# Verslag: Prestatieanalyse van Bucket Sort

Naam: Sietse Neve

Studentnummer: 1810364

Git: <https://github.com/DrZisnotavailable/hpp-bucketsort.git>

**Inleiding**

Bucket Sort is een efficiënt sorteeralgoritme dat gebruikmaakt van het principe van distributie en herverzameling van elementen in buckets. Dit verslag beschrijft de implementatie en prestatieanalyse van Bucket Sort in C++. Daarnaast wordt de tijdscomplexiteit van het algoritme onderzocht aan de hand van experimentele timing-tests.

**Implementatie**

De implementatie bestaat uit verschillende stappen:

**1. Bepalen van het aantal cijfers van een getal**

De functie getNumDigits berekent het aantal cijfers van een getal, wat nodig is om te bepalen hoeveel iteraties het sorteerproces moet doorlopen.

int getNumDigits(T num) {

if constexpr (std::is\_signed<T>::value) {

num = std::abs(num);

}

int digits = 1;

while (num >= 10) {

num /= 10;

digits++;

}

return digits;

}

**2. Distributie-pass**

De functie distributionPass plaatst elk getal in de juiste bucket op basis van het huidige cijfer (index), waarbij een offset wordt gebruikt om negatieve getallen correct te verwerken.

std::array<std::vector<T>, 19> distributionPass(const std::vector<T>& lst, int index) {

std::array<std::vector<T>, 19> buckets;

for (T num : lst) {

int digit = (num / static\_cast<int>(pow(10, index))) % 10;

int bucketIndex = digit + 9;

buckets[bucketIndex].push\_back(num);

}

return buckets;

}

**3. Gathering-pass**

De gatheringPass functie verzamelt de elementen terug uit de buckets en zet ze in de oorspronkelijke lijst.

std::vector<T> gatheringPass(const std::array<std::vector<T>, 19>& buckets) {

std::vector<T> sortedList;

for (const auto& bucket : buckets) {

for (T num : bucket) {

sortedList.push\_back(num);

}

}

return sortedList;

}

**4. Bucket Sort**

De bucketSort-functie voert iteratief distributie- en gathering-passes uit, afhankelijk van het maximale aantal cijfers in de invoerlijst.

std::vector<T> bucketSort(std::vector<T> lst) {

int maxDigits = 0;

for (T num : lst) {

maxDigits = std::max(maxDigits, getNumDigits(num));

}

for (int index = 0; index < maxDigits; index++) {

std::array<std::vector<T>, 19> buckets = distributionPass(lst, index);

lst = gatheringPass(buckets);

}

return lst;

}

**Prestatietests en Experimentele Analyse**

De prestaties van het algoritme worden geëvalueerd door sorteertijden te meten voor verschillende invoergroottes. De resultaten worden opgeslagen in een CSV-bestand voor verdere analyse in Python.

void runSortTest(const std::string& filename, int minVal, int maxVal, unsigned int step, unsigned int maxSize) {

std::ofstream file(filename);

file << "n,time\n";

srand(0);

for (unsigned int i = step; i <= maxSize; i += step) {

std::vector<int> lst;

for (unsigned int j = 0; j < i; j++) {

lst.push\_back(rand() % (maxVal - minVal + 1) + minVal);

}

auto start = std::chrono::high\_resolution\_clock::now();

lst = bucketSort(lst);

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> elapsed = end - start;

file << i << "," << elapsed.count() << "\n";

}

file.close();

}

In de main-functie wordt de sorteertest uitgevoerd met een bereik van -1000 tot 1000, met een stapgrootte van 100 tot een maximum van 10.000.

int main() {

runSortTest("bucketSortTimings.csv", -1000, 1000, 100, 10000);

std::cout << "Timing data opgeslagen in bucketSortTimings.csv\n";

return 0;

}

**Complexiteitsanalyse**

getNumDigits functie:

Gebruikt een while loop die deelt door 10 tot er één cijfer over is

Voor een getal n kost dit O(log n) tijd

Initiële loop om maxDigits te vinden:

Roept getNumDigits aan voor elk element in de lijst

Als n het grootste getal in de lijst is en k de lengte van de lijst:

Time complexity: O(k \* log n)

Hoofdsorteerlus:

Draait maxDigits keer (laten we dit d keer noemen)

Voor elke iteratie:

distributionPass: O(k) om elk getal één keer te verwerken

gatheringPass: O(k) om alle getallen terug te verzamelen

Totaal voor de hoofdlus: O(d \* k)

Totale time complexity:

Initiële maxDigits berekening: O(k \* log n)

Hoofdsorteerlus: O(d \* k)

Waarbij d = log₁₀(n) (aantal cijfers in het grootste getal)

Daarom is de totale complexiteit: O(k \* log n + k \* log n)

Dit vereenvoudigt naar: O(k \* log n)

**Conclusie**

Bucket Sort is een efficiënte sorteermethode voor datasets met een beperkte bereikgrootte. De implementatie toont aan dat het algoritme goed schaalbaar is, en de prestaties zijn geanalyseerd via experimentele timing-tests. De theoretische analyse bevestigt dat de tijdscomplexiteit in de orde van **O(k \* log n)** ligt. Verdere optimalisaties kunnen worden onderzocht door alternatieve bucket-indelingen en parallelle verwerking toe te passen.