Algorytmy Sortowania

Projekt 1
Projektowanie i Analiza Algorytmów

Danil Krassotov 282656

Wstęp

Celem niniejszego sprawozdania jest implementacja wybranych algorytmów sortowania oraz analiza ich efektywności. W ramach pracy zaimplementowano algorytmy Timsort, Dual-pivot Quicksort i sortowanie przez kopcowanie. Dla każdego z algorytmów zbadano czas sortowania dla tablic o różnych rozmiarach oraz wygenerowano wykresy pokazujące ich złożoności.

1 Opis Algorytmów

1.1 Timsort

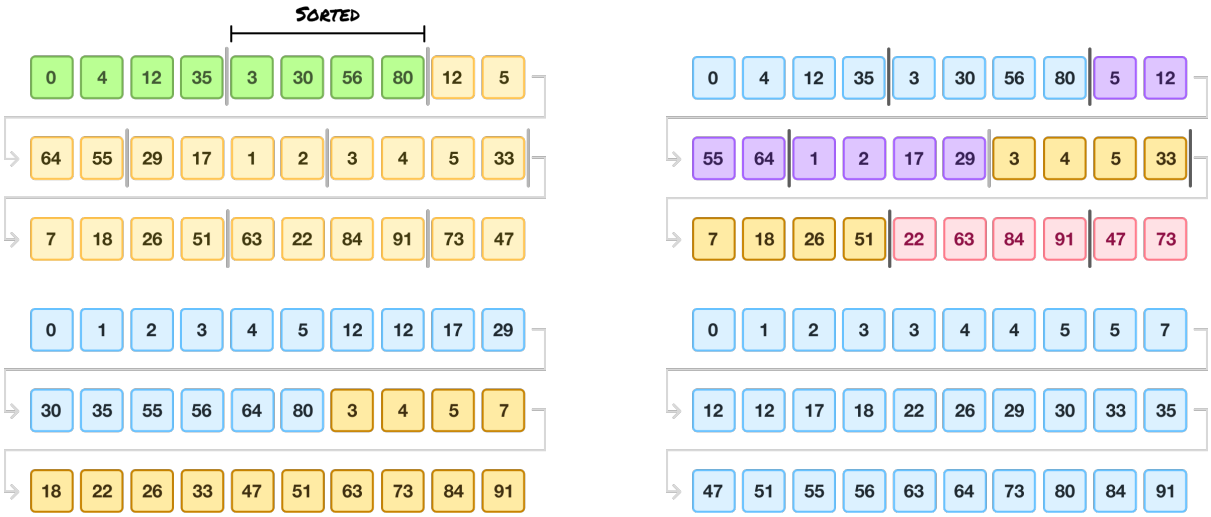
Timsort to hybrydowy algorytm sortowania, który łączy **sortowanie przez wstawianie (insertion_sort)** i **sortowanie przez scalanie (merge)**. Algorytm, pokazany na Listing 1 jest uproszczoną wersją Timsorta, zawierającą podstawowe elementy takie jak dzielenie na runy, sortowanie przez wstawianie oraz scalanie runów metodą merge sort. Nie uwzględnia jednak pełnej logiki stosu runów oraz zaawansowanych optymalizacji. Timsort jest skuteczny przy sortowaniu dużych zbiorów danych z już częściowym porządkiem, jednak w sytuacjach, gdzie dane są całkowicie losowe, może nie mieć znaczącej przewagi nad innymi algorytmami.
Złożoność obliczeniowa:

- Najlepszy przypadek: *O(n)* – przy już posortowanych danych.
- Średni przypadek: *O(n log(n))*
- Najgorszy przypadek: *O(n log(n))*
- Złożoność pamięciowa w najgorszy przypadku: *O(n)*

```cpp
void timsort(vector<int>& arr) {
    int n = arr.size();
    int mr = minrun(n);

    for (int i = 0; i < n; i += mr) {
        int end = min(i + mr - 1, n - 1);
        insertion_sort(arr, i, end);
    }

    for (int size = mr; size < n; size *= 2) {
        for (int start = 0; start < n; start += 2 * size) {
            int mid = start + size - 1;
            int end = min(start + 2 * size - 1, n - 1);

            if (mid < end) {
                merge(arr, start, mid, end);
            }
        }
    }
}
```

Listing 1: Timsort - implementacja w C++

Wizualnie algorytm można przedstawić w następujący sposób:



Rysunek 1: Timsort

1.2 Sortowanie przez kopcowanie

Ten algorytm oparty jest na strukturze danych zwanej kopcem. Algorytm buduje kopiec z tablicy wejściowej, a następnie usuwa elementy z korzenia kopca i umieszcza je na końcu tablicy.
Algorytm przedstawiony na Listingu 2 zawiera dwie wersje sortowania:

- **heapsort()** - sortowanie rosnące,
- **heapsort_reverse** - sortowanie malejące.

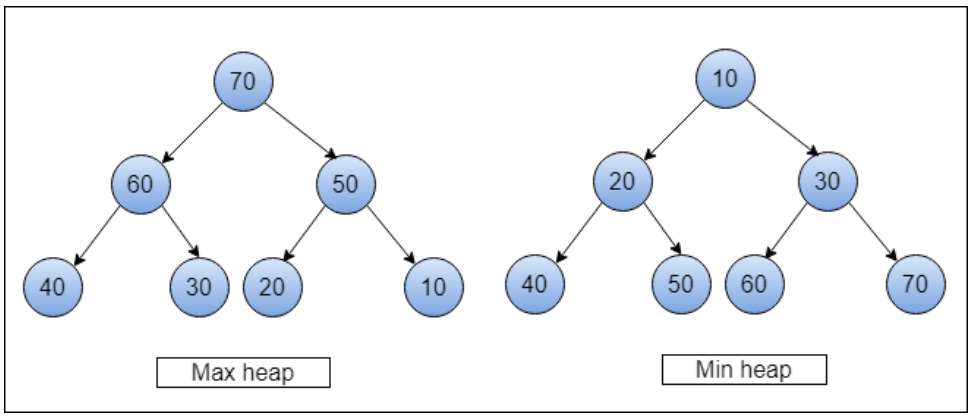
```
void heapsort(vector<int> &arr) {
    int n = arr.size();
    for (int i = n / 2 - 1; i >= 0; i--) {
        kopiec_max_to_min(arr, i, n);
    }
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        kopiec_max_to_min(arr, 0, i);
    }
}

void heapsort_reverse(vector<int> &arr) {
    int n = arr.size();
    for (int i = n / 2 - 1; i >= 0; i--) {
        kopiec_min_to_max(arr, i, n);
    }
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        kopiec_min_to_max(arr, 0, i);
    }
}
```

Listing 2: Heapsort - implementacja w C++

W obu przypadkach dane są najpierw przekształcane w kopiec binarny, po czym największy/najmniejszy element jest zamieniany z ostatnim i usuwany z kopca. Następnie przywracana jest struktura kopca i proces powtarza się do momentu posortowania całej tablicy. Algorytm nie wymaga dodatkowej pamięci i gwarantuje złożoność obliczeniową równą O(n log(n)) niezależnie od danych wejściowych. Wada to nieco niższa wydajność w porównaniu do np. Timsorta.
Funkcja z kopcem minimalnym (**heapsort_reverse**) została użyta do utworzenia tablicy posortowanej w odwrotnej kolejności podczas pomiaru czasu działania funkcji (**heapsort()**) na różnych tablicach.
Złożoność obliczeniowa:

- Najlepszy przypadek: *O(n log(n))*
- Średni przypadek: *O(n log(n))*
- Najgorszy przypadek: *O(n log(n))*

- Złożoność pamięciowa: *O(1)* – algorytm działa w miejscu (in-place)



Rysunek 2: Wizualizacja maksymalnego i minimalnego kopca

1.3 Dual-Pivot Quicksort

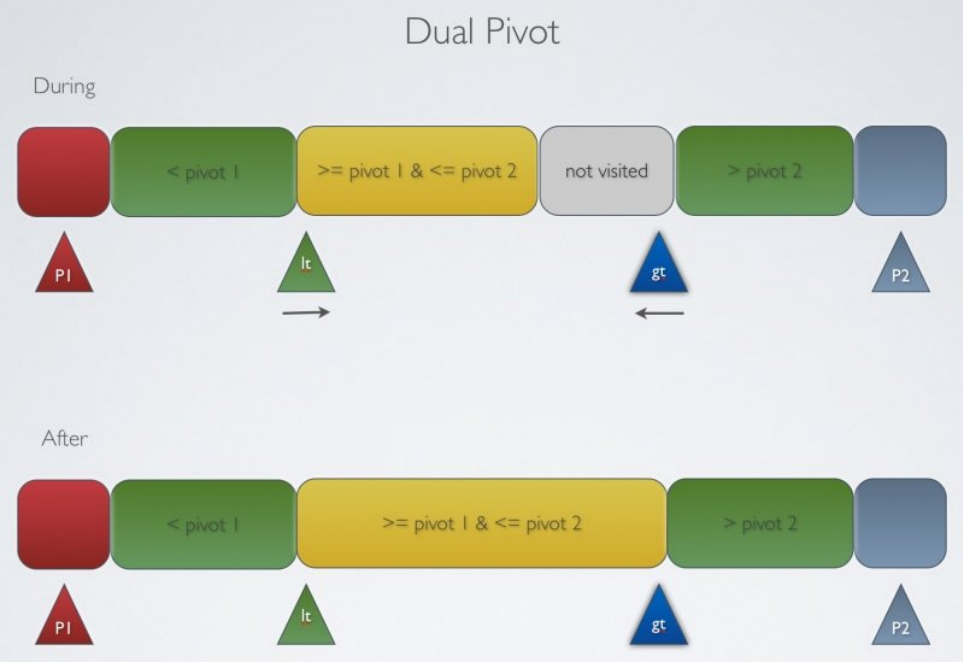
Dual-pivot quicksort to wariant algorytmu quicksort, który zamiast jednego pivot'u wykorzystuje dwa. Tablica dzielona jest na trzy części: elementy mniejsze od pierwszego pivotu, elementy pomiędzy pivotami oraz elementy większe od drugiego pivot'u. Algorytm powtarza się dla każdej części, aż nie zostanie posortowana tablica.
W implementacji przedstawionej na Listing 3 znajdują się dwie wersje algorytmu: jedna sortująca rosnąco (qsort), druga odwrotnie (qsort_reverse). Podobnie jak w przypadku poprzedniego algorytmu wersja odwrotna używana była do utworzenia tablicy już uporządkowanej, ale w kolejności malejącej, dla obserwacji zachowania algorytmu w teoretycznie "najgorszym"przypadku.
Dual-pivot quicksort jest prosty do implementacji, szybki i wymaga dodatkowej pamięci jedynie dla rekurencji. Jednak jest bardzo wrażliwy na dane wejściowe i dobór pivot'ów: jeśli dane są często duplikowane lub już posortowane, może to znacznie zmniejszyć wydajność. Dla dużych i niekorzystnych danych algorytm może powodować bardzo głębokie wywołania funkcji co skutkuje przepełnieniem stosu i zatrzymaniem programu.
Złożoność obliczeniowa:

- Najlepszy przypadek: *O(n log(n))*i.
- Średni przypadek: *O(n log(n))*
- Najgorszy przypadek: $O(n^2)$ – przy złym wyborze pivotów.
- Złożoność pamięciowa: *O(log(n))* – w wersji rekurencyjnej, bez dodatkowej alokacji pamięci.

```
1 void dualpivot(vector<int> &arr, int left, int right) {
2     ...
3
4     while (i <= rk) {
5         if (arr[i] < lp) {
6             swap(arr[i], arr[lk]);
7             lk++;
```

```
8               i++;
9           } else if (arr[i] > rp) {
10              swap(arr[i], arr[rk]);
11              rk--;
12          } else {
13              i++;
14          }
15      }
16
17      ...
18
19      dualpivot(arr, left, lk - 1);
20      dualpivot(arr, lk + 1, rk - 1);
21      dualpivot(arr, rk + 1, right);
22  }
23
24  void qsort(vector<int> &arr) {
25      if (arr.size() > 1) {
26          dualpivot(arr, 0, arr.size() - 1);
27      }
28  }
```

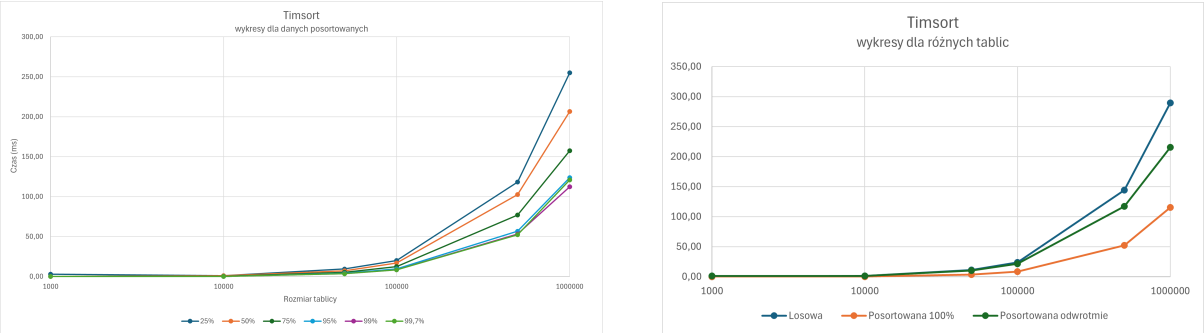
Listing 3: Dual-Pivot Quicksort - implementacja w C++



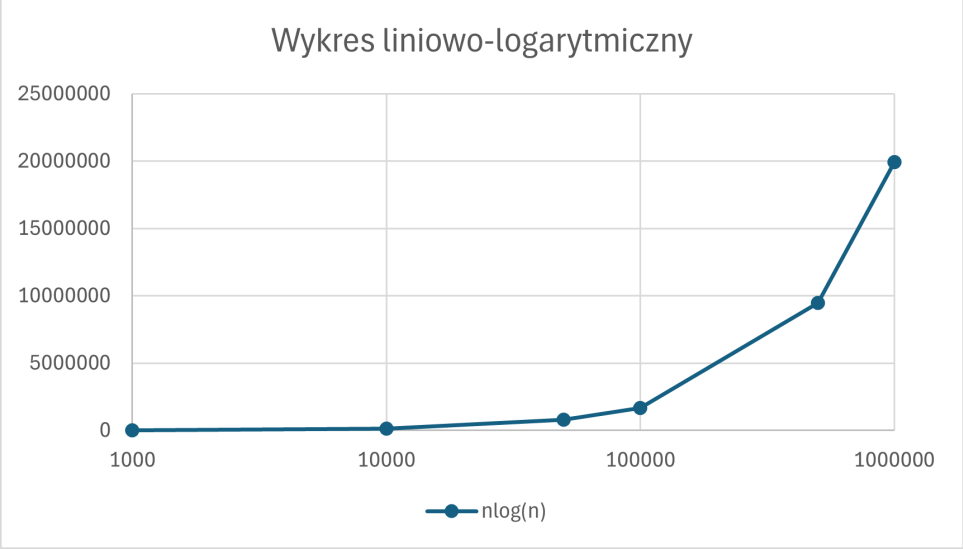
Rysunek 3: Wizualizacja funkcji *dualpivot()*

2 Analiza algorytmów

Dla każdego algorytmu przeprowadzono pomiar czasu wykonania sortowania dla tablic o rozmiarach N = 1000, 10000, 50000, 100000, 500000, 1000000. Na podstawie tych danych stworzono wykresy w celu dopasowania teoretycznych modeli złożoności obliczeniowej do wyników pomiarowych.
Wszystkie zaimplementowane algorytmy mają złożoność czasową *O(nlog(n))*, więc dopasowany model oparty jest na funkcji
$f(n) = n * log(n)$,
gdzie n - rozmiar tablicy
Wykresy:

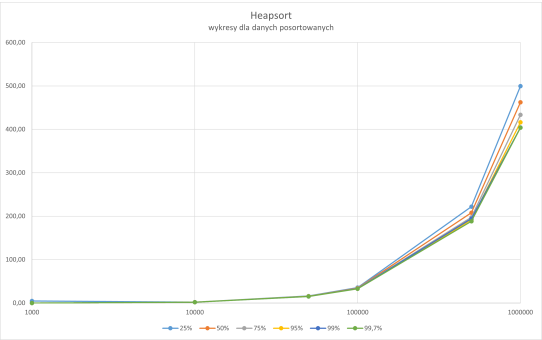


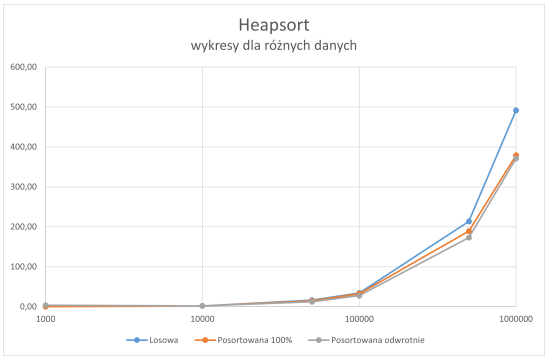
Rysunek 4: Pomiary dla Timsort



Rysunek 5: Wykres $f(n) = n * log(n)$

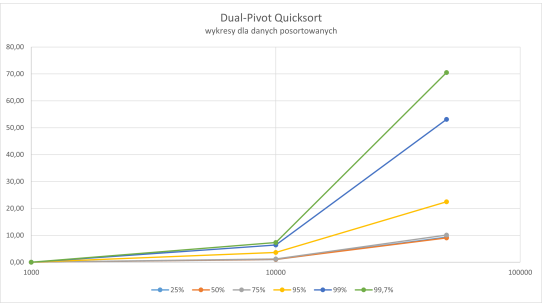
- Złożoność czasowa wygląda jak *O(nlog(n))*, co odpowiada oczekiwaniom teoretycznym.
- Wzrost czasu jest niższy przy posortowanych danych.

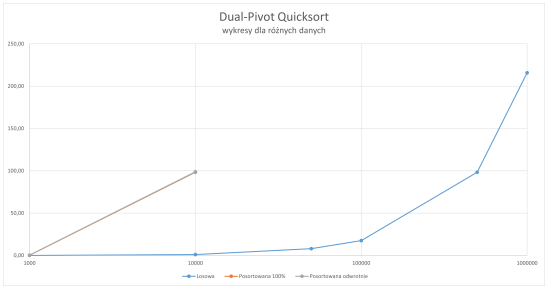


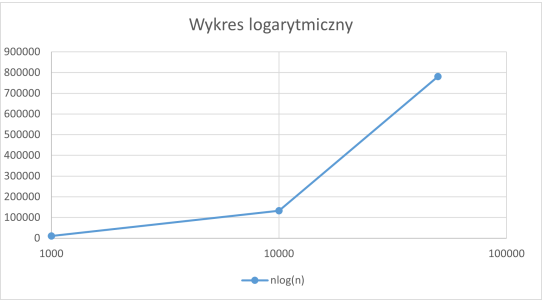


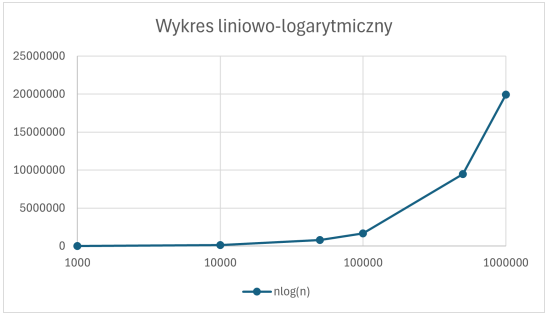
Rysunek 6: Pomiary dla Heapsort

- Złożoność czasowa wygląda jak $O(nlog(n))$ i jest stabilna, co odpowiada oczekiwaniom teoretycznym.
- Widać, że trochę wolniejszy niż Timsort
- Zachowanie jest prawie identyczne dla różnych typów danych.









Rysunek 7: Pomiary dla Dual-Pivot Quicksort

- Dla danych losowych jest bardzo szybki, złożoność czasowa wygląda jak $O(nlog(n))$
- Dla danych posortowanych, które są dla tego algorytmu niekorzystne, czas gwałtownie rośnie

3 Wnioski

Na podstawie analizy algorytmów sortowania, teoretycznej złożoności oraz przeprowadzonych badań można stwierdzić, że Timsort ma przewagę dla danych częściowo posortowanych - czas sortowania był wtedy znacznie krótszy niż przy danych losowych. Sortowanie przez kopcowanie ma stabilny czas pracy niezależnie od danych wejściowych. Dual-Pivot Quicksort osiągnął bardzo dobre wyniki dla danych losowych, jednak jego czas działania drastycznie wzrósł przy danych posortowanych, co najprawdopodobniej wynika ze słabej optymalizacji - program przepełnia stos, wielokrotnie wywołując funkcję. Mimo takiej samej złożoności teoretycznej, algorytmy różnią się znacznie pod względem czasu działania w praktyce i mogą być wrażliwe na określone typy danych.