Politechnika Wrocławska

Projektowanie i analiza algorytmów

Projekt 2

Freddy Ms

19 kwietnia 2024

1 Wstęp

Sortowanie jest powszechnym zagadnieniem w informatyce. My się zajmiemy trzema z nich - quciksort, bucketsort i mergesort. Każdy z tych algorytmów ma swoje unikalne cechy i własnoći.

Quicksort to algorytm sortowania oparty na zasadzie dziel i zwyciężaj. Polega on na wyborze elementu, nazywanego "pivotem", a następnie dzieleniu listy na dwa podzbiory: jeden, który zawiera elementy mniejsze od pivota oraz drugi, który zawiera elementu większe od pivota. Następnie te podzbiory są sortowane rekurencyjnie. Quicksort jest jednym z najszybszych algorytmów sortowania i często wykorzystywany w praktyce.

Mergesort to algorytm sortowania, który używa strategii podobnej do quicksorta. Dzieli on listę na pół, sortuje obie połówki osobno, a następnie scala je w jedną posortowaną listę. Mergesort jest algorytmem stabilnym i jest efektywny dla dużych zestawów danych.

Bucketsort jest algorytmem sortowania nie przypominającym żadnego z dwóch poprzednich. Zamiast porównywać bezpośrednio elementy, bucketsort dzieli elementy na "kubełki"na podstawie ich wartości. Następnie każdy kubełek jest sortowany osobno. Po posorotwaniu kubełków, elementy są scalane w jedną całość. Bucketsort jest szczególnie efektywny, gdy dane są równomiernie rozłożone w zakresie wartości i można go wykorzystać do sortowania liczb zmiennoprzecinkowych.

2 Funkcja wczytująca dane

W trakcie implementacji programu do sortowania danych napotkano na znaczące wyzwanie związane z funkcją odpowiedzialną za wczytywanie wartości z pliku. Pierwotna wersja funkcji zakładała wczytywanie wartości znajdujących się po trzecim przecinku w linii tekstu i próbowała konwersji tego fragmentu ze znakowego formatu na liczbę. Jednak po dokładnej analizie okazało się, że niektóre tytuły danych zawierały przecinki w swoich nazwach, co zakłócało prawidłowe działanie funkcji. Problem został rozwiązany poprzez odpowiednią modyfikację funkcji wczytującej dane z pliku.

```
template <typename T>
void Sort<T>::ReadCSV()
{
    ifstream file("projekt2_dane.csv");
    if (!file.is_open()){
        cout << "Error opening file" << endl;
        return;
    }
    string line,data;
    size_t position = string::npos;
    for (int i = 0; i < this->size && getline(file,line); i++){
        position = line.find_last_of(",");
        if(position != string::npos){
            data = line.substr(position + 1);
        try{
            this->arr[i] = stoi(data);
        } catch(const invalid_argument& e){
            i--;
        }}
    }
    file.close();
}
```

Zmienna "this->size"zawierała liczbę elementów, które należało posortować. Funkcja operuje w następujący sposób: lokalizuje pozycję ostatniego przecinka w danej linii tekstu, następnie odetnie wszystko po tej pozycji i próbuje dokonać konwersji pozyskanego fragmentu na liczbę. W przypadku niepowodzenia konwersji, proces ten jest powtarzany dla kolejnych linii tekstu, aż do momentu poprawnego wczytania odpowiedniej liczby elementów.

3 Quicksort

3.1 Złożoności

Złożoność czasowa i pamięciowa w najlepszym przypadku:

- Czasowa O(n log n)
- Pamięciowa O(log n)

W najgorszym przypadku:

- Czasowa O(n^2)
- Pamięciowa O(n)

3.2 Implementacja kodu

```
template <typename T>
void Sort<T>::quickSort(int left, int right){
    if (right == -1)
        right = size - 1;
    int i = left, j = right;
    int pivot = arr[(left + right) / 2];
    while (i <= j) {
        while (arr[i] < pivot)
            i++;
        while (arr[j] > pivot)
            j--;
        if (i <= j) {
            swap(arr[i], arr[j]);
            i++;
            j--;
        }
    }
    if (left < j)
        quickSort(left, j);
    if (i < right)
        quickSort(i, right);
}
```

Funkcja działa w oparciu o algorytm, w którym wybiera środkowy element z tablicy i ustanawia go jako element pivot. Następnie przeprowadza operację partycjonowania, zamieniając miejscami elementy tablicy w taki sposób, że wszystkie elementy mniejsze od pivota znajdują się po jego lewej stronie, a większe po prawej. Dla dwóch uzyskanych podtablic, funkcja quicksort jest wywoływana rekurencyjnie w celu sortowania każdej z nich. W efekcie, algorytm quicksort jest stosowany rekurencyjnie dla każdej z podtablic, aż do uzyskania posortowanej tablicy.

4 Mergesort

4.1 Złożoności

Złożoność czasowa i pamięciowa:

- Czasowa O(n log n)
- Pamięciowa O(n)

4.2 Implementacja kodu

Funkcja ta została podzielona na dwie: jedna część odpowiedzialna za sortowanie, a druga za scalanie tablicy:

```
template <typename T>
void Sort<T>::merge(int left, int middle, int right){
    int i, j, k;
    int n1 = middle - left + 1;
    int n2 = right - middle;
    T* L = new T[n1];
    T* R = new T[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[middle + 1 + j];
    i = 0; // Index of left array
    j = 0; // Index of right array
    k = left; // Index of main array
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
    delete[] L;
    delete[] R;
}
```

Funkcja tworzy dwie podtablice - lewą i prawą o określonym rozmiarze. Następnie przepisuje wartości z głównej tablicy do odpowiednich podtablic. W procesie sortowania porównuje wartości w obu podtablicach i wstawia mniejszy element do głównej tablicy, kontynuując ten proces do momentu, gdy obie podtablice są niepuste. Gdy jedna z podtablic staje się pusta, pozostałe elementy z drugiej podtablicy są kopiowane do głównej tablicy. Na zakończenie funkcji pamięć dynamicznie zaalokowana dla dodatkowych podtablic jest zwalniana.

```
template <typename T>
void Sort<T>::mergeSort(int left, int right){
    if (right == -1)
        right = size - 1;
    if (left < right) {
        int middle = left + (right - left) / 2;
        mergeSort(left, middle);
        mergeSort(middle + 1, right);
        merge(left, middle, right);
    }
}
```

5 Bucketsort

5.1 Złożoności

Złożoność czasowa w najlepszym i najgorszym przypadku:

- O(n + k), n - liczba elementów, k - liczba kubełków
- $O(n^2)$

Złożoność pamięciowa:

- O(n + k)

5.2 Implementacja kodu

```
template<typename T>
void Sort<T>::bucketSort() {
    T max_val = arr[0], min_val = arr[0];
    for (int i = 1; i < size; ++i) {
        if (arr[i] > max_val)
            max_val = arr[i];
        if (arr[i] < min_val)
            min_val = arr[i];
    }

    int num_buckets = 10;
    int range = max_val - min_val + 1;

    vector<T>* buckets = new vector<T>[num_buckets];

    for (int i = 0; i < size; ++i) {
        buckets[(arr[i] - min_val) * num_buckets / range].push_back(arr[i]);
    }

    int index = 0;
    for (int i = 0; i < num_buckets; ++i) {
        sort(buckets[i].begin(), buckets[i].end());
        for (T element : buckets[i]) {
            arr[index++] = element;
        }
    }
    delete[] buckets;
}
```

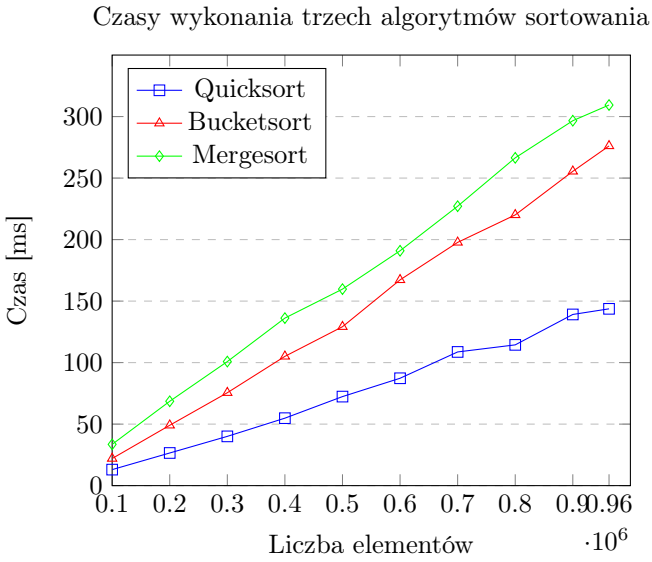
Funkcja rozpoczyna działanie od identyfikacji najmniejszego i największego elementu w danych wejściowych w celu określenia ich rozstępu. Następnie tworzone są kubełki w odpowiedniej liczbie, a każdy element z danych wejściowych jest umieszczany w odpowiednim kubełku na podstawie jego wartości. Po umieszczeniu wszystkich elementów w odpowiednich kubełkach, funkcja sortuje każdy kubełek indywidualnie. W kontekście danych z zakresu od 1 do 10, z uwagi na ograniczoną liczbę możliwych wartości, sortowanie kubełków nie jest konieczne. Na końcu kubełki są scalane w jedną posortowaną całość.

6 Badania

W ramach badań przeprowadzonych dla powyżej opisanych algorytmów przetestowano ich wydajność w praktyce. Poniżej przedstawiono wyniki w postaci tabeli, w której zawarto ilość elementów oraz czas sortowania w milisekundach dla każdego z algorytmów.

Tabela 1: Czasy wykonania trzech algorytmów sortowania dla różnych liczb elementów

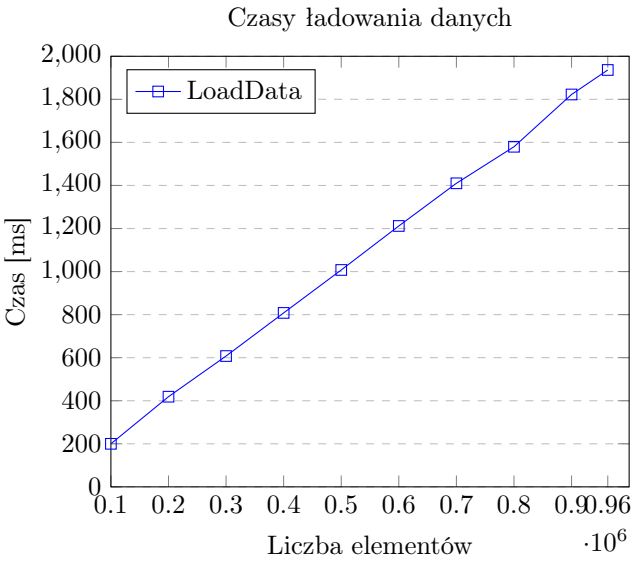
Liczba elementów	Quicksort [ms]	Bucketsort [ms]	Mergesort [ms]
100000	13.013	22.023	33.547
200000	26.532	49.056	68.568
300000	40.038	75.515	100.835
400000	54.834	105.076	136.176
500000	72.37	129.171	159.749
600000	87.278	167.258	190.906
700000	108.666	197.721	227.134
800000	114.419	220.061	266.448
900000	139.143	255.486	296.636
962903	143.712	276.128	309.383



Rysunek 1: Wykres czasów wykonania trzech algorytmów sortowania

Tabela 2: Czasy ładowania danych dla różnych liczb elementów

Liczba elementów	LoadData [ms]
100000	199.948
200000	418.75
300000	607.671
400000	807.939
500000	1007.43
600000	1211.8
700000	1410.02
800000	1579.71
900000	1822.54
962903	1936.39



Rysunek 2: Wykres czasów ładowania danych

Tabela 3: Średnie wartości i mediany elementów

Liczba elementów	Average	Median
100000	6.08993	7
200000	6.41498	7
300000	6.53519	7
400000	6.58598	7
500000	6.66572	7
600000	6.66067	7
700000	6.66552	7
800000	6.67074	7
900000	6.65017	7
962903	6.63661	7

7 Wnioski

Z przeprowadzonych badań wynika, że każdy z algorytmów działa zgodnie z oczekiwaniami i posiada wykresy czasu sortowania zgodne z ich założeniami obliczeniowymi. Analizując zebrane wyniki, można stwierdzić, że quicksort wyróżnia się jako najbardziej efektywny spośród analizowanych trzech algorytmów. Oprócz wysokiej wydajności w badaniach, quicksort charakteryzuje się także brakiem konieczności dynamicznego alokowania pamięci, co jest korzystne w kontekście zarządzania zasobami pamięci komputera. Wszystkie operacje sortowania przeprowadzane są na jednym obszarze w pamięci, co przyczynia się do optymalizacji zużycia zasobów systemowych.

8 Bibliografia

- https://pl.wikipedia.org/wiki/Sortowanie_przez_scalanie
- https://pl.wikipedia.org/wiki/Sortowanie_szybkie
- https://pl.wikipedia.org/wiki/Sortowanie_kubełkowe
- Cormen T.; Leiserson C.E.; Rivest R.L.; Stein C.: Wprowadzenie do algorytmów, WNT