2022

Verslag – Bucket Sort

IN DIT DOCUMENT WORD ER BESCHREVEN WAT DE TIJD EN RUIMTE COMPLEXITEIT IS VAN DE BUCKET SORT.

CEYHUN ÇAKIR - 1784480

Inhoud

Inleiding

In deze document ga ik uitleggen wat de tijd en ruimte complexiteit is van de bucket sort algoritme. Ook ga ik uitleggen hoe ik dit heb gedaan en wat de uiteindelijke uitkomsten zijn.

Hoe heb ik het gemaakt?.

Ik ben begonnen met het analyseren van het opdracht. Na dat ik de opdracht had geanalyseerd ben ik begonnen met het opzetten van het environment. Toen ik klaar was met het opzetten van het opdracht ben begonnen met het aanmaken van het bucket sort functie. Voor dit functie hebben we een template aangewezen om vectoren binnen te krijgen met meerdere datatypes. In dit functie gaf ik twee parameters mee, de array die gesorteerd moest worden en de max element binnen die array.

```
template <typename T>
std::vector<T> bucket_sort(std::vector<T> array, int max_elem) {
```

Binnen de functie ben ik begonnen met het schrijven van een for loop die verlaagt vanaf de lengte van de max_elem tot aan 0. Aangezien we vanaf de 100 tallen naar de een tallen willen sorteren was dit noodzakelijk in het functie. Na dat ik de for loop had geschreven had ik ook een vector aangemaakt met buckets. Dit was voornamelijk noodzakelijk voor het sorteren van de getalen.

```
std::vector<std::vector<T>> buckets = {{}, {}, {}, {}, {}, {}, {}, {}, {}, {}};
```

Verder ben ik door gegaan met de onderdeel van distribution pass. In dit onderdeel werd er verwacht dat de meest rechtse cijfer (de een waarde) geplaats werd in een van de correspondente bucket. Ik deed dit door een for loop te schrijven die door de gegeven unsorted array heen loopt. Binnen de loop vereenvoudig ik de absolute waarde van de element tot de macht van 10. Dit doe ik zodat wanneer de index waarde 3 is en de waarde wat gepakt word een index van 2 heeft, dat het nog wel mogelijk is dat die ook in een bucket word gestopt. Uiteindelijk stop ik de element in een bucket door de meest rechtse waarde van het element als index te gebruiken.

```
for(int elements = 0; elements < array.size(); elements++) {
  // creert nullen voor de cijfers zodat wanneer er 3 getallen er zij
  int big_int = std::abs(array[elements]) * std::pow(10, index - 2);

  // we pushbacken de laatse integer als bucket en de element
  buckets[big_int % 10].push_back(array[elements]); //
}
```

Na de onderdeel van het distribution pass verwijder ik alle elementen binnen de array met unsorted elementen. dit doe ik zodat de gesorteerde nummers voor een ronde in een lege array komen. Toen alle elementen verwijdert waren ben ik begonnen met het maken van het gathering pass onderdeel. Hierin werd gevraagt dat je alle elementen binnen de buckets naar de originele array terug zet. Dit heb ik kunnen realiseren door twee for loops te schrijven die nested waren. Een for loop om door de buckets heen te gaan en een for loop om door de elementen binnen de bucket te gaan.

```
for(int bucket = 0; bucket < buckets.size(); bucket++) {
  // we itereren voor elke bucket in buckets om de getallen te krijgen.
  for(int integers = 0; integers < buckets[bucket].size(); integers++) {
    // we pushen uiteindelijk alle nummers die we steeds tegen komen in
    array.push_back(buckets[bucket][integers]);
  }
}
```

Uiteindelijk na de gathering pass onderdeel is de algoritme compleet. Als de waarde 0 is van de eerste for loop dan returnen we de array met de gesorteerde nummers.

Uitkomsten

Hieronder in de afbeelding zien we dat er 10 getallen gegenereerd zijn met een min en max waarde. We zien de ongesorteerde array en de uiteindelijke gesorteerde array.

```
desired amount of integers: 10
desired range of random generated numbers (min/max) -9999/9999
given array - -9958 | 8468 | -3665 | -3497 | 9170 | 5725 | 1479 | -639 | -3035 | -5533 |
sorted array - -9958 | -5533 | -3665 | -3497 | -3035 | -639 | 1479 | 5725 | 8468 | 9170 |
```

Testcases

Hieronder in de afbeelding zien we de worst naar best case scenario's waar de algoritme op heeft afgedraaid. We zien bijvoorbeeld dat bij de worst case scenario een meerwaarde heeft van 39 milliseconde wat te verwachte was aangezien de elementen die gesorteerd moeten worden omgekeerd staan.

Verder zien we dat bij de average case scenario een run time had van 38 seconden wat ongeveer zelfde runtime had als de best case scenario. De elementen die gesorteerd moesten worden in de average case scenario zijn gerandomiseerd en bij de best case als gesorteerd was.

```
----------User inputted outcome---------
given array - -159 | -133 | 134 | -100 | 169 | -76 | 78 | -42 | -38 | -136 |
sorted array - -159 | -136 | -133 | -100 | -76 | -42 | -38 | 78 | 134 | 169 |
Time taken by function: 49 microseconds

----------Algorithm sorted validation checkt---------
de uitkomst van de eigen gemaakte bucket sort tegenover de sorting algoritme van c++: True

----------Worst case scenario of the bucket sort---------

given test array - -10 | -20 | -400 | -4000 | -15000 | 110000 | 20000 | 15000 | 4000 | 400 |
sorted test array - -15000 | -4000 | -400 | -20 | -10 | 400 | 4000 | 15000 | 20000 | 110000 |
Time taken by function: 41 microseconds

----------Average case scenario of the bucket sort---------

given test array - -63211 | 27079 | 88632 | -72690 | 10250 | 25437 | -17285 | 85871 | -53096 | 13023 |
sorted test array - -72690 | -63211 | -53096 | -17285 | 10250 | 13023 | 25437 | 27079 | 85871 | 88632 |
Time taken by function: 39 microseconds

----------Best case scenario of the bucket sort---------

given test array - -91243 | -87564 | -68757 | -2186 | 14406 | 20622 | 23253 | 31020 | 76913 | 89908 |
sorted test array - -91243 | -87564 | -68757 | -2186 | 14406 | 20622 | 23253 | 31020 | 76913 | 89908 |
Time taken by function: 39 microseconds
```

Tijd complexiteit

We beginnen met de eerste for loop waar we als operations een n pakken. Aangezien de algoritme binnen de for loop gebeurt doen we op het moment een (n *). We zien een variabele die buckets heet. Aangezien we via internet zagen dat variabele declaraties een big o operations hebben van O(1). Dus hierbij geven de variabele buckets een O(1). Na de variabele zien we weer een for loop waar een variabelen in zit met een pushback.

We zetten in de eerste plek een n voor de for loop. Binnen de for loop zetten we twee keer een 1 aangezien alle variabele declaraties en de pushbacks een 1 krijgen. Na dat krijgen we een vector.clear(). Voor dit operations functie geld de notatie n. dat de clear functie kregen we de gathering pass.

In de gathering pass word er twee for loops uitgevoerd wat er voor zorgt dat elke element uiteindelijk terug gestopt word in de originele array. Voor dit operations kunnen we in geheel n zeggen. Uiteindelijk geven we de array terug, als het gesorteerd is. En dit functie operation geven we een 1. De totale formule gaat als volgt: n * (1 + n * (1 + 1) + n + n + 1)

n * (1 + n * (1 + 1) + n + n + 1) // de gehele formule

n * (2n + n + n) // we halen alle constante weg

n * (4n) // 2n + n + n maakt 4n

n * (n) // 4n kunnen we versimpelen naar n

O(n ^ 2) // n * n kunnen we versimpelen naar n ^ 2

```
template <typename T>
std::vector<T> bucket_sort(std::vector<T> array, int max_elem) {

  // initialiseren we een lijst met vectoren er in waar integers in zitten.

  for(int index = 0; index <= max_elem; index++) {        n

    // we creeren 9 lege buckets
    std::vector<std::vector<T>> buckets = {{}, {}, {}, {}, {}, {}, {}, {}, {}, {}};    1
    // we itereren door de lijst met getallen die gesorteerd moeten worden.
    for(int elements = 0; elements < array.size(); elements++) {   n          n * (1 + 1)

      // creert nullen voor de cijfers zodat wanneer er 3 getallen er zijn maar de index is
      int big_int = std::abs(array[elements]) / (int) std::pow(10, index);
                                                                          1
      // we pushbacken de laatse integer als bucket en de element
      buckets[big_int % 10].push_back(array[elements]);  1
    }

    // we clearen de array zodat wanneer de nieuwe in komen dat de oude run er niet in zit.
    array.clear();    n

    for(int bucket = 0; bucket < buckets.size(); bucket++) {                    n
        // we itereren voor elke bucket in buckets om de getallen te krijgen.
        for(int integers = 0; integers < buckets[bucket].size(); integers++) {
          // we pushen uiteindelijk alle nummers die we steeds tegen komen in de buckets naar
          array.push_back(buckets[bucket][integers]);
        }
    }
  }
  return array;    O(1)
}
```

Ruimte complexiteit

De ruimte complexiteit voor de bucket sorting algoritme is in dit geval in de worst case:

Array: O(n)

max_elem: O(1)

buckets: 10n = O(n)

we geven de array een O(n) aangezien dit linear is. De max element geven we een O(1) aangezien dit een constante is wat aan duid dat een integer 4 bytes is en als we 4 x 3 doen is dat 12 bytes wat wilt zeggen dat we geen extra space nodig hebben wat zegt dat het een O(1) is. Uiteindelijk krijg de buckets een 10n aangezien er ook maar 10 buckets er in zitten. We kunnen dit versimpelen naar een O(n).