Sprawozdanie 03: rekurencja Metoda dziel i zwyciężaj na przykładzie wyszukiwania binarnego i QuickSort

Dmytro Sakharskyi - L02 - 31259 03.04.2025

Metody Programowania

Informatyka 1 rok. 2 semestr

Wstęp

W ramach ćwiczenia skupiono się na dwóch klasycznych algorytmach implementujących tę metodę: wyszukiwaniu binarnym oraz sortowaniu szybkim (QuickSort). Oba algorytmy zostały zaimplementowane i przetestowane w różnych scenariuszach, w tym także pod kątem efektywności czasowej i liczby operacji. Celem było nie tylko zrozumienie działania samych algorytmów, ale również ocena ich praktycznej wydajności w zależności od rozmiaru danych wejściowych.

Zadanie 1. Dynamiczne generowanie tablicy

Stworzenie programu, który generuje tablicę losowych liczb, sortuje ją oraz umożliwia wyszukiwanie wskazanej liczby przez użytkownika.

Kod 1.

```
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Funkcja swap - zamienia miejscami wartości dwóch zmiennych typu int.

Funkcja partition - dzieli tablicę względem pivota – wszystkie elementy mniejsze od pivota idą na lewo, większe na prawo. Zwraca indeks pivota po podziale.

Funkcja quickSort - rekurencyjnie sortuje tablicę.

Kod 2.

```
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

void search(int arr[], int x) {
    int l;
    bool istnieje = false;
    cout << "Podaj liczbe do wyszukiwania ";
    cin >> l;

    for (int y = 0; y < x; y++) {
        if (arr[y] == l) {
            cout << "Znaleziono pod indeksem: " << y << endl;
            istnieje = true;
        }
    }
    if (istnieje == false) {
        cout << "Liczby " << l << " w tablice nie ma";
    }
}
```

Funkcja printArray - wyświetla wszystkie elementy tablicy na ekranie.

Funkcja search - pozwala użytkownikowi wpisać liczbę i sprawdza, czy znajduje się ona w tablicy.

Test sortowania

```
Podaj rozmiar tablicy 10
Podaj poczotek zakresa: 5
Podaj koniec zakresa: 20
16 7 9 15 19 9 8 8 12 19
Posortowana tablica:
7 8 8 9 9 12 15 16 19 19
Podaj liczbe do wyszukiwania 8
Znaleziono pod indeksem: 1
Znaleziono pod indeksem: 2
```

1. Program działa poprawnie: potrafi wylosować liczby, posortować je, a następnie wyszukać wskazaną wartość.

2. Funkcja search znajduje wszystkie wystąpienia danej liczby i wypisuje ich indeksy, co jest bardziej dokładne niż tylko potwierdzenie jej obecności.

3. QuickSort dobrze radzi sobie z sortowaniem liczb losowych.

4. Dzięki użyciu zakresów użytkownik może testować działanie programu w różnych warunkach.

Zadanie 2. Analiza czasowa

Porównanie czasu działania QuickSort i sort() z STL dla dużych zbiorów danych oraz analiza wyników.

Kod 1.

```
roznica = kon - pocz;

int* table = new int[x];
int* table1 = new int[x];
for (int y = 0; y < x; y++) {
    table1[y] = table[y] = rand() % roznica + pocz;
}

auto start = high_resolution_clock::now();
quickSort(table, 0, x - 1);
auto end = high_resolution_clock::now();
auto duration = duration_cast<milliseconds>(end - start);
cout << "Czas wykonania QuickSort: " << duration.count() << " ms" << endl;

auto start2 = high_resolution_clock::now();
sort(table1, table1 + x);
auto end2 = high_resolution_clock::now();
auto duration2 = duration_cast<milliseconds>(end2 - start2);
cout << "Czas sort() STL: " << duration2.count() << " ms" << endl;
return 0;
```

auto start = high_resolution_clock::now(); - Początek pomiaru czasu.

auto end = high_resolution_clock::now(); - Koniec pomiaru czasu.

auto duration = duration_cast<milliseconds>(end - start); - Obliczenie czasu działania.

Test sortowania

```
Podaj rozmiar tablicy 10000
Podaj poczotek zakresa: 5
Podaj koniec zakresa: 12
Czas wykonania QuickSort: 7 ms
Czas sort() STL: 0 ms
```

Gdy dane są bardzo *powtarzalne* (od 5 do 12), QuickSort potrzebuje więcej czasu na przetwarzanie, ponieważ musi obsłużyć wiele duplikatów. STL sort() w tym przypadku okazał się znacznie szybszy – prawdopodobnie optymalizuje lepiej dane z powtórzeniami.

```
Podaj rozmiar tablicy 20000
Podaj poczotek zakresa: 1
Podaj koniec zakresa: 100
Czas wykonania QuickSort: 4 ms
Czas sort() STL: 3 ms
```

Zwiększenie rozmiaru tablicy o x2 (20000) przy nieznacznym zwiększeniu zakresu wartości nie wpłynęło znacząco na czas działania — zarówno QuickSort, jak i STL działają szybko, ale różnica między nimi zaczyna się zacierać.

```
Podaj rozmiar tablicy 20000
Podaj poczotek zakresa: 1
Podaj koniec zakresa: 10000
Czas wykonania QuickSort: 3 ms
Czas sort() STL: 8 ms
```

Dla tablicy z *zróżnicowanymi danymi* (od 1 do 10000), QuickSort okazał się nawet szybszy niż STL sort(). Może to być efektem dobrze dobranych pivotów i efektywnego podziału danych. STL sort() może mieć dodatkowe narzuty lub inaczej działać z losowymi danymi.

Zadanie 3. Analiza liczby operacji

Zbadanie wydajności wyszukiwania binarnego i QuickSort dla różnych długości tablic oraz analiza liczby operacji i czasu wykonania.

```
auto start = high_resolution_clock::now();
quickSort(table, 0, x - 1);
auto end = high_resolution_clock::now();
auto duration = duration_cast<milliseconds>(end - start);
cout << "Czas wykonania QuickSort: " << duration.count() << " ms" << endl;

int n = sizeof(table) / sizeof(table[0]);

int target;
cout << "Podaj co szukamy: ";
cin >> target;

auto start1 = high_resolution_clock::now();
int result = binarySearch(table, 0, n - 1, target);
auto end1 = high_resolution_clock::now();
auto duration1 = duration_cast<milliseconds>(end1 - start1);
cout << "Czas wykonania wyszukiwania binarnego: " << duration1.count() << " ms" << endl;

if (result != -1)
    cout << "Element znaleziony na indeksie " << result << endl;
else
    cout << "Element nie znaleziony" << endl;

cout << "Ilosc wywolania swap: " << licznikSwap << endl;
cout << "Ilosc wywolania binarySearch: " << licznikSearch << endl;

cout << "Czas laczny wykonania obu operacji: " << duration.count() + duration.count() << "ms" << endl;
```

Łącząc wszystkie poprzednie metody (Zadania 1 i 2), a także dodając liczniki i dodatkowe pomiary czasu, możemy kompleksowo porównać oba algorytmy, aby sprawdzić, jak się zachowują w różnych warunkach.

Test sortowania

```
Podaj rozmiar tablicy 1000
Podaj poczotek zakresa: 1
Podaj koniec zakresa: 100
Czas wykonania QuickSort: 0 ms
Podaj co szukamy: 20
Czas wykonania wyszukiwania binarnego: 0 ms
Element nie znaleziony
Ilosc wywolania swap: 4618
Ilosc wywolania binarySearch: 3
Czas laczny wykonania obu operacji: 0ms
```

Dla tablicy (1000) operacje są prawie natychmiastowe. Mimo to wykonano **ponad 4 tysiące zamian**, co świadczy o dużej liczbie powtórzeń i małym zakresie danych. Binary search działa wykonał tylko 3 wywołania rekurencji.

```
Podaj rozmiar tablicy 10000
Podaj poczotek zakresa: 1
Podaj koniec zakresa: 100
Czas wykonania QuickSort: 1 ms
Podaj co szukamy: 20
Czas wykonania wyszukiwania binarnego: 0 ms
Element nie znaleziony
Ilosc wywolania swap: 68175
Ilosc wywolania binarySearch: 3
Czas laczny wykonania obu operacji: 2ms
```

Pomimo większej tablicy (1000), czas wykonania 2ms. Ilość operacji swap wzrosła 15-krotnie – to pokazuje, że **wydajność QuickSort bardzo cierpi przy dużej liczbie duplikatów**. BinarySearch nadal szybki – to zależy tylko od głębokości drzewa (logarytmicznie).

```
Podaj rozmiar tablicy 100000
Podaj poczotek zakresa: 1
Podaj koniec zakresa: 100
Czas wykonania QuickSort: 39 ms
Podaj co szukamy: 20
Czas wykonania wyszukiwania binarnego: 0 ms
Element nie znaleziony
Ilosc wywolania swap: 441966
Ilosc wywolania binarySearch: 3
Czas laczny wykonania obu operacji: 78ms
```

Wzrost liczby swapów (ponad 4 miliony) przy tylko 100 tys. elementów. To pokazuje, że **QuickSort z 10000+ liczbą powtórzeń jest bardzo nieefektywny**, bo wykonuje masę zbędnych zamian. BinarySearch nadal niezależny od rozmiaru tablicy – wciąż tylko 3 wywołania.

Wnioski

Na podstawie przeprowadzonych testów można zauważyć, że algorytm QuickSort działa szybciej w przypadku tablic w zakresie liczb **od 1 do 100**, natomiast funkcja sort() z biblioteki STL jest bardziej efektywna, gdy w tablicy znajdują się liczby w zakresie **100+.**

Dodatkowo, przy rozmiarach tablicy około 500 000 – 1 000 000+ pojawiają się błędy i awarie programu. Może to wynikać z ograniczeń pamięci operacyjnej lub braku odpowiedniej optymalizacji algorytmów.

Podsumowując, przeprowadzone eksperymenty pozwoliły nie tylko porównać dwa podejścia do sortowania i wyszukiwania, ale również lepiej zrozumieć ich zalety i ograniczenia. Dzięki temu można świadomie dobierać algorytmy w zależności od rodzaju i rozmiaru danych, co ma kluczowe znaczenie w praktycznym programowaniu i optymalizacji działania aplikacji.