Politechnika Wrocławska

Projektowanie i analiza algorytmów

Projekt 2

ArtsyKimiko

13 maja 2024

1 Wstęp

Sortowanie jest jednym z fundamentalnych zagadnień informatyki, istotnym elementem wielu algorytmów i aplikacji. Algorytmy sortowania różnią się pod względem wydajności, złożoności czasowej i pamięciowej oraz zastosowań praktycznych. W tym sprawozdaniu omówiono cztery algorytmy sortowania: QuickSort, IntroSort oraz BucketSort.

Quicksort to algorytm sortowania, który wykorzystuje strategię "dziel i zwyciężaj", dzieląc listę na części na podstawie wybranego elementu zw. pivotem. Następnie rekurencyjnie sortuje obie części. Jest szybkim algorytmem o złożoności średnio O(n log n), ale może osiągnąć $O(n^2)$ w najgorszym przypadku. Aby zmniejszyć ryzyko tego przypadku, stosuje się różne strategie wyboru pivota, takie jak losowanie lub wybór mediany.

IntroSort to algorytm hybrydowy, który łączy ze sobą różne strategie sortowania, takie jak Quicksort, Heapsort i Insertion Sort. Jego działanie polega na zastosowaniu Quicksorta do szybkiego sortowania danych, jednak w przypadku wystąpienia pesymistycznego scenariusza, gdzie Quicksort może działać wolno, przechodzi do Heapsorta lub Insertion Sorta. Dzięki temu jest on efektywny zarówno dla danych przypadkowych, jak i dla danych zbliżonych do posortowanych, osiągając złożoność czasową O(n log n).

BucketSort, znany również jako sortowanie kubełkowe, jest algorytmem sortowania liniowego, który dzieli zbiór elementów na przedziały, zwane kubełkami, a następnie sortuje każdy kubełek osobno, często wykorzystując inny algorytm sortowania, np. sortowanie przez wstawianie. Jest to skuteczna metoda dla danych równomiernie rozłożonych w zakresie wartości, osiągającą złożoność czasową O(n + k), gdzie n to liczba elementów, a k to liczba kubełków.

2 Opis algorytmów

2.1 Filtrowanie danych

Metoda `ReadFile` służy do wczytania danych z pliku "`projekt2_dane.csv`"oraz filtrowania błędnych danych. Plik jest otwierany, a następnie wczytywany linia po linii. Dla każdej linii, program znajduje ostatnie wystąpienie przecinka, odczytuje dane po nim i konwertuje je na liczby całkowite. Jeśli konwersja się nie powiedzie, linia jest ignorowana. Po wczytaniu wszystkich poprawnych danych lub osiągnięciu końca pliku, plik jest zamykany.

```
template <typename T>
void Sort<T>::ReadFile()
{
    ifstream file("projekt2_dane.csv");
    if (!file.is_open())
        return;

    string line, data;
    size_t pos = string::npos;
    for (int i = 0; i < this->size && getline(file,line); i++)
    {
        pos = line.find_last_of(",");
        if(pos != string::npos)
        {
            data = line.substr(pos + 1);
            try { this->arr[i] = stoi(data); }
            catch(const invalid_argument& e){ i--; }
        }
    }
    file.close();
}
```

2.2 Sortowanie szybkie

Funkcja `QuickSort` implementuje algorytm sortowania szybkiego (QuickSort) dla tablicy. Pierwszym krokiem jest wyznaczenie pivota, który jest środkowym elementem tablicy. Następnie tablica jest przeszukiwana, aby elementy mniejsze od pivota znalazły się po lewej stronie, a większe po prawej.

Algorytm wykonuje się rekurencyjnie dla dwóch podtablic: od low do high2 oraz od low2 do high.

- **Złożoności w najlepszym przypadku:**
  - Czasowa: O(n log n)
  - Pamięciowa: O(log n)
- **Złożoności w najgorszym przypadku:**
  - Czasowa: O(n^2)
  - Pamięciowa: O(n)

```
void Sort<T>::QuickSort(int low, int high)
{
    if(high == -1)
        high = size - 1;
    int low2 = low, high2 = high;
    T pivot = arr[(low + high) / 2];
    while (low2 <= high2)
    {
        while (arr[low2] < pivot)
            low2++;
        while (arr[high2] > pivot)
            high2--;
        if (low2 <= high2)
        {
            swap(arr[low2], arr[high2]);
            low2++;
            high2--;
        }
    }
    if (low < high2)
        QuickSort(low, high2);
    if (low2 < high)
        QuickSort(low2, high);
}
```

2.3 Sortowanie introspectywne

Metoda `IntroSort` inicjuje sortowanie hybrydowe algorytmem IntroSort przez wywołanie funkcji rekurencyjnej `IntroSortRecursion` z odpowiednimi parametrami. W funkcji `IntroSortRecursion`, jeśli maksymalny poziom rekursji osiągnie 0, sortowanie jest przeprowadzane za pomocą MergeSort. W przeciwnym razie, używany jest QuickSort. Algorytm ten dzieli tablicę na elementy mniejsze i większe od wybranego pivota, a następnie rekurencyjnie sortuje obie części.

- **Złożoności w najlepszym przypadku:**
  - Czasowa: O(n log n)
  - Pamięciowa: O(n log n)
- **Złożoności w najgorszym przypadku:**
  - Czasowa: O(n log n)
  - Pamięciowa: O(n log n)

```
template<typename T>
void Sort<T>::IntroSort()
{
    int max_depth = 2 * log(size);
    IntroSortRecursion(0, size - 1, max_depth);
}
template<typename T>
void Sort<T>::IntroSortRecursion(int low, int high, int max_depth)
{
    if (max_depth == 0)
    {
        MergeSort(low, high);
        return;
    }
    else
    {
        int low2 = low, high2 = high;
        T pivot = arr[(low + high) / 2];
        while (low2 <= high2)
        {
            while (arr[low2] < pivot)
                low2++;
            while (arr[high2] > pivot)
                high2--;
            if (low2 <= high2)
            {
                swap(arr[low2], arr[high2]);
                low2++;
                high2--;
            }
        }
        if (low < high2)
            IntroSortRecursion(low, high2, max_depth - 1);
        if (low2 < high)
            IntroSortRecursion(low2, high, max_depth - 1);
    }
}
```

2.4 Sortowanie kubełkowe

Metoda `BucketSort` sortuje dane za pomocą algorytmu BucketSort. Na początku znajdowana jest maksymalna wartość w tablicy. Następnie tworzona jest tablica `bucket` o rozmiarze równym maksymalnej wartości zwiększonej o 1. Dane są następnie przypisywane do odpowiednich kubełków na podstawie ich wartości. Kolejno, dane są zbierane z kubełków i umieszczane ponownie w tablicy w posortowanej kolejności. Na końcu, tablica `bucket` jest usuwana.

- **Złożoności w najlepszym przypadku:**
  - Czasowa: O(n+k)
  - Pamięciowa: O(n+k)
- **Złożoności w najgorszym przypadku:**
  - Czasowa: O(n^2)
  - Pamięciowa: O(n+k)

Gdzie k to liczba kubełków.

```
template<typename T>
void Sort<T>::BucketSort()
{
    double max = arr[0];
    for(int i = 1; i < size; i++)
        if(arr[i] > max)
            max = arr[i];

    int bucket_size = static_cast<int>(max) + 1;
    double *bucket = new double[bucket_size];

    for(int i = 0; i < bucket_size; i++)
        bucket[i] = 0;
    for(int i = 0; i < size; i++)
        bucket[static_cast<int>(arr[i])]++;
    for(int i = 0, j = 0; i < bucket_size; i++)
    {
        while(bucket[i] > 0)
        {
            arr[j++] = i;
            bucket[i]--;
        }
    }
    delete bucket;
}
```

3 Badania

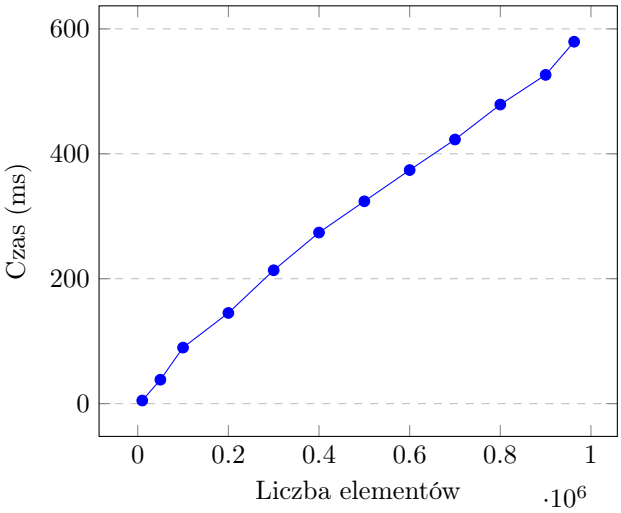
W ramach przeprowadzonych badań przetestowano wydajność wczytywania danych oraz trzech różnych algorytmów sortowania: QuickSort, IntroSort i BucketSort. Poniżej przedstawiono wyniki oraz wnioski płynące z przeprowadzonych eksperymentów.

3.1 Wczytywanie danych

Wyniki wczytywania danych przedstawiono w tabeli 1 oraz na wykresie 1. Zgodnie z oczekiwaniami, czas wczytywania danych wzrasta wraz z ilością elementów. Jednak tempo wzrostu czasu nie jest liniowe, sugerując możliwe ograniczenia sprzętowe lub oprogramowania przy większych zbiorach danych.

Tabela 1: Czas wczytywania danych

Ilość elementów	Czas (ms)
10000	4.97
50000	38.30
100000	89.68
200000	145.19
300000	213.56
400000	273.93
500000	323.93
600000	373.98
700000	422.87
800000	478.87
900000	526.38
962903	579.46



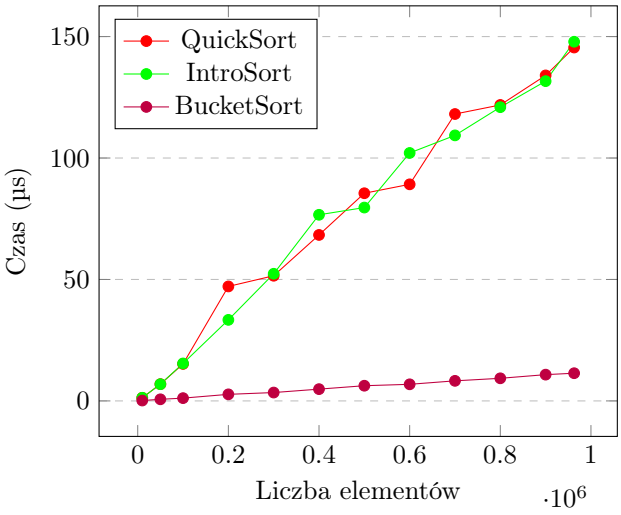
Rysunek 1: Czas wczytywania danych

3.2 Sortowanie danych

Kolejne eksperymenty dotyczyły wydajności algorytmów sortowania. Wyniki przedstawiono w tabeli 2 oraz na wykresie 2.

Tabela 2: Czasy sortowania (w mikrosekundach)

Liczba elementów	QuickSort	IntroSort	BucketSort
10000	1.29	1.28	0.16
50000	6.91	6.88	0.66
100000	15.23	15.43	1.15
200000	47.13	33.37	2.71
300000	51.53	52.35	3.44
400000	68.33	76.63	4.85
500000	85.52	79.61	6.26
600000	89.17	102.08	6.84
700000	118.11	109.32	8.26
800000	121.85	120.93	9.31
900000	134.00	131.65	10.81
962903	145.49	147.88	11.41



Rysunek 2: Czas sortowania

3.3 Analiza median i wartości średnich

Analizując zmierzone wartości na podstawie danych wczytanych z pliku, przedstawionych w tabeli 3, możemy zauważyć, że wartości średnie oraz mediany zbliżają się do siebie w miarę wzrostu liczby elementów. Oznacza to, że dane wydają się być rozkładane równomiernie, co może być pozytywnym znakiem dla skuteczności algorytmów sortowania.

Tabela 3: Analiza wartości na podstawie danych wczytanych z pliku

Liczba elementów	Wartości średnie	Mediany
10000	5.460	5
50000	5.491	5.5
100000	6.090	7
200000	6.415	7
300000	6.535	7
400000	6.585	7
500000	6.665	7
600000	6.660	7
700000	6.665	7
800000	6.670	7
900000	6.650	7
962903	6.636	7

4 Wioski

1. **Wydajność algorytmów sortowania**

   Przeprowadzone eksperymenty pozwoliły na porównanie wydajności wykorzystanych algorytmów sortowania. Z wyników można zauważyć, że w przypadku mniejszych zbiorów danych (do około 100 tysięcy elementów), algorytmy QuickSort i IntroSort osiągają podobne czasy sortowania, z lekką przewagą dla QuickSort. Natomiast dla większych zbiorów danych, algorytm BucketSort okazuje się być znacznie bardziej wydajny, osiągając czas sortowania nawet o kilka rzędów wielkości krótszy od pozostałych algorytmów.

2. **Ograniczenia czasowe i pamięciowe**

   Otrzymane wyniki eksperymentów są zgodne z oczekiwanymi złożonościami czasowymi i pamięciowymi dla poszczególnych algorytmów. Czasy sortowania rosną zgodnie z przewidywaną złożonością czasową dla QuickSort, a zużycie pamięci w algorytmie BucketSort jest wyższe ze względu na potrzebę alokacji dodatkowej tablicy.

3. **Równomierność danych**

   Analiza median i średnich wartości na podstawie danych wczytanych z pliku potwierdza równomierny rozkład danych, co jest zgodne z oczekiwaniami. Stabilność mediany wokół wartości 7 dla wszystkich rozważanych liczebności elementów oraz zbliżenie wartości średnich do tej samej liczby potwierdzają brak skrajnych wartości lub znaczących odchyleń w danych. Takie właściwości danych są korzystne dla efektywności działania algorytmów sortowania, ponieważ umożliwiają im operowanie na zbiorach danych o relatywnie jednorodnej strukturze.

Literatura

[1] https://11dhanushs.medium.com/intro-sort-a-brief-introduction-f06b419674de

[2] https://www.geeksforgeeks.org/introsort-or-introspective-sort/

[3] https://eduinf.waw.pl/inf/alg/003_sort/0018.php

[4] https://pl.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort

[5] https://binarnie.pl/sortowanie-kubelkowe/

[6] https://www.zsnr1.com/leszeksosnowski/Informatyka_p/Lekcja_p_D1.pdf