Counting Sort

Table of contents

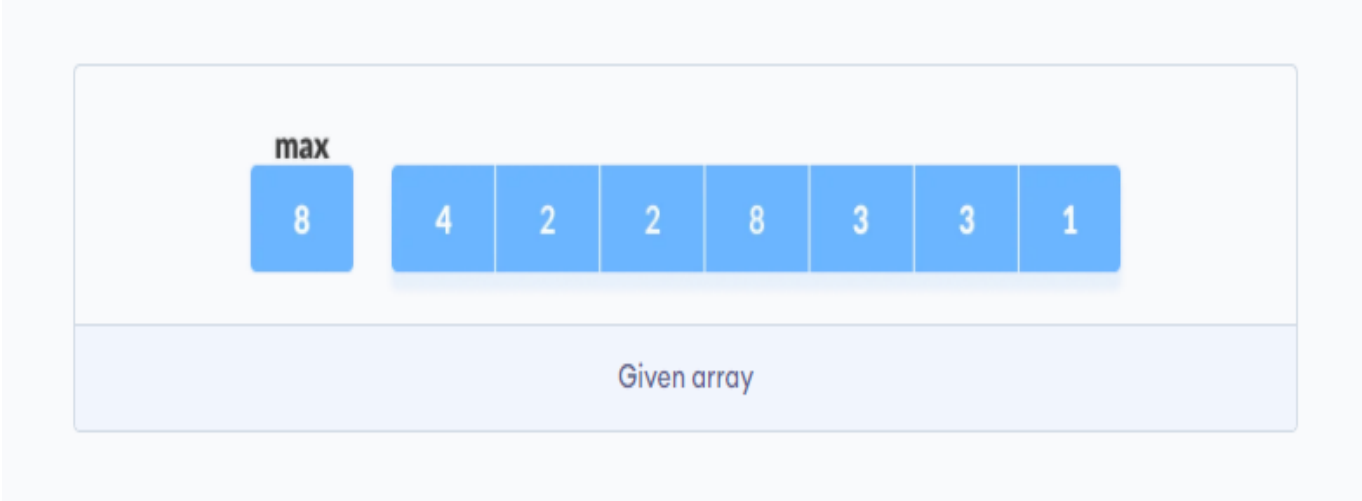
(Hình ảnh và nội dung tham khảo từ programmingquiz và journaldev).

Idea

Counting Sort là thuật toán sắp xếp mảng bằng cách đếm số lần xuất hiện của những phần tử riêng biệt có trong mảng. Số lần đếm được này được lưu trong mảng phụ và thuật toán kết thúc khi ánh xạ các số lần đếm này thành index của mảng.

Các bước thực hiện:

1. Tìm phần tử lớn nhất trong mảng, gọi giá trị này là `max`



- Độ phức tạp thời gian $O(n)$, không gian $O(1)$.

2. Khởi tạo một mảng có độ dài `max + 1` với mọi giá trị đều là 0. Mảng này dùng để lưu các giá trị đếm số lần xuất hiện trong mảng của các phần tử.

- Độ phức tạp thời gian và không gian $O(max)$.

3. Tiến hành đếm số lần xuất hiện của giá trị `k` bất kỳ và cho vào vị trí `count[k]` của mảng đếm.

- Độ phức tạp thời gian $O(n)$, không gian $O(1)$.

4. Thực hiện cộng tích lũy các phần tử trong mảng đếm, phục vụ cho việc ánh xạ sang mảng chính để sắp xếp. Với `count[k] = count[k] + count[k - 1]` (Tại sao nó phục vụ cho việc ánh xạ ?).

- Độ phức tạp thời gian $O(max)$, không gian $O(1)$.

Chú ý rằng giá trị cuối mảng `count[]` sau khi tính tích lũy luôn là tổng giá trị có trong mảng `count[]` trước khi tính tích lũy.

> Nếu thực hiện Counting Sort sắp xếp giảm dần, chỉ cần tính tích lũy theo chiều ngược lại.

5. Tìm vị trí của phần tử mảng, chẳng hạn 4 thông qua mảng đếm và ánh xạ đến đúng vị trí của nó (có trừ cho 1 ?).

- Độ phức tạp thời gian $O(n)$, không gian $O(n)$ (do cần chép qua một mảng phụ).

6. Sau khi ánh xạ thì giảm giá trị của `count[4]` đi 1.

- Độ phức tạp thời gian $O(n)$, không gian $O(1)$.

Properties

Không dựa vào các phép so sánh như các thuật toán thông thường.

Điểm mạnh

- Stable Sort.
- Độ phức tạp tuyến tính rất nhanh.
- Giảm không gian phụ nếu khoảng giá trị là nhỏ (nhiều tần số).

Điểm yếu

- Cả độ phức tạp thời gian và không gian tăng đáng kể nếu dữ liệu đầu vào lớn và khoảng giá trị rộng.
- Chỉ hoạt động cho các giá trị rời rạc như số nguyên.
- Nếu dùng cho số âm thì Complexity và cài đặt đều tăng.
- Tiêu tốn nhiều bộ nhớ không cần thiết nếu dữ liệu phân bố thành cụm (chẳng hạn như dữ liệu có 1000 phần tử và có 1000 phần tử giá trị 1. Lúc này mảng đếm có 1000 vị trí nhưng 999 vị trí trong số đó là trống).

Complexity Analysis

Như đã ghi chú ở phần Idea, các vòng for ở mỗi có độ phức tạp thời gian lần lượt là:

1. $O(n)$
2. $O(max)$
3. $O(n)$
4. $O(max)$
5. $O(n)$
6. $O(n)$

Tổng độ phức tạp thời gian là $O(n + max)$. Cả ba cases đều có độ phức tạp thời gian là $O(n + max)$, vì cho dù dữ liệu đầu vào có như thế nào, nó vẫn duyệt qua `n + max` lần. Điều này cho thấy đây vẫn là một thuật toán không phụ thuộc dãy đầu vào.

Tuy nhiên độ phức tạp không gian là khá cao, $O(max)$.

Complexity

Cases	Complexity
Best case	$O(n + max)$
Worst case	$O(n + max)$
Average case	$O(n + max)$

Space Complexity: $O(max)$.

với max là giá trị lớn nhất của các phần tử.

Code

```
void countSort(int array[], int size)
{

    int output[10];
    int count[10] = {0}; //Đã khởi tạo mảng đếm
```

```
        int max = array[0];

        // Tìm phần tử lớn nhất
        for (int i = 1; i < size; i++)
        {
            if (array[i] > max)
                max = array[i];
        }

        // Đếm tần số xuất hiện của các phần tử
        for (int i = 0; i < size; i++)
        {
            count[array[i]]++;
        }

        // Đếm giá trị tích lũy
        for (int i = 1; i <= max; i++)
        {
            count[i] += count[i - 1];
        }

        // Ánh xạ giá trị qua mảng tạm
        for (int i = size - 1; i >= 0; i--)
        {
            output[count[array[i]] - 1] = array[i];
            count[array[i]]--;
        }

        // Chép mảng tạm vào mảng chính
        for (int i = 0; i < size; i++)
        {
            array[i] = output[i];
        }
    }
```