BÁO CÁO

Đánh giá độ phức tạp của các thuật toán sắp xếp

Thái Xuân Đăng
19120003

MỤC LỤC

Giới thiệu 12 thuật toán sắp xếp

Trong phần này, ta sẽ giới thiệu về ý tưởng, cách cài đặt và độ phức tạp về thời gian, bộ nhớ của 12 thuật toán: sắp xếp chọn (selection sort), sắp xếp chèn (insertion sort), sắp xếp chèn kết hợp tìm kiếm nhị phân (binary insertion sort), sắp xếp nổi bọt (bubble sort), sắp xếp cocktail (shaker sort), sắp xếp Shell (Shellsort), sắp xếp vun đống (heapsort), sắp xếp trộn (merge sort), sắp xếp nhanh (quicksort), sắp xếp đếm (counting sort), sắp xếp theo cơ số (radix sort), sắp xếp flash (flashsort).

Để tiện cho việc trình bày, ta giả sử rằng dãy cần sắp xếp là $a_0, a_1, \dots, a_{n-1}$ với n là độ dài của dãy và thứ tự dãy ta muốn đạt được là không giảm. Tức là ta cần hoán vị các phần tử trên để đảm bảo điều kiện $a_0 \le a_1 \le \cdots \le a_{n-1}$.

1. Sắp xếp chọn (selection sort)

a) Ý tưởng

Đi từ đầu đến cuối dãy, với mỗi phần tử a_i ta cần tìm phần tử a_j $(i \le j < n)$ nhỏ nhất và tráo đổi a_i với a_j. Vì vậy, ta luôn giữ được điều kiện $a_i \le a_j$ $(0 \le i < j < n)$ nên dãy đã cho sẽ được sắp xếp.

b) Cài đặt và mã giả

Duyệt i từ 0 đến $n-2$, với mỗi i, duyệt $n-i$ lần tìm chỉ số j $(i \le j < n)$ sao cho a_j là nhỏ nhất và tiến hành tráo đổi a_i với a_j.

```
for i = 0 to n – 2
    jmin = i
    for j = i + 1 to n – 1
        if a[j] < a[jmin]
            jmin = j
    exchange a[j] with a[jmin]
```

c) Độ phức tạp

Số lần thực hiện phép so sánh (không phụ thuộc gì vào tình trạng ban đầu của dãy a) là $(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}$. Vậy thuật toán có độ phức tạp là $\Theta(n^2)$.

2. Sắp xếp chèn (insertion sort)

a) Ý tưởng

Đi từ đầu đến cuối dãy, với mỗi phần tử a_i ta luôn đảm bảo dãy $a_0, a_1, \dots, a_{i-1}$ đã được sắp xếp. Việc của ta lúc này chỉ là chèn a_i vào đúng vị trí trong dãy này thì ta sẽ đảm bảo được dãy $a_0, a_1, \dots, a_i$ được sắp xếp và đi tới phần tử a_{i+1} tiếp theo.

b) Cài đặt và mã giả

Duyệt i từ 1 đến $n-1$, với mỗi i, tiến hành so sánh a_i với phần tử a_{i-1} trước nó xem đã đảm bảo điều kiện sắp xếp chưa. Nếu chưa, thì tiến thành dịch chuyển a_{i-1} vào vị trí của a_i và xét tiếp với a_{i-2}… cứ như vậy cho đến khi điều kiện được thỏa mãn và a_i được đặt vào đúng vị trí.

```
for i = 1 to n – 1
    x = a[i]
    j = i – 1
    // chèn a[i] vào đúng vị trí
    while j >= 0 and a[j] > x
        a[j + 1] = a[j]
        j = j – 1
    a[j + 1] = x
```

c) Độ phức tạp

Trong trường hợp dãy đã được sắp xếp, ta chỉ thực hiện đúng n lần thực hiện phép so sánh, do đó ta có độ phức tạp trong trường hợp tốt nhất là $\Theta(n)$.

Trong trường hợp dãy được sắp xếp ngược, với mỗi i, ta cần thực hiện i phép so sánh cũng như dịch chuyển, do đó ta có độ phức tạp trong trường hợp xấu nhất là $\Theta(n^2)$.

Trong trường hợp dãy được sắp xếp ngẫu nhiên, ta xem xác suất mỗi giá trị là giống nhau thì có thể coi với mỗi i, ta cần trung bình $\frac{i}{2}$ phép so sánh và thuật toán cũng có độ phức tạp trung bình là $\Theta(n^2)$.

3. Sắp xếp chèn kết hợp tìm kiếm nhị phân (binary insertion sort)

a) Ý tưởng

Tương tự thuật toán sắp xếp chèn, chỉ khác lúc tìm vị trí chèn ta sử dụng thuật toán tìm kiếm nhị phân.

b) Cài đặt và mã giả

Duyệt i từ 1 đến $n-1$, với mỗi i, sử dụng tìm kiếm nhị phân để tìm vị trí đúng của a_i trong dãy $a_0, a_1, \dots, a_{i-1}$ và chèn vào đó.

```
for i = 1 to n – 1
    x = a[i]
    // tìm kiếm nhị phân k là vị trí gần i nhất mà a[k] > x
```

```
// hoặc k = i nếu không có vị trí nào thỏa mãn
k = binary-search(a, i, x)
for j = i – 1 downto k
    a[j + 1] = a[j]
a[k] = x
```

c) Độ phức tạp

Thuật toán có độ phức tạp trong trường hợp xấu nhất tương tự như thuật toán sắp xếp chèn là $\Theta(n^2)$. Tuy nhiên, đối với dãy mà việc so sánh các phần tử là tốn nhiều thời gian thì thuật toán có sự hiệu quả vượt trội so với thuật toán sắp xếp chèn khi mà chỉ sử dụng $\Theta(n \log n)$ phép so sánh so với $O(n^2)$ phép so sánh của thuật toán sắp chèn.

4. Sắp xếp nổi bọt (bubble sort)

a) Ý tưởng

Tương tự như thuật toán sắp xếp chọn, nhưng thay vì chọn phần tử nhỏ nhất thì ta phần tử đó "nổi" lên từ cuối về vị trí i.

b) Cài đặt và mã giả

Duyệt i từ 0 đến $n-2$, với mỗi i, duyệt từ cuối j từ cuối $(n-1)$ về $i+1$, nếu $a_{j-1} > a_j$ thì đổi chỗ a_{j-1} và a_j.

```
for i = 0 to n – 2
    for j = n – 1 downto i + 1
        if a[j] < a[j – 1]
            exchange a[j] with a[j – 1]
```

c) Độ phức tạp

Số lần thực hiện phép so sánh (không phụ thuộc gì vào tình trạng ban đầu của dãy a) là $(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}$. Vậy thuật toán có độ phức tạp là $\Theta(n^2)$.

5. Sắp xếp cocktail (shaker sort)

a) Ý tưởng

Cải tiến từ thuật toán sắp xếp nổi bọt, sau khi đưa phần tử nhỏ nhất về đầu dãy, thuật toán sẽ giúp chúng ta đưa phần tử lớn nhất về cuối dãy. Do đưa các phần tử về đúng vị trí ở cả hai đầu nên thuật toán sắp xếp cocktail sẽ giúp cải thiện thời gian sắp xếp dãy số.

b) Cài đặt và mã giả

Tương tự như thuật toán sắp xếp nổi bọt, nhưng ta ghi nhớ thêm hai chỉ số l và r cho biết ta chỉ cần sắp xếp các phần tử trong đoạn $a_l, a_{l+1}, \dots, a_r$.

```
l = 0
r = n – 1
k = 0
while l < r
   for i = l to r – 1
      if a[i] > a[i + 1]
         exchange a[i] with a[i + 1]
         k = i
   r = k
   for i = r downto l + 1
      if a[i] < a[i – 1]
         exchange a[i] with a[i – 1]
         k = i
   l = k
```

c) Độ phức tạp

Trong trường hợp tốt nhất lúc dãy đã được sắp xếp, thuật toán có độ phức tạp là $\Theta(n)$ do chỉ so sánh n lần và được ghi nhớ được vùng nhớ cần sắp xếp là rỗng.

Trong trường hợp xấu nhất và trung bình, thuật toán có độ phức tạp là $\Theta(n^2)$ tương tự như bubble sort.

6. Sắp xếp Shell (Shellsort)

a) Ý tưởng

Nhược điểm của thuật toán sắp xếp chèn là khi mà ta luôn phải chèn một phần tử vào vị trí gần đầu dãy. Từ đó, thuật toán sắp xếp Shell có ý tưởng rằng sử dụng thuật toán sắp xếp chèn với những bước nhảy xa.

b) Cài đặt và mã giả

Lần lượt thực hiện thuật toán sắp chèn với những bước nhảy xa khác nhau tương ứng với các dãy con khác nhau. Sau đó thực hiện thuật toán chèn với bước nhảy bằng 1 (tức là thuật toán sắp xếp chèn thông thường) với dãy.

```
for gap in gaps
    for i = gap to n – 1
        // thực hiện sắp xếp chèn với độ dài bước nhảy là gap
        x = a[i]
        j = i – gap
        while j >= 0 and a[j] > x
            a[j + gap] = a[j]
            j = j – gap
        a[j + gap] = x
```

c) Độ phức tạp và các cải tiến

Độ phức tạp của thuật toán phụ thuộc hoàn toàn việc chọn dãy các bước nhảy. Với việc chọn dãy bước nhảy như cách cài đặt ở trên $\left(\left[\frac{n}{2}\right], \left[\frac{n}{4}\right], \dots, 1\right)$, độ phức tạp của thuật toán trong trường hợp xấu nhất là $\Theta(n^2)$. Tuy nhiên, đối với các bộ dữ liệu được sử dụng trong báo cáo (ngẫu nhiên, đã sắp xếp, …), thuật toán hoạt động rất nhanh. Trong trường hợp tốt nhất, dãy đã cho đã được sắp xếp, thuật toán thực hiện $1 + 2 + 4 \dots + \frac{n}{2} + n = \Theta(n)$ phép so sánh do đó có độ phức tạp là $\Theta(n)$.

Tham khảo Gap sequences, ta thấy rằng có thể đạt được độ phức tạp $\Theta(n \log^2 n)$ với cách chọn bước nhảy có dạng các số nguyên liên tiếp có dạng $2^p 3^q$. Tuy nhiên, để đơn giản, trong bài làm này chỉ cài đặt bước nhảy có dạng $\left[\frac{n}{2^k}\right]$.

7. Sắp xếp vun đống (heapsort)

a) Ý tưởng

Sử dụng cấu trúc đống nhị phân (binary heap) để thực hiện việc sắp xếp. Đống ở đây (trường hợp này là đống tối đa (max heap)) là một dãy đảm bảo điều kiện $a_i > a_{2i+1}$ và a_{2i+2} (nếu $2i + 2 < n$) với mọi i sao cho $0 \le i \le \frac{n-2}{2}$. Ta lần lượt lấy các phần tử lớn nhất và đặt chúng từ cuối về đầu của dãy bằng cấu trúc này, dãy đã cho sẽ được sắp xếp.

b) Cài đặt và mã giả

Đầu tiên, ta xây dựng dãy đã cho thành một đống có n phần tử.

Sau đó, ta duyệt i từ $n-1$ về 1, với mỗi i, ta đảm bảo có $a_0, a_1, \ldots, a_i$ là một đống; ta tráo đổi hai phần tử a_0 và a_i thì a_i sẽ là phần tử lớn nhất trong đống này; tiếp theo, ta xem như đống còn i phần tử (đã loại phần tử a_i ra) và điều chỉnh đống thỏa mãn điều kiện từ a_0.

Các thao tác trên đều giả sử rằng ta đã biết cách xây dựng và điều chỉnh đống trong cách cài đặt tối ưu nhất với độ phức tạp lần lượt là $O(n \log n)$ và $O(\log n)$.

Thao tác điều chỉnh đống ta có thể cài đặt như sau.

```
adjust-heap(i, n)
    x = a[i]
    j = i * 2 + 1
    while j < n
        if j + 1 < n and a[j + 1] > a[j]
            j = j + 1
        if x >= a[j]
            break
        a[i] = a[j]
        i = j
        j = i * 2 + 1
    a[i] = x
```

Thuật toán hoàn chỉnh được cài đặt đơn giản như mã giả dưới đây.

```
// xây dựng đống
for parent = (n – 2) / 2 downto 0
    adjust-heap(parent, n)
// sắp xếp
for i = n – 1 downto 1
    exchange a[0] with a[i]
    adjust-heap(0, i)
```

c) Độ phức tạp

Mỗi thao tác lấy phần tử lớn nhất của đống đều có thể thực hiện trong $\Theta(1)$ và thao tác điều chỉnh đống có thể thực hiện trong $O(\log n)$.

Để xây dựng đống, ta sẽ thực hiện $\frac{n}{2}$ lần điều chỉnh đống và có độ phức tạp là $O\left(\frac{n}{2}\log n\right) = O(n \log n)$. Thao tác tiếp theo, cũng sẽ thực hiện n lần điều chỉnh đống và có độ phức tạp là $O(n \log n)$. Do đó, tổng độ phức tạp là $O(n \log n)$.

8. Sắp xếp trộn (merge sort)

a) Ý tưởng

Với hai dãy x và y có độ dài lần lượt là n và m, ta có thể tạo một dãy z có độ dài $n + m$ với mỗi phần tử trong x và y đều nằm trong z đúng một lần. Thao tác này gọi là thao tác "trộn" và có thể cài đặt trong độ phức tạp $\Theta(n + m)$. Thuật toán sắp xếp trộn sẽ thực hiện sắp xếp sau đó trộn hai dãy con không giao nhau, có độ dài bằng nhau (hoặc chênh lệch 1 đơn vị nếu độ dãy là lẻ). Việc sắp xếp hai dãy con cũng được thực hiện bằng cách như vậy. Sau khi thực hiện xong, dãy đã cho sẽ được sắp xếp.

b) Cài đặt và mã giả

Ta có thể viết thủ tục đệ quy mã giả dưới đây và để sắp xếp dãy a, ta gọi thủ tục merge_sort(a, 0, n – 1).

```
merge-sort(a, l, r)
  if l < r
    m = (l + r) / 2
    // sắp xếp hai dãy a[1..m] và a[m + 1..r]
    merge-sort(a, l, m)
    merge-sort(a, m + 1, r)
    // tiến hành trộn hay dãy a[l..m] và a[m + 1..r] lại với nhau
    merge(a, l, m, r)
```

c) Độ phức tạp và cải tiến

Ta sẽ có $\Theta(\log n)$ lần trộn trên toàn bộ dãy n phần tử, mỗi lần ta tốn tổng độ phức tạp là $\Theta(n)$. Do đó, tổng độ phức tạp của thuật toán về mặt thời gian là $\Theta(n \log n)$. Vì phải sử dụng một mảng tạm để lưu trữ kết quả trộn nên độ phức tạp của thuật toán về mặt bộ nhớ là $\Theta(n)$.

Mặc dù ý tưởng và cách cài đặt nêu trên là dễ hiểu và dễ cài đặt nhưng trong báo cáo này, em đã cài đặt thuật toán trên bằng cách khử đệ quy để đạt được tốc độ nhanh hơn. Cách cài

đặt cũng khá đơn giản: thực hiện trộn các các bộ hai dãy có độ dài bằng 1, rồi từ đó trộn các dãy đã sắp xếp có độ dài bằng 2, tiếp theo là các dãy đã sắp xếp có độ dài bằng 4… cho đến khi tạo được một dãy duy nhất.

9. Sắp xếp nhanh (quicksort)

a) Ý tưởng

Ta sử dụng ý tưởng chia để trị như sau. Với dãy a ban đầu, ta sẽ chia thành hai dãy $a_0, a_1, \dots, a_k$ và $a_{k+1}, a_{k+2}, \dots, a_{n-1}$ sao cho mọi phần tử trong dãy thứ nhất luôn không lớn hơn mọi phần tử trong dãy thứ hai. Sau đó, ta tiếp tục làm việc tương tự đối với hai dãy này. Kết thúc thuật toán, dãy đã cho sẽ được sắp xếp.

b) Cài đặt và mã giả

Ta viết một hàm đệ quy để thực hiện việc chia để trị. Để chia ra dãy a thành hai dãy như trên (thực ra là ba trong cách cài đặt này), ta cần chọn một phần tử nào đó trong dãy làm "chốt". Mọi phần tử không lớn hơn chốt sẽ nằm trong dãy thứ nhất và mọi phần tử không nhỏ hơn chốt sẽ nằm trong dãy thứ hai. Việc này ta có thể thực hiện một cách dễ dàng trong độ phức tạp $\Theta(n)$. Sẽ có một số phần tử không nằm trong hai dãy này mà nằm ở giữa hai dãy. Chúng là các phần tử có giá trị bằng với chốt và ta không cần sắp xếp đoạn này.

Tiếp theo, ta gọi hàm đệ quy đối với hai dãy con này cho đến khi độ dài của dãy ta cần sắp xếp là bằng 1 hoặc rỗng.

```
quicksort(a, l, r)
  if l >= r
    return
  pivot = a[(l + r) / 2] // tùy theo trường hợp cụ thể mà ta có cách chọn chốt riêng
  i = l
  j = r
  repeat
    while a[i] < pivot
      i = i + 1
    while a[j] > pivot
      j = j – 1
    if i <= j
      exchange a[i] with a[j]
```

```
        i = i + 1
        j = j – 1
    until i > j
    quicksort(a, l, j)
    quicksort(a, j, r)
```

c) Độ phức tạp và cách chọn chốt

Trong trường hợp tốt nhất, khi ta chọn được phần tử chốt là phần tử trung vị của đoạn, độ phức tạp về mặt thời gian là $\Theta(n \log n)$ và độ phức tạp bộ nhớ dùng cho việc gọi đệ quy là $\Theta(\log n)$.

Trong trường hợp xấu nhất, khi ta chọn phần tử chốt là phần tử lớn nhất hoặc nhỏ nhất, chia ra thành hai dãy có kích thước là 1 và $n - 1$ thì độ phức tạp thời gian của thuật toán là $\Theta(n^2)$ và độ phức tạp bộ nhớ của thuật toán là $\Theta(n)$.

Độ phức tạp của thuật toán quicksort phụ thuộc hoàn toàn vào việc chọn chốt để thực hiện phân đoạn. Ta biết rằng, nếu chốt được chọn là trung vị của đoạn đó thì thuật toán sẽ đạt tới độ phức tạp trong trường hợp tốt nhất là $\Theta(n \log n)$. Nếu ta chọn chốt một cách ngẫu nhiên, ta cũng có thể đạt được độ phức tạp trung bình là $\Theta(n \log n)$ và thật khó để tìm ra một bộ dữ liệu làm thuật toán thực thi chậm hơn so với độ phức tạp đó. Tuy nhiên, để chắc chắn hơn, vấn đề chọn chốt có thể giải quyết nếu như chúng ta có thể đưa ra một thuật toán có thể tìm được trung vị của đoạn trong độ phức tạp $\Theta(n)$. Trên thực tế, có một phương pháp có thể làm được như vậy nhưng khá phức tạp và giới hạn trong báo cáo này không thể trình bày một cách ngắn gọn. Từ đó, ta nói rằng thuật toán quicksort có thể cài đặt được với độ phức tạp $\Theta(n \log n)$ kể cả trong trường hợp xấu nhất.

10. Sắp xếp đếm (counting sort)

a) Ý tưởng

Nếu mỗi giá trị trong dãy đều có thể đếm được nhanh chóng thì ta có thể đếm tất cả các giá trị có trong dãy và các giá trị nhỏ hơn sẽ được xếp trước đúng với số lần xuất hiện của nó.

b) Cài đặt và mã giả

Ta cài đặt thuật toán với dãy a là một dãy số nguyên bằng cách đếm mỗi giá trị nguyên xuất hiện bao nhiêu lần bằng một mảng đếm. Sau đó, đặt những giá trị này vào đúng vị trí như ý tưởng vừa nêu trên.

```
for i = 0 to n – 1
    cnt[a[i]] = cnt[a[i]] + 1
j = 0
```

```
for i = min to max:
    while cnt[i] > 0
        cnt[i] = cnt[i] – 1
        a[j] = i
        j = j + 1
```

c) Độ phức tạp

Độ phức tạp về mặt thời gian của thuật toán là $\Theta(n+k)$ với $k = max\{a_0, a_1, \dots, a_{n-1}\} - min\{a_0, a_1, \dots, a_{n-1}\}$. Độ phức tạp bộ nhớ của thuật toán là $\Theta(k)$.

11. Sắp xếp theo cơ số (radix sort)

a) Ý tưởng

Đối với các số tự nhiên, ta có ý tưởng sắp xếp các số này theo các chữ số có nghĩa nhất cho đến các chữ số ít ý nghĩa nhất (chữ số hàng đơn vị). Đầu tiên ta sắp xếp các số theo độ lớn của chữ số có ý nghĩa nhất. Tiếp đến, đối với mỗi nhóm có cùng chữ số đó thì ta tiếp tục tục thực hiện sắp xếp trong nhóm đó cho đó, … Cứ như vậy cho đến khi ta thực hiện đến hàng đơn vị.

b) Cài đặt và mã giả

Ý tưởng là vậy nhưng cách cài đặt lại ngược lại với cách hiểu trên. Đầu tiên, ta sử dụng một thuật toán sắp xếp các số theo độ lớn của chữ số hàng đơn vị. Trong các lần tiếp theo, ta dùng một thuật toán sắp xếp ổn định sắp xếp các số theo chữ số hàng chục (đối với hệ cơ số 10), … Cứ như vậy cho đến khi ta hoàn thành việc sắp xếp các số theo chữ số có nghĩa nhất của số lớn nhất. Do thuật toán ta sử dụng là thuật toán sắp xếp ổn định nên với mỗi chữ số ta sắp xếp thì các chữ số ít ý nghĩa hơn (đã được sắp xếp từ trước) sẽ được sắp xếp không giảm nếu những số này có chữ số đang xét bằng nhau.

Thuật toán mà ta sử dụng trong mỗi lần sắp xếp như vậy là phải ổn định và đã được tốc độ cao nên ta sử dụng thuật toán sắp xếp đếm phân phối. Và ta không chỉ có thể sắp xếp theo hệ cơ số 10 mà có thể thực hiện thuật toán với hệ cơ số R bất kỳ với điều kiện $R \geq 2$ để có thể phân lớp.

```
base = 1
while base <= max
    // thực hiện sắp xếp a với a[i] theo giá trị a[i] / base % R bằng counting sort
    counting-sort(a, base)
    base = base * R
```

c) Độ phức tạp và biến thể

Do thuật toán sắp xếp bằng đếm phân phối có độ phức tạp là $\Theta(n + R)$, nhưng do R là hằng số ta đã chọn từ trước nên khi n lớn, thuật toán độ phức tạp $\Theta(n)$. Ta thực hiện thuật toán sắp xếp bằng đếm phân phối này d lần với d là số chữ số của số lớn nhất trong dãy, nên độ phức tạp của thuật toán về mặt thời gian là $\Theta(nd)$. Do phải tốn bộ nhớ cho việc thực hiện thuật toán sắp xếp bằng đếm phân phối, độ phức tạp bộ nhớ là $\Theta(R)$.

Thuật toán trình bày ở trên cũng có thể hoạt động được với dãy các số nguyên mà có tồn tại phần tử âm bằng một số sửa đổi nhỏ. Thí dụ, ta có thể cộng tất cả các phần tử của dãy lên cùng một lượng đủ lớn để các phần tử này luôn không âm. Hoặc ta cũng có thể thực hiện sắp xếp các phần tử có giá trị âm một cách riêng bằng cách sắp xếp các phần tử này theo giá trị tuyệt đối của chúng theo thứ tự không tăng rồi đặt chúng vào đầu dãy.

12. Sắp xếp flash (flashsort)

a) Ý tưởng

Đầu tiên, ta chia các số vào m phân lớp như trong trường hợp lý tưởng rằng các số đều được phân bố đều, mỗi giá trị xuất hiện một lần bằng công thức $k_{a_i} = \left[(m-1)\frac{a_i - min}{max - min}\right]$ với $max = max\{a_0, a_1, \dots, a_{n-1}\}$ và $min = min\{a_0, a_1, \dots, a_{n-1}\}$. Lúc này, ta đảm bảo các phần tử nằm trong phân lớp nhỏ hơn sẽ luôn nằm trước các phần tử nằm phân lớp lớn hơn. Sau đó, với mỗi phân lớp, ta thực hiện thuật toán sắp xếp chèn để sắp xếp các phần tử trong phân lớp đó đúng vị trí. Tính đúng đắn của thuật toán được hiểu một cách dễ dàng như cách hoạt động của thuật toán sắp xếp theo cơ số.

b) Cài đặt và mã giả

Cài đặt thuật toán này được nêu khá rõ ràng trong phần ý tưởng ở trên.

Vấn đề ta cần quan tâm là chọn giá trị m nào sao cho thuật toán thực hiện một cách nhanh nhất. Qua thực nghiệm, người ta thấy rằng giá trị $m = 0{,}42n$ sẽ cho thời gian thực hiện tối ưu nhất. Việc phân lớp ta sẽ thực hiện tương tự với thuật toán sắp xếp đếm phân phối nhưng có sửa đổi một chút. Việc sử dụng thuật toán sắp xếp chèn cho việc sắp xếp các phần tử trong phân lớp là bởi vì qua thực nghiệm, người ta thấy rằng nó mang đến hiệu suất cao nhất trong trường hợp này.

Mã giả thuật toán được cho dưới đây.

```
m = 0.42 * n

// phân hoạch dãy đã cho thành m lớp

for i = 0 to n – 1

    k = (m – 1) * (a[i] – min) / (max – min)

    cnt[k] = cnt[k] + 1
```

```
for i = 1 to m – 1
    cnt[i] = cnt[i] + cnt[i – 1]
move = 0
i = 0
k = 0
while move < n
    while i >= cnt[k]
        i = i + 1
        k = (m – 1) * (a[i] – min) / (max – min)
    x = a[i]
    while i < cnt[k]
        k = (m – 1) * (x – min) / (max – min)
        cnt[k] = cnt[k] – 1
        exchange x with a[cnt[k]]
        move = move + 1
// thực hiện sắp xếp chèn đối với mỗi phân lớp
for k = 1 to m – 1
    for i = cnt[k] – 2 downto cnt[k – 1]
        x = a[i]
        j = i + 1
        while x > a[j]
            a[j – 1] = a[j]
            j = j + 1
        a[j – 1] = x
```

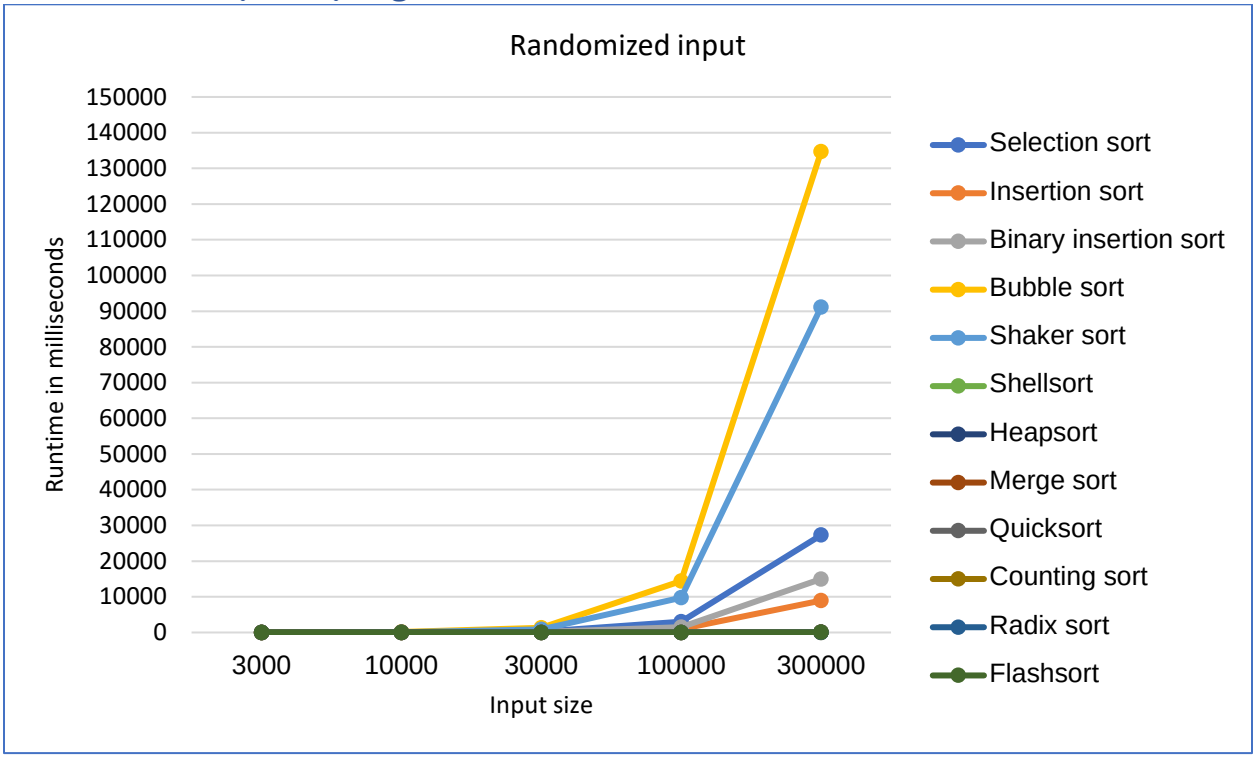
c) *Độ phức tạp*

Độ phức tạp của thuật toán trong trường hợp tốt nhất cũng như trung bình khi mà dữ liệu phân bố đều, mỗi phân lớp sẽ có kích thước xấp xỉ $\frac{n}{m}$. Mỗi lần thực hiện sắp xếp chèn có

độ phức tạp $O\left(\left(\frac{n}{m}\right)^2\right)$. Lúc này độ phức tạp thuật toán về mặt thời gian là $O\left(m\left(\frac{n}{m}\right)^2\right) = O\left(\frac{n^2}{m}\right) = O(n)$. Độ phức tạp bộ nhớ dùng cho việc phân lớp là $\Theta(m)$.

Thực nghiệm so sánh giữa các thuật toán sắp xếp

1. Đối với bộ dữ liệu ngẫu nhiên



a) Thuật toán nhanh nhất, chậm nhất

Qua số liệu thu được từ thực nghiệm, ta thấy rằng thuật toán sắp xếp đếm (counting sort) có thời gian thực hiện ngắn nhất (nhanh nhất), thuật toán sắp xếp nổi bọt (bubble sort) có thời gian thực hiện dài nhất (chậm nhất). Điều này được giải thích một cách dễ dàng từ lý thuyết. Thuật toán counting sort có độ phức tạp là $\Theta(n+k)$ trong mọi trường hợp và sử dụng rất ít thao tác nên hằng số lập trình rất nhỏ. Thuật toán bubble sort thì phải tốn $\Theta(n^2)$ lần so sánh, cùng với việc tráo đổi hai phần tử được thực hiện nhiều lần do đây là bộ dữ liệu ngẫu nhiên nên hằng số lập trình rất lớn khiến bubble sort chạy lâu.

b) Các cải tiến của các thuật toán

Trước hết, ta phải nói rằng vì đây là bộ dữ liệu ngẫu nhiên nên các thuật toán đều luôn thực thi với tốc độ trung bình. Do đó, một số cải tiến của các thuật toán có khả năng phát huy tác dụng.

Thuật toán shaker sort là một cải tiến của thuật toán bubble sort, đã thể hiện sự cải thiện về thời gian đáng kể so với bubble sort khi kích thước dữ liệu lớn dần. Và trên thực tế, đối với mọi bộ dữ liệu, thuật toán shaker sort luôn có thời gian thực hiện nhanh hơn bubble sort vì những cải tiến đã được ở phần giới thiệu thuật toán của báo cáo.

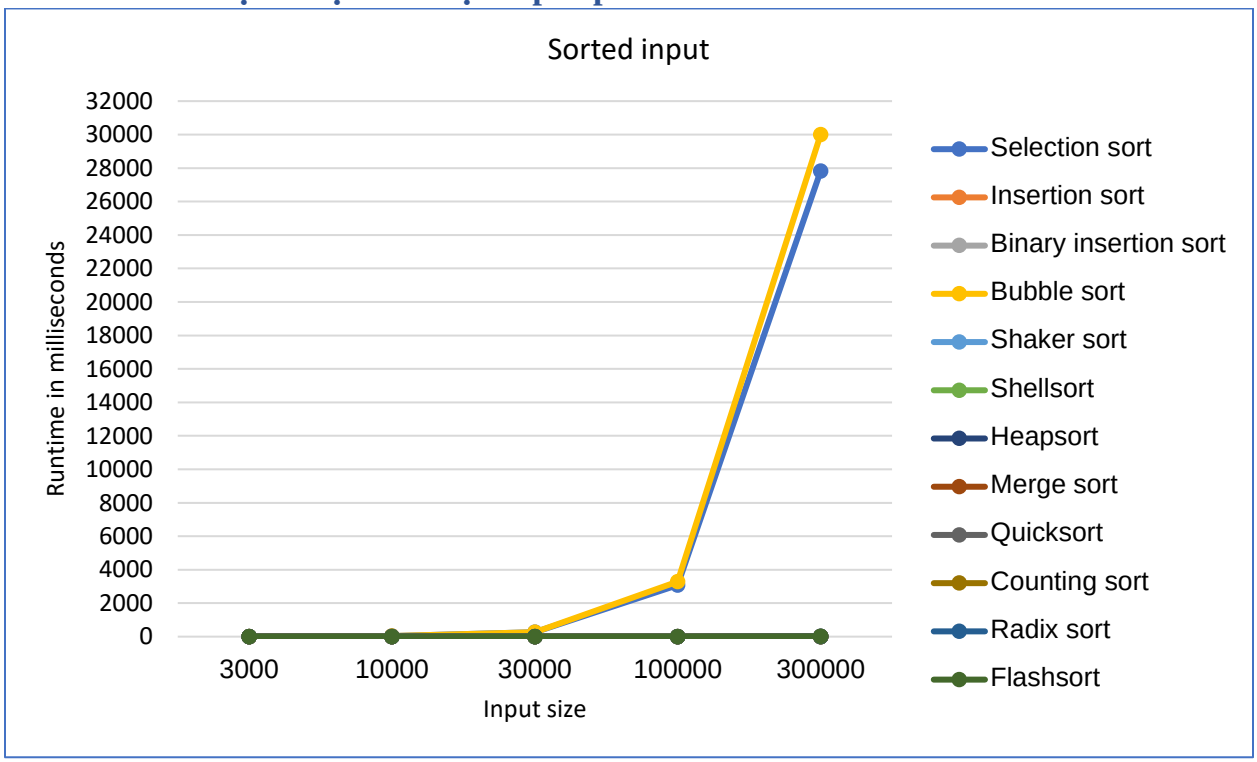
Thuật toán selection sort mặc dù cũng có độ phức tạp $\Theta(n^2)$ như bubble sort nhưng số lần tráo đổi chỉ là $O(n)$ nên có thời gian thực hiện nhỏ hơn hẳn so với bubble sort.

Thuật toán binary insertion sort không thể hiện tốc độ cao hơn insertion sort trong bộ dữ liệu này cũng như các bộ dữ liệu sau. Thuật toán này chỉ phát huy hiệu quả đối với dãy gồm các phần tử, mà thời gian so sánh các phần tử này là rất tốn thời gian. Ví dụ, nếu dãy các phần tử là các xâu thì thuật toán này hoạt động rất tốt, tốt hơn so với insertion sort thông thường do số thao tác so sánh được giảm đi nhờ tìm kiếm nhị phân. Nhưng 4 bộ dữ liệu ta thử nghiệm đều chỉ là các số nguyên, việc so sánh chúng là rất nhanh nên binary insertion sort không chỉ không nhanh hơn mà còn gây chậm so với insertion sort.

Một cải tiến khác của insertion sort là Shellsort. Shellsort đã thể hiện rất tốt ở bộ dữ liệu này khi có tốc độ tương đương với các thuật toán có độ phức tạp $O(n)$, $O(n \log n)$. Các bộ dữ liệu sau Shellsort cũng có tốc độ rất tốt mặc dù việc ta chỉ cài đặt dãy các bước nhảy một cách thông thường $\left(\left[\frac{n}{2^k}\right]\right)$.

Các lớp thuật toán có độ phức tạp trung bình $O(n \log n)$ và $O(n)$ như heapsort, merge sort, quicksort, counting sort, radix sort, flashsort đều được thực thi với tốc độ tương đương nhau và thật khó để so sánh chúng trong trường hợp này. Ta cần phải thử nghiệm chúng với kích thước dữ liệu lớn hơn hoặc thực hiện chúng nhiều lần mới thấy sự khác biệt. Tuy nhiên, về cơ bản, đối với bộ dữ liệu ngẫu nhiên những thuật toán này đều hoạt động tốt.

2. Đối với bộ dữ liệu đã được sắp xếp



Với bộ dữ liệu này, gần như các thuật toán được trình bày ở trên đều đạt được tốc độ nhanh nhất có thể với độ phức tạp trong trường hợp tốt nhất.

a) Thuật toán nhanh nhất, chậm nhất

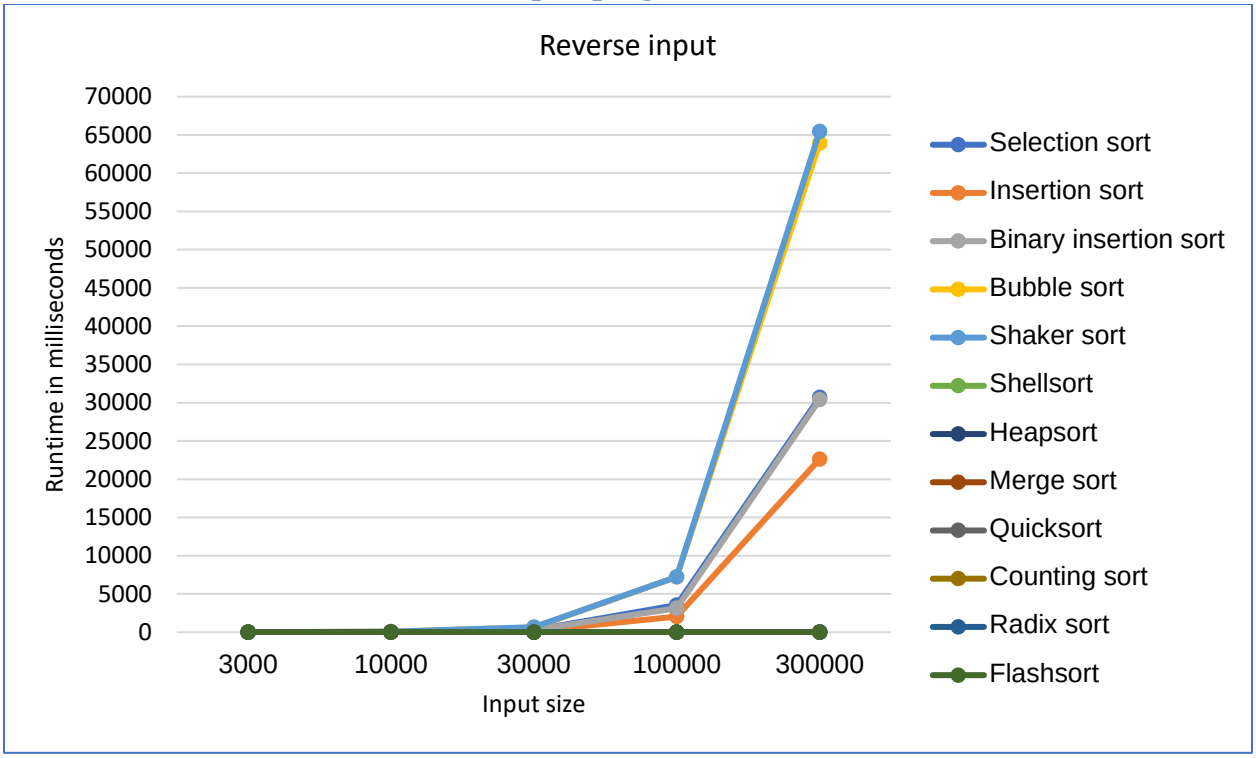
Lần này, hai thuật toán thực thi nhanh nhất dựa vào số liệu đo được là insertion sort và shaker sort. Với bộ dữ liệu đã được sắp xếp như vậy, cả hai thuật toán chỉ việc thực hiện $\Theta(n)$ lần so sánh và không cần đổi chỗ phần tử nào. Đó cũng là cách giải thích về lý do tại sao hai thuật toán này lại thực hiện nhanh chóng đến vậy, và ta gọi chúng là các thuật toán nhận biết được được dãy đã sắp. Như cũ thì thuật toán bubble sort vẫn là thuật toán chạy chậm nhất bởi nó vẫn phải thực hiện $\Theta(n^2)$ lần so sánh. Selection sort cũng là một thuật toán chạy chậm trong trường hợp này với nguyên nhân tương tự.

Các thuật toán còn lại đều đạt được tốc độ nhanh nhất có thể với độ phức tạp trong trường hợp tốt nhất. Cụ thể, các thuật toán đều không cần đổi chỗ phần tử nào cả; quicksort luôn chọn được chốt là trung vị (do cách cài đặt trong thử nhiệm này thì chốt được chọn là phần tử ở giữa); các phân lớp của flashsort được phân bố đều…

b) Các cải tiến của các thuật toán

Ngoại trừ shaker sort cải thiện được tốc độ so với bubble sort ra thì các thuật toán còn lại đều không thể cải tiến thêm về tốc độ trong trường hợp này bởi chúng đã được thực hiện trong trường hợp lý tưởng nhất.

3. Đối với bộ dữ liệu đã được sắp xếp ngược



Khác với những bộ dữ liệu ở trên, bộ dữ liệu này làm cho một số thuật toán thực thi với độ phức tạp trong trường hợp tốt nhất, một số thuật toán khác lại được thực thi với độ phức tạp trong trường hợp xấu nhất.

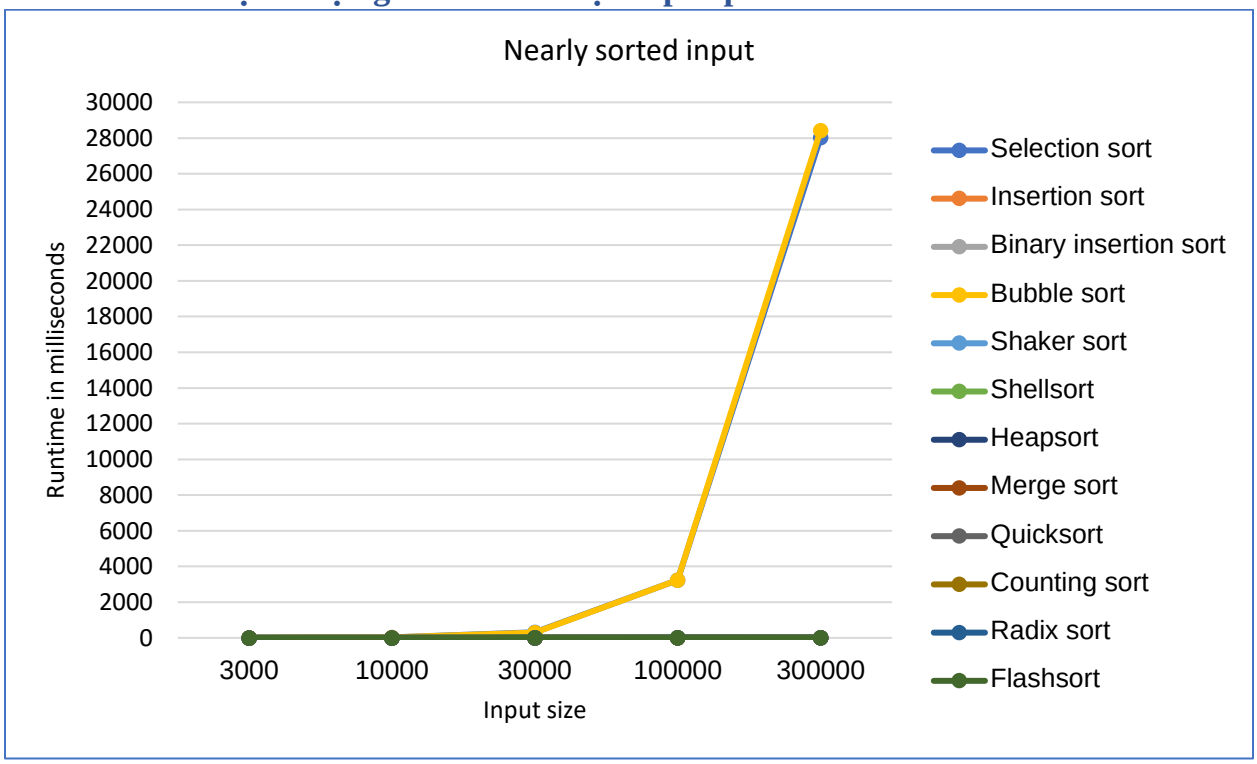
a) Thuật toán nhanh nhất, chậm nhất

Theo số liệu đo được, thuật toán counting sort lại trở thành thuật toán chạy nhanh nhất như cách nó đã thể hiện ở bộ dữ liệu ngẫu nhiên. Các thuật toán có độ phức tạp $O(n^2)$ đều chạy chậm ở bộ dữ liệu này bởi việc so sánh và đổi chỗ phần tử luôn được thực hiện. Đặc biệt, hai thuật toán bubble sort và shaker sort là những thuật toán chạy lâu nhất do hằng số lập trình lớn. Tuy nhiên, mặc dù luôn phải đổi chỗ các phần tử liên tục, bubble sort ở bộ dữ liệu này lại chạy nhanh hơn rõ rệt so với lúc chạy ở bộ dữ liệu ngẫu nhiên. Điều này được giải thích bởi cách thực thi mã của máy tính, khi biểu thức so sánh để đổi chỗ các phần tử của bubble sort (cũng như shaker sort) là luôn đúng khiến máy tính luôn đi vào đoạn mã đổi chỗ đã chuẩn bị sẵn giúp nó hoạt động nhanh hơn.

b) Các cải tiến của các thuật toán

Các cải tiến của các thuật toán hầu như không đạt được hiệu quả ở bộ dữ liệu này. Các thuật toán heapsort, quicksort, flashsort đều được thực hiện với độ phức tạp trong trường hợp tốt nhất. Các thuật toán insertion sort, binary insertion sort được thực hiện với độ phức tạp trong trường hợp xấu nhất. Shellsort là thuật toán duy nhất trong trường hợp này vẫn hoạt động tốt mặc dù nó là cải tiến của insertion sort. Các thuật toán có độ phức tạp tổng quát là $O(n)$ và $O(n \log n)$ vẫn hoạt động tốt như thường lệ.

4. Đối với bộ dữ liệu gần như đã được sắp xếp



Biểu đồ này có vẻ gần giống với biểu đồ đối với bộ dữ liệu đã được sắp xếp bởi hai bộ dữ liệu này gần giống nhau làm các thuật toán thực thi với số thao tác cũng gần giống nhau.

a) Thuật toán nhanh nhất, chậm nhất

Như nhận xét ở trên, các thuật toán chạy nhanh nhất và chậm nhất ở đây cũng giống với bộ dữ liệu đã được sắp xếp: counting sort là thuật toán chạy nhanh nhất; bubble sort và selection sort là hai thuật toán chạy chậm nhất.

b) Các cải tiến của các thuật toán

Cũng như đối với bộ dữ liệu đã được sắp xếp, thuật toán shaker sort ở trường hợp này cũng cải thiện rõ rệt tốc độ so với thuật toán bubble sort và tốc độ đó tương đương với các thuật toán có độ phức tạp $O(n)$. Các thuật toán còn lại đều được thực thi với tốc độ gần như trong trường hợp lý tưởng nhất.

Đánh giá và kết luận

1. Đánh giá chung về độ phức tạp của các thuật toán sắp xếp

a) Lớp các thuật toán có độ phức tạp $O(n^2)$

Trong mọi trường hợp thử nghiệm ở trên, thuật toán bubble sort luôn là thuật toán có thời gian thực thi lâu nhất. Các thuật toán có độ phức tạp lớn, trong trường hợp xấu nhất là $O(n^2)$ thường là những thuật toán dễ cài đặt (selection sort, insertion sort, bubble sort). Tuy vậy, mặc dù Shellsort cũng là một thuật toán dễ cài đặt nhưng vẫn đảm bảo tốc độ cao trong nhiều bộ dữ liệu nên đây là một thuật toán phù hợp trong rất nhiều trường hợp.

b) Lớp các thuật toán có độ phức tạp $O(n \log n)$

Các thuật toán heapsort và merge sort cho độ phức tạp trong trường hợp xấu nhất cũng như trung bình là $O(n \log n)$ cho tốc độ cao, chạy được với nhiều loại dữ liệu chứ không chỉ là số nguyên.

Thuật toán quicksort cho độ phức tạp trong trường hợp trung bình giống như heapsort và merge sort là $O(n \log n)$ và cũng chạy được với nhiều loại dữ liệu khác nhau. Nhưng trong trường hợp xấu nhất khi phần tử chốt được chọn luôn là giá trị nhỏ nhất hoặc nhỏ nhất trong đoạn đang xét thì thuật toán sẽ có độ phức tạp là $\Theta(n^2)$. Tuy nhiên, ta vẫn có thể khắc phục nhược điểm này bằng cách chọn chốt một cách ngẫu nhiên từ cái giá trị trong đoạn hoặc bằng một số phương pháp khác. Điều này giúp quicksort tốt hơn heapsort và merge sort trong hầu hết các trường hợp khi mà qua thực nghiệm quicksort luôn đạt tốc độ cao hơn so với heapsort và merge sort nhờ hằng số lập trình nhỏ. Một điểm cộng cho quicksort nữa là cách cài đặt có phần dễ hơn, ngắn gọn hơn so với heapsort và merge sort cùng với việc không tốn thêm bộ nhớ như merge sort.

c) Lớp các thuật toán có độ phức tạp $O(n)$

Ba thuật toán counting sort, radix sort, flashsort đều chỉ hoạt động trên tập dữ liệu là các số nguyên. Với tập các số nguyên có giới hạn nhỏ, counting sort trở nên là một thuật toán "không có đối thủ" khi luôn là thuật toán chạy nhanh nhất. Với tập các số nguyên có giới hạn lớn, ta cần phải dùng đến radix sort hoặc flashsort. Radix sort tỏ ra là thuật toán tốt hơn hẳn flashsort khi mà có độ phức tạp trong trường hợp xấu nhất vẫn rất nhanh là $\Theta(nd)$ như đã giải thích ở phần giới thiệu. Trong trường hợp trung bình, qua các thử nghiệm, radix sort vẫn là thuật toán chạy nhanh hơn so với flashsort trong đa số trường hợp. Theo cảm nhận của em, radix sort cũng là thuật toán dễ cài đặt hơn so với flashsort. Nhược điểm của cả ba thuật toán này là đều phải tốn thêm một lượng bộ nhớ đáng kể.

2. Tính ổn định của các thuật toán sắp xếp

Một thuật toán sắp xếp được gọi là ổn định nếu nó bảo toàn được thứ tự tương đối ban đầu của các phần tử mang giá trị được xem là bằng nhau trong dãy ban đầu.

Dựa vào cách hoạt động cũng như cách cài đặt, ta có thể phân chia các thuật toán sắp xếp trên thành hai loại ổn định và không ổn định. Các thuật toán sắp xếp tổng quát không ổn

định vẫn có thể biến đổi nó thành ổn định bằng cách tốn thêm $\Theta(n)$ bộ nhớ cho việc lưu trữ chỉ số của các phần tử để việc sắp xếp có thể dựa thêm thông tin đó.

- Các thuật toán sắp xếp ổn định: selection sort, insertion sort, binary insertion sort, bubble sort, shaker sort, merge sort, counting sort, radix sort;
- Các thuật toán sắp xếp không ổn định: Shellsort, heapsort, quicksort, flashsort.

Các thuật toán sắp xếp có tính ổn định luôn có một lợi thế nhất định khi sử dụng so với các thuật toán không có tính chất đó. Ví dụ, trong thuật toán radix sort, nhờ tính ổn định của counting sort nên radix sort mới có thể hoạt động chính xác. Trong một số trường hợp thực tế, khi có những yêu cầu về tính ổn định và giới hạn bộ nhớ, các thuật toán sắp xếp ổn định lại khẳng định thêm vai trò quan trọng của mình.

3. Kết luận

Nói chung, những kết quả về thời gian cũng như tốc độ của các thuật toán sắp xếp trên chỉ là thử nghiệm trên những bộ dữ liệu cụ thể và không thể đánh giá bao quát hết mọi trường hợp được. Mỗi thuật toán có thể chạy nhanh với bộ dữ liệu này nhưng cũng có thể chậm hơn rõ rệt với bộ dữ liệu khác cùng kích thước. Các con số thực nghiệm trên cũng phụ thuộc vào công cụ lập trình cụ thể và máy tính cụ thể được sử dụng do sự tối ưu của trình biên dịch, hệ điều hành hay phần cứng của mỗi máy tính là khác nhau. Với bộ dữ liệu khác, công cụ lập trình khác hay máy tính khác thì con số thực nghiệm có thể khác. Tuy nhiên, ta vẫn có thể đánh giá chính xác đối với các thuật toán có tốc độ chênh lệch rõ rệt như những nhận xét đã nêu ở trên.

Tùy vào mục đích sử dụng, đầu vào dữ liệu cũng như những yêu cầu của bài toán đặt ra mà chúng ta sẽ sử dụng thuật toán nào cho phù hợp. Với những bộ dữ liệu có kích thước nhỏ, chúng ta có thể sử dụng các thuật toán đơn giản, dễ cài đặt như selection sort, insertion sort hay Shellsort. Đối với đầu vào là dãy số nguyên có kích thước lớn, ta cần phải sử dụng các thuật toán như counting sort, radix sort, flashsort để việc sắp xếp được đảm bảo thời gian. Các thuật toán sắp xếp tổng quát mà vẫn nhanh như heapsort, merge sort, quicksort được sử dụng rộng rãi trong thực tế bởi tính di động của nó. Heapsort hay quicksort là sự lựa chọn tốt hơn đối với những bài toán yêu cầu về giới hạn bộ nhớ. Theo thực nghiệm, quicksort được xem là một thuật toán có hiệu suất cao nhất so với hai thuật toán còn lại. Bởi lẽ này mà quicksort được sử dụng nhiều hơn cả trong thực tế. Với yêu cầu về tính ổn định, ta cũng cần phải chọn thuật toán phù hợp để đạt hiệu suất cao nhất, ví dụ như là merge sort.

Ghi chú

- Trong tệp DataGenerator.cpp, em đã sửa đổi hàm `rand()` để số ngẫu nhiên được hàm này trả về có giá trị đảm bảo lớn hơn 300000, khác với hàm mặc định có giá trị trả về lớn nhất đạt `RAND_MAX` (thông thường là 32767).
- Trong cài đặt thuật toán Shellsort, em đã sử dụng dãy các bước nhảy độ dài lần lượt là $\left[\frac{n}{2}\right], \left[\frac{n}{4}\right], \dots, 1$.
- Trong cài đặt thuật toán quicksort, em đã chọn phần tử làm chốt là phần tử ở giữa đoạn do các bộ dữ liệu chúng ta đang xét đều là ngẫu nhiên, đã được sắp xếp không giảm (không tăng) hoặc là gần như đã được sắp xếp. Với việc chọn chốt như vậy, trong các bộ dữ liệu này, thuật toán luôn đạt được tốc độ thực thi trung bình hoặc tốt nhất.
- Trong cài đặt thuật toán radix sort, để đảm bảo thể hiện rõ cách thuật toán thực hiện và với điều kiện (kích thước $n \leq 300000$ và $0 \leq a_i < n$) của các bộ dữ liệu, em đã sử dụng hệ cơ số $R = 2^{16} = 65536$.
- Trong cài đặt thuật toán flashsort, em đã sử dụng giá trị $m = 0{,}42n$ theo cách giải thích đã nói ở phần giới thiệu của thuật toán.
- Các thử nghiệm để đưa ra những con số, kết quả trên được thực hiện bằng trình biên dịch GCC cho C++ và trên hệ điều hành Windows.

Các bảng số liệu thực nghiệm

Các số liệu dưới đây đều có đơn vị là mili giây.

1. Thời gian thực nghiệm đối với bộ dữ liệu ngẫu nhiên

Thuật toán \ Kích thước dữ liệu	3000	10000	30000	100000	300000
Selection sort	4	32	275	2987	27275
Insertion sort	1	10	85	939	8930
Binary insertion sort	2	18	139	1497	14979
Bubble sort	14	148	1311	14399	134656
Shaker sort	5	80	865	9783	91131
Shellsort	0	1	2	12	40
Heapsort	1	0	3	10	34
Merge sort	1	0	2	8	26
Quicksort	1	1	2	7	23
Counting sort	0	0	1	1	3
Radix sort	1	0	0	3	8
Flashsort	1	1	1	6	19

2. Thời gian thực nghiệm đối với bộ dữ liệu đã được sắp xếp

Thuật toán \ Kích thước dữ liệu	3000	10000	30000	100000	300000
Selection sort	3	30	270	3078	27814
Insertion sort	0	0	0	0	0
Binary insertion sort	0	0	2	2	9
Bubble sort	3	31	274	3287	30000
Shaker sort	0	0	0	0	0
Shellsort	0	0	0	2	5
Heapsort	0	0	2	6	18
Merge sort	0	0	1	2	5
Quicksort	0	0	0	1	4
Counting sort	0	0	0	0	1
Radix sort	0	0	0	1	5
Flashsort	1	0	1	4	10

3. Thời gian thực nghiệm đối với bộ dữ liệu đã được sắp xếp ngược

Thuật toán \ Kích thước dữ liệu	3000	10000	30000	100000	300000
Selection sort	3	32	295	3541	30681
Insertion sort	2	19	168	2055	22618
Binary insertion sort	2	30	269	3107	30386
Bubble sort	7	72	638	7242	63914
Shaker sort	6	70	646	7231	65441
Shellsort	0	0	1	3	10
Heapsort	0	1	2	6	21
Merge sort	0	0	0	1	5
Quicksort	0	0	0	1	5
Counting sort	0	0	0	1	1
Radix sort	0	0	0	1	5
Flashsort	0	0	1	3	10

4. Thời gian thực nghiệm đối với bộ dữ liệu gần như đã được sắp xếp

Thuật toán \ Kích thước dữ liệu	3000	10000	30000	100000	300000
Selection sort	3	29	321	3244	28024
Insertion sort	0	0	0	0	1
Binary insertion sort	0	0	1	3	10
Bubble sort	3	30	292	3234	28427
Shaker sort	0	0	1	1	3
Shellsort	0	0	1	3	9
Heapsort	0	0	2	5	18
Merge sort	0	0	0	1	5
Quicksort	0	0	1	2	4
Counting sort	0	0	0	1	1
Radix sort	0	0	0	2	5
Flashsort	0	0	1	3	10