HPP #1 Bucket sort

BIG O COMPLEXITY

Joey Welvaadt
HOGESCHOOL UTRECHT | 18-02-2025

In dit document zal de code van het bucket sort algoritme worden behandeld en wordt er gekeken naar welke tijd complexiteit de code heeft.

Om te beginnen worden er een aantal afbeeldingen weergegeven samen met een korte beschrijving zodat duidelijk is welke keuzes er zijn genomen. In figuur 1 worden de utility functies weergegeven, get_digit wordt gebruikt om van een bepaalde plek in een nummer te krijgen, voorbeeld: get_digit(189, 10) -> 8. Vervolgens wordt get_largest_decimal_amount gebruikt om van een vector gevuld met floats het getal te halen die het meest aantal decimalen bevat, zodat dit later gebruikt kan worden om de alle decimalen weg te werken.

```
/** START UTILS FUNCTIONS */
int get_digit(int number, int place) {
    // If place value is higher than the number itself
    return (number / place) % 10;
}

int get_largest_decimal_amount (vector<float> numbers) {
    int answer = 0;
    for (float number : numbers) {
        string stringNum =  to_string(number);
        int decimalPos = stringNum.find(".");
        string newString = stringNum.substr(decimalPos);

        while (!newString.empty() && newString.back() == '0') {
            newString.pop_back();
        }

        int tmp = newString.length() - 1;
        if(tmp > answer) {
            answer = tmp;
        }
    }
    return answer;
}

/** END UTILS FUNCTIONS */
```

Figure 1 Uitlity function bucket sort

Nu de verschillende utility functies duidelijk zijn kan het algoritme van de bucket sort worden toegelicht. Zie figuur 2 voor een snippet van de code, hier worden aan het begin twee variabelen gecreëerd: places (het aantal cijfers in het grootste getal) & minimum (het laagste getal in de gegeven lijst). Als minimum lager is dan nul en we dus werken met min getallen, lopen we door de lijst heen en voegen de minimum toe bij elk getal. Dit zorgt ervoor dat het laagste nummer in de lijst 0 is.

Aan het eind van het bucket sort algoritme trekken we de minimum weer van elk getal in de lijst af, zodat we de normale waardes weer terug krijgen.

```
void bucket_sort(vector<int>& list) {
    int place = 1;

    vector<vector<int>> buckets(10);

    int minimum = *min_element(begin(list), end(list));

    if(minimum < 0) {
        valarray<int> list_val(list.data(), list.size());
        list_val += -minimum;
        list.assign(begin(list_val), end(list_val));
    }

    int places = to_string(*max_element(begin(list), end(list))).length() + 1;
    // 9i + 1
    for (int i = 0; i < places; i++)
    {

        // Distribution Pass
        for (size_t j = 0; j < list.size(); j++)
        {
            int digit = get_digit(list[j], place); // 2 arithmetic operations, 1 lookup
            // cout << list[j] << ", " << digit << endl;

            buckets[digit].push_back(list[j]); // 1 lookup, 1 write
            // cout << "help me " << endl;
        }


        // cout << " " << endl;
        int index = 0;
        // Gathering Pass
        for (size_t k = 0; k < buckets.size(); k++)
        {
            for (size_t l = 0; l < list.size(); l++)
            {
                if(l < buckets[k].size()) {
                    list[index++] = buckets[k][l]; // 2 lookup, 1 write, 1 increment
                }
            }
            buckets[k].clear();

        }
        place *= 10;  // 1 arithmetic
    }


    if(minimum < 0) {
        valarray<int> list_val(list.data(), list.size());
        list_val += minimum;
        list.assign(begin(list_val), end(list_val));
    }
}
```

Figure 2 Core bucket sort algoritme

Het laatste stuk van het bucket sort algoritme is de support voor nummers met een of meerdere decimalen (floats). Er is een override mogelijkheid voor de bucket_sort functie (zie figuur 3). Deze override functie neemt in plaats van een vector met integers, een vector met floats. In combinatie met de get_largest_decimal_amount utility functie worden de floats veranderd naar integers, deze lijst word dan mee gegeven aan de functie in figuur 2, op deze manier zijn er geen grote veranderingen nodig in het originele algoritme.

```
void bucket_sort(vector<float>& numbers) {
    int multiplier = pow(10, get_largest_decimal_amount(numbers));
    valarray<float> numbers_val(numbers.data(), numbers.size());

    numbers_val *= multiplier;
    vector<int> int_numbers(numbers_val.size());
    int_numbers.assign(begin(numbers_val), end(numbers_val));
stexpr auto std::end<std::vector<int>>(std::vector<int> &__cont)->std::vector<int>::iterator
overloads

    transform(begin(int_numbers), end(int_numbers), begin(numbers),
              [multiplier](int n) { return static_cast<float>(n) / multiplier; });

}
```

Figure 3 Override bucket sort functie met float vector parameter

Het laatste stuk wat in dit document behandeld word is de time complexiteit die verzameld is vanuit het programma. Figuur 4 is een grafiek van de verschillende tijd complexiteiten en hoe de gemeten data hier tussen valt.

De gemeten data valt onder O(log(n)) en boven O(n).

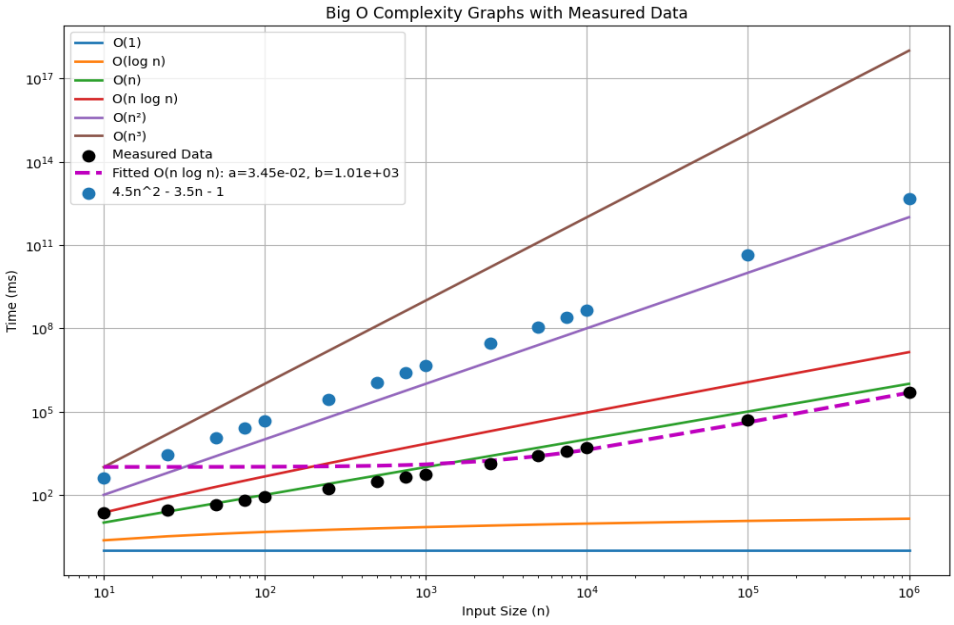

Figure 4 Tijd complexiteit verzamelde data