Timmy Nord
2022-08-14

Counting sort

Counting sort är en sorteringsalgoritm. Algoritmen används för att sortera element givet i ett specifikt intervall och detta görs med hjälp av att räkna antalet unika förekommande element i array:en. Frekvensen av antalet unika element lagras därefter i en ytterligare array , där indatat sorteras genom att mappa räkningen av unika element till ett index för en sorterad array.
Två egenskaper Counting sort har är följande:

- Out-of-place
- Stable

Out-of-place innebär att sorteringsalgoritmen behöver extra resurser för att sortera indata. Detta görs i Counting Sort när man allokerar en array av med storlek av största elementet i indatat.
Stable innebär att Counting Sort tar hänsyn till att bevara ordningen av lika eller upprepade element relativt till deras position i array:en.

Tidskomplexitet

Average case time complexity

Counting sort har en generell tidskomplexitet på **O(N + K).**
N = antal element, indatat.
K = största talet i array:en, bildar en talserie från 0 -> … **K**

Counting Sort är generellt en effektiv sorteringsalgoritm därmed average case under villkoret att **K** inte är mycket större än **N**, vilket ger **O(N + K).**

Best case time complexity

uppnås när alla element i indatat har samma maxstorlek av 1. Att räkna antalet förekommande element tar då konstant tid och att mappa räkningen av unika element till index tar **N** mycket tid, vilket ger oss en tidskomplexitet på **O(N)**. Det betyder att tiden är linjär.

Worst case time complexity

uppnås när **K** är mycket större än storleken av indatat **N**, vilket ger en tidskomplexitet på **O(K)**.
Exempel: Låt **N** = 1000 och **K** = 1 000 000. Vi har en array av storleken 1000 att sortera och det största elementet är 1 000 000. Vi kommer därmed att skapa och iterera genom en array av storlek 1 000 000 => **K** = 10^6 = **N**^2.

Implementation

1) Vi börjar med att skapa upp två arrayer, **output** som kommer vara vår sorterade array, därefter hittar vi det största elementet i vår osorterade array **arr** och initierar vår tredje array **count** med största elementet **max** + 1

```
void counting_sort(std::vector<int>* arr)
{
    std::vector<int> output(arr->size() + 1);
    int max = *(std::max_element(arr->begin(), arr->end()));
    std::vector<int> count(max + 1);
```

2) Därefter tilldelar / nollställer vi vår count array.
För att då räkna antal förekomster av ett unikt element.
Detta görs för att vi ska tilldela indexet för resp. tal i array:en.

```
for (int i = 0; i <= max; ++i)
{
    count[i] = 0;
}

for (int i = 0; i < arr->size(); i++)
{
    count[arr->at(i)]++;
}
```

3) Sedan adderar vi alla antalet räknade element kumulativt, givet i första for-loopen.
Därefter i den andra for-loopen börjar vi på det sista elementet (“det högra elementet”) och tilldelar talen i den nya arrayen samt minskar antalet unikt räknade tal med ett för varje iteration.
Genom att börja längst till höger och iterera mot vänster gör algoritmen stabil, för att ifall det fanns någon sorts ordning sedan innan på talen (lika tal) som gör de särskiljbar så består det under denna iteration.

```
for (int i = 1; i <= max; i++)
{
    count[i] += count[i - 1];
}

for (int i = arr->size() - 1; i >= 0; i--)
{
    output[count[arr->at(i)] - 1] = arr->at(i);
    count[arr->at(i)]--;
}
```

4) i sista steget överskrider vi får originella osorterade array med den sorterade vi fick i från steg 3.

```
    for (int i = 0; i < arr->size(); i++)
    {
        arr->at(i) = output[i];
    }
}
```

Exempel:
A[] = { 1,4,4,1,6,5 }
output(size+1) = { x, x, x, x, x, x, x}
max = 6
count(max+1) = { 0, 0, 0, 0, 0, 0, 0}
Hitta unika element:
{ [0], [1], [2], [3], [4], [5], [6] }
0 2 0 0 2 1 1
kumulativt:
{ [0], [1], [2], [3], [4], [5], [6] }
0 2 2 2 4 5 6
tilldela output och minska med 1:
{ [0], [1], [2], [3], [4], [5], [6] }
1 1 4 4 5 6 0
iterera till originala array:
{ [0], [1], [2], [3], [4], [5] }
1 1 4 4 5 6

Data och tidsmätningar

indatat var slumpmässigt vald mellan intervallet av den initiala array-storleken till storleken multiplicerat med antal repetitioner, i detta fall: 500 000 * 10 = 5 000 000.
Repetitioner = 10

N	T[ms]	dev[ms]	Samples
500000	37009.5	5949.93	100
1000000	32456	833.884	100
1500000	32473.2	867.775	100

2000000	33134.9	2085.22	100
2500000	33050.6	2090.02	100
3000000	33501	2925.49	100
3500000	33939.4	2654.29	100
4000000	34677.1	4802.41	100
4500000	33465.7	2979.8	100
5000000	33471.3	2571.3	100