Analiza porównawcza algorytmów sortowania

28 października 2025

1 Wprowadzenie

W niniejszym sprawozdaniu przedstawiono implementację i analizę sześciu algorytmów sortowania:

1. `INSERTION_SORT`,
2. `INSERTION_SORT_DOUBLE`,
3. `MERGE_SORT`,
4. `MERGE_SORT3`,
5. `HEAP_SORT`
6. `HEAP_SORT_TERNARY`.

Dla tablic o różnych rozmiarach zmierzono czas działania oraz zliczano liczbę porównań i przypisań. Wykresy w skali log-log potwierdzają teoretyczne rzędy złożoności - algorytmy $O(n^2)$ szybko przegrywają z wariantami $O(n \log n)$ już dla średnich n. Dla każdego algorytmu mierzono czas wykonania oraz liczby porównań i przypisań, a następnie wizualizowano wyniki w postaci tabel i wykresów.

2 Metodyka pomiarów

Pomiary wykonano w języku C++ z użyciem `std::chrono::high_resolution_clock` i zapisem wyników do pliku `wyniki.csv`. Metodyka zapewnia sprawiedliwe porównanie algorytmów:

Rozmiary i powtórzenia. Dla każdego $n \in \{100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000, 100000\}$ wykonano $R = 10$ niezależnych prób dla każdego algorytmu. Dla każdej próby generowano nową tablicę `baza` wypełnioną losowymi liczbami wygenerowanymi przez funkcję `fill_random`.

Losowanie danych. Elementy tablicy losowano z przedziału liczb całkowitych $[-10^6, 10^6]$.

Pomiar czasu. Czas mierzono za pomocą `t1 = high_resolution_clock::now()` i `t2 = ...::now()`, a różnicę przeliczano na nanosekundy przez `duration_cast<nanoseconds>(t2 - t1).count()`. Dla każdego (n, r) zapisywano czasy wszystkich sześciu algorytmów, a następnie uśredniano po R próbach.

Liczniki operacji (struktura `Stats`). W algorytmach zliczano:

- `por` — liczbę porównań wartości danych (np. `A[j] > x`, `L[i] <= R[j]`),
- `przyp` — liczbę przypisań przenoszących wartości danych (np. `A[l] = L[i]` lub zamiany).

Dla każdego algorytmu uśredniano `por` i `przyp` po R próbach.

Format wyników. Plik `wyniki.csv` zawiera kolumny: `n`, a dla każdego algorytmu (INS, INSD, M2, M3, H2, H3) trójkę: `_czas_ns`, `_por`, `_przyp`. Do uśrednienia 10 wyników danego algorytmu dla ustalonego n wykorzystano `llround`.

TLDR;

- Każde wywołanie programu ma swój własny seed losowany na podstawie czasu systemowego `std::srand(std::time(NULL));`,
- Na podstawie tego użyto `unsigned int base_seed = std::rand();` i otrzymano `base_seed`, czyli prawie już pierwszy używalny seed do generowania,
- Prawie, ponieważ `base_seed` będzie deterministycznie modyfikowany dla ustalonego $n \in$ `Ns` w każdej z R prób. Użyto — jak widać poniżej — na żaden sposób niewyróżniającego się wzoru (n, r) podczas wywołania funkcji `fill_random`.

```
      for (int r = 0; r < R; ++r) {
        int* baza = new int[n];
        fill_random(baza, n, base_seed + n*37 + r);
```

- Warto dodać, że zastosowano *funkcję lambda* `avg`, która po każdej iteracji pętli zewnętrznej iterującej po $n \in$ `Ns`, czyli po R prób dla ustalonego n przyjmowała sumy: czasów [ns], powtórzeń oraz przypisań jako `long double x` i miała bardzo proste zadanie - uśrednić wynik za pomocą `llround`:

```
      auto avg = [R](long double x)
      { return (long long) llround(x / R); };
```

Została użyta podczas *printowania* do `ofstream out("wyniki.csv", ios::trunc);` wiersza wyników dla n:

```
      out << n << ","
    << avg(sum_tINS)  << "," << avg(sum_cINS)  << "," << avg(sum_aINS)  << ","
       ...
```

3 Najciekawsze fragmenty kodu z wyjaśnieniami

INSERTION_SORT_DOUBLE — wstawianie podwójne

```
    //INSERTION SORT modyfikacja
    void INSERTION_SORT_DOUBLE(int A[], int n, Stats& stats)
    {
      if (n < 2) return;

      //sortowanie pierwszej pary
      stats.porownania++;
      if (A[0] > A[1]) {
        int x = A[0];
        A[0] = A[1];  stats.przypisania++;
        A[1] = x;     stats.przypisania++;
      }

      //kolejne pary
      for (int i = 2; i < n; i += 2) {
        //gdy nie ma pelnej pary
        if (i + 1 >= n) {
          int x = A[i];
          int j = i - 1;
          while (j >= 0) {
            stats.porownania++;
            if (A[j] > x) {
              A[j + 1] = A[j]; stats.przypisania++;
              j--;
            } else break;
```

```
26            }
27            A[j + 1] = x; stats.przypisania++;
28            break;
29          }
30
31          //nastepna para tak, by a <= b
32          int a = A[i];
33          int b = A[i + 1];
34          stats.porownania++;
35          if (a > b) {
36            int x = a;
37            a = b;
38            b = x;
39          }
40
41          //wstaw wiekszy (b) - przesuwamy o 2
42          int j = i - 1;
43          while (j >= 0) {
44            stats.porownania++;
45            if (A[j] > b) {
46              A[j + 2] = A[j]; stats.przypisania++;
47              j--;
48            } else break;
49          }
50          A[j + 2] = b; stats.przypisania++;
51
52          //wstaw mniejszy (a) - przesuwamy o 1
53          while (j >= 0) {
54            stats.porownania++;
55            if (A[j] > a) {
56              A[j + 1] = A[j]; stats.przypisania++;
57              j--;
58            } else break;
59          }
60          A[j + 1] = a; stats.przypisania++;
61        }
62      }
```

Mechanika działania. Algorytm działa podobnie do klasycznego wstawiania, ale zamiast pojedynczego elementu wstawia *parę* elementów (a, b) tak, aby $a \leq b$. Najpierw wstawiany jest element większy, przesuwany o dwa pozycje, a następnie mniejszy o jedną. Dzięki temu redukuje się liczbę przesunięć, ponieważ wewnętrzna pętla wykorzystuje częściowo przesunięte dane z poprzedniego wstawienia.

Uzasadnienie poprawności. Dla każdej iteracji z indeksem i (rozpatrywanej pary):

$$A[0 \ldots i-1] \text{ jest posortowane niemalejąco.}$$

Po wstawieniu pary (a, b), gdzie $a \leq b$, niezmiennik zostaje zachowany, ponieważ: 1. Wstawienie b nie narusza porządku w lewym fragmencie (zachodzi $A[j] \leq b$), 2. Następnie a jest wstawiane w dokładne miejsce, gdzie $A[j] \leq a \leq A[j+1]$. Indukcyjnie dla wszystkich i mamy, że po każdej iteracji fragment $A[0 \ldots i+1]$ jest posortowany.

MERGE_SORT3

```
1     //MERGE_SORT modyfikacja
2     void MERGE_SORT3(int A[], int p, int k, Stats& stats)
3     {
4       if (p >= k) return;
```

```
    int len = k - p + 1;
    int t1 = p + (len/3) - 1;
    int t2 = p + (2*len/3) - 1;

    // korekta granic
    if (t1 < p) t1 = p;
    if (t2 < t1 + 1) t2 = t1 + 1;
    if (t2 > k - 1) t2 = k - 1;

    MERGE_SORT3(A, p,    t1, stats);
    MERGE_SORT3(A, t1+1, t2, stats);
    MERGE_SORT3(A, t2+1, k,  stats);

    MERGE(A, p,   t1, t2, stats);
    MERGE(A, p,   t2, k,  stats);
  }
```

Mechanika działania. Zamiast dzielić tablicę na dwie połowy, algorytm dzieli ją na trzy części o zbliżonej długości:

$$[p,\ t_1],\ [t_1{+}1,\ t_2],\ [t_2{+}1,\ k].$$

Każdy fragment jest sortowany rekurencyjnie, a następnie wyniki są łączone dwuetapowo (pierwsze scalanie łączy 1. i 2. część, drugie — wynik z 3. częścią).

Uzasadnienie poprawności.

- **Funkcja `MERGE`.**
  Funkcja `MERGE` scala dwa posortowane fragmenty tablicy w jeden większy, zachowując porządek niemalejący. Porównuje elementy z lewej i prawej części, zawsze wybierając mniejszy i dopisując go do wyniku. Dzięki temu po zakończeniu działania cały fragment $A[p..k]$ jest posortowany.
- **Algorytm `MERGE_SORT3`.**
  Algorytm dzieli tablicę na trzy części, z których każda jest następnie sortowana rekurencyjnie. Gdy wszystkie trzy fragmenty są już posortowane, następują dwa kolejne scalenia: najpierw łączenie pierwszej i drugiej części, a następnie połączenie powstałego wyniku z trzecią częścią. Ponieważ każda operacja `MERGE` zachowuje porządek, po dwóch scaleniach otrzymujemy posortowany fragment $A[p..k]$. Po zakończeniu wszystkich wywołań rekurencji cała tablica A jest uporządkowana niemalejąco.

HEAP_SORT_TERNARY

```
  //HEAP SORT modyfikacja
  void heapify_ternary(int A[], int n, int i, Stats& stats)
  {
    int c1 = 3*i + 1;
    int c2 = 3*i + 2;
    int c3 = 3*i + 3;
    int largest = i;

    if (c1 < n) { stats.porownania++; if (A[c1] > A[largest]) largest = c1; }
    if (c2 < n) { stats.porownania++; if (A[c2] > A[largest]) largest = c2; }
    if (c3 < n) { stats.porownania++; if (A[c3] > A[largest]) largest = c3; }

    if (largest != i) {
      swap_count(A[i], A[largest], stats);
      heapify_ternary(A, n, largest, stats);
    }
  }
```

Mechanika działania. Każdy węzeł ma do trzech dzieci $(3i+1, 3i+2, 3i+3)$. Funkcja `heapify_ternary` utrzymuje własność kopca: element w indeksie i jest nie mniejszy od swoich dzieci. W fazie *build–heap* kopiec jest budowany od dołu, a następnie sortowanie polega na wielokrotnym przenoszeniu elementu maksymalnego na koniec tablicy.

Uzasadnienie poprawności.

- **Funkcja `heapify_ternary` (lokalnie w poddrzewie).**
  Funkcja `heapify_ternary(i)` dba o to, aby element w węźle i był większy (lub równy) od swoich dzieci. Jeśli któreś z dzieci jest większe, elementy zostają zamienione miejscami, a procedura wywoływana jest rekurencyjnie dla tego dziecka. Po zakończeniu działania całe poddrzewo o korzeniu i spełnia własność kopca maksymalnego – rodzic nie jest mniejszy od żadnego ze swoich potomków.

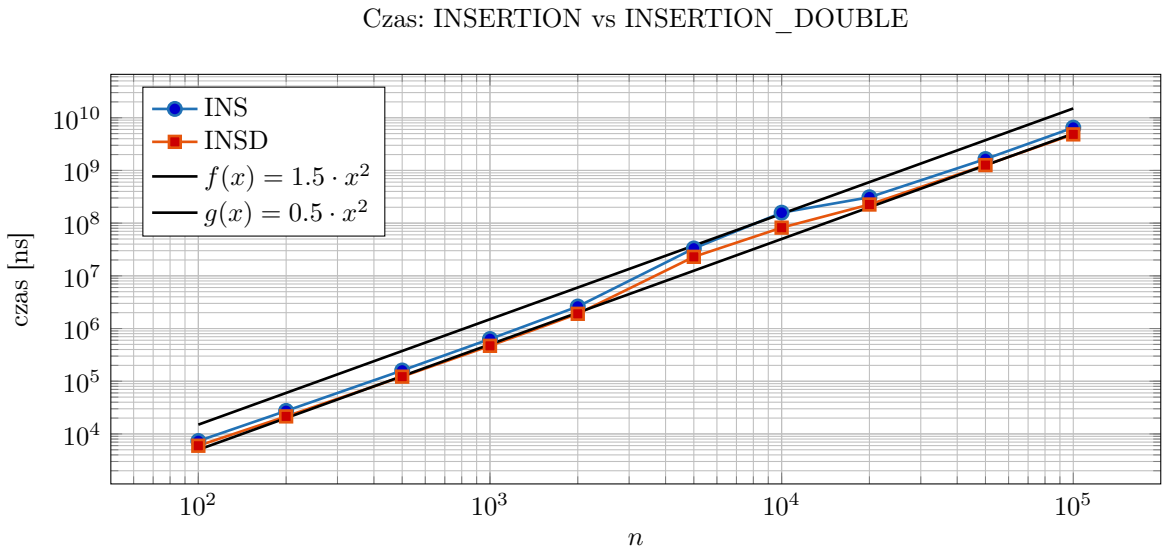
- **Algorytm `HEAP_SORT_TERNARY` (całość).**
  W pierwszej fazie budowany jest kopiec, dzięki czemu największy element znajduje się na początku tablicy. Następnie w każdej iteracji największy element (z korzenia) zamieniany jest z ostatnim nieposortowanym, po czym wywoływana jest funkcja `heapify_ternary`, aby przywrócić strukturę kopca w pozostałej części tablicy. W ten sposób z każdą iteracją rośnie posortowany fragment na końcu tablicy, a reszta pozostaje kopcem. Po zakończeniu wszystkich kroków cała tablica jest posortowana w porządku rosnącym.

4 Wyniki pomiarów

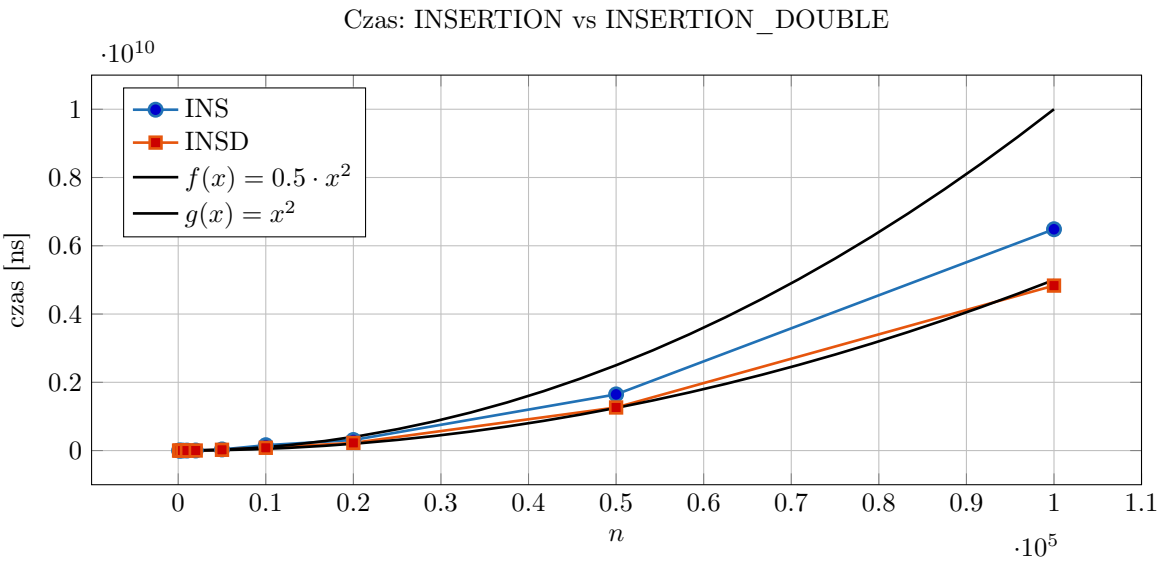
4.1 Tabela (czas w ns)

n	INS	INSD	M2	M3	H2	H3
100	7340	5970	20690	20860	10520	9560
200	27370	21510	42460	45470	24180	21430
500	160030	122150	108580	130900	71770	61640
1000	632010	469960	219440	239390	159870	138470
2000	2614220	1902930	483530	498590	357680	308780
5000	33089180	22949150	2480610	2603560	1519290	1330030
10000	157938310	82056700	3701390	4263850	3048180	2558480
20000	310030700	225582910	6299860	6109440	5469100	4628730
50000	1646121180	1262847360	14091930	14960820	13297230	11482840
100000	6484806260	4833272160	27762510	31965820	27549340	23682240

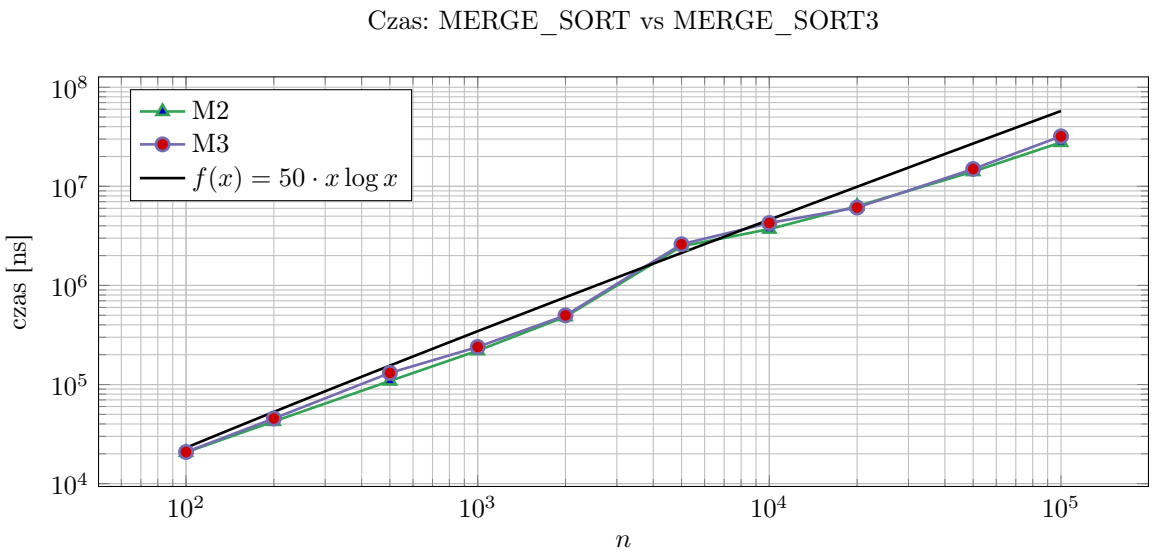
4.2 Wykresy czasu



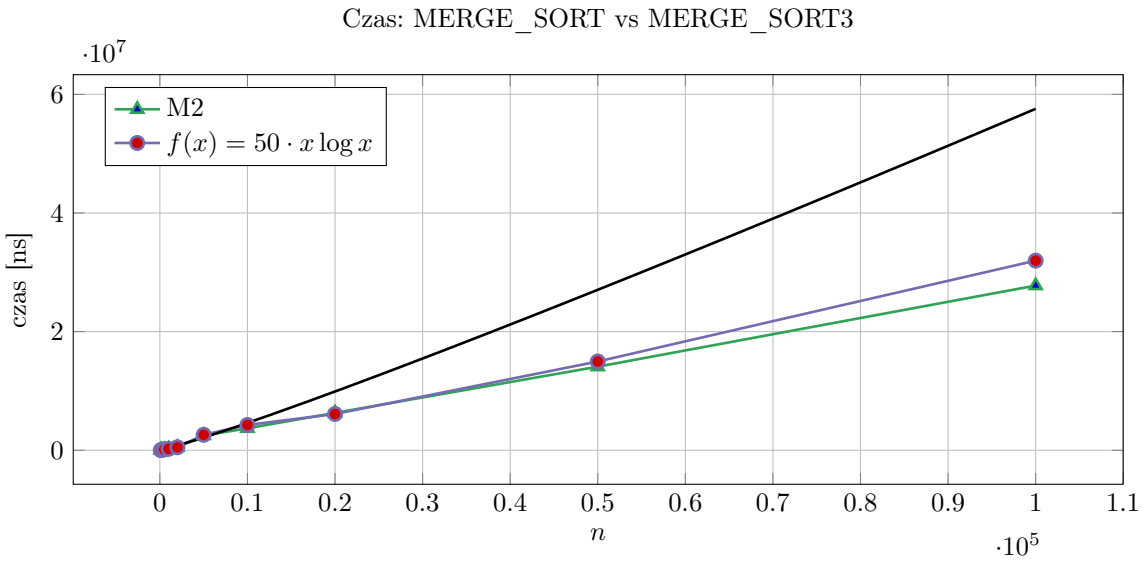
Rysunek 1: Porównanie czasu: INSERTION i INSERTION_DOUBLE w *log–log* z odniesieniami



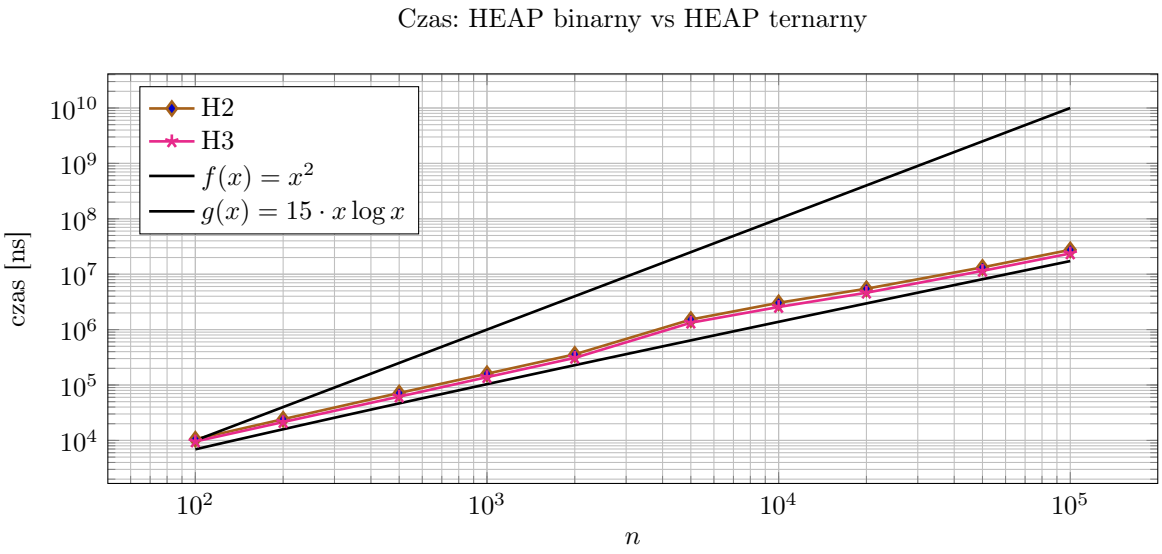
Rysunek 2: Porównanie czasu: INSERTION i INSERTION_DOUBLE w *lin–lin* z odniesieniami



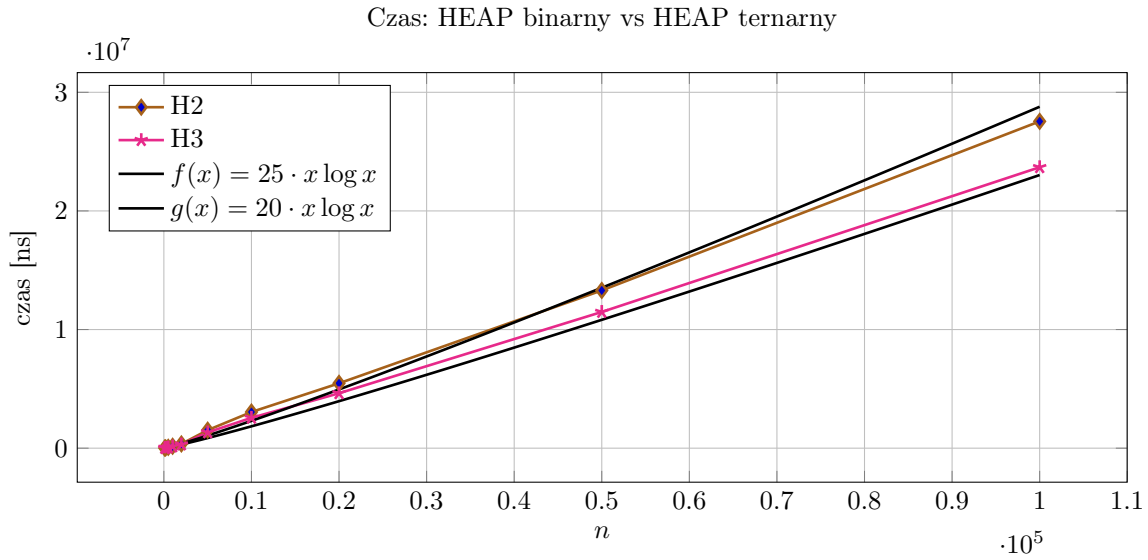
Rysunek 3: Porównanie czasu: MERGE_SORT vs MERGE_SORT3 w *log–log* z odniesieniami.



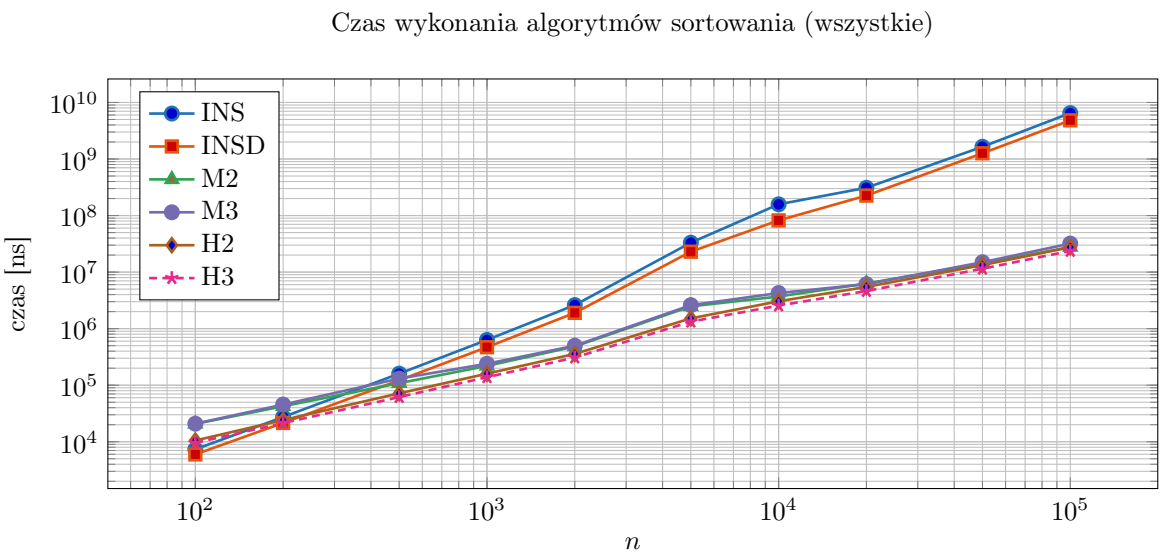
Rysunek 4: Porównanie czasu: MERGE_SORT vs MERGE_SORT3 w *lin–lin* z odniesieniami.



Rysunek 5: Porównanie czasu: HEAP_SORT binarny vs ternarny *log–log* z odniesieniami.

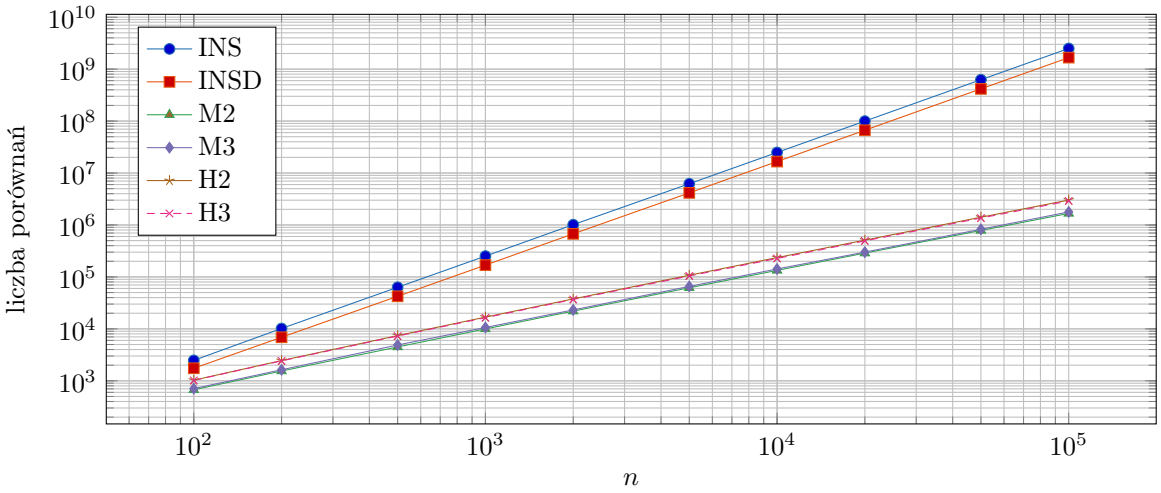


Rysunek 6: Porównanie czasu: HEAP_SORT binarny vs ternarny w *lin–lin* z odniesieniami

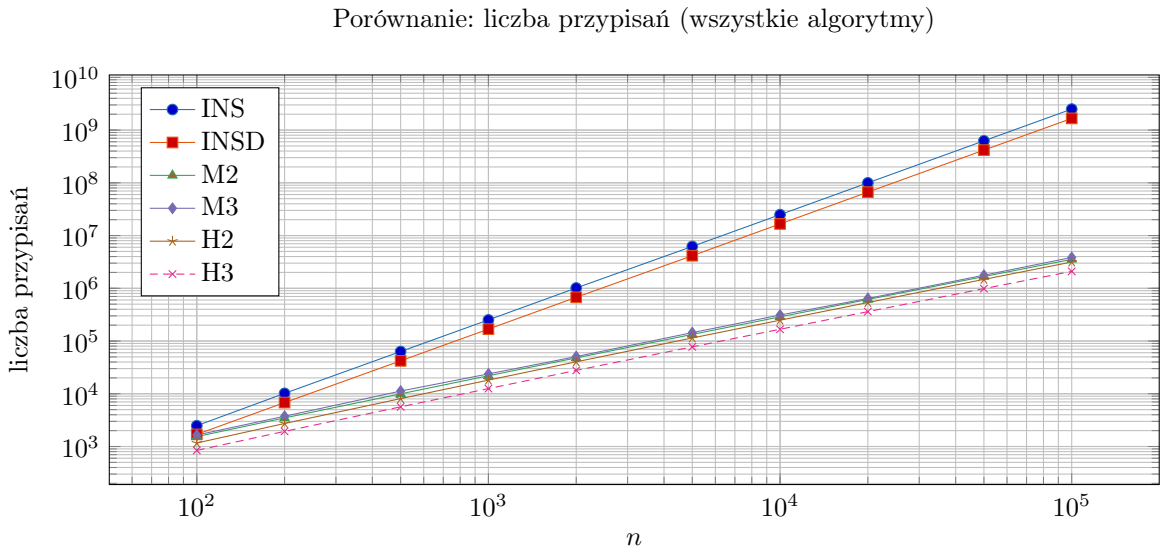


Rysunek 7: Porównanie czasu działania dla wszystkich algorytmów *log–log*.

4.3 Wykresy liczby porównań i przypisań



Rysunek 8: Liczba porównań w funkcji rozmiaru danych.



Rysunek 9: Liczba przypisań w funkcji rozmiaru danych.

INS/INSD: Algorytmy wstawiania wykazują wzrost kwadratowy; `INSERTION_SORT_DOUBLE` ogranicza liczbę ruchów dzięki wstawianiu par, jednak dla rosnącego n i tak przegrywa z metodami $O(n \log n)$.

MERGE_SORT vs MERGE_SORT3: Dzielenie na 3 części zmniejsza głębokość rekurencji i może skutkować lepszym wykorzystaniem pamięci podręcznej; w wynikach widać nieznaczne korzyści czasowe dla większych n.

HEAP: Kopiec potrójny redukuje wysokość drzewa względem binarnego, ale każde `heapify` porównuje do trzech dzieci.

5 Wnioski

- Algorytmy $O(n^2)$ (`INS`, `INSD`) stają się nieefektywne już dla $n \gtrsim 2000$.
- `MERGE_SORT` oraz `HEAP_SORT` są złożoności $O(n \log n)$, więc wypadają lepiej od `INS`, `INSD`.
- `MERGE_SORT3` wypada minimalnie lepiej od zwykłego MERGE'a dla większych n, co przypisujemy mniejszej głębokości rekurencji.
- `HEAP_SORT_TERNARY` stosuje płytsze drzewo, dzięki czemu czasy są minimalnie lepsze.
- Dla praktycznych, dużych danych najlepsze są te o złożoności $O(n \log n)$.