Bucket Sort

Student	Stan Merlijn
Email	stan.merlijn@student.hu.nl
Studentennummer	1863967
link github	https://github.com/StanMerlijn/HPP-bucket-Sort.git

1 Inleiding

In deze opdracht wordt het bucket sort-algoritme geïmplementeerd. Voor negatieve positieve getallen en comma getallen. Het doel van deze opdracht is om het bucket sort-algoritme te implementeren en te testen met verschilende groottes van arrays.

2 aanpak

2.1 1. Distribution pass

De eerste stap van het bucket sort-algoritme is de distribution pass. In deze stap wordt de input array verdeeld in verschilende buckets. Zowel voor de gehelle getallen als voor de kommagetallen is er een aparte functie geschreven.

De functies zijn hieronder te zien. het enige verschil tussen de twee functies is dat de `float` functie de input array eerst cast naar een int en daarna de digit berekent. De functie om de digit te bereken:

```
int digit = (num / int(std::pow(10, i))) % 10;
```

Listing 1: digit berekenen

Waar:

- num: het getal dat in het bucket moet worden geplaatst
- i: de index van de digit waar het getal op moet worden gesorteerd 0 is de laatste digit.

Deze functie haalt de digit van het getal op de i-de plaats. Dit geld voor gehelle getallen. Voor kommagetallen wordt het getal eerst gecast naar een `int` en daarna de digit berekent. Dit verliest wel de decimalen van het getal. Daarom worden de kommagetallen eerst geschaald met een factor. Met dit cijfer kan het getal in de overeenkomstige bucket geplaatst worden.

```
void distributePassInt(std::vector<int>& input, std::vector<std::vector<int>>& buckets, int
  i)
{
    for (int j = 0; j < input.size(); j++) {
        int num = input[j];
        int digit = (num / int(std::pow(10, i))) % 10;
        buckets[digit].emplace_back(num);
    }
}

void distributePassFloat(std::vector<float>& input, std::vector<std::vector<float>>& buckets
  , int i)
{
    for (int j = 0; j < input.size(); j++) {
        int num = static_cast<int>(input[j]);
        int digit = int(num / int(std::pow(10, i))) % 10;
        buckets[digit].emplace_back(num);
    }
}
```

Listing 2: Distribution pass

2.2 2. gathering pass

In de gathering pass worden de buckets weer samengevoegd tot één array. Deze functie is voor zowel gehele getallen als kommagetallen identiek, waardoor er gebruik wordt gemaakt van templates. Hiermee wordt van een 2D-array een 1D-array gecreëerd.

```cpp
template<typename T>
std::vector<T> gatheringPass(const std::vector<std::vector<T>>& input)
{
    std::vector<T> output;
    for (int i = 0; i < input.size(); i++) {
        for (int j = 0; j < input[i].size(); j++){
            output.emplace_back(input[i][j]);
        }
    }
  return output;
}
```

Listing 3: Gathering pass

2.3 3. Negatieve getallen

Om negatieve getallen te sorteren, worden de negatieve en positieve getallen eerst gescheiden. De negatieve getallen worden omgezet naar positieve waarden en vervolgens gesorteerd met behulp van het Bucket Sort-Algoritme; hetzelfde geldt voor de positieve getallen.

```cpp
// NOTE: dit is voor integers
std::vector<T> negatives, nonNegatives;
for (auto num : input) {
    if (num < 0) {
        // Store the absolute value for sorting the negatives
        negatives.push_back(-num);
    } else {
        nonNegatives.push_back(num);
    }
}
// NOTE: dit is voor floats ze worden geschaald met een factor
std::vector<T> negatives, nonNegatives;
for (auto num : input) {
    if (num < 0) {
        negatives.push_back(-num * scaleFactor);
    } else {
        nonNegatives.push_back(num * scaleFactor);
    }
}
```

Listing 4: Negatieve getallen

De originele negatieve vector wordt vervolgens omgedraaid, omdat de negatieve getallen (na omzetting naar positieve waarden) in omgekeerde volgorde zijn gesorteerd. Hierdoor staat het oorspronkelijk grootste (in absolute waarde) getal als eerste. Bijvoorbeeld: de getallen -1, -2 en -3 worden na omzetting en sortering als 3, 2, 1 gerangschikt. Daarna worden beide verzamelingen samengevoegd. Het samen voegen gebeurd zowel voor gehele getallen als kommagetallen op dezelfde manier.

```cpp
// Concatenate the negative and non-negative numbers
negatives.insert(negatives.end(), nonNegatives.begin(), nonNegatives.end());

// Divide each float by scaleFactor
for (int i = 0; i < negatives.size(); i++) {
    negatives[i] = negatives[i] / scaleFactor;
}
return negatives;
```

Listing 5: getallen samenvoegen

2.4 4. Comma getallen

Zoals eerder vermeld worden de kommagetallen van de input vector geschaald met een factor. Deze factor is op het moment 1000.0f dit betekent dat de kommagetallen worden geschaald met 3 decimalen. Dit kan er voor zorgen dat de kommagetallen precisie verliezen. Dit is een trade-off tussen snelheid en precisie.

```cpp
for (int i = 0; i < negatives.size(); i++) {
          negatives[i] = -negatives[i];
      }
```

Listing 6: Terug schalen

3 complexiteitsanalyse

Om het Bucket Sort-Algoritme te testen, is het algoritme met arrays van verschillende groottes uitgevoerd; voor elke grootte is het algoritme 100 keer gerund. De resultaten zijn in de onderstaande grafiek weergegeven. Deze grafiek toont aan dat het Bucket Sort-Algoritme een tijdcomplexiteit van $O(n)$ heeft.

Hieronder zie je de resultaten van de Bucket Sort-benchmark (CLI-output). 'Est. Run Time' geeft de geschatte totale uitvoeringstijd weer, 'Mean' is de gemiddelde tijd die het algoritme nodig heeft over de 100 uitvoeringen en 'Std Dev' geeft de standaardafwijking van de meetwaarden aan.

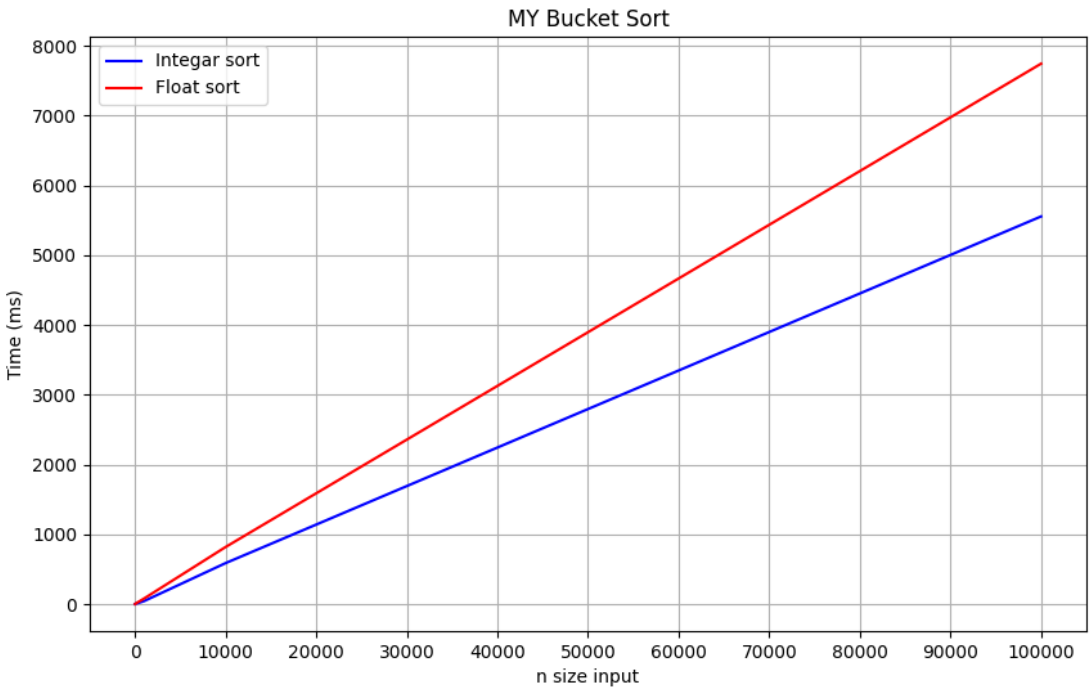
Tabel 1: Bucket Sort Benchmark Results: Large `ints` (detailed)

Benchmark	Iter.	Est. Run Time	Mean	Std Dev
Bucket Sort 10	2	4.4852 ms	22.4083 us	808.702 ns
Bucket Sort 100	1	7.7749 ms	78.0013 us	1.56197 us
Bucket Sort 1000	1	45.5004 ms	455.9 us	2.43898 us
Bucket Sort 10000	1	589.814 ms	5.89859 ms	28.6057 us
Bucket Sort 100000	1	5.55511 s	55.6646 ms	253.145 us

Tabel 2: Bucket Sort Benchmark Results: Large `floats` (detailed)

Benchmark	Iter.	Est. Run Time	Mean	Std Dev
Bucket Sort 10	1	3.3439 ms	33.6925 us	1.27137 us
Bucket Sort 100	1	12.4503 ms	126.133 us	2.4091 us
Bucket Sort 1000	1	82.2361 ms	811.04 us	3.01316 us
Bucket Sort 10000	1	819.995 ms	8.16308 ms	40.902 us
Bucket Sort 100000	1	7.74322 s	77.2987 ms	274.485 us

Uit deze resultaten is de est run time er uit gehaald en geplot tegen de grootte van de array:



Figuur 1: Complexity of the bucket sort-algoritme

4 Performance

- **Distributiepass:**
  Elke distributiepass scant het gehele inputarray en plaatst elk element in één van de 10 buckets. Dit levert een tijdcomplexiteit van O(n) per pass op.

- **Gathering pass:**
  In de gathering pass worden alle buckets samengevoegd in één vector. Dit kost ook O(n) per pass.

- **Aantal passes:**
  Het totale aantal passes wordt bepaald door het aantal cijfers (digit count) van het maximale element. In het slechtste geval (bijvoorbeeld voor grote aantallen) is dit $\log_{10}(\text{max})$ wat vrij beperkt blijft, zodat de overhead laag is.

- **Extra bewerkingen:**
  Voor negatieve getallen wordt er een extra stap uitgevoerd voor het omkeren en weer aanvullen van de negatieve subvector. Voor kommagetallen wordt er een schaaloperatie uitgevoerd vóór en na het sorteren. Deze stappen zijn lineair en hebben een kleine constante extra kost.