Lista 2

Mateusz Kaczkowski

November 2025

Lista 2 polegała na napisaniu 3 algorytmów: quicksort, radixsort i bucketsort oraz modyfikacji quicksorta i własnej struktury listy z sortowaniem insertionsort, a następnie porównaniu wydajności tych algorytmów.

1 Najciekawsze fragmenty kodu

1.1 QuickSort-3

Jaki jest quicksort każdy widzi, ale jego modyfikacja dzieląca na 3 jest znacznie bardziej skomplikowana. Oprócz wyznaczania drugiego piwota (tutaj używam elementu przedostatniego), musi wyznaczyć 3 różne sekcje na kolejne elementy - przed pierwszym piwotem, między i po drugim. Żeby to uzyskać trzymam 2 indeksy - element po pierwszej i, oraz po drugiej sekcji j.

Jeśli element powinien trafić do drugiej sekcji zostaje zamieniony z elementem j, a następnie zwiększamy j by powiększyć sekcję 2 o ten nowy element. Jeżeli coś ma trafić do sekcji 1 to zamieniamy nowy element z elementem j, a następnie j z i oraz zwiększamy oba indeksy o 1. W ten sposób nowy element jest dodawany do sekcji 2, a następnie pierwszy element sekcji 2 zamieniany jest z ostatnim/nowym i obie sekcje powiększają się by objąć nowe elementy.

Ostatnim krokiem jest przeniesienie piwotów na odpowiednie miejsca jednocześnie zachowując sekcje i posortowanie każdej sekcji osobno.

```
void quickSort3(float arr[], int low, int high) {
    if (low < high) {

        int mid = high - 1;

        if (arr[mid] > arr[high])
            swap(arr[mid], arr[high]);

        float pivot1 = arr[mid];
        float pivot2 = arr[high];

        int i = low;
```

```
        int j = low;

        for (int n = low; n < high - 1; n++) {
            if (arr[n] <= pivot1) {
                swap(arr[j], arr[n]);
                swap(arr[i], arr[j]);
                i++;
                j++;
            }
            else if (arr[n] <= pivot2) {
                swap(arr[j], arr[n]);
                j++;
            }
        }

        swap(arr[j], arr[mid]);
        swap(arr[i], arr[j]);
        j++;
        swap(arr[j], arr[high]);

        quickSort3(arr, low, i - 1);
        quickSort3(arr, i + 1, j - 1);
        quickSort3(arr, j + 1, high);
    }
}
```

1.2 Radixsort

Radixsort zmienia się niewiele by móc obsługiwać inne podstawy i wartości ujemne. Ilość sekcji zmienia się z 10 (0...9) do $2d-1$ $(-(d-1)...d-1)$ a wartość największa zmienia się na największą wartość bezwzględną.

```
void radixsort(int arr[], int n, int d)
{
    int m = abs(arr[0]);
    for (int i = 1; i < n; i++)
        if (abs(arr[i]) > m)
            m = abs(arr[i]);


    for (int ex = 1; m / ex > 0; ex *= d) {
        int* output = new int[n];
        int* count = new int[(2 * d) - 1]{ 0 };
```

```
        for (int i = 0; i < n; i++)
            count[((arr[i] / ex) % d) + d - 1]++;

        for (int i = 1; i < 2 * d - 1; i++)
            count[i] += count[i - 1];

        for (int i = n - 1; i >= 0; i--) {
            output[count[((arr[i] / ex) % d) + d - 1] - 1] = arr[i];
            count[((arr[i] / ex) % d) + d - 1]--;
        }

        for (int i = 0; i < n; i++)
            arr[i] = output[i];

        delete[] output;
        delete[] count;
    }
}
```

1.3 Lista i insertionsort

Listy nie będę wklejał lub opisywał, większość tego kodu to zarządzanie pamięcią i dbanie o wskaźniki. Lista po prostu trzyma node początkowy i końcowy, każdy node ma wartość oraz odniesienie do poprzednika i następcy. Funkcje dodają element, usuwają, dają dostęp i iterują po całej liście, np do wyświetlenia.

Sam insertionsort napisałem trochę inaczej niż klasyczna wersja na tablicy, która zamiast "ruszać" elementami, nadpisuje kolejne elementy poprzednimi, zostawiając jeden duplikat jako wolne miejsce. Zamiast tego, jako że elementy listy istnieją osobno i tylko trzymają do siebie odniesienia, napisałem ten algorytm aby sortowany element był odczepiany od swojego miejsca i doczepiany gdzie powinien być.

```
void insertionSort() {
    if (count <= 1) return;
    if (count == 2) {
        if (start->key > head->key) swap(start->key, head->key);
        return;
    }
    Node* currentHead = start->next;

    for (Node* currentHead = start->next; currentHead != nullptr; ) {
        Node* nextHead = currentHead->next, * sortingHead = currentHead->prev;

        if (sortingHead->key > currentHead->key) {
```

```
            currentHead->prev->next = currentHead->next;
            if (currentHead->next != nullptr)
                currentHead->next->prev = currentHead->prev;
            else
                head = currentHead->prev;

            while (sortingHead != nullptr && sortingHead->key > currentHead->key)
                sortingHead = sortingHead->prev;

            if (sortingHead == nullptr) {
                start->prev = currentHead;
                currentHead->prev = nullptr;
                currentHead->next = start;
                start = currentHead;
            }
            else {
                currentHead->prev = sortingHead;
                currentHead->next = sortingHead->next;
                if (sortingHead->next != nullptr)
                    sortingHead->next->prev = currentHead;
                else
                    head = currentHead;
                sortingHead->next = currentHead;
            }
        }
        currentHead = nextHead;

    }
}
```

1.4 Bucketsort

Normalny bucketsort działa wyłącznie dla wartości od 0 do 1, aby podzielić je na n bucketów wybierając bucket przez mnożenie wartości razy n. Aby zmienić go na działający dla dowolnych wartości, wystarczy odpowiednio przeskalować wybieranie bucketa przez znalezienie wartości minimalnej i maksymalnej - z $n \times x$ do $n \times \frac{x-min}{max+min+1}$.

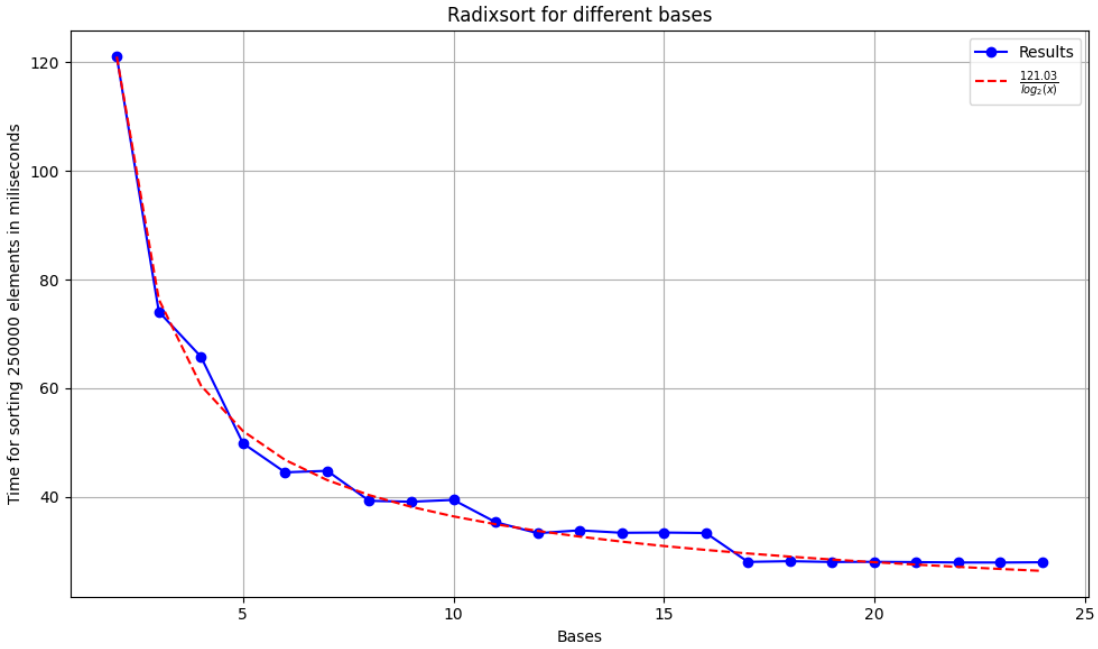
```
void bucketSort(float arr[], int n) {
    float max = arr[0], min = arr[0];
    for (int i = 0; i < n; i++) {
        if (arr[i] > max) max = arr[i];
        if (arr[i] < min) min = arr[i];
    }
```

```
    List<float>* b = new List<float>[n];
    for (int i = 0; i < n; i++)
        b[(int)(n * (arr[i] - min) / (max - min + 1))].push(arr[i]);

    for (int i = 0; i < n; i++)
        b[i].insertionSort();

    int index = 0;

    for (int i = 0; i < n; i++) {
        List<float>::Iterum val;
        while (b[i].iterate(val))
            arr[index++] = val;
    }
    delete[] b;
}
```

2 Porównanie działania algorytmów

2.1 Radixsort dla rożnych podstaw

Testy radixsorta zostały przeprowadzone dla rozsądnie dużej tablicy (250 000 elementów) i dla kolejnych podstaw od 2 do 24, z wynikiem podanym w mikrosekundach. Czas wykonania może się zmieniać zależnie od sprzętu i obciążenie, ale jako że wszystkie testy wykonują się bezpośrednio jedno po drugim, pokazuje to ich względną prędkość.



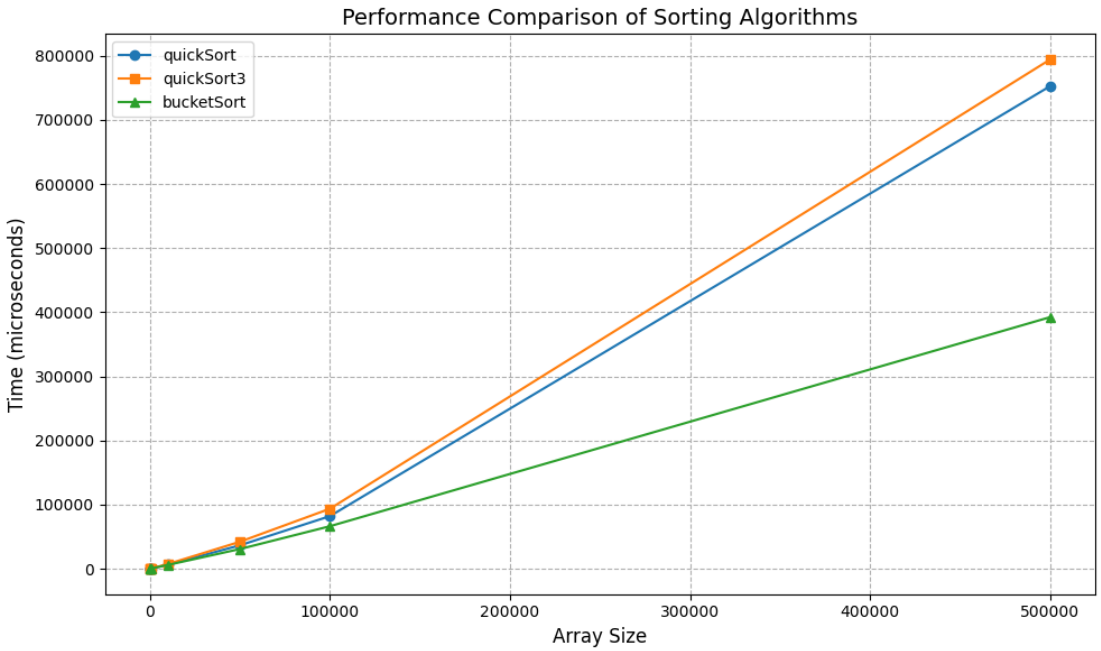
Można zauważyć, że czas wykonania spada wraz ze wzrostem podstawy d proporcjonalnie do $\frac{1}{\log d}$. Dokładniej $\frac{czas\ d_1}{czas\ d_2} = \frac{\ln d_2}{\ln d_1}$.

2.2 Porównanie pozostałych algorytmów

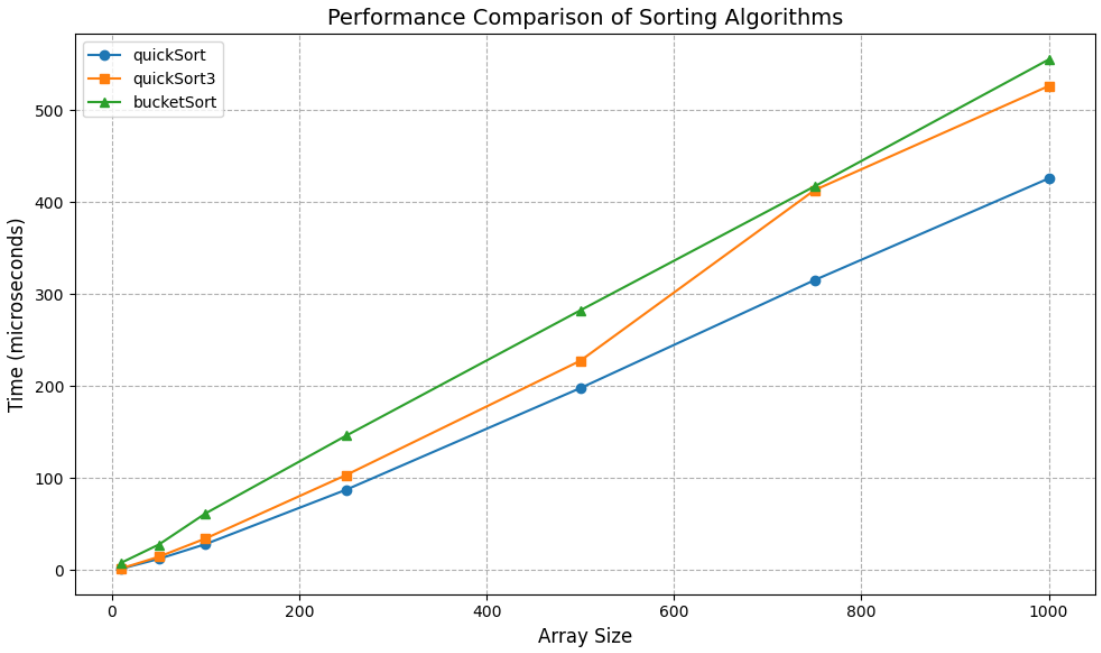
Quicksort i quicksort3 mają lepszą wydajność dla małych tablic ale w pewnym momencie (∼10 000 elementów) teoretyczna prędkość $O(n)$ bucketsorta przewyższa $O(n \log n)$ i utrzymuje przewagę już do końca. Znacznie ciekawsze wydaje mi się, że quicksort3 nie tylko nie ma spodziewanej przewagi nad standardową wersją algorytmu ($\log_2 3$), jest on odrobinę wolniejszy.

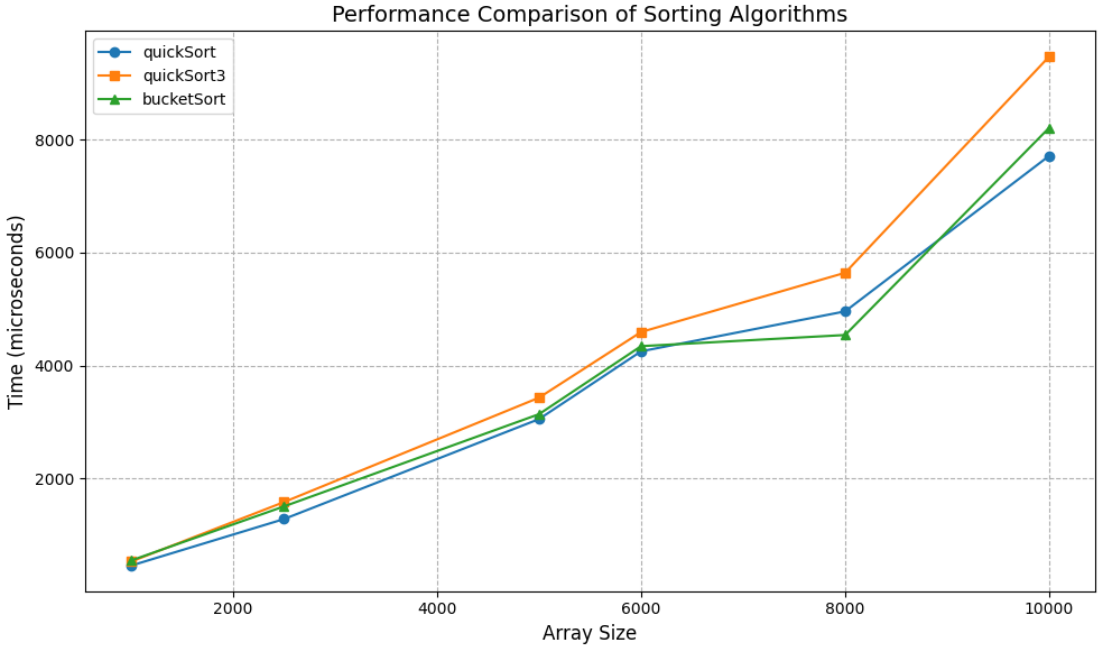
Array length	**quickSort**	**quickSort3**	**bucketSort**
10	1.78	2.22	5.57
50	12.66	14.92	29.88
100	29.77	37.87	58.04
1000	424.91	502.69	516.12
10000	6207.42	7196.90	5818.73
50000	36374.40	41965.70	30616.20
100000	81902.20	93448.00	66192.30
500000	752485.00	794100.00	392243.00

Tabela 1: Performance comparison of sorting algorithms (in microseconds)



Dla lepszej widoczności uwzględniłem też osobny graf dla małych tablic oraz dla "średnich", czyli poniżej tysiąca elementów i od 1000 do 10 000. W tych przedziałach trudno było mi zaobserwować konkretne punkty w których jeden algorytm staje się szybszy od drugiego. Podejrzewam że są to artefakty mierzenia czasu na rzeczywistej maszynie, nie matematycznej abstrakcji.


Performance Comparison of Sorting Algorithms
quickSort
quickSort3
bucketSort
Time (microseconds)
Array Size
0
100
200
300
400
500
0
200
400
600
800
1000


Performance Comparison of Sorting Algorithms
quickSort
quickSort3
bucketSort
Time (microseconds)
Array Size
2000
4000
6000
8000
2000
4000
6000
8000
10000

3 Wnioski

Radixsort najczęściej pisany jest dla podstawy w okolicach 10, co na grafie wygląda na uzasadnione. Mniejsze podstawy znacznie zwiększają konieczny czas, a większe nie dają benefitów proporcjonalnych do dodatkowego zużycia pamięci. Podejrzewam jednak że z samej architektury komputera wyniknie, że najbardziej optymalną wartością jest 8 lub 16, bo umożliwiają one zastąpienie dzielenia przez shift bitowy.

Co do algorytmów quicksort, quicksort3 i bucketsort zaskoczyły mnie one bardzo. Wcześniej nie słyszałem o algorytmie sortującym w czasie $O(n)$, a ten nie zużywa nawet wcale tak dużo pamięci. Równie dużym zaskoczeniem był fakt, że quicksort3 radził sobie stabilnie gorzej od normalnej wersji, choć nie bardzo - z testów jakie wykonałem nie wygląda to na asymptotyczną różnicę. Jest to sprzeczne z wnioskami z poprzedniej listy, gdzie algorytmy oparte na systemie trójkowym były zauważalnie lepsze. Zastanawia mnie jakie wyniki osiągnęłyby dla inteligentnie wybieranych wartości pośrednich.