Analiza i porównanie algorytmów sortowania

Tomasz Warzecha

December 5, 2024

1 Wstęp

W sprawozdaniu moim celem jest analiza i porównanie wydajności trzech algorytmów sortowania wraz z ich modyfikacjami. Quick sort, bucket sort, radix sort zostały zaimplementowane, tak aby przeprowadzić na nich różne testy wydajności. Badania te skupiają się na generowaniu losowych danych, które następnie są używane do obliczenia czasu wykonania algorytmów. Wyniki tych testów mają na celu porównanie algorytmów, a także pokazanie różnic między teoretycznymi, a praktycznymi wynikami.

2 Implementacja algorytmów

Algorytmy zostały zaimplementowane zgodnie z klasycznymi wersjami algorytmów sortujących, korzystając z pseudokodów zamieszczonych na stronie przez prof. Szymona Żeberskiego. Wprowadzone zostały także modyfikacje, które delikatnie zmieniają działanie algorytmów. Algorytmy quick sort oraz bucket zostały zaimplementowane dla tablic z liczbami typu `float`, natomiast radix sort - `int`.

Najciekawsze fragmenty kodu

Poniżej przedstawię najciekawsze fragmenty kodu, wraz z objaśnieniami, tak aby łatwiej zrozumieć działanie algorytmów.

2.1 *Radix sort*

Do sortowania w sortowaniu pozycyjnym potrzebujemy dodatkowego algorytmu sortującego - w tym wypadku użyliśmy sortowania przez zliczanie. Sortowanie pozycyjne iteruje przez kolejne cyfry liczb w ustalonej podstawie, a następnie sortuje tablice względem danej cyfry za pomocą counting sort.

```
for (int poz = 1; max / poz > 0; poz *= baza) {
    coutingSort(A, n, baza, poz);
}
```

Funkcja `coutingSort` stabilnie sortuje nasze liczby według ustalonej cyfry. Robi to za pomocą dzielenia całkowitego i reszty z dzielenia przez podstawę systemu liczbowego:

```
int znak = (A[i] / poz) % baza;
B[C[znak] - 1] = A[i];
C[znak]--;
```

2.2 *Quick sort*

Sortowanie szybkie wykorzystuje strategię dziel i zwyciężaj. Wybieramy nasz pivot(w tym wypadku ostatni element) i za pomocą funkcji partition dzielimy tablicę na dwie części - liczby mniejsze od wartości pivota i większe od niej:

```
void quickSort(float A[],int p, int k){
    if(p < k){
        int s = partition(A,p,k);
        quickSort(A,p,s-1);
        quickSort(A,s+1,k);
    }
}
```

2.3 *Bucket sort*

Sortowanie kubełkowe grupuje dane w przedziały(kubełki), a następnie wewnątrz nich stosuje dodatkowe sortowanie (w tym wypadku insertion sort). Posortowane kubełki złącza w następujący sposób w naszą posortowaną listę.

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

3 Porównanie algorytmów

3.1 Porównanie teoretyczne

Poniżej zestawiono złożoności czasowe, stabilność oraz złożoność pamięciową dla klasycznych wersji algorytmów:

Algorytm	**Optymistyczna**	**Pesymistyczna**	**Pamięciowa**	**Stabilność**
Radix sort	$O(nk)$	$O(nk)$	$O(n+k)$	Tak
Quick sort	$O(n \log n)$	$O(n^2)$	$O(\log n)$	Nie
Bucket sort	$O(n+k)$	$O(n^2)$	$O(n+k)$	Tak

Table 1: Porównanie teoretyczne algorytmów sortowania

3.2 Porównanie czasów działania

Zaprezentowano wyniki pomiarów na wykresach:

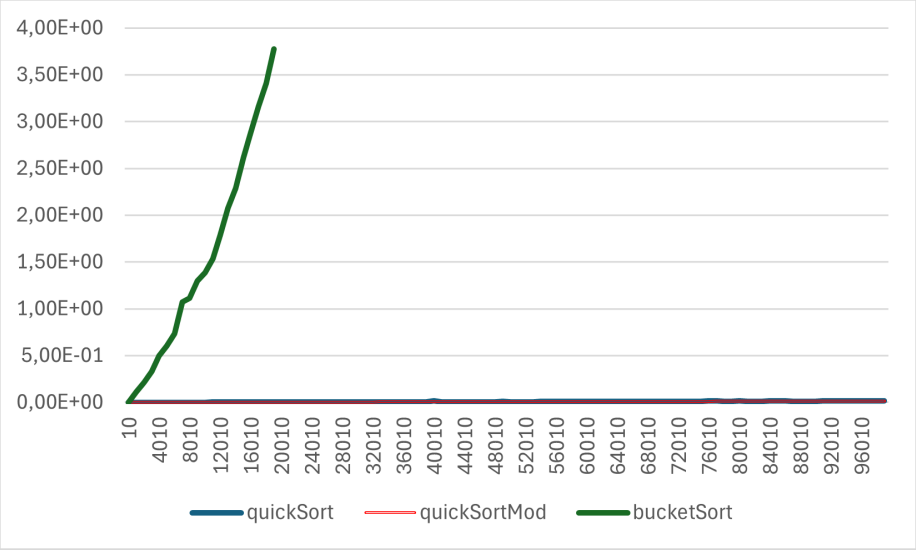

Figure 1: Porównanie czasów działania *quick sort*, jego modyfikacji i *bucket sort*

Na wykresie możemy zobaczyć, że quick sort jak i jego modyfikacja są znacząco szybsze. Jest to spowodowane dużo większą złożonością algorytmu sortowania kubełkowego. Natomiast możemy również zobaczyć, że nasza modyfikacja szybkiego sortowania znacząco nie zmieniła wyników.

Quick sort i radix sort Porównamy teraz czasy działania dwóch podobnych złożeniowo algorytmów.

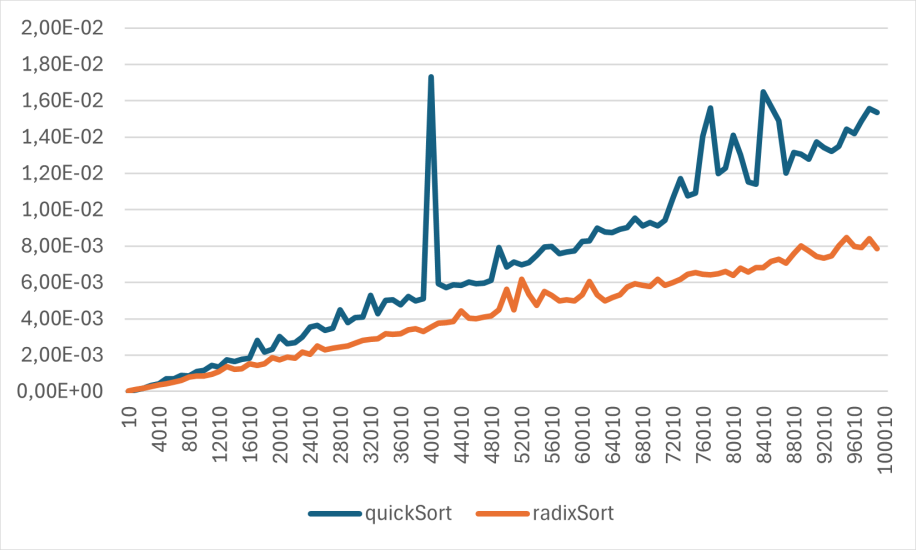

Figure 2: Porównanie czasów działania *quick sort* i *radix sort*

Na wykresie możemy zobaczyć, że quick sort jak i radix sort mają bardzo podobne czasy działania, jest to spowodowane podobną złożonością obliczeniową, różnice są na poziomie stałej

4 Porównanie modyfikacji algorytmów

4.1 *Radix sort*

Zmodyfikowana wersja obsługuje liczby ujemne, dzieląc dane na dodatnie i ujemne, następnie ujemne zmienia na dodatnie, sortuje i w odwrotnej kolejności dodaje do pierwotnej tablicy wraz ze zmienionym znakiem

```
for(int i = 0; i<n; i++){
    if(A[i] < 0){
        U[j] = -A[i];
        j++;
    }else{
        D[l] = A[i];
        l++;
    }
}

radixSort(D, l, baza);
radixSort(U, j, baza);
```

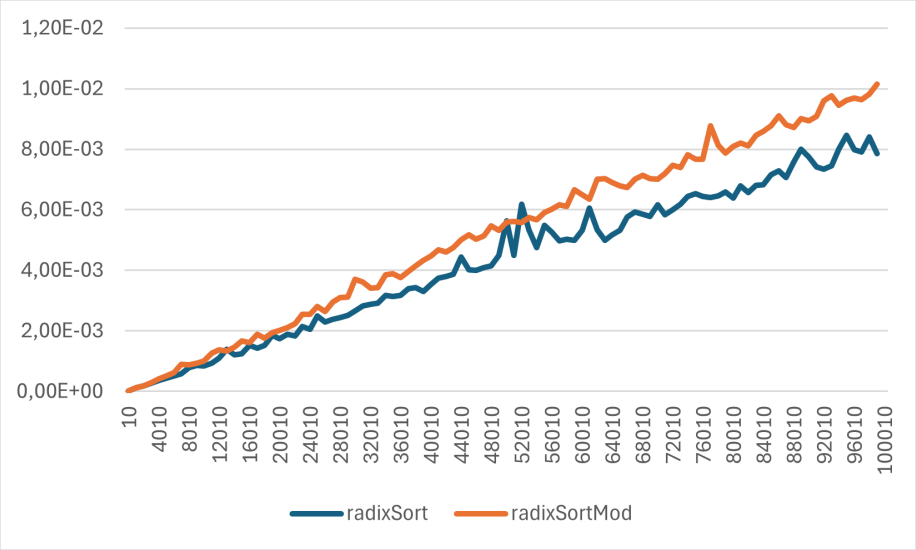

Figure 3: Porównanie czasów działania *radix sort* i jego modyfikacji

Dodanie obsługi liczb ujemnych zwiększa elastyczność algorytmu kosztem dodatkowej pamięci.

4.2 *Quick sort*

Modyfikacja *quick sort* polega na podziale tablicy za pomocą dwóch pivotów, a następnie rozdzieleniu na trzy części tablicy, tak jak jest to zrobione w poniższym kodzie:

```
        if (A[j] < x) {
            std::swap(A[i], A[j]);
            i++;
        } else if (A[j] > x2) {
            std::swap(A[k], A[j]);
            k--;
        }
```

4.3 *Bucket sort*

Zmodyfikowana wersja dynamicznie dopasowuje przedziały kubełków - normalizuje je tak, aby nie wypaść za przedział miedzy 0 a 1, nastepnie tak jak w oryginale, przypisuje do kubełków i sortuje wewnątrz:

```
int b = n * (A[i] - min) / (max - min);
```

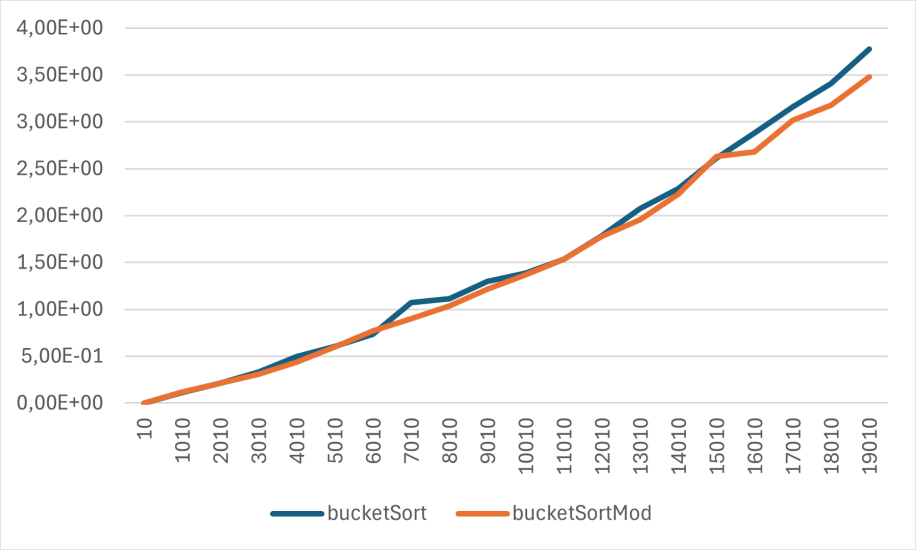

Figure 4: Porównanie czasów działania *bucket sort* i jego modyfikacji

5 *Insertion sort* w listach jednokierunkowych

5.1 Opis działania

Struktura listy jednokierunkowej została zainicjowana wraz z podaniem wartości:

```
SLel *L;
SLel *e,* p;
int n=6;
int values[] = {5, 2, 9, 1, 5, 6};
```

A następnie *Insertion sort* szuka miejsca dla każdego elementu z *values* a następnie wstawia w nie łącząc się z poprzednim oraz następnym miejscem:

```
for (p = L; v > p->next->data; p = p->next);
e = new SLel;
e->data = v;
e->next = p->next;
p->next = e;
```

Algorytm cechuje się prostotą implementacji, ale ma złożoność $O(n^2)$.

6 Wnioski

Przeprowadzone badania pokazują, że algorytmy *radix sort* i *bucket sort* są bardziej wydajne w scenariuszach z dużymi danymi. *Quick sort* pozostaje uniwersalnym wyborem dzięki prostocie implementacji. Modyfikacje poprawiają

działanie w specyficznych przypadkach, jednak kosztem większego skomplikowania kodu.