Lista 2

Maciej Kmieciak, 277516

11.2024

1 Wstęp

Zaimplementowałem algorytmy: Quick Sort z modyfikacją polegającą na dzieleniu na trzy części; Radix Sort względem dowolnej podstawy `d`; implementację Insertion Sort na liście (stworzonej jako własna struktura) oraz Bucket Sort z modyfikacją, która daje działania dla dowolnych liczb (a nie tylko z przedziału $(0, 1]$. Porównałem działanie Radix Sort dla różnych podstaw, a także Quick Sort z Bucket Sort.

2 Najciekawsze fragmenty kodu

Poniżej znajdują się kluczowe fragmenty zmodyfikowanych wersji algorytmów. Tutaj usunąłem z nich liczniki porównań i przypisań.

2.1 Modyfikacja Quick Sort

```
array<int, 2> partition_mod(int A[], int p, int k) {
    if (A[p] > A[k]) {
        swap(A[p], A[k]);
    }
    int leftPivot = A[p];
    int rightPivot = A[k];

    int i = p + 1,
        j = p + 1,
        g = k - 1;

    while (j <= g) {
        if (A[j] < leftPivot) {
            swap(A[i], A[j]);
            ++i;
        }
        else if (A[j] > rightPivot) {
            while (A[g] > rightPivot && j < g) {
```

```
                --g;
            }

            swap(A[j], A[g]);
            --g;

            if (A[j] < leftPivot) {
                swap(A[i], A[j]);
                ++i;
            }
        }
        ++j;
    }

    --i;
    ++g;
    swap(A[p], A[i]);
    swap(A[k], A[g]);

    return { i, g };
}
```

To zmodyfikowana funkcja `partition`. Indeks `i` przechowuje pozycję, na którą powienien trafić następny element mniejszy od `leftPivot`. `j` przechodzi przez tablicę z lewa na prawo. `g` przechowuje pozycję, na którą ma trafić następny element większy od `rightPivot`. Przechodząc przez tablicę zamieniamy elementy odpowiednio, jeśli trzeba, tzn. przerzucamy je na lewo od lewego pivota, na prawo od prawego lub pozostawiamy na "środku", jeśli liczba jest pomiędzy wartościami pivotów. Na koniec wstawiamy same pivoty we właściwe miejsca i zwracamy ich indeksy.

2.2 Modyfikacja Radix Sort

```
void radix(int A[], int d) {
    int max_val = A[0];

    for (int i = 1; i < n; ++i) {
        if (A[i] > max_val) {
            max_val = A[i];
        }
    }

    int num_digits = log(max_val) / log(d) + 1;

    int digit_place = 1;
    for (int i = 0; i < num_digits; ++i) {
```

```
        counting(A, digit_place, d);
        digit_place *= d;
    }
}

void counting(int A[], int digit_place, int d) {
    int* C = new int[d]();
    int* B = new int[n];

    for (int i = 0; i < n; ++i) {
        int digit = (A[i] / digit_place) % d;
        ++C[digit];
    }

    for (int j = 1; j < d; ++j) {
        C[j] += C[j - 1];
    }

    for (int i = n - 1; i >= 0; --i) {
        int digit = (A[i] / digit_place) % d;
        B[C[digit] - 1] = A[i];
        --C[digit];
    }

    for (int i = 0; i < n; ++i) {
        A[i] = B[i];
    }
}
```

Najpierw szukamy maksimum, żeby potem określić liczbę cyfr do posortowania z użyciem logarytmu o podstawie `d`. Następnie, dla każdej cyfry-miejsca, zaczynając od najmniej znaczącej, wykonujemy procedurę `counting`. Wewnątrz niej, wydobywamy cyfrę na danym miejscu dla każdej liczby z tablicy. Tworzymy "pudełka" odpowiadające kolejnym cyfrom i umieszczcamy tam liczbę, która ma daną cyfrę na danym miejscu. Następnie przepisujemy liczby z powrotem do oryginalnej tablicy, w kolejności od najmniejszej cyfry (na danym miejscu) do największej i przechodzimy do kolejnej, bardziej znaczącej cyfry-miejsca.

2.3 Insertion Sort zaimplementowane na liście

```
void insert(Node** sorted, Node* newNode) {
    if (!*sorted || (*sorted)->data >= newNode->data) {
        newNode->next = *sorted;
        *sorted = newNode;
    }
    else {
```

```
        Node* current = *sorted;
        while (current->next && current->next->data < newNode->data) {
            current = current->next;
        }
        newNode->next = current->next;
        current->next = newNode;
    }
}
```

Powyżej znajduje się procedura `insert`, która wstawia nowy element do posortowanej listy, zgodnie z zasadą działania Insertion Sort. Przesuwamy się po kolejnych elementach listy, zaczynając od najmniejszego tak długo, jak element do wstawienia jest od nich mniejszy. Następnie wstawiamy nowy element pomiędzy ostatni, który był mniejszy a jego następcę.

2.4 Modyfikacja Bucket Sort

```
float* bucket_mod(const float* unsorted) {
 float* A = new float[n];

 for (int i = 0; i < n; ++i) {
  A[i] = unsorted[i];
 }

 float min_val = A[0], max_val = A[0];
 for (int i = 1; i < n; ++i) {
  if (A[i] < min_val) {
   min_val = A[i];
  }

  if (A[i] > max_val) {
   max_val = A[i];
  }
 }

 float range = max_val - min_val;

 List* B = new List[n];
 for (int j = 0; j < n; ++j) {
  B[j] = List();
 }

 for (int i = 0; i < n;++i) {
  int bucket_idx = (int)floor(n * (A[i] - min_val) / range);
  if (bucket_idx == n) bucket_idx = n - 1;
  B[bucket_idx].append(A[i]);
```

```
 }

 for (int j = 0; j < n; ++j) {
  B[j].insertionSort();
 }

 int i = 0;
 for (int j = 0; j < n; ++j) {
  for (int k = 0; k < B[j].getSize(); ++k) {
   A[i] = B[j].getElement(k);
   ++i;
   if (i >= n) break;
  }
 }

 return A;
}
```

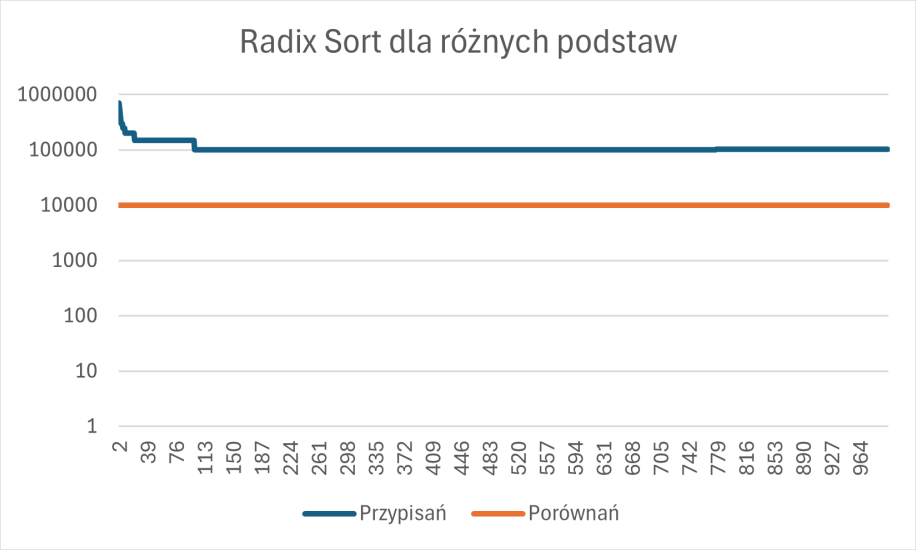
Określamy zakres wartości (najmniejszą i największą liczbę) tablicy. Znając go, dla każdej liczby z tablicy określamy, do którego "wiadra" ją przydzielić, czyli do którego podprzedziału zakresu wartości należy.

```
(int)floor(n * (A[i] - min_val) / range)
```

Po przydzieleniu, sortujemy każde "wiadro" z osobna z użyciem Insertion Sort na liście. Następnie przepisujemy zawartość kolejnych "wiader" do jednej, wynikowej tablicy czyniąc ją posortowaną.

3 Porównanie działania algorytmów

3.1 Radix Sort dla rożnych podstaw d



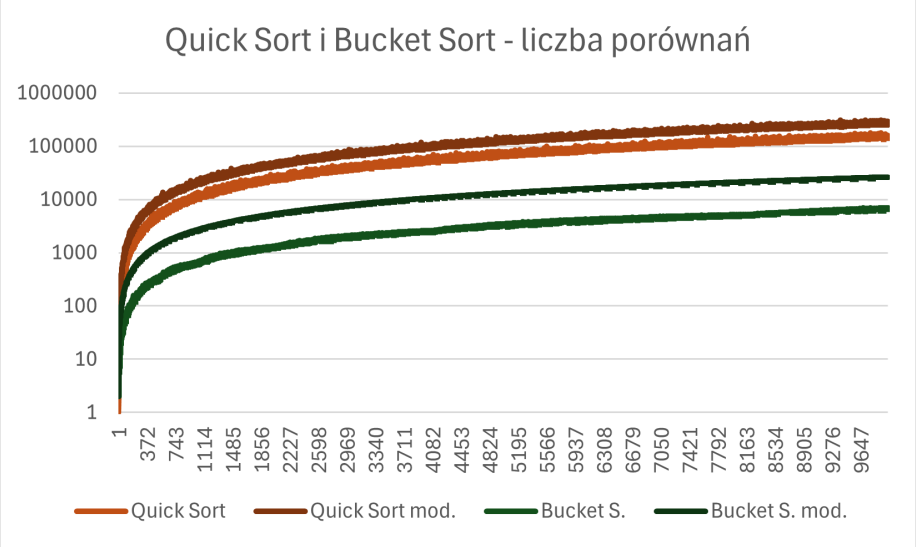
Dla zmniejszenia ilości przypisań, które musi wykonać Radix Sort, korzystne jest używanie systemów liczbowych składających się z wielu cyfr. Można zauważyć, że gdy liczba cyfr systemu rośnie, liczba przypisań spada istotnie w pewnych odstępach, co daje wizualne "schodki" na wykresie.

d	Liczba przypisań
5	300030
6	300036
7	250036
8	250041
9	250046
10	200042
11	200046

d	Liczba przypisań
20	200082
21	200086
22	150069
23	150072

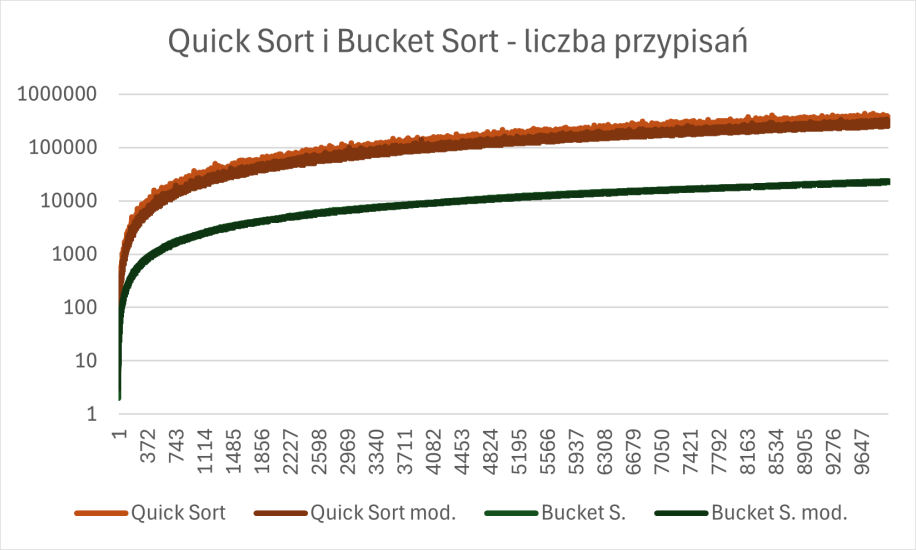
d	Liczba przypisań
98	150297
99	150300
100	100204
101	100206

3.2 Quick Sort kontra Bucket Sort



Liczba porównań dla danych różnych rozmiarów:

n	Quick Sort	Quick Sort mod.	Bucket Sort	Bucket Sort mod.
10	25	66	5	23
100	690	1285	60	263
1000	11266	20161	638	2618
10000	156495	283309	6953	26352



Liczba przypisań dla danych różnych rozmiarów:

n	Quick Sort	Quick Sort mod.	Bucket Sort	Bucket Sort mod.
10	82	49	20	24
100	1655	1236	220	236
1000	26761	20024	2272	2251
10000	363352	284213	23601	22727

4 Wnioski

Radix Sort działa bardziej efektywnie, gdy podstawa jest większa. Pozwala to zredukować liczbę iteracji dla kolejnych, bardziej znaczących cyfr. Należy jednak pamiętać, że wzrost podstawy powoduje większe wykorzystanie pamięci.

Na podstawie danych, Quick Sort wykazuje znacznie większą liczbę porównań i przypisań niż Bucket Sort. Modyfikacja zwiększyła liczbę porównań dla obu algorytmów, w przypadku Bucket Sort bardziej znacznie. Zmniejszyła liczbę przypisań w przypadku Quick Sort, dla Bucket Sort różnica jest zaniedbywalna (dla badanych rozmiarów tablicy).