒ Số phép so sánh và hoán vị cần thực hiện là O(n^2) trong đó trường hợp tốt là O(n) với mảng đã được sắp xếp sẵn với n lần so sánh. Từ đó ta có kết luận.

- Trường hợp tốt: O(n^2).
- Trung bình: O(n^2).
- Trường hợp xấu: O(n^2).
- Bộ nhớ: O(1).

- Tính ổn định: Không.

4. Heap sort.

Ý tưởng:

Ban đầu ta xây dựng cây nhị phân cho mảng cần sắp xếp với quy tắc một node có hai nhánh, nhánh bên phải lớn hơn nhánh cha, nhánh trái nhỏ hơn nhánh con. Thuật toán sẽ đưa phần tử lớn nhất lên phần tử root và xếp nó vào mảng khác.

Thuật toán:

- Xây dựng cây nhị phân.
- Duyệt cây nhị phân và tìm phần tử lớn nhất đẩy lên làm phần tử gốc.
- Bốc phần tử gốc ra mảng khác.
- Lặp lại các bước trên và có được mảng đã sắp xếp.

Ví dụ:

Cho mảng a = [86, 4, 6905, 43, 7, 659]

Xây dựng cây nhị phân lần 1:

86
/ \
4 6905
/ \ \
43 7 659

Heapify 1: swap(43, 4)

86
/ \
43 6905
/ \ \
4 7 659

Heapify 2: swap(86, 6905)

6905
/ \
43 86
/ \ \
4 7 659

Heapify 3: swap(86, 659)

6905
/ \
43 695
/ \ \
4 7 86

arr = | 6905 |

Xây dựng cây nhị phân lần 2:

86

/ \

43 659

/ \

4 7

Xây dựng cây nhị phân lần 3:

86

/

43

/ \

4 7

arr = | 86 | 659 | 6905 |

Heapify 4: swap(86, 659)

659

/ \

43 86

/ \

4 7

arr = | 659 | 6905 |

Xây dựng cây nhị phân lần 4:

43

/ \

4 7

arr = |4|7|43|86|659|6950|

⇒ mảng đã được sắp xếp.

Nhận xét:

Độ phức tạp của heap sort trong xây dựng cây nhị phân sẽ là O(n), trong khi đẩy phần tử max lên root sẽ là O(nlogn). Vì vậy, trong trường hợp tốt nhất thuật toán sẽ có độ phức tạp là O(n) và xấu nhất là O(nlogn).

- Trường hợp tốt: O(n).
- Trung bình: O(nlogn).
- Trường hợp xấu: O(nlogn).
- Bộ nhớ: O(1).
- Tính ổn định: Không.

5. Merge sort.

Ý tưởng:

Sắp xếp trộn là thuật toán dựa trên so sánh và phương pháp chia để trị của John Von Neumann.

- Chia (divide): chia dãy gồm n phần tử cần sắp xếp thành 2 dãy, mỗi dãy có n/2 phần tử.
- Trị (conquer): sắp xếp mỗi dãy con một cách đệ quy sử dụng sắp xếp trộn. Khi dãy chỉ còn một phần tử thì trả lại phần tử này.
- Tổ hợp (combine): trộn (merge) hai dãy con được sắp xếp để thu được dãy đã sắp xếp gồm tất cả các phần tử của cả hai dãy con.

Thuật toán:

- Chia mảng ra làm hai phần bằng nhau (đối với mảng chẵn) và lệch nhau một với mảng lẻ.
- So sánh hai phần tử đầu ở hai mảng và, lấy phần tử nhỏ hơn cho vào dãy mới. Tiếp tục như vậy cho tới khi một trong hai dãy rỗng.
- Khi một trong hai dãy rỗng, ta lấy phần còn lại của dãy kia cho vào cuối dãy mới. Khi đó, ta sẽ thu được dãy cần tìm.

Ví dụ:

Cho mảng a = [86, 4, 6905, 43, 7, 659]

| 86 | 4 | 6905 | 43 | 7 | 659 | = | 86 | 4 | 6905 | + | 43 | 7 | 659 |

Mảng con 1:

| 86 | 4 | 6905 | = | 86 | 4 | + | 6905 |

| 86 | 4 | & | 6905 | : (4 < 6905) ⇒ | 4 |

| 86 | & | 6905 | : (86 < 6905) ⇒ | 4 | 86 | 695 |

Mảng con 2:

| 43 | 7 | 659 | = | 43 | 7 | 659 |

| 43 | 7 | & | 659 | : (7 < 659) ⇒ | 7 |

| 43 | & | 659 | : (43 < 695) ⇒ | 7 | 43 | 695 |

Gộp mảng và làm tương tự như trên:

| 7 | 43 | 659 | & | 4 | 86 | 6905 | : (4 < 7) ⇒ | 4 |

| 7 | 43 | 659 | & | 86 | 6905 | : (7 < 86) ⇒ | 4 | 7 |

| 43 | 659 | & | 86 | 6905 | : (43 < 86) ⇒ | 4 | 7 | 43 |

| 659 | & | 86 | 6905 | : (86 < 659) ⇒ | 4 | 7 | 43 | 86 |

| 659 | & | 6905 | : (659 < 6905) ⇒ | 4 | 7 | 43 | 86 | 659 | 6905 |

⇒ mảng đã được sắp xếp.

Nhận xét:

Độ phức tạp thuật toán của sắp xếp trộn là O(nlogn) ở mọi trường hợp, ta chứng minh như sau:

Gọi T(n) là thời gian thực thi của thuật toán sắp xếp trộn cho mảng có n phần tử.

⇒ Ta có hệ thức đệ quy sau: T(n) = 2 * T(n/2) + n. Vì ta cần giải bài toán con cho 2 mảng con với độ dài n/2, và tốn n cho phép trộn 2 mảng con.

T(n) = 4*T(n/4) + n + (n/2)*2 = 2^logn * T(n/2^logn) + nlogn = n + nlogn = O(nlogn).

Khi đó bộ nhớ cần dùng sẽ là O(n) với n độ dài mảng và logn tầng đệ quy.

- Trường hợp tốt: O(nlogn).
- Trung bình: O(nlogn).
- Trường hợp xấu: O(nlogn).
- Bộ nhớ: O(n).
- Tính ổn định: Có.

6. Shell sort.

Ý tưởng:

Sắp xếp vỏ sò tượng tự như sắp xếp chèn nhưng trước khi sử dụng sắp xếp chèn chúng ta sẽ xét trên các khoảng và đổi chỗ các nghịch thế. điều này giúp làm giảm việc phải so sánh quá nhiều phần tử như sắp xếp chèn.

Thuật toán:

- Ban đầu chúng ta lấy gap = n/2 và và đổi chỗ các nghịch thế cách nhau gap phần tử.
- Tiếp tục giảm gap đi một nửa là lặp lại bước trên cho đến khi gap = 1
- Đến lúc gap = 1 thì tương đương là sắp xếp chèn.

Ví dụ:

Cho mảng a = [86, 4, 6905, 43, 7, 659]

gap = 3: | 86 | 4 | 6905 | 43 | 7 | 659 | ⇒ swap(86, 43)

| 43 | 4 | 6905 | 86 | 7 | 659 |

| 43 | 4 | 6905 | 86 | 7 | 659 | ⇒ swap(6905, 659)

| 43 | 4 | 659 | 86 | 7 | 6905 |

gap = 2: | 43 | 4 | 659 | 86 | 7 | 6905 |

| 43 | 4 | 659 | 86 | 7 | 6905 |

| 43 | 4 | 659 | 86 | 7 | 6905 | ⇒ swap(659, 7)

| 43 | 4 | 7 | 86 | 659 | 6905 |

| 43 | 4 | 7 | 86 | 659 | 6905 |

gap = 1: | 43 | 4 | 7 | 86 | 659 | 6905 | ⇒ swap(43, 4) ⇒ swap(43, 7)

| 4 | 7 | 43 | 86 | 659 | 6905 |

⇒ mảng đã được sắp xếp.

Nhận xét:

Độ phức tạp của sắp xếp vỏ sò là chưa xác định được vì còn phụ thuộc vào cách chọn gap. Trong trường hợp chọn gap như trên thì bộ nhớ sẽ là O(1+m) với m là độ dài gap.

Khi danh sách mảng đã cho đã được sắp xếp, tổng số phép so sánh của mỗi khoảng bằng kích thước của mảng đã cho. Vì vậy, độ phức tạp của trường hợp tốt nhất là Ω (nlogn).

Trường hợp xấu nhất là O(n^2) khi mảng sắp xếp từ lớn đến bé, khi này mọi gap sẽ phải hoán vị và duyệt n lần.

- Trường hợp tốt: O(nlogn).
- Trung bình: phụ thuộc.
- Trường hợp xấu: O(n^2).
- Bộ nhớ: O(1+gap).
- Tính ổn định: Không.

7. Quick sort.

Ý tưởng:

Quick sort là thuật toán sắp xếp chia để trị tương tự như merge sort. Với mảng dưới hai phần tử ta sẽ thoát vòng đệ quy. Ngược lại, ta chọn phần tử chốt và chuyển tất cả các phần tử lớn hơn vứt sang bên phải, các phần tử bé hơn vứt sang bên trái.

Thuật toán:

- Chọn phần tử làm phần tử chốt ví dụ chọn phần tử cuối mảng.
- Duyệt qua cả mảng và so sánh từng phần tử với phần tử chốt, nếu nhỏ hơn quăng qua bên trái, lớn hơn quăng qua bên phải.
- Đệ quy hai mảng hai phần tử chốt lặp lại các bước trên và thu được mảng đã sắp xếp.

Ví dụ:

Cho mảng a = [86, 4, 6905, 43, 7, 659]

| 86 | 4 | 6905 | 43 | 7 | 659 |: chọn pivot = 659

Duyệt mảng từ trái qua phải so sánh từng phần tử với pivot, phần tử nào lớn hơn để sang bên trái pivot và ngược lại:

⇒ Mảng mới: | 86 | 4 | 43 | 7 | 659 | 6905 |

Chia mảng làm hai từ vị trí pivot tiếp tục thực hiện bước trên cho hai mảng con:

| 86 | 4 | 43 | 7 | 659 | 6905 | = | 86 | 4 | 43 | 7 | + | 659 | 6905 |

| 86 | 4 | 43 | 7 |: chọn pivot = 7

Duyệt mảng từ trái qua phải so sánh từng phần tử với pivot, phần tử nào lớn hơn để sang bên trái pivot và ngược lại:

⇒ Mảng mới: | 4 | 7 | 86 | 43 |

Chia mảng làm hai từ vị trí pivot tiếp tục thực hiện bước trên cho hai mảng con:

| 4 | 7 | 86 | 43 | = | 4 | 7 | + | 86 | 43 |

| 86 | 43 |: tương tự.

Gộp các mảng lại ⇒ | 4 | 7 | 43 | 86 | 659 | 6905 |

⇒ mảng được sắp xếp

Nhận xét:

Độ phức tạp của quick sort phụ thuộc vào việc chọn phần tử chốt, trong trường hợp tốt nhất khi phần tử chốt chia mảng ra làm hai mảng có độ dài gần bằng nhau thì độ phức tạp của quick sort sẽ là O(nlogn), chứng minh tương tự merge sort. Trong trường hợp xấu nhất độ phức tạp của thuật toán sẽ là O(n^2) khi việc chọn phần tử chốt liên tục chia làm hai dãy có độ dài là 1 và n-1. Khi đó bộ nhớ cần dùng sẽ là O(n) còn trong trường hợp trung bình sẽ là O(logn).

- Trường hợp tốt: O(nlogn).
- Trung bình: O(nlogn).
- Trường hợp xấu: O(n^2).
- Bộ nhớ: O(logn).
- Tính ổn định: Không.

8. Shaker sort.

Ý tưởng:

Shaker sort là thuật toán cải tiến của bubble sort khi đồng thời đẩy phần tử lớn nhất về cuối mảng và phần tử nhỏ nhất lên đầu mảng.

Thuật toán:

- Duyệt mảng từ đầu mảng đến cuối mảng và đổi chỗ các nghịch thế.

- Duyệt mảng từ cuối mảng lại đầu mảng và đổi chỗ các nghịch thế.
- Lặp lại hai bước trên và thu được mảng đã sắp xếp.

Ví dụ:

Cho mảng a = [86, 4, 6905, 43, 7, 659]
Duyệt từ trái sang:
| 86 | 4 | 6905 | 43 | 7 | 659 | ⇒ swap(86, 4)
| 4 | 86 | 6905 | 43 | 7 | 659 | ⇒ swap(6905, 43)
| 4 | 86 | 43 | 6905 | 7 | 659 | ⇒ swap(6905, 7)
| 4 | 86 | 43 | 7 | 6905 | 659 | ⇒ swap(6905, 659)
Duyệt từ phải sang:
| 4 | 43 | 86 | 7 | 659 | 6905 | ⇒ swap(7, 86)
| 4 | 43 | 7 | 86 | 659 | 6905 | ⇒ swap(7, 43)
| 4 | 7 | 43 | 86 | 659 | 6905 |
⇒ mảng đã được sắp xếp.

Nhận xét:

Độ phức tạp của shaker sort bằng độ phức tạp của bubble sort chia hai vì shaker sort duyệt mảng cả đi lẫn về trong khi bubble sort chỉ duyệt mảng đi.

- Trường hợp tốt: O(n/2).
- Trung bình: O(n^2/2).
- Trường hợp xấu: O(n^2/2).
- Bộ nhớ: O(1).
- Tính ổn định: Có.

9. Radix sort.

Ý tưởng:

Radix sort là thuật toán sắp xếp không dựa vào so sánh mà vào việc phân loại lần lượt các chữ số: hàng đơn vị, hàng chục, hàng trăm,...
Thuật toán:

- Đầu tiên chia mảng vào các nhóm có cùng hàng đơn vị, sắp xếp lại mảng theo thứ tự từ 0 đến 9.
- Tiếp theo chia mảng vào các nhóm có cùng hàng chục, sắp xếp lại mảng theo thứ tự từ 0 đến 9.

- Lặp lại tương tự cho hàng trăm, hàng nghìn,... và ta có được mảng đã sắp xếp.

Ví dụ:

Cho mảng a = [86, 4, 6905, 43, 7, 659]
Ban đầu chúng ta xếp thành các nhóm theo hàng đơn vị:
Nhóm 3: | 43 |
Nhóm 4: | 4 |
Nhóm 5: | 6905 |
Nhóm 6: | 86 |
Nhóm 7: | 7 |
Nhóm 9: | 659 |
Xếp lại mảng theo các nhóm: | 43 | 4 | 6905 | 86 | 7 | 659 |
Tiếp theo chúng ta xếp thành các nhóm theo hàng chục:
Nhóm 0: | 6905 | 4 | 7 |
Nhóm 4: | 43 |
Nhóm 5: | 659 |
Nhóm 8: | 86 |
Xếp lại mảng theo các nhóm: | 6905 | 4 | 7 | 43 | 659 | 86 |
Tiếp theo chúng ta xếp thành các nhóm theo hàng trăm:
Nhóm 0: | 4 | 7 | 43 | 86 |
Nhóm 6: | 659 |
Nhóm 9: | 6905 |
Xếp lại mảng theo các nhóm: | 4 | 7 | 43 | 86 | 659 | 6905 |
⇒ Như vậy đến đây chúng ta đã có được mảng đã sắp xếp, nếu có phần tử lớn hơn chúng ta cứ tiếp tục xếp nhóm như trên.

Nhận xét:

Độ phức tạp của radix sort là O(nlogk) với n là độ dài mảng và k là số lớn nhất trong mảng. Vì ta cần gọi k lần đệ quy và xử lý n phần tử, khi đó trường hợp tốt nhất sẽ là O(0) khi k = 1. Về bộ nhớ cần dùng thì sẽ là O(logk). Vì thế chúng ta có kết luận.

- Trường hợp tốt: O(0).
- Trung bình: O(nlogn).
- Trường hợp xấu: O(nlogn).
- Bộ nhớ: O(logn).

- Tính ổn định: Không.

10. Flash sort.

Ý tưởng:

- Flash sort là thuật toán áp dụng bucket sort đưa về mảng gần như đã sắp xếp và sử dụng insertion sort để hoàn thành việc sắp xếp.
- Thuật toán:
- Tìm giá trị nhỏ nhất của các phần tử trong mảng(min) và vị trí phần tử lớn nhất của các phần tử trong mảng(max).
- Khởi tạo 1 vector L có m phần tử (ứng với m lớp, chọn số lớp bằng 0.45n).
- Đếm số lượng phần tử các lớp theo quy luật, phần tử a[i] sẽ thuộc lớp k = int((m - 1) * (a[i]-min) / (a[max] - min)).
- 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.
- 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.

Ví dụ:

Cho mảng a = [86, 4, 6905, 43, 7, 659]
Cho số phân đoạn bằng 3
Như vậy ta có phần tử và phân đoạn như sau:

Phần tử	Phân đoạn
86	2
4	1
6905	3
43	2
7	1

659	3

Mảng sau khi phân hoạch: | 4 | 7 | 86 | 43 | 6905 | 659 |
Sau đó chúng ta sử dụng sắp xếp chèn từng phân đoạn và thu được mảng đã sắp xếp: | 4 | 7 | 43 | 86 | 659 | 6905 |

Nhận xét:

Độ phức tạp của thuật toán flash sort:

- Giai đoạn phân hạch có độ phức tạp là O(n) vì mỗi phần tử t chỉ duyệt qua một lần.
- Giai đoạn sắp xếp ta sẽ có độ phức tạp là O(n).
- Trong trường hợp tệ nhất, nếu dữ liệu phân bố k đều thì độ phức tạp có thể lên tới O(n^2).
- Bộ nhớ cần dùng là O(m) = O(n).

Ta có kết luận:

- Trường hợp tốt: O(n).
- Trung bình: O(n).
- Trường hợp xấu: O(n^2).
- Bộ nhớ: O(n).
- Tính ổn định: Không.

Tính ổn định: Không.

11. Count sort.

Ý tưởng:

Count sort là thuật toán sắp xếp cực nhanh mang các phần tử là số nguyên không âm hoặc một danh sách các kí tự được ánh xạ theo dạng số để sắp xếp theo bảng chữ cái.

Thuật toán:

- Lấy phần tử lớn nhất trong mảng là max.
- Tạo mảng mới có độ dài là max + 1.
- Lưu trữ số lượng phần tử tương ứng với chỉ số của mảng mới.
- Sửa đổi mảng đếm bằng cách thêm các số trước đó và tạo thành tổng tích lũy của của một mảng.

- Tạo mảng trống khác có số phần tử bằng mảng cần sắp xếp và đặt các phần từ vào đúng vị trí của nó và giảm số lượng đã đếm đi một.
- Lặp lại bước liền trên và thu được mảng đã được sắp xếp.

Ví dụ:

Do để thuận tiện cho việc chạy tay chúng ta sẽ sử dụng mảng có phần tử max không quá lớn so với số phần tử.
Cho mảng a = [3, 5, 1, 6, 7, 3]
Ban đầu ta tìm max của mảng: max = 7
Chúng ta sẽ khởi tạo mảng mới có đô dài bằng max + 1 với tất cả phần tử bằng 0.
b = [0, 0, 0, 0, 0, 0, 0, 0]
Sau đó chúng ta sẽ đếm các phần tử của mảng a và lưu trữ vào vị trí bằng phần tử đó vào mảng b.
b = [0, 1, 0, 2, 0, 1, 1, 1]
Bây giờ, ta sẽ sửa đổi mảng đếm bằng cách cộng thêm các số trước đó và tạo tổng tích lũy của một mảng
b = [0, 1, 1, 3, 3, 4, 5, 6]
Vì có 6 đầu vào trong mảng ban đầu, ta sẽ tạo một mảng trống khác có 7 vị trí để lưu trữ dữ liệu đã sắp xếp và đặt các phần tử vào đúng vị trí của chúng và giảm số lượng đi 1, lặp lại bước này và thu được mảng đã sắp xếp.
| 4 | 7 | 43 | 86 | 659 | 6905 | ⇒ mảng đã được sắp xếp

Nhận xét:

Độ phức tạp của count sort phụ thuộc vào mảng đầu vào là O(n + k) với n là độ dài mảng và k là chênh lệch giữa phần tử lớn nhất của mảng và số phần từ trong mảng. Bộ nhớ cần dùng là O(k) với k < n.

- Trường hợp tốt: O(n).
- Trung bình: O(n).
- Trường hợp xấu: O(n).
- Bộ nhớ: O(n).
- Tính ổn định: Có.

12. Binary Insertion sort.

Ý tưởng:

Sắp xếp chèn nhị phân tương tự như sắp xếp chèn nhưng thay vì tìm kiếm tuần tự ta sẽ tìm kiếm nhị phân để tìm phần tử min.

Thuật toán:

- Xây dựng cây nhị phân tìm kiếm.
- Tìm kiếm phần tử nhỏ nhất trong cây.
- Thêm phần tử đó vào mảng và xóa phần tử đó đi.
- Lặp lại hai bước trên và thu được mảng đã sắp xếp.

Ví dụ:

Cho mảng a = [86, 4, 6905, 43, 7, 659]

Với thuật toán sắp xếp chèn nhị phân phần đã được sắp xếp sẽ bắt đầu với một phần tử: | 86 |

Phần tử tiếp theo được chèn vào từ tìm kiếm nhị phân: | 4 | 86 |

Phần tử tiếp theo được chèn vào từ tìm kiếm nhị phân: | 4 | 86 | 6905 |

Phần tử tiếp theo được chèn vào từ tìm kiếm nhị phân: | 4 | 43 | 86 | 6905 |

Phần tử tiếp theo được chèn vào từ tìm kiếm nhị phân: | 4 | 7 | 43 | 86 | 6905 |

Phần tử tiếp theo được chèn vào từ tìm kiếm nhị phân: | 4 | 7 | 43 | 86 | 659 | 6905 |

⇒ mảng đã được sắp xếp

Nhận xét:

Độ phức tạp của thuật toán chèn nhị phân:

- Ở mọi trường hợp ta cần O(logn) phép so sánh để tìm vị trí thích hợp.
- Ở trường hợp tốt nhất khi dãy đã được sắp xếp ta cần tổng cộng O(1) phép gán còn trường hợp xấu nhất là O(n) phép gán.

Như vậy ta có:

- Trường hợp tốt: O(nlogn).
- Trung bình: O(n^2).
- Trường hợp xấu: O(n^2).
- Bộ nhớ: O(1).
- Tính ổn định: Có.

Về độ ổn định:

Độ ổn định là phép đánh giá tính ổn định của thuật toán sắp xếp thông qua việc xáo trộn vị trí các phần tử trước và sau khi sắp xếp. Một thuật toán ổn định là khi một phần tử ban đầu nằm đúng vị trí thì sẽ giữ nguyên vị trí sau khi sắp xếp ví dụ như là: 3 3 4 4 5 5. Còn về thuật toán không ổn định thì mảng có thể sẽ là: 3 3 4 4 5 5.

TÀI LIỆU THAM KHẢO.

Sách cấu trúc dữ liệu và giải thuật, tác giả Phạm Thế Bảo.

Sorting Algorithms - GeeksforGeeks

Các thuật toán sắp xếp cơ bản (viblo.asia)

Data Structure and Algorithms - Shell Sort (tutorialspoint.com)

Counting Sort in Python (Code with Example) | FavTutor

https://blog.luyencode.net/ctdl-gt/

PHẦN CODE PYTHON VÀ BIỂU ĐỒ

Import statements

In [5]:

```
from random import randint
import time
import pandas
import matplotlib.pyplot as plt
import math
```

Array creators

In [27]:

```
#--------- Random array ---------
def rand_arr(n):
    sample_arr = []
    for i in range(n):
        sample_arr.append(randint(1, n))
    return sample_arr

#------- Min sorted array -------
def min_sorted_arr(n):
    sample_arr = []
    for i in range(1, n+1):
        sample_arr.append(i)
    return sample_arr

#------- Max sorted array -------
def max_sorted_arr(n):
    sample_arr = []
    for i in range(1, n+1):
        sample_arr.append(n+1-i)
    return sample_arr

def create_array(state, size):
    if (state == "rand"):
        return rand_arr(size)
    elif (state == "min"):
        return min_sorted_arr(size)
    else:
        return max_sorted_arr(size)
```

Insertion Sort

In [6]:

```
test_arr = [4, 3, 2, 10, 12, 1, 5, 6]
def insertionSort(a):
    for i in range(len(a)):
        key = a[i]
        j = i - 1
        while (j >= 0 and a[j] > key):
            a[j+1] = a[j]
            j -= 1
        a[j+1] = key

insertionSort(test_arr)
print(test_arr)
```

```
[1, 2, 3, 4, 5, 6, 10, 12]
```

Bubble Sort

In [29]:

```
test_arr = [7, 4, 1, 9, 2]
def bubbleSort(a):
    for i in range(len(a)-1):
        for j in range(len(a)-i-1):
            if (a[j] > a[j+1]):
                temp = a[j]
                a[j] = a[j+1]
                a[j+1] = temp

bubbleSort(test_arr)
print(test_arr)
```

```
[1, 2, 4, 7, 9]
```

Selection Sort

In [30]:

```
test_arr = [7, 5, 4, 2]
def selectionSort(a):
    for i in range(len(a)):
        min = i
        for j in range(i, len(a)):
            if (a[j] < a[min]):
                min = j
        temp = a[min]
        a[min] = a[i]
        a[i] = temp

selectionSort(test_arr)
print(test_arr)
```

```
[2, 4, 5, 7]
```

Merge Sort

In [31]:

```
test_arr = [38, 27, 43, 3, 9, 82, 10]
def merge(a, l, m, r):
    n1 = m - l + 1
    n2 = r - m
    L = [x for x in range(n1)]
    R = [x for x in range(n2)]
    for i in range(0, n1):
        L[i] = a[l + i]
    for j in range(0, n2):
        R[j] = a[m + 1 + j]
    i = 0
    j = 0
    k = l
    while (i < n1 and j < n2):
        if (L[i] <= R[j]):
            a[k] = L[i]
            i += 1
        else:
            a[k] = R[j]
            j += 1
        k += 1
    while (i < n1):
        a[k] = L[i]
        i += 1
        k += 1
    while (j < n2):
        a[k] = R[j]
        j += 1
        k += 1
```

```
def mergeSort(a, l, r):
    if (l < r):
        m = int(l + (r-l)/2)
        mergeSort(a, l, m)
        mergeSort(a, m+1, r)
        merge(a, l, m, r)

mergeSort(test_arr, 0, 6)
print(test_arr)
```

```
[3, 9, 10, 27, 38, 43, 82]
```

Quick Sort

In [32]:

```
test_arr = [13, 81, 92, 43, 65, 31, 57, 26, 75, 0]
def partition(a, low, high):
    pivot = a[high]
    i = low - 1
    for j in range(low, high):
        if a[j] <= pivot:
            i = i + 1
            (a[i], a[j]) = (a[j], a[i])
    (a[i + 1], a[high]) = (a[high], a[i + 1])
    return i + 1

def quickSort(a, low, high):
    if (low < high):
        pi = partition(a, low, high)

        quickSort(a, low, pi - 1)
        quickSort(a, pi + 1, high)

quickSort(test_arr, 0, 9)
print(test_arr)
```

```
[0, 13, 26, 31, 43, 57, 65, 75, 81, 92]
```

Heap Sort

In [33]:

```
test_arr = [1, 3, 5, 4, 2]
def heapify(arr, N, i):
    largest = i        # Gán phần tử max = root
    l = 2 * i + 1      # Trái = 2*i + 1
    r = 2 * i + 2      # Phải = 2*i + 2

    if l < N and arr[largest] < arr[l]:
        largest = l

    if r < N and arr[largest] < arr[r]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # swap
        heapify(arr, N, largest)

def heapSort(arr):
    N = len(arr)

    for i in range(N//2 - 1, -1, -1): #Vun đống: phần tử max = root
        heapify(arr, N, i)

    for i in range(N-1, 0, -1): #Rút phần tử max = root ra khỏi cây
        arr[i], arr[0] = arr[0], arr[i]  # swap
        heapify(arr, i, 0)
```

```
heapSort(test_arr)
print(test_arr)
```

```
[1, 2, 3, 4, 5]
```

Shell Sort

In [34]:

```
test_arr = [35, 33, 42, 10, 19, 27, 44]
def shellSort(arr):
    n = len(arr)
    gap = int(n/2)
    while gap > 0:
        for i in range(gap,n):
            temp = arr[i]
            j = i
            while  j >= gap and arr[j-gap] >temp:
                arr[j] = arr[j-gap]
                j -= gap
            arr[j] = temp
        gap = round(gap / 2)

shellSort(test_arr)
print(test_arr)
```

```
[10, 19, 27, 33, 35, 42, 44]
```

Shaker Sort

In [35]:

```
test_arr = [1, 3, 5, 4, 2]
def shakerSort(a):
    n = len(a)
    swapped = True
    start = 0
    end = n-1
    while (swapped == True):
        swapped = False
        for i in range(start, end):
            if (a[i] > a[i + 1]):
                a[i], a[i + 1] = a[i + 1], a[i]
                swapped = True
        if (swapped == False):
            break
        swapped = False
        end = end-1
        for i in range(end-1, start-1, -1):
            if (a[i] > a[i + 1]):
                a[i], a[i + 1] = a[i + 1], a[i]
                swapped = True
        start = start + 1

shakerSort(test_arr)
print(test_arr)
```

```
[1, 2, 3, 4, 5]
```

Radix Sort

In [36]:

```
test_arr = [43, 613, 831, 987, 17, 210, 1990, 1234]
def countingSort(arr, exp1):
    n = len(arr)
    output = [0] * (n)
    count = [0] * (10)
```

```
    for i in range(0, n):
        index = (arr[i]/exp1)
        count[int((index)%10)] += 1
    for i in range(1,10):
        count[i] += count[i-1]
    i = n-1
    while i>=0:
        index = (arr[i]/exp1)
        output[ count[ int((index)%10) ] - 1] = arr[i]
        count[int((index)%10)] -= 1
        i -= 1
    i = 0
    for i in range(0,len(arr)):
        arr[i] = output[i]

def radixSort(arr):
    max1 = max(arr)
    exp = 1
    while max1/exp > 0:
        countingSort(arr,exp)
        exp *= 10

radixSort(test_arr)
print(test_arr)
```

```
[17, 43, 210, 613, 831, 987, 1234, 1990]
```

Counting Sort

In [37]:

```
test_arr = [3, 5, 1, 6, 7, 8, 3]
def countSort(input):
    output = [0] * len(input)
    max = input[0]
    min = input[0]
    for i in range(1, len(input)):
        if input[i] > max:
            max = input[i]
        elif input[i] < min:
            min = input[i]
    k = max - min + 1
    count_list = [0] * k
    for i in range(0, len(input)):
        count_list[input[i] - min] += 1
    for i in range(1, k):
        count_list[i] += count_list[i - 1]
    for i in range(0, len(input)):
        output[count_list[input[i] - min] - 1] = input[i]
        count_list[input[i] - min] -= 1
    for i in range(0, len(input)):
        input[i] = output[i]

countSort(test_arr)
print(test_arr)
```

```
[1, 3, 3, 5, 6, 7, 8]
```

Binery Insertion Sort

In [38]:

```
test_arr = [1, 5, 3, 4, 8, 6, 3, 4]

def binaryInsertionSort(arr):
   for i in range(1, len(arr)):
      temp = arr[i]
      pos = binarySearch(arr, temp, 0, i) + 1
      for k in range(i, pos, -1):
```

```
            arr[k] = arr[k - 1]
        arr[pos] = temp

def binarySearch(arr, key, start, end):
    if end - start <= 1:
        if key < arr[start]:
            return start - 1
        else:
            return start
    mid = (start + end)//2
    if arr[mid] < key:
        return binarySearch(arr, key, mid, end)
    elif arr[mid] > key:
        return binarySearch(arr, key, start, mid)
    else:
        return mid

binaryInsertionSort(test_arr)
print(test_arr)
```

```
[1, 3, 3, 4, 4, 5, 6, 8]
```

Flash Sort

In [39]:

```
test_arr = [-5, 2, -1, 0, 6, 3, 9]
def flashSort(arr):
    bucket = math.floor(float(0.45 * len(arr)))
    a = [0]* bucket
    max = arr[0]
    min = arr[0]
    for i in range(1, len(arr)):
        if (arr[i] < min):
            min = arr[i]
        if (arr[i] > max):
            max = arr[i]
    for i in range(len(arr)):
        k = math.floor(float((bucket-1) * (arr[i]-min)) / (max-min))
        a[k] += 1
    for i in range(1, bucket):
        a[i] = a[i] + a[i-1]
    hold = arr[0]
    move = 0
    flash = 0
    k = 0
    t = 0
    j = 0
    while (move < (len(arr)-1)):
        while(j > (a[k]-1)):
            j += 1
            k = math.floor(float((bucket-1) * (arr[j]-min)) / (max-min))
        flash = arr[j]
        while (j != a[k]):
            k = math.floor(float((bucket-1) * (arr[j]-min)) / (max-min))
            t = a[k] - 1
            hold = arr[t]
            a[k] -= 1
            arr[t] = hold
            flash = hold
            move += 1
    insertionSort(arr)

flashSort(test_arr)
print(test_arr)
```

```
[-5, -1, 0, 2, 3, 6, 9]
```

Variables

In [51]:

```
# size_set = [1000, 3000, 10000, 30000, 100000]
size_set = [1000, 3000, 10000]
state_set = ["rand", "min", "max"]
algorithms_set = [
            insertionSort,
            bubbleSort,
            selectionSort,
            heapSort,
            mergeSort,
            shellSort,
            # quickSort,
            shakerSort,
            radixSort,
            flashSort,
            countSort,
            binaryInsertionSort
]
run_time_data = {
    "Input state": [],
    "Input Size": [],
    "Insertion": [],
    "Bubble": [],
    "Selection": [],
    "Heap": [],
    "Merge": [],
    "Shell": [],
    # "Quick": [],
    "Shaker": [],
    "Radix": [],
    "Flash": [],
    "Count": [],
    "Binary Insertion": [],
}
```

RUN TIME

In [52]:

```
for state in state_set:
    for size in size_set:
        run_time_data["Input state"].append(str(state))
        run_time_data["Input Size"].append(str(size))
        for algo in algorithms_set:
            # create new array
            array = create_array(state, size)

            # get the start time
            st = time.time()

            # Execution function
            if (algo == mergeSort or algo == quickSort):
                algo(array, 0, len(array)-1)
            else:
                algo(array)

            # get the end time
            et = time.time()

            # get the execution time
            elapsed_time = round(et - st, 5)
            if (algo == insertionSort):
                with open(f"result/Insertion_{state}_{size}.txt", 'w') as fp: #get the t
xt result file
                    for a in array:
                        fp.write(f" {a}")
                run_time_data["Insertion"].append(elapsed_time)
            elif (algo == bubbleSort):
                with open(f"result/Bubble_{state}_{size}.txt", 'w') as fp:
```

```
                    for a in array:
                        fp.write(f" {a}")
                run_time_data["Bubble"].append(elapsed_time)
            elif (algo == selectionSort):
                with open(f"result/Selection_{state}_{size}.txt", 'w') as fp:
                    for a in array:
                        fp.write(f" {a}")
                run_time_data["Selection"].append(elapsed_time)
            elif (algo == heapSort):
                with open(f"result/Heap_{state}_{size}.txt", 'w') as fp:
                    for a in array:
                        fp.write(f" {a}")
                run_time_data["Heap"].append(elapsed_time)
            elif (algo == mergeSort):
                with open(f"result/Merge_{state}_{size}.txt", 'w') as fp:
                    for a in array:
                        fp.write(f" {a}")
                run_time_data["Merge"].append(elapsed_time)
            elif (algo == shellSort):
                with open(f"result/Shell_{state}_{size}.txt", 'w') as fp:
                    for a in array:
                        fp.write(f" {a}")
                run_time_data["Shell"].append(elapsed_time)
            # elif (algo == quickSort):
            #     with open(f"result/Quick_{state}_{size}.txt", 'w') as fp:
            #         for a in array:
            #             fp.write(f" {a}")
            #     run_time_data["Quick"].append(elapsed_time)
            elif (algo == shakerSort):
                with open(f"result/Shaker_{state}_{size}.txt", 'w') as fp:
                    for a in array:
                        fp.write(f" {a}")
                run_time_data["Shaker"].append(elapsed_time)
            elif (algo == radixSort):
                with open(f"result/Radix_{state}_{size}.txt", 'w') as fp:
                    for a in array:
                        fp.write(f" {a}")
                run_time_data["Radix"].append(elapsed_time)
            elif (algo == flashSort):
                with open(f"result/Flash_{state}_{size}.txt", 'w') as fp:
                    for a in array:
                        fp.write(f" {a}")
                run_time_data["Flash"].append(elapsed_time)
            elif (algo == binaryInsertionSort):
                with open(f"result/Binary_Insertion_{state}_{size}.txt", 'w') as fp:
                    for a in array:
                        fp.write(f" {a}")
                run_time_data["Binary Insertion"].append(elapsed_time)
            elif (algo == countSort):
                with open(f"result/Count_{state}_{size}.txt", 'w') as fp:
                    for a in array:
                        fp.write(f" {a}")
                run_time_data["Count"].append(elapsed_time)
            # print(f'Execution time {(size)} {(state)}: ', elapsed_time, 'seconds')

data = pandas.DataFrame(run_time_data)
data.to_csv("Runtime.csv")
```

Data Exploration

In [53]:

```
df = pandas.read_csv('Runtime.csv')
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9 entries, 0 to 8
Data columns (total 14 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   Unnamed: 0        9 non-null      int64
```

```
 1   Input state        9 non-null      object
 2   Input Size         9 non-null      int64
 3   Insertion          9 non-null      float64
 4   Bubble             9 non-null      float64
 5   Selection          9 non-null      float64
 6   Heap               9 non-null      float64
 7   Merge              9 non-null      float64
 8   Shell              9 non-null      float64
 9   Shaker             9 non-null      float64
 10  Radix              9 non-null      float64
 11  Flash              9 non-null      float64
 12  Count              9 non-null      float64
 13  Binary Insertion   9 non-null      float64
dtypes: float64(11), int64(2), object(1)
memory usage: 1.1+ KB
```

In [54]:

```
df.describe()
```

Out[54]:

	Unnamed: 0	Input Size	Insertion	Bubble	Selection	Heap	Merge	Shell	Shaker	Radix	Fla
count	9.000000	9.000000	9.000000	9.000000	9.000000	9.000000	9.000000	9.000000	9.000000	9.000000	9.000(
mean	4.000000	4666.666667	3.176734	7.995559	3.471573	0.078457	0.087437	0.045326	4.835711	4.044529	2.8178
std	2.738613	4092.676386	5.630665	12.028375	5.083568	0.081350	0.090048	0.052690	8.910101	3.429760	5.0257
min	0.000000	1000.000000	0.000000	0.139630	0.055850	0.009970	0.012970	0.004990	0.000000	0.796880	0.0069
25%	2.000000	1000.000000	0.006980	0.273270	0.139630	0.012970	0.019950	0.011970	0.001990	1.400260	0.0608
50%	4.000000	3000.000000	0.221410	1.296530	0.747000	0.032910	0.056850	0.030920	0.372010	2.355710	0.1815
75%	6.000000	10000.000000	1.493010	11.548680	6.081950	0.134640	0.105720	0.037900	2.009630	6.702090	1.2506
max	8.000000	10000.000000	14.303780	33.361220	14.665830	0.222400	0.252330	0.169550	24.935880	10.486990	13.2606

In [55]:

```
reshape_df = df[df["Input state"] == "rand"]
randomize_input = reshape_df[["Input Size", "Insertion", "Bubble", "Selection", "Heap",
"Merge", "Shell", "Shaker", "Radix", "Count", "Binary Insertion", "Flash"]]
randomize_input.head()
```

Out[55]:

	Input Size	Insertion	Bubble	Selection	Heap	Merge	Shell	Shaker	Radix	Count	Binary Insertion	Flash
0	1000	0.08577	0.19249	0.05585	0.01296	0.01596	0.00899	0.15858	0.79688	0.00199	0.05784	0.08677
1	3000	0.77393	1.29653	0.74700	0.03291	0.03391	0.03092	1.27958	2.35571	0.00499	0.68424	0.70811
2	10000	11.70373	33.36122	7.80814	0.18251	0.22142	0.16955	14.76274	10.48699	0.01197	4.79020	9.77588

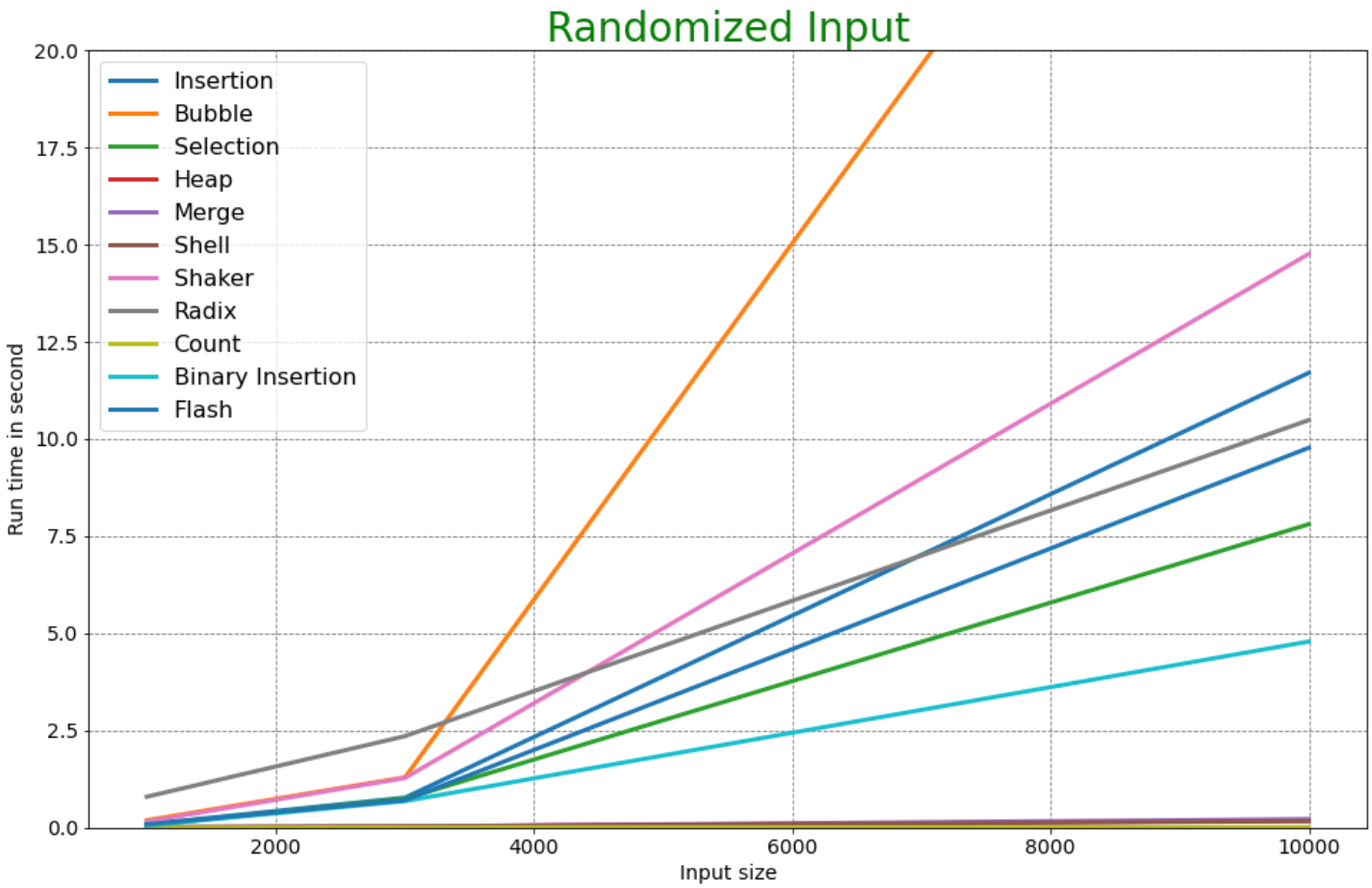
Randomize input

In [56]:

```
plt.figure(figsize=(16,10))
plt.grid(color='grey', linestyle='--')
plt.title('Randomized Input', fontsize=28, color="green")
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.xlabel('Input size', fontsize=14)
plt.ylabel('Run time in second', fontsize=14)
plt.ylim(0, 20)
```

```
for column in randomize_input.columns:
    if (column != "Input Size"):
        plt.plot(randomize_input["Input Size"], randomize_input[column], label=randomize
_input[column].name, linewidth=3)

plt.legend(fontsize=16)
plt.show()
```



What we got from the chart

- **Từ biểu đồ với dãy hoàn toàn ngẫu nhiên thì sắp xếp nổi bọt có thới gian chạy lớn nhất từ chuỗi có độ dài lớn hơn 3000.**
- **Radix sort có thời gian chạy tăng tuyến tính theo độ lớn dãy input.**
- **Các thuật toán như shell sort, count sort, merge sort và heap sort có thời gian chạy rất nhanh gần như bằng 0.**

Min Sorted Input

In [57]:

```
reshape_df = df[df["Input state"] == "min"]
min_sorted_input = reshape_df[["Input Size", "Insertion", "Bubble", "Selection", "Heap",
"Merge", "Shell", "Shaker", "Radix", "Count", "Binary Insertion", "Flash"]]
min_sorted_input.head()
```

Out[57]:

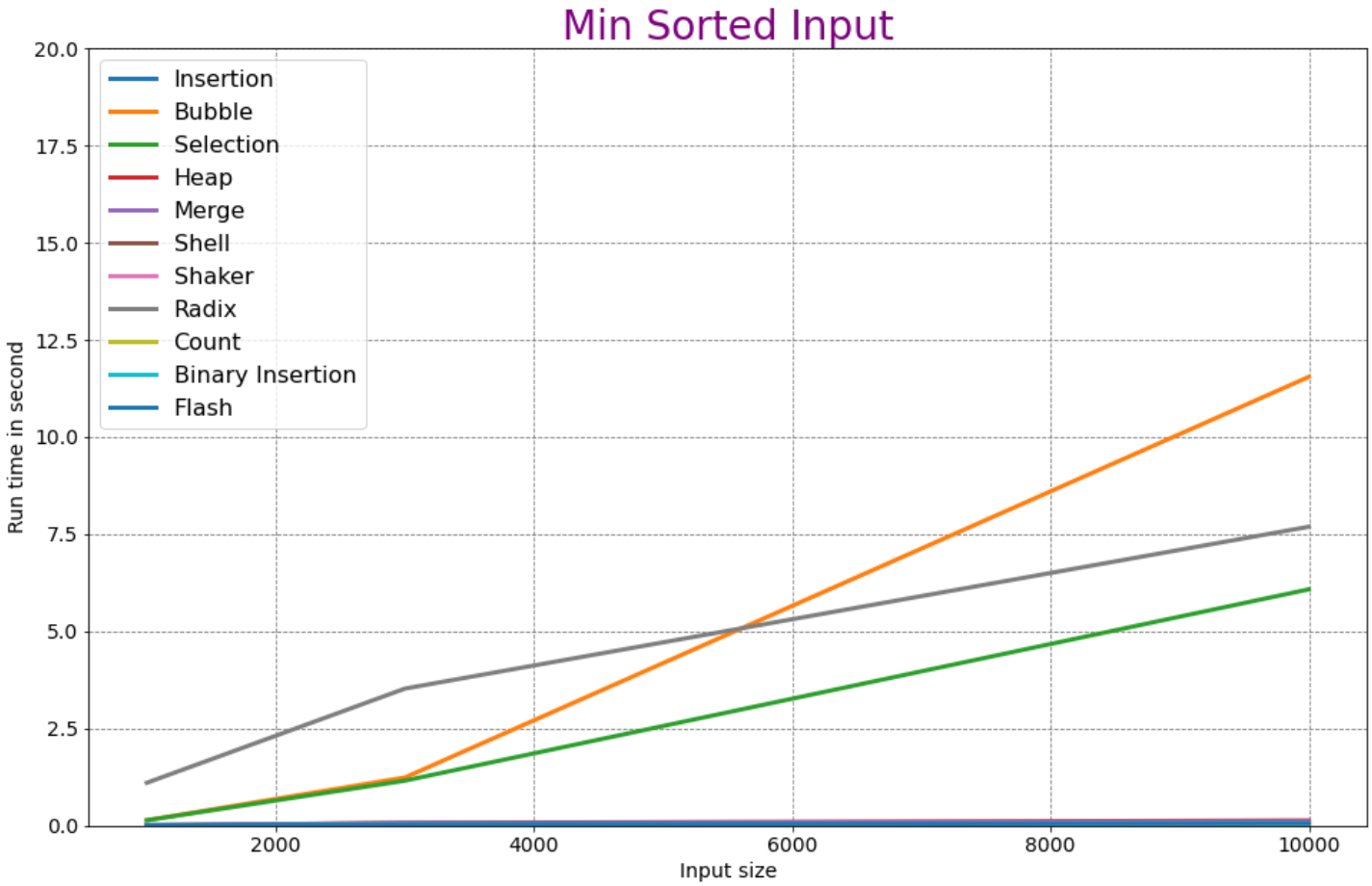
	Input Size	Insertion	Bubble	Selection	Heap	Merge	Shell	Shaker	Radix	Count	Binary Insertion	Flash
3	1000	0.00000	0.13963	0.13963	0.01297	0.01297	0.01197	0.00000	1.10405	0.00199	0.01396	0.00698
4	3000	0.00200	1.24068	1.15932	0.07281	0.06782	0.02294	0.00099	3.52857	0.00499	0.03491	0.02892
5	10000	0.00698	11.54868	6.08195	0.13464	0.10572	0.03790	0.00199	7.69244	0.02094	0.06483	0.06084

In [58]:

```
plt.figure(figsize=(16,10))
plt.grid(color='grey', linestyle='--')
plt.title('Min Sorted Input', fontsize=28, color="purple")
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.xlabel('Input size', fontsize=14)
plt.ylabel('Run time in second', fontsize=14)
plt.ylim(0, 20)

for column in min_sorted_input.columns:
    if (column != "Input Size"):
        plt.plot(min_sorted_input["Input Size"], min_sorted_input[column], label=min_sor
ted_input[column].name, linewidth=3)

plt.legend(fontsize=16)
plt.show()
```



What we got from the chart

- **Với mảng đã sắp xếp từ nhỏ đến lớn.**
- **Bubble sort, selection sort và radix sort có thời gian chạy khá lâu.**
- **Các thuật toán còn lại do đã thỏa điều kiện tốt nhất nên chạy rất nhanh.**

Max Sorted Input

In [59]:

```
reshape_df = df[df["Input state"] == "max"]
max_sorted_input = reshape_df[["Input Size", "Insertion", "Bubble", "Selection", "Heap",
"Merge", "Shell", "Shaker", "Radix", "Count", "Binary Insertion", "Flash"]]
max_sorted_input.head()
```

Out[59]:

	Input Size	Insertion	Bubble	Selection	Heap	Merge	Shell	Shaker	Radix	Count	Binary Insertion	Flash

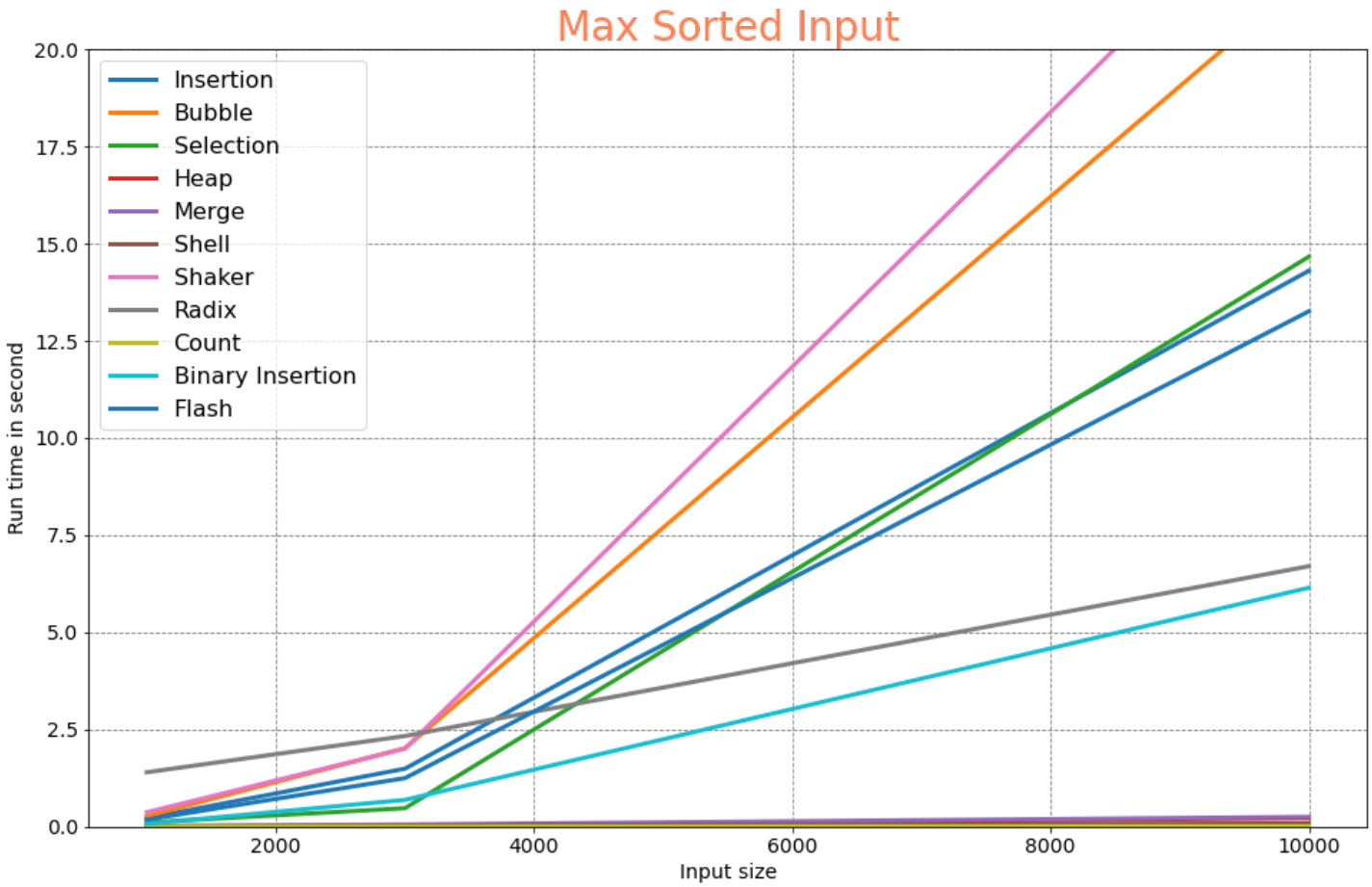
Input Size	Insertion	Bubble	Selection	Heap	Merge	Shell	Shaker	Radix	Count	Binary Insertion	Flash
6	1000	0.22141	0.27327	0.10971	0.00997	0.01995	0.00499	0.37201	1.40026	0.00299	0.07579	0.18152
7	3000	1.49301	2.02060	0.47673	0.02494	0.05685	0.03291	2.00963	2.33377	0.00399	0.19948	1.25066
8	10000	14.30378	21.88693	14.66583	0.22240	0.25233	0.08776	24.93588	6.70209	0.01396	6.14757	13.26060

In [60]:

```
plt.figure(figsize=(16,10))
plt.grid(color='grey', linestyle='--')
plt.title('Max Sorted Input', fontsize=28, color="coral")
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.xlabel('Input size', fontsize=14)
plt.ylabel('Run time in second', fontsize=14)
plt.ylim(0, 20)

for column in max_sorted_input.columns:
    if (column != "Input Size"):
        plt.plot(max_sorted_input["Input Size"], max_sorted_input[column], label=max_sor
ted_input[column].name, linewidth=3)

plt.legend(fontsize=16)
plt.show()
```



What I got from the chart

- **Với mảng đã sắp xếp từ lớn đến bé.**
- **Bubble sort và shaker sort có thời gian chạy rất lâu.**
- **Selection sort, flash sort, insertion sort cũng có thời gian chạy khá lâu.**
- **Binary insertion sort và radix sort có thời gian chạy tuyến tính với độ dài mảng input.**
- **Các thuật toán còn lại chạy rất nhanh dưới 1 giây ở mọi độ dài mảng.**

Reference links

- https://www.geeksforgeeks.org/sorting-algorithms/?ref=gcse
- https://viblo.asia/p/cac-thuat-toan-sap-xep-co-ban-Eb85ooNO52G

- https://www.tutorialspoint.com/data_structures_algorithms/shell_sort_algorithm.htm
- https://favtutor.com/blogs/counting-sort-python