Bucket Sort

Student	Stan Merlijn
Email	stan.merlijn@student.hu.nl
Studentennummer	1863967
link github(origineel)	https://github.com/StanMerlijn/HPP-bucket-Sort.git
link github(classroom)	https://github.com/AI-S4-2024/inleveropgave-1-bucket-sort-StanMerlijn.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 vectors.

2 aanpak

2.1 Distribution pass (stap 1)

De eerste stap van het bucket sort-algoritme is de distribution pass. In deze stap wordt de input vector verdeeld in verschilende buckets. Voor beide de kommagetallen en de gehelle getallen is er maar één functie nodig.

De code om de digit te bereken:

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

Listing 1: digit berekenen

Deze code gebruikt drie variabelen:

- num: het getal dat in het bucket moet worden geplaatst
- divisor: de factor waarmee het getal wordt gedeeld om de digit te berekenen
- 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.

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

Listing 2: Distribution pass

2.2 Gathering pass(stap 2)

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

```cpp
template<typename T>
void gatheringPass(std::vector<T>& input, const std::vector<std::vector<T>>& buckets)
{
    // Flatten the buckets
    input.clear(); // clear the input vector but keep the capacity
    for (int i = 0; i < buckets.size(); i++) {
        for (int j = 0; j < buckets[i].size(); j++){
            input.push_back(buckets[i][j]);
        }
    }
}
```

Listing 3: Gathering pass

2.3 Bucket Sort (stap 3)

De functie `bucketSortSteps` is de hoofdfunctie van het bucket sort-algoritme. Hier worden de distributie- en gathering pass uitgevoerd. Deze functie heeft drie parameters:

- input: de vector die gesorteerd moet worden
- buckets: de 2D-vector waarin de getallen worden verdeeld
- passes: het aantal passes dat nodig is om de vector te sorteren

```cpp
template<typename T>
std::vector<T> bucketSortSteps(std::vector<T>& input, std::vector<std::vector<T>>& buckets,
  int passes)
{
    // 3. Loop through the digits
    for (int i = 0; i < passes; i++) {
        // 1. Distribution pass: distribute
        distributePass(input, buckets, i);
        // 2. Gathering pass: flatten the buckets
        gatheringPass(input, buckets);

        // Clear the buckets
        for (int j = 0; j < 10; j++) {
            buckets[j].clear();
        }
    }
    return input;
}
```

De `int` passes is de lengte van het grootste getal in de vector(input). De functie calculatePasses berekent het aantal passes dat nodig is om de vector te sorteren. Beide voor de gehelle getallen en de kommagetallen wordt hieronder weergegeven.

```cpp
// Bucket sort the negative numbers
if (!negatives.empty()) {
    bucketSortSteps(negatives, buckets, calculatePasses(negatives));
}

// Bucket sort the non-negative numbers
if (!nonNegatives.empty()) {
    bucketSortSteps(nonNegatives, buckets, calculatePasses(nonNegatives));
}
```

2.4 Negatieve getallen(stap 4)

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.emplace_back(-num);
    } else {
        nonNegatives.emplace_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.emplace_back(-num * scaleFactor);
    } else {
        nonNegatives.emplace_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. Het gesorteerde vector van negatieve getallen wordt dan weer omgezet naar negatieve getallen. Daarna worden beide verzamelingen samengevoegd. Het samen voegen gebeurd zowel voor gehele getallen als kommagetallen op dezelfde manier.

```cpp
// Reverse the negative numbers
std::reverse(negatives.begin(), negatives.end());
for (int i = 0; i < negatives.size(); i++) {
    negatives[i] = -negatives[i];
}

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

Listing 5: getallen samenvoegen

2.5 Comma getallen (opdracht 4)

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 (auto num : input) {
    if (num < 0) {
        negatives.push_back(-num * scaleFactor);
    } else {
        nonNegatives.push_back(num * scaleFactor);
    }
}
```

Listing 6: Schaal factor

Terug schalen van de kommagetallen gebeurd na het sorteren van de kommagetallen.

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

Listing 7: Terug schalen

3 Complexiteitsanalyse (opdracht 1 en 3)

Om het Bucket Sort-Algoritme te testen, is het algoritme met vectors 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 naar json geformateerd). '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: `ints`

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: `floats`

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

Performance verbeteringen

Na verder kijk naar het geïmplementeerde algoritme zijn er nog wat performance verbeteringen gemaakt; De vectors zijn met gebruik can `reserve` gereserveerd. Dit zorgt voor een snelheidswinst, omdat de vector niet elke keer hoeft te worden vergroot wat steeds nieuwe allocaties kost.

Voor beide de gehelle getallen en de kommagetallen is er gebruik gemaakt van `reserve` om de negatieve en de non negative vector te reserveren. Daarnaast worden in alle functies de vectoren doorgegeven als referentie, waar dit bij sommige functies eerder niet het geval was. Dit voorkomt dat de vectoren gekopieerd worden. Deze aanpassingen zorgen voor een snelheidswinst. Die je terug ziet in de onderstaande tabellen.

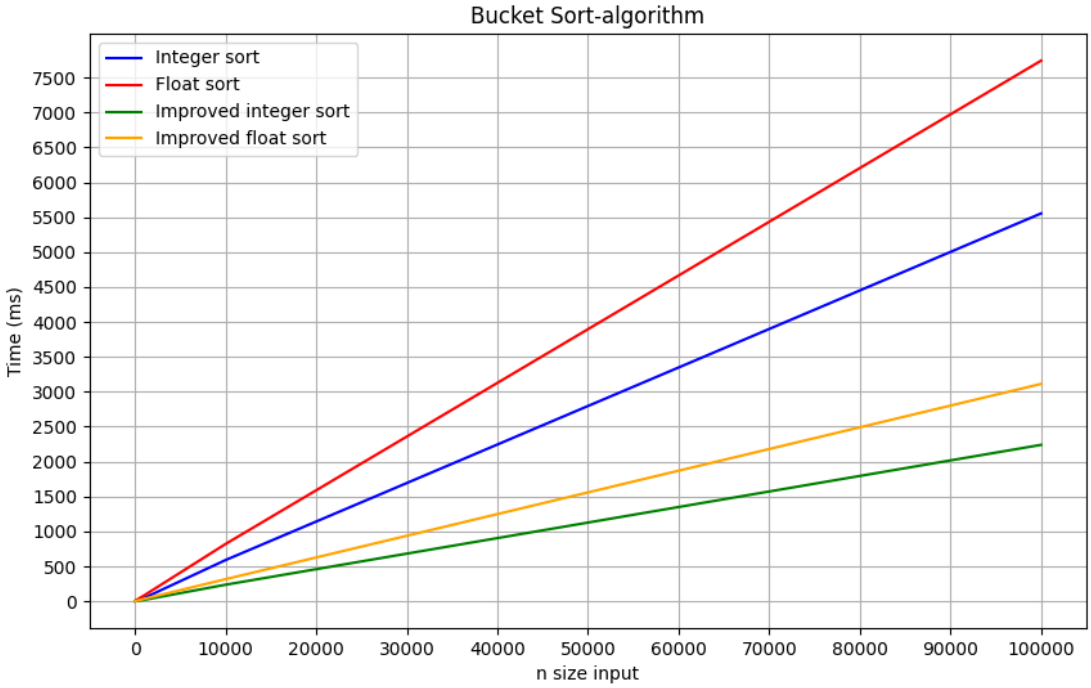
Tabel 3: Improved Bucket Sort Benchmark Results: `ints`

Benchmark	Iter.	Est. Run Time	Mean	Std Dev
Bucket Sort 10	3	2.6265 ms	8.78159 us	372.258 ns
Bucket Sort 100	1	3.4312 ms	34.5852 us	1.57385 us
Bucket Sort 1000	1	19.5995 ms	198.198 us	6.33555 us
Bucket Sort 10000	1	236.632 ms	2.35983 ms	34.5517 us
Bucket Sort 100000	1	2.24007 s	22.5052 ms	152.575 us

Tabel 4: Improved Bucket Sort Benchmark Results: `floats`

Benchmark	Iter.	Est. Run Time	Mean	Std Dev
Bucket Sort 10	2	2.7788 ms	13.8443 us	403.738 ns
Bucket Sort 100	1	4.9579 ms	49.2992 us	1.22971 us
Bucket Sort 1000	1	33.2223 ms	330.474 us	2.3857 us
Bucket Sort 10000	1	316.069 ms	3.12869 ms	16.7833 us
Bucket Sort 100000	1	3.1129 s	30.9041 ms	304.188 us

Uit deze resultaten is de est run time er uit gehaald en geplot tegen de grootte van de vector(n):



Figuur 1: Complexity of the bucket sort-algoritme

Je kan zien dat de complexiteit van het algoritme lineair is. Dit is te verwachten omdat de distributie en gathering pass beide een tijdcomplexiteit van $O(n)$ hebben.

4 Performance

- **Distributiepass:**
  Elke distributiepass scant het gehele input vector 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.

- **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 constante extra kost. Dit levert het verschill op wat je ziet in de tabellen 3, 4 en de grafiek 1.

5 Geheugenverbruik (opdracht 5)

Het geheugenverbruik (memory usage) is gemeten door het project in Visual Studio te openen en de memory profiler te gebruiken. Hiervoor heb ik de test.cpp aangepast voor een vector met een grootte van 100000 elementen. Hier is de code die ik heb gebruikt:

```
  TEST_CASE("tes bucketsort")
{
    std::vector<int> ns = {100000};
    std::vector<std::vector<float>> vecs;
    std::vector<std::vector<float>> results;

    for (int n : ns) {
        // Generate a random vector to sort
        std::random_device rnd_device;
        std::mt19937 mersenne_engine {rnd_device()};
        std::uniform_real_distribution<float> dist {1, 1000};
        auto gen = [&dist, &mersenne_engine]() {
            return dist(mersenne_engine);
        };
        std::vector<float> vec(n);
        generate(std::begin(vec), std::end(vec), gen);

       std::cout << "vec: " << vec[0] << std::endl;

       std::vector<float> result = bucketSort(vec);

       std::cout << "result: " << result[0] << std::endl;
}
```

Met gebruik van breakpoints en snapshots is er gekeken naar het geheugenverbruik van de test en het sorteer-algoritme. De grootte van de vector "vec1ïn de profiler is 400.047 bytes. Dit kan je ook ongeveer berekenen als 100000 * 4 bytes (voor een float), wat 400000 bytes oplevert. De extra 47 bytes vormen de overhead van de vector.

Tijdens de runtime van de bucket sort is het geheugenverbruik op de heap gestegen naar 1.771,08 KB. Op dat moment zijn er 13 vectoren aangemaakt: de vectoren negatives, nonNegatives, input en de 10 buckets. De drie grootste vectoren, namelijk input, negatives en nonNegatives, hebben elk een grootte van 400.047 bytes. Bovendien heeft elke bucket een grootte van precies 48.599 bytes. Dit komt waarschijnlijk door de wijze waarop de pseudo-random getallen worden gegenereerd. Er wordt een `uniform_real_distribution` gebruikt, wat zorgt voor kommagetallen tussen 1 en 1000. Deze verdeling spreidt de getallen gelijkmatig over het bereik 1 tot 1000, waardoor de getallen evenwichtig over de buckets worden verdeeld.

Omdat de bucket sort de input als referentie meekrijgt en de output als return-waarde teruggeeft, blijft het geheugenverbruik van de input en output gelijk, maar voor de functieaanroep is er extra geheugen nodig. Hiervoor kan een eenvoudige schattingsformule worden opgesteld. Hierbij is n de grootte van de inputvector en t het type van de inputvector:

$$\text{geheugenverbruik} = 3 * n * \text{sizeof}(t) + 13 * \text{sizeof}(\texttt{std::vector<t>}) \tag{1}$$

Het getal 3 in de formule staat voor de negatives, nonNegatives en de buckets die worden aangemaakt. De term 13 × sizeof(`std::vector<t>`) betreft de 10 buckets, de inputvector en de outputvector. Als we voor (n = 100000) invullen en aannemen dat (sizeof(`std::vector<t>`) = 48) bytes (een geschatte waarde voor de begin-, eind- en capaciteits-pointers) en dat voor (t = `float`) geldt dat (sizeof(t) = 4) bytes, dan krijgen we:

$$\text{geheugenverbruik} = 3 * 100000 * 4 + 13 * 47 = 1.200.624\text{bytes} \tag{2}$$

Als we dit vergelijken met het via de profiler gemeten geheugenverbruik:

$$\text{geheugenverbruik} = 400,047 * 3 + 48,599 * 10 = 1.200.624\text{bytes} \tag{3}$$

Beide waarden komen overeen, wat betekent dat de formule ongeveer correct is en dat het geheugenverbruik van de bucket sort op deze manier berekend kan worden. Deze schatting geldt uitsluitend voor het geheugenverbruik op de heap dat wordt gebruikt voor de vectoren. Daarnaast wordt er nog extra geheugen benut voor de functieaanroep en de stack. Concluderend is ook het geheugenverbruik van het algoritme lineair, aangezien de formule lineair toeneemt met (n) en de overige variabelen constant blijven.