Sprawozdanie – Algorytmy sortowania
Lista 2

Alexander Genov

1 Wprowadzenie

Na tej liście trzeba było zaimplementować kilka klasycznych algorytmów sortowania: QUICK SORT (także wersję z trzema częściami), RADIX SORT dla różnych podstaw d, INSERTION SORT na własnej liście, oraz BUCKET SORT i jego modyfikację dla dowolnych danych.

Wszystkie pomiary zrobione dla danych losowych generowanych `mt19937` w C++.

2 Wybrane fragmenty kodu

2.1 Najciekawszy fragment – modyfikacja BUCKET SORT

To moim zdaniem najciekawsza część całej listy, bo trzeba było "rozciągnąć" klasyczny BUCKET SORT (który działa tylko na $[0, 1)$) na dowolne dane. Zrobiłem to po prostu przez normalizację:

$$x' = \frac{x - \min}{\max - \min}.$$

Działa to niestety nie za bardzo dobrze, bo trzeba jeszcze zabiezpieczyć się od nieścisłości obliczeniowej, która występuje w C++. Najpierw miałem pomysł jeszcze dzielić przez jaką bardzo małą liczbę, żeby ppo normalizacji liczby były mniejsze od jedynki.

$$x' = \frac{x - \min}{\max - \min + \min_{float}}.$$

Ale okazało się, że o tyle małej liczby nie starczy. Więc, w bucket sort trzeba było zrobić w taki sposób:

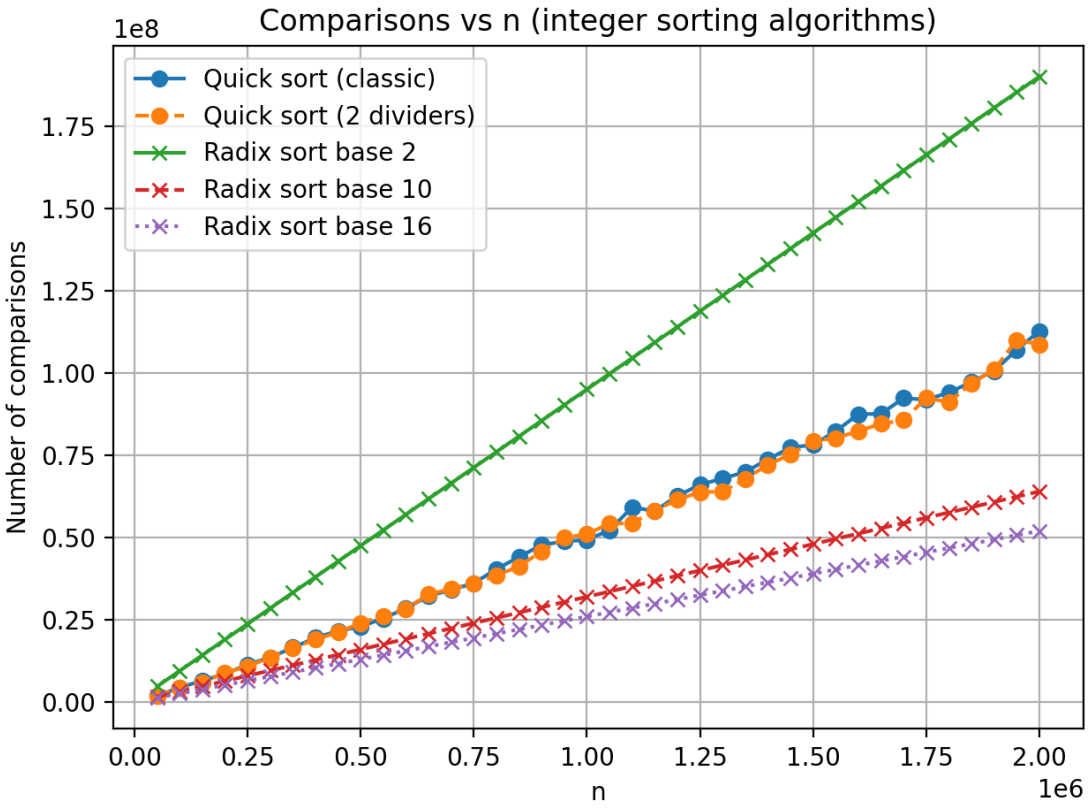
```
        if (index < 0) index = 0;        // just in case of some
           weird float values
        if (index >= n) index = n - 1;
```

```
for (int i = 0; i < n; i++)
   {
       list[i] = (list[i] - min_val)/(max_val - min_val);
   }

   if (max_val == min_val) {
       return;
   }

```

```
10      bucket_sort(list, n);
11
12      for (int i = 0; i < n; i++)
13      {
14          list[i] = list[i] * (max_val - min_val) + min_val;
15      }
```

3 Wyniki testów

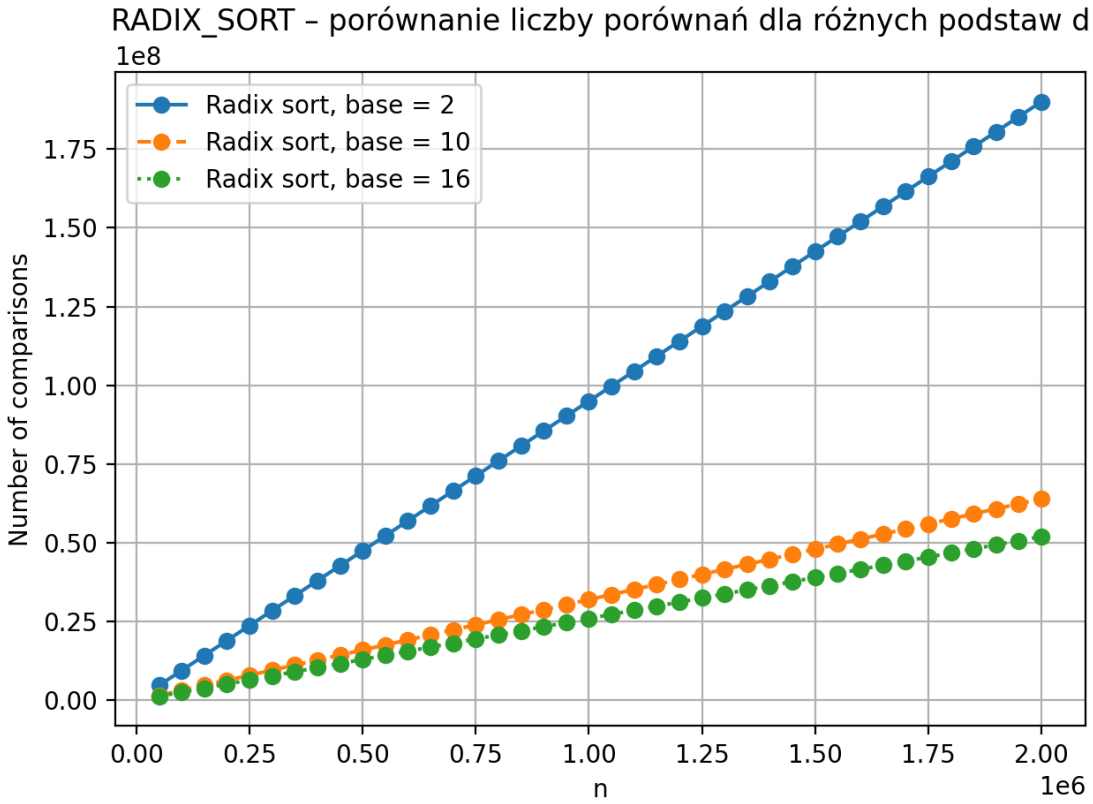
3.1 Porównanie algorytmów całkowitoliczbowych



Rysunek 1: Porównania – QUICK, QUICK (3 części) i RADIX (2,10,16).

Na wykresie widać, że dla większych danych RADIX SORT wygrywa, zwłaszcza dla podstawy 16. Ale modyfikacja z `base=2` ma najwięcej operacji.

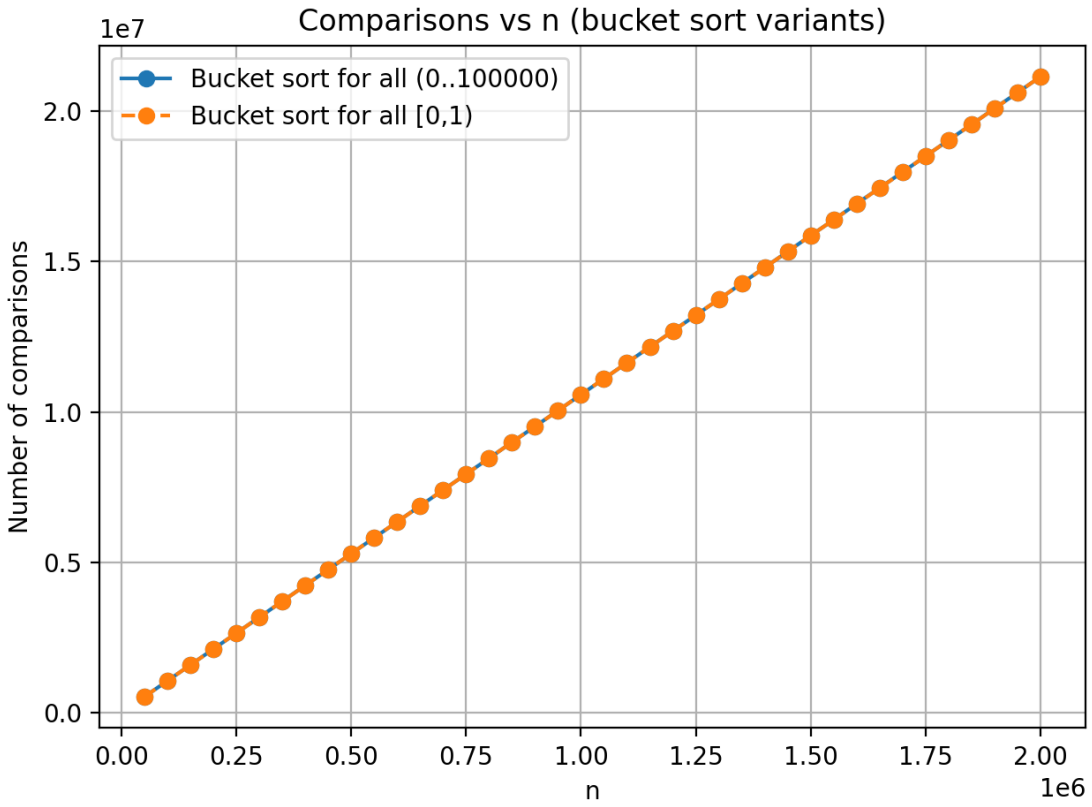
3.2 RADIX SORT – różne podstawy



Rysunek 2: RADIX: wpływ podstawy d.

Baza 2 robi bardzo dużo kroków (każdy bit osobno), więc wypada najgorzej. Bazy 10 i 16 mają o wiele mniej etapów i dzięki temu krzywa rośnie wolniej. Najlepsze wyniki dała baza 16.

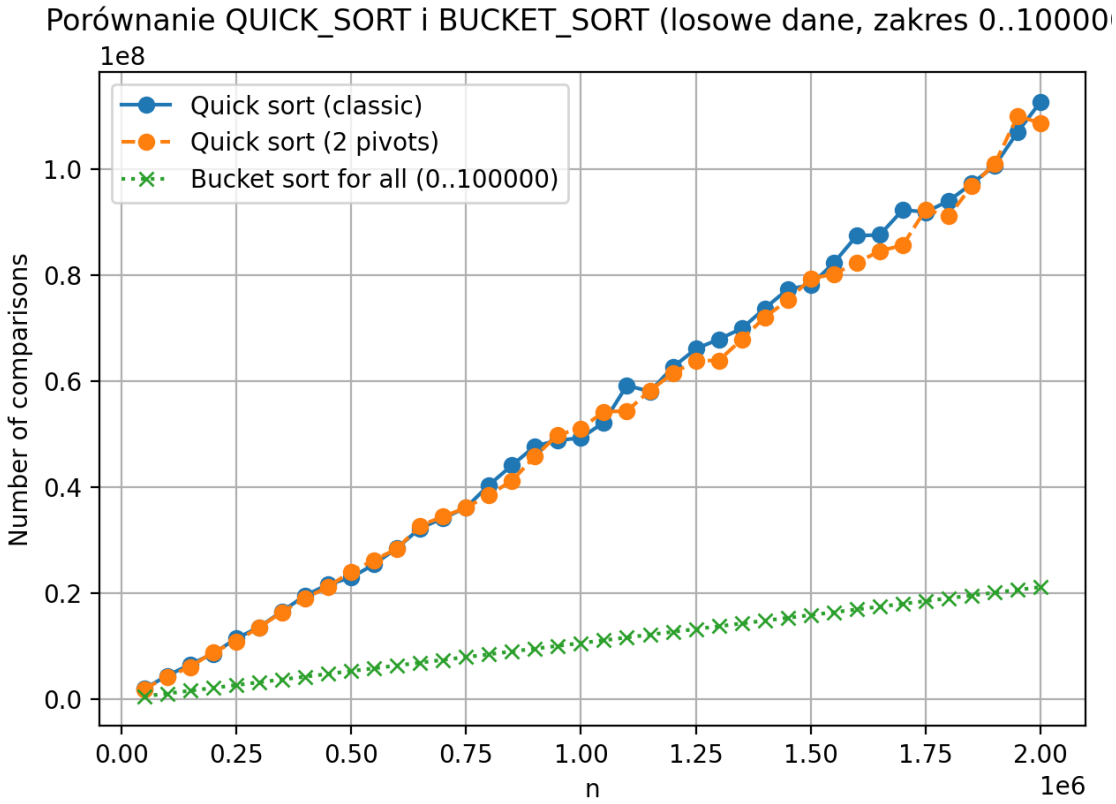
3.3 BUCKET SORT – dane $(0, 1)$ vs dane $0..100000$



Rysunek 3: BUCKET – porównania dla dwóch zakresów.

Co ciekawe, wykresy na różnych zakresch danych prawie idealnie się pokrywają. To oznacza, że poprawnie działa zarówno modyfikacja, jak i sam generator C++ (rozkłada dane równomiernie, bo intuicyjnie wydawało się, że na większych przedziałach elementy będą się rozkładały nierównomiernnie).

3.4 QUICK SORT vs BUCKET SORT



Rysunek 4: Porównania: QUICK i BUCKET.

Dla losowych danych BUCKET SORT radzi sobie zaskakująco dobrze i przy dużych n ma mniej operacji niż QUICK (nawet ten z dwoma pivotami).

4 Wnioski

- RADIX SORT (szczególnie baza 16) jest najszybszy dla liczb całkowitych.
- QUICK z dwoma pivotami działa trochę lepiej niż zwykły QUICK, ale czasami może być nawet wolnieszy.
- BUCKET SORT wygrywa z QUICK SORT dla równomiernych danych.
- Generator losowy w C++ faktycznie rozrzuca liczby równomiernie.

Cały kod znajduję się w katalogu `lista2` na GitHub.