Algorytmy i złożoność obliczeniowa – Projekt 1

Badanie algorytmów z omówieniem ich złożoności obliczeniowej

Ilya Shnitko

Spis treści

1. Wprowadzenie

1.1. Sortowanie przez wstawianie (zwykłe)

Sortowanie przez wstawianie (ang. *Insertion Sort*) jest prostym algorytmem sortującym, który działa na zasadzie iteracyjnego budowania posortowanego podciągu w miejscu. Opiera się na koncepcji wstawiania kolejnych elementów w odpowiednie pozycje w już uporządkowanej części tablicy.

Procedure INSERTION-SORT(A, n)

Inputs and Result: Same as SELECTION-SORT.

1. For $i = 2$ to n:
   A. Set *key* to $A[i]$, and set j to $i - 1$.
   B. While $j > 0$ and $A[j] > key$, do the following:
      i. Set $A[j + 1]$ to $A[j]$.
      ii. Decrement j (i.e., set j to $j - 1$).
   C. Set $A[j + 1]$ to *key*.

Algorytm 1 - Pseudokod algorytmu Insertion-Sort. Źródło: Thomas H. Cormen - Algorithms Unlocked (2013)

Dane wejściowe i wyniki dla *Insertion-Sort*:

Inputs:

- A: Tablica elementów.

- n: Liczba elementów w tablicy.

Result: Posortowana niemalejąco tablica A.

Algorytm 2 - Opis danych wyjściowych i wejściowych dla algorytmu Insertion-Sort. Źródło: Thomas H. Cormen - Algorithms Unlocked (2013)

1.1.1. Zasada działania

- Początek: Algorytm rozpoczyna się od drugiego elementu tablicy (indeks **j = 2**), traktując pierwszy element (**A[1]**) jako początkowy podciąg posortowany.
- Iteracja: Dla każdego kolejnego elementu **A[j]** (od **j = 2** do **j = n**):
  - Element **A[j]** jest porównywany z elementami w posortowanym podciągu **A[1...j-1]**.
  - Elementy większe od **A[j]** są przesuwane w prawo, aby zrobić miejsce dla **A[j].**
  - Element **A[j]** jest wstawiany w odpowiednią pozycję, tak aby podciąg **A[1...j]** pozostał posortowany.
- Zakończenie: Po przetworzeniu ostatniego elementu (**j = n**) cała tablica jest posortowana.

1.1.2. Niezmiennik pętli

Na początku każdej iteracji (dla ustalonego **j**) podciąg **A[1...j-1]** zawiera pierwotne elementy **A[1...j-1]** w **stanie posortowanym**. Dzięki temu po każdej iteracji posortowany podciąg rośnie o jeden element.

1.1.3. Złożoność obliczeniowa

- **Przypadek średni**: $O(n^2)$ – występuje, gdy elementy są częściowo uporządkowane.
- **Przypadek najgorszy**: $O(n^2)$ – występuje, gdy tablica jest początkowo posortowana odwrotnie (każdy element musi być przesunięty na początek).
- **Przypadek najlepszy**: $O(n)$ – gdy tablica jest już posortowana (każdy element trafia od razu na właściwą pozycję).
- Złożoność pamięciowa: $O(1)$ (sortowanie w miejscu).

1.2. Sortowanie przez wstawianie (binarne)

Sortowanie przez wstawianie binarne (ang. *Binary Insertion Sort*) jest modyfikacją klasycznego algorytmu sortowania przez wstawianie, w której binarne wyszukiwanie zastępuje sekwencyjne porównania w celu znalezienia pozycji wstawienia elementu. W porównaniu z klasycznym *InsertionSort* liczba porównań jest zmniejszona z $O(n^2)$ do $O(nlogn)$, chociaż to nie poprawia asymptotyczną złożoność czasową ze względu na dominującą rolę przesunięć.

Procedure BINARY-INSERTION-SORT(A, n)

Inputs:

- A: Tablica elementów.

- n: Liczba elementów w tablicy.

Result: Posortowana niemalejąco tablica A.

1. For i = 1 to n - 1:

    A. Set key to A[i].

    B. Set pos to BINARY-SEARCH(A, key, 0, i - 1).

    C. For j = i - 1 downto pos:

        i. Set A[j + 1] to A[j].

    D. Set A[pos] to key.

Algorytm 3 - Pseudokod algorytmu BINARY-INSERTION-SORT. Źródło: opracowanie własne.

1.2.1. Zasada działania

Algorytm działa w następujących krokach (zgodnie z pseudokodem):

1. Iteracja po tablicy: Dla każdego elementu A[i] (od i = 1 do n-1):
    - Wybór klucza: Przechowywanie wartości A[i] w zmiennej key.
    - Wyszukiwanie binarne: Użycie funkcji BINARY-SEARCH, aby znaleźć indeks pos, gdzie key powinien zostać wstawiony w posortowanym podciągu A[0..i-1].
    - Przesunięcie elementów: Przesunięcie wszystkich elementów od A[pos] do A[i-1] o jedną pozycję w prawo (użycie pętli j = i-1 downto pos).
    - Wstawienie klucza: Umieszczenie key na pozycji A[pos].

Funkcja BINARY-SEARCH znajduje najmniejszy indeks pos, dla którego A[pos] ≥ key, przeszukując podtablicę A[low..high] metodą "dziel i zwyciężaj".

1.2.2. Złożoność obliczeniowa

- **Średni przypadek**: $O(n^2)$ - przesunięcia dominują nad operacjami porównania, nawet przy redukcji liczby porównań do O(nlogn).
- **Najgorszy przypadek**: $O(n^2)$ - mimo że liczba porównań spada z $O(n^2)$ (w klasycznej wersji) do O(nlogn) dzięki binarnemu wyszukiwaniu, przesunięcia elementów nadal wymagają $O(n^2)$ operacji w najgorszym przypadku (np. dla danych odwrotnie posortowanych).
- **Przypadek najlepszy**: $O(n)$ – gdy tablica jest już posortowana (każdy element trafia od razu na właściwą pozycję).
- **Złożoność pamięciowa**: $O(1)$ (sortowanie w miejscu).

Function BINARY-SEARCH(A, key, low, high)

Inputs:

- A: Posortowany podciąg.

- key: Szukana wartość.

- low, high: Zakres wyszukiwania.

Result: Indeks pozycji dla key.

1. While low ≤ high:
    A. Set mid to low + (high - low) / 2.
    B. If A[mid] < key:
        i. Set low to mid + 1.
    C. Else:
        i. Set high to mid - 1.
2. Return low.

Algorytm 4 - Pseudokod dla funkcji BINARY-SEARCH. Źródło: opracowanie własne.

1.3. Sortowanie przez kopcowanie

Zanim możliwe będzie zastosowanie sortowania przez kopcowanie, konieczne jest odpowiednie przekształcenie tablicy wejściowej w kopiec maksymalny. Proces ten, znany jako **budowanie kopca maksymalnego** (ang. *BUILD-MAX-HEAP*), polega na uporządkowaniu elementów w taki sposób, aby każdy rodzic posiadał wartość większą lub równą wartościom swoich dzieci.

Kopiec binarny (ang. *binary heap*) to struktura danych w formie tablicy, która odpowiada drzewu binarnemu o specjalnej organizacji. Struktura ta spełnia warunek tzw. prawie pełnego drzewa binarnego, czyli wszystkie poziomy są całkowicie zapełnione, poza ewentualnie ostatnim, który uzupełniany jest kolejno od lewej strony.

Algorytm działa od środka tablicy do początku, ponieważ węzły z końca nie mają dzieci, więc są już kopcami jednoelementowymi. W każdej iteracji wywoływana jest procedura porządkująca lokalnie strukturę drzewa binarnego, zaczynając od danego węzła w dół, co umożliwia nadbudowanie pełnej struktury kopca.

Procedure HEAPSORT(A, n)

Inputs:

- A: Tablica elementów.

- n: Liczba elementów w tablicy.

Result: Posortowana tablica A (rosnąco).

1. Build a max-heap from array A:

   A. For i = ⌊n/2⌋ - 1 downto 0:

      i. Call HEAPIFY(A, n, i).

2. Extract elements from the heap:

   A. For i = n - 1 downto 1:

      i. Swap A[0] with A[i].

      ii. Call HEAPIFY(A, i, 0).

Algorytm 5 - Pseudokod dla funkcji HEAPSORT. Źródło: opracowanie własne i Thomas H. Cormen - Algorithms Unlocked (2013)

 # Procedure HEAPIFY(A, n, i)

Inputs:

- A: Tablica reprezentująca kopiec.

- n: Rozmiar kopca.

- i: Indeks węzła, dla którego przywracana jest własność kopca.

Result: Kopiec z przywróconą własnością dla węzła i.

1. Set largest = i.

2. Set left = 2*i + 1.

3. Set right = 2*i + 2.

4. If left < n and A[left] > A[largest]:

   A. Set largest = left.

5. If right < n and A[right] > A[largest]:

   A. Set largest = right.

6. If largest ≠ i:

   A. Swap A[i] with A[largest].

   B. Call HEAPIFY(A, n, largest).

Algorytm 6 - Pseudokod dla funkcji HEAPIFY. Źródło: opracowanie własne i Thomas H. Cormen - Algorithms Unlocked (2013)

1.3.1. Zasada działania

Budowa kopca (max-heap):

- Przekształcenie tablicy w strukturę kopca, gdzie wartość każdego rodzica jest większa lub równa wartościom jego dzieci.
- Proces rozpoczyna się od ostatniego nie-liścia (**⌊n/2⌋ - 1**) i iteracyjnie naprawia poddrzewa za pomocą **HEAPIFY**.

Ekstrakcja elementów:

- Największy element (korzeń) jest zamieniany z ostatnim elementem tablicy.
- Rozmiar kopca jest zmniejszany o 1, a **HEAPIFY** przywraca własność kopca dla nowego korzenia.
- Powtarzane aż do posortowania całej tablicy.

1.3.2. Złożoność obliczeniowa

- **Budowa kopca**: *O(n)* – pomimo pozornie , suma operacji dla wszystkich węzłów jest liniowa.
- **Ekstrakcja elementów**: *O(nlogn)* – każda z **n−1** iteracji wymaga *O(logn)* czasu.
- **Całkowita złożoność**: O(nlogn) dla wszystkich przypadków (najlepszy, średni, najgorszy).

1.4. Szybkie sortowanie

Sortowanie szybkie (ang. *QuickSort*) jest algorytmem opartym na technice dziel i zwyciężaj (ang. *divide and conquer*). Proces sortowania rozpoczyna się od wyboru elementu odniesienia, zwanego pivotem[1], który może zostać wybrany dowolnie (np. jako ostatni, środkowy, losowy element). Następnie tablica jest dzielona na dwie części względem pivota:

- Lewa część: elementy mniejsze lub równe pivotowi,
- Prawa część: elementy większe od pivota.

Każda z tych części jest rekurencyjnie sortowana według tego samego schematu. Proces kończy się, gdy podzbiory stają się jednoelementowe lub puste, co oznacza pełne uporządkowanie tablicy.

Procedure QUICKSORT(A, p, r)

Inputs and Result: Same as MERGE-SORT.

1. If $p \geq r$, then just return without doing anything.
2. Otherwise, do the following:
   A. Call PARTITION(A, p, r), and set q to its result.
   B. Recursively call QUICKSORT$(A, p, q-1)$.
   C. Recursively call QUICKSORT$(A, q+1, r)$.

Algorytm 7 - Pseudokod dla funkcji Quicksort. Źródło: Thomas H. Cormen - Algorithms Unlocked (2013)

1.4.1. Zasada działania

1. Wybór pivota: W implementacji z procedurą PARTITION (Algorytm 8) pivotem jest ostatni element podtablicy (A[r]).
2. Podział tablicy (PARTITION):

[1] Pivot - element osiowy. Dalej te definicje będą używane zamiennie.

- Inicjalizacja indeksu q = p.
- Dla każdego elementu A[u] w zakresie [p, r-1]:
  - Jeśli A[u] ≤ A[r], zamień miejscami A[q] z A[u] i zwiększ q o 1.
- Zamień A[q] z A[r], umieszczając pivota na pozycji q.
- Zwróć q jako granicę podziału.

3. Rekurencja:
   - Wywołaj QuickSort dla lewej części (A[p...q-1]).
   - Wywołaj QuickSort dla prawej części (A[q+1...r]).

Procedure PARTITION(A, p, r)

Inputs: Same as MERGE-SORT.

Result: Rearranges the elements of $A[p\,..\,r]$ so that every element in $A[p\,..\,q-1]$ is less than or equal to $A[q]$ and every element in $A[q+1\,..\,r]$ is greater than q. Returns the index q to the caller.

1. Set q to p.
2. For $u = p$ to $r-1$ do:
   A. If $A[u] \le A[r]$, then swap $A[q]$ with $A[u]$ and then increment q.
3. Swap $A[q]$ with $A[r]$ and then return q.

Algorytm 8- Pseudokod dla funkcji Partition. Źródło: opracowanie własne i Thomas H. Cormen - Algorithms Unlocked (2013)

1.4.2. Złożoność obliczeniowa

- **Średni przypadek**:*O(nlogn)* - efektywny podział tablicy na dwie części o zbliżonych rozmiarach, co redukuje liczbę rekurencyjnych wywołań. Dominują operacje porównania i zamiany, które w średnim przypadku wykonywane są logarytmiczną liczbę razy względem **n**.
- **Najgorszy przypadek**: *O(n²)* - nierównomierny podział tablicy, np. gdy pivot jest skrajnym elementem, a dane są już posortowane lub odwrotnie posortowane. W takim przypadku rekurencja degeneruje się do **n-1** poziomów, a liczba operacji porównania i zamiany rośnie kwadratowo.
- **Przypadek najlepszy**: *O(nlogn)* - idealnie zrównoważony podział tablicy na dwie równe części przy każdym wywołaniu rekurencyjnym (np. gdy pivot jest medianą).
- **Złożoność pamięciowa**: *O(nlogn)* - zużycie pamięci zależy od głębokości rekurencyjnego stosu wywołań. W średnim przypadku głębokość stosu jest logarytmiczna względem **n**.

2. Plan eksperymentu

2.1. Cel eksperymentu

Analiza złożoności czasowej algorytmów opisanych we wcześniejszej części wprowadzenia. Eksperyment obejmował pomiary średniego czasu sortowania dla różnych typów danych i rozmiarów tablic. Do tworzenia danych wykorzystano generatory liczb pseudolosowych, które zapewniają różnorodność i nieprzewidywalność wyników. Dla każdej próby losowania generowany był nowy klucz inicjujący, co eliminuje powtarzalność między eksperymentami. Zakresy wartości dobrano tak, aby uniknąć przepełnień i zapewnić reprezentatywność dla danego typu (np. znaki ograniczono do małych liter).

2.2. Metodologia badań

2.2.1. Próbki i rozmiary tablic:

- Dla każdego algorytmu wykonano **100 prób** pomiarowych.
- Badano tablice o rozmiarach:
  10^4, $2 \cdot 10^4$, $4 \cdot 10^4$, $6 \cdot 10^4$, $8 \cdot 10^4$, $16 \cdot 10^4$ elementów.

2.2.2. Typy danych:

- **int** (liczby całkowite),
- **char** (znaki),
- **float** (liczby zmiennoprzecinkowe).

2.2.3. Rodzaje tablic wejściowych:

- **Losowe** (elementy w przypadkowej kolejności) - *random*
- **Uporządkowane rosnąco** - *sorted*
- **Uporządkowane malejąco** – *rev sorted*
- **Częściowo uporządkowane**:
- **33% początkowe elementy** posortowane niemalejąco, reszta losowa – *part sort 33*
- **66% początkowe elementy** posortowane niemalejąco, reszta losowa – *part sort 66*

2.3. Proces przeprowadzenia eksperymentu

Dla każdej kombinacji (rozmiar × typ danych × rodzaj tablicy) zmierzono **średni czas sortowania** na podstawie 100 prób. Każda próba wykorzystywała nowy zestaw danych, co eliminuje ryzyko powtarzalności wyników. Wyniki zostały zestawione w tabelach i na wykresach, aby porównać wydajność algorytmów w różnych scenariuszach.

2.4. Kluczowe założenia

Wszystkie testy przeprowadzono na tym samym środowisku sprzętowym i programistycznym. Uwzględniono różne rozkłady danych, aby zbadać zachowanie algorytmów w ekstremalnych i typowych przypadkach. Dla algorytmu *QuickSort* pivot wybierany jako element środkowy.

3. Omówienie przebiegu eksperymentów i przedstawienie uzyskanych wyników

Eksperymenty przeprowadzone w ramach badania miały na celu zweryfikowanie teoretycznych założeń dotyczących złożoności czasowej algorytmów sortowania oraz analizę wpływu różnych czynników na ich wydajność. Wyniki potwierdzają ogólne prawidłowości, ale ujawniły również szereg niuansów, które wymagają pogłębionej interpretacji. Poniżej przedstawiono szczegółowe omówienie, łączące obserwacje z próbą wyjaśnienia ich przyczyn.

3.1. Wpływ rozmiaru danych

Zgodnie z oczekiwaniami, rozmiar danych okazał się kluczowym czynnikiem determinującym czas sortowania. Dla algorytmów o złożoności O(n log n), takich jak *Quick Sort* i *Heap Sort,* wzrost czasu był zbliżony do teoretycznych przewidywań, ale nie identyczny. Na przykład dla *Quick Sort* i danych losowych (random int) czas wykonania zwiększył się z **0,526** m**s** dla 10^4 elementów do **21,888** ms dla $16 \cdot 10^4$, co oznacza wzrost około **41,6-krotny**. Teoretyczny wzrost, oparty na zależności n log n, powinien wynosić około 19,2× dla 16-krotnego zwiększenia rozmiaru. Różnica ta może wynikać z narzutu związanego z implementacją algorytmu — np. kosztem rekurencji, który staje się znaczący przy bardzo dużych tablicach, lub fragmentacją pamięci, utrudniającą efektywne wykorzystanie cache'u procesora.

Dla algorytmów kwadratowych, takich jak *Insertion Sort*, wzrost czasu był dramatyczny. Dla random int czas *Insertion Sort (Binary)* zwiększył się z **13,999** ms do **3526,59** ms (wzrost ~252×), co jest bliskie teoretycznemu n^2 ($16^2 = 256$×). Ta zależność potwierdza, że nawet niewielkie zwiększenie rozmiaru danych może uczynić te algorytmy całkowicie niepraktycznymi w zastosowaniach wymagających obsługi dużych zbiorów. Warto jednak zauważyć, że dla małych rozmiarów (np. 10^4 elementów) Insertion Sort pozostaje konkurencyjny, co wynika z niskich stałych ukrytych w notacji O.

3.2. Wpływ typu danych

Różnice w czasie sortowania między typami danych (*int, float, char*) okazały się znaczące i wymagają uwzględnienia w praktycznych implementacjach. Dla liczb całkowitych (*int*) czas sortowania był najkrótszy, co można wytłumaczyć prostotą operacji porównania — w przypadku liczb całkowitych jest to pojedyncza instrukcja procesora, która może być zoptymalizowana sprzętowo. Dla danych typu float czas był nieco dłuższy, co wynika ze złożoności porównań liczb zmiennoprzecinkowych, wymagających analizy zarówno wykładnika, jak i mantysy. Dodatkowo, liczby zmiennoprzecinkowe mogą generować tzw. liczby denormalne[2], które spowalniają obliczenia.

Najwolniejsze przetwarzanie dotyczyło typu *char*. Dla *Quick Sort* czas sortowania $16 \cdot 10^4$ elementów wyniósł **371,158** ms — ponad 17× więcej niż dla int. Przyczyną jest konieczność porównywania kodów ASCII znaków, które często wymagają iteracji po całych łańcuchach (np. przy sortowaniu leksykograficznym). Ponadto, operacje na znakach mogą generować dodatkowe narzuty związane z konwersją typów lub zarządzaniem pamięcią.

Warto zauważyć, że czas sortowania algorytmu QuickSort dla typu float okazał się niższy niż dla int przy rozmiarze $16 \cdot 10^4$ (**12,067** ms vs. **21,888** ms). Możliwym wyjaśnieniem jest wykorzystanie przez kompilator lub procesor instrukcji wektorowych, które optymalizują operacje na liczbach zmiennoprzecinkowych, grupując je w pakiety.

3.3. Anomalie i ich interpretacja

3.3.1. Skok czasu Quick Sort dla random int przy $8 \cdot 10^4$:

Nagły wzrost czasu z **9,744** ms do **21,888** ms dla $16 \cdot 10^4$ elementów może wynikać z kumulacji nieoptymalnych podziałów. Przy dużych rozmiarach danych nawet pojedyncze błędne wybory pivota (np. skrajne wartości) prowadzą do głębokiej rekurencji i zwiększenia narzutu.

---

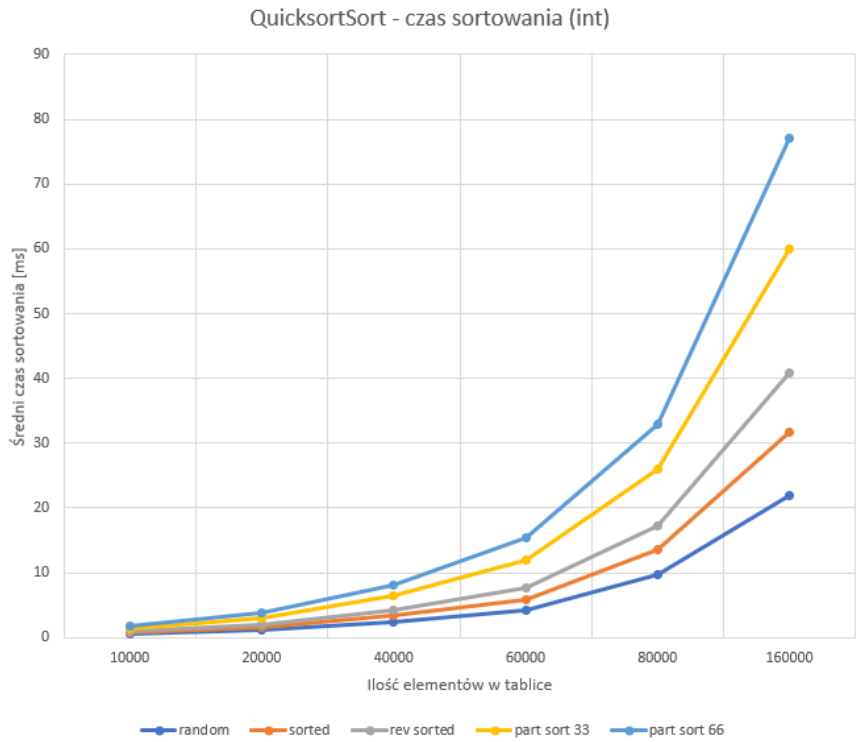
[2] Liczby denormalne (ang. denormal numbers) to liczby zmiennoprzecinkowe o bardzo małej wartości bezwzględnej, mniejsze od najmniejszej znormalizowanej liczby reprezentowanej w danym formacie (np. IEEE 754). Ich reprezentacja wymaga rezygnacji z domyślnej normalizacji mantysy, co prowadzi do utraty precyzji i znacząco spowalnia obliczenia ze względu na konieczność obsługi wyjątków sprzętowych. Procesory często traktują je jako przypadki brzegowe, generujące dodatkowe opóźnienia.

3.3.2. Nietypowy spadek czasu Heap Sort dla part sort 66 float:

Czas sortowania $8 \cdot 10^4$ elementów spadł z **7,747** ms (*part sort 33*) do **6,718** ms (*part sort 66*). Być może częściowa sortacja poprawiła lokalność pamięciową danych, redukując liczbę cache misses podczas przebudowy kopca.

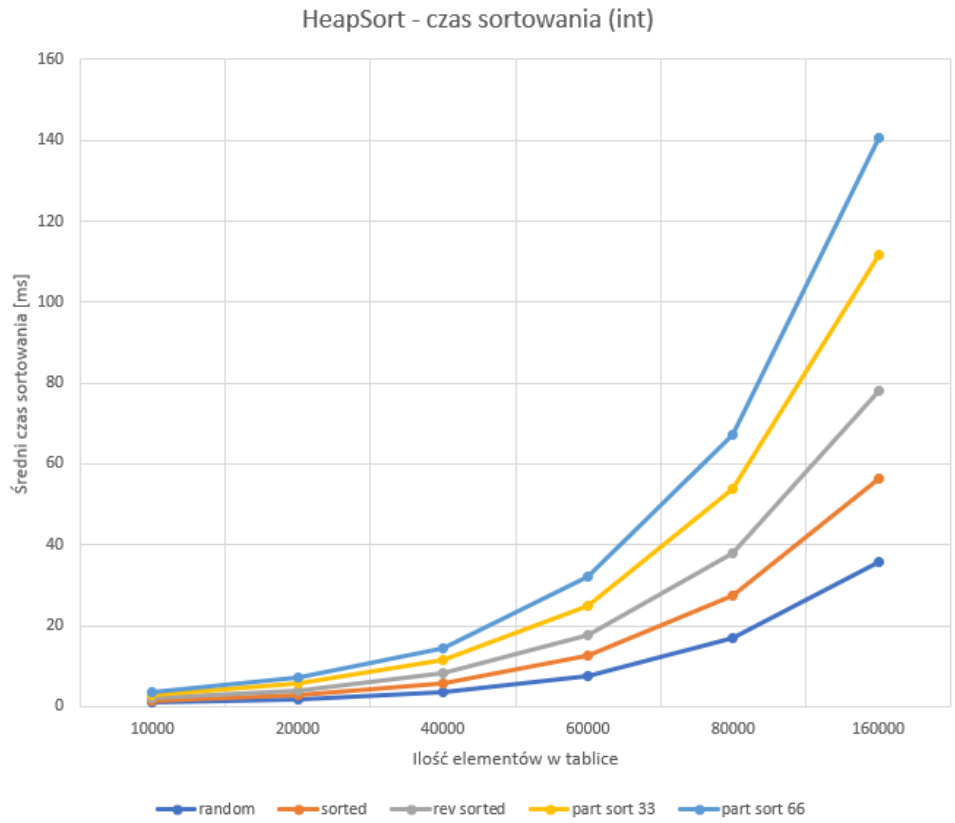
3.3.3. Różnice w czasie dla float vs. int:

Niższy czas *Quick Sort* dla typu *float* w porównaniu do *int* może wynikać z optymalizacji kompilatora specyficznych dla operacji zmiennoprzecinkowych (np. wykorzystanie rejestrów SSE[3]) lub lepszej lokalizacji pamięciowej danych. Wymaga to jednak dalszej weryfikacji, np. poprzez analizę kodu maszynowego.

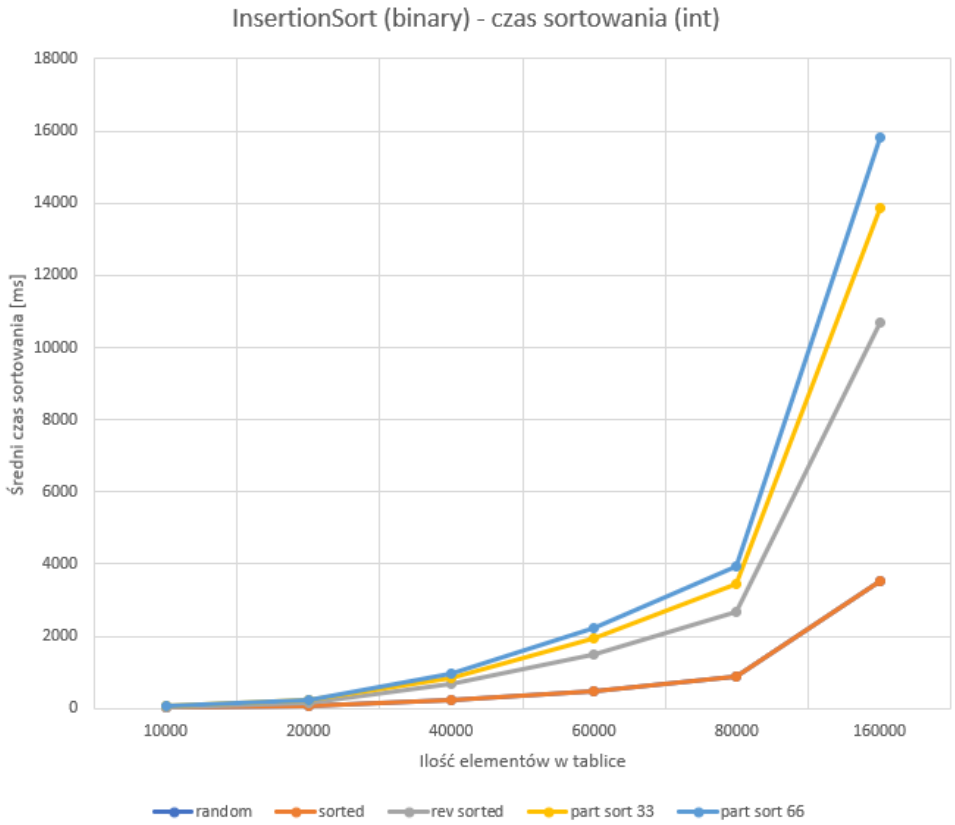


Rysunek 1 - Porównanie czasu algorytmu QuickSort dla liczb całkowitych (int) w funkcji rozmiaru tablicy wejściowej z uwzględnieniem różnych typów tablic wejściowych. Źródło: opracowanie własne.

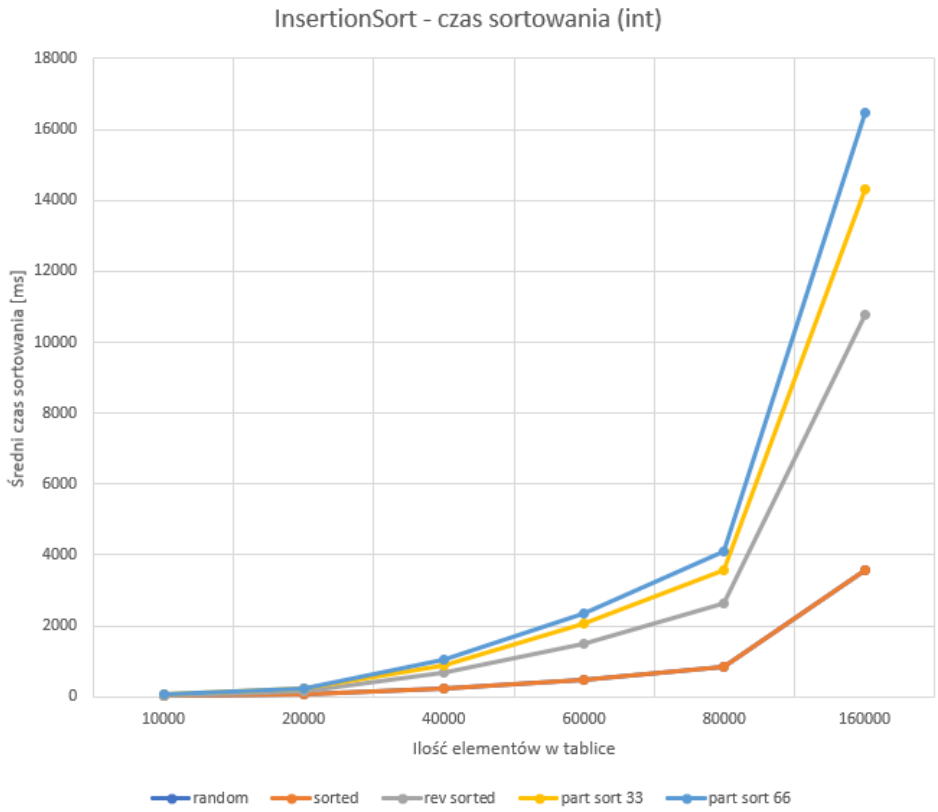
[3] Rejestry SSE (ang. Streaming SIMD Extensions) to zestaw instrukcji procesora umożliwiający równoległe przetwarzanie wielu danych w jednej operacji (SIMD). Rejestry te mają rozmiar 128 bitów i pozwalają np. na jednoczesne porównanie czterech liczb zmiennoprzecinkowych 32-bitowych (float). Kompilatory mogą wykorzystywać je do optymalizacji operacji na tablicach, nawet bez jawnego użycia intrinsics w kodzie.



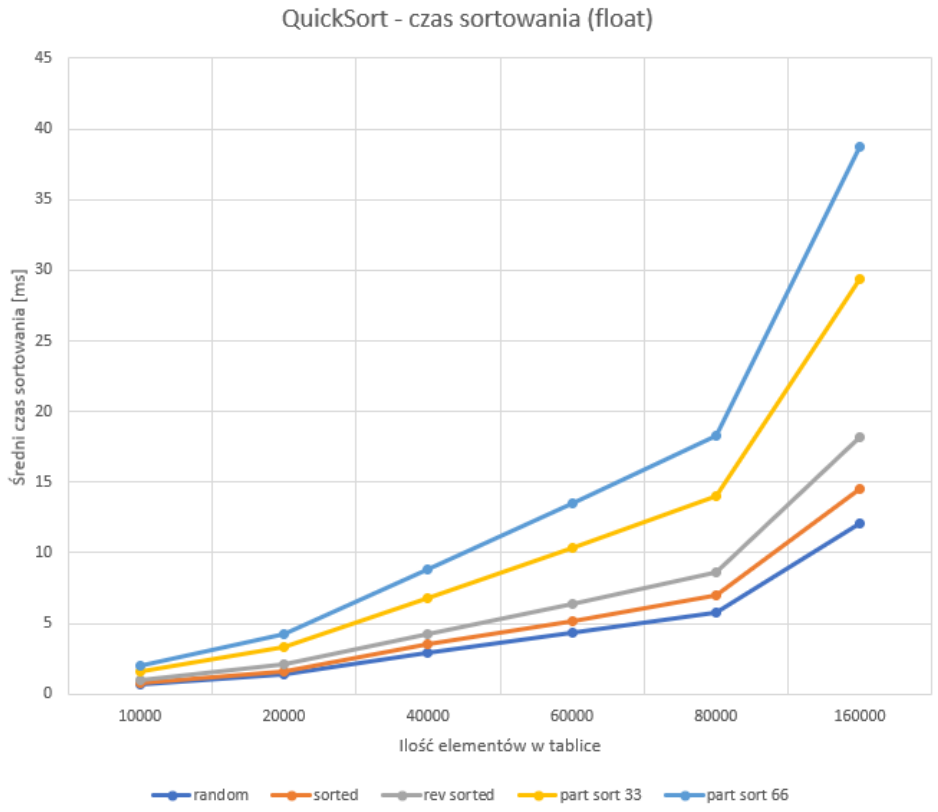
Rysunek 2 - Porównanie czasu algorytmu HeapSort dla liczb całkowitych (int) w funkcji rozmiaru tablicy wejściowej z uwzględnieniem różnych typów tablic wejściowych. Źródło: opracowanie własne.



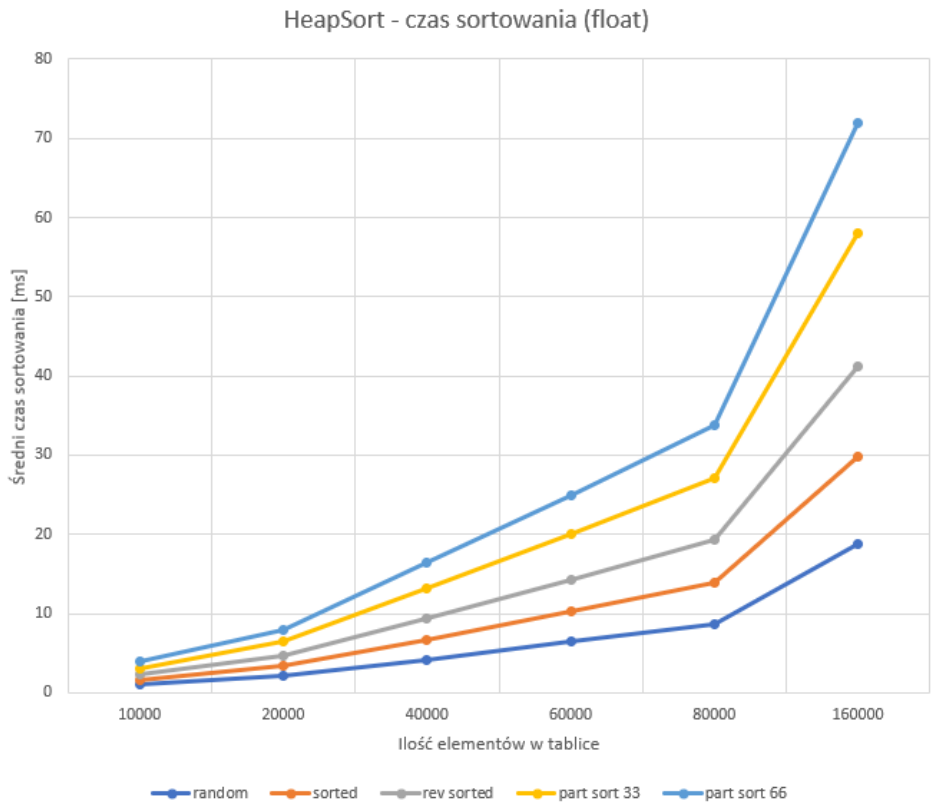
Rysunek 3 - Porównanie czasu algorytmu InsertionSort (Binary) dla liczb całkowitych (int) w funkcji rozmiaru tablicy wejściowej z uwzględnieniem różnych typów tablic wejściowych. Źródło: opracowanie własne.



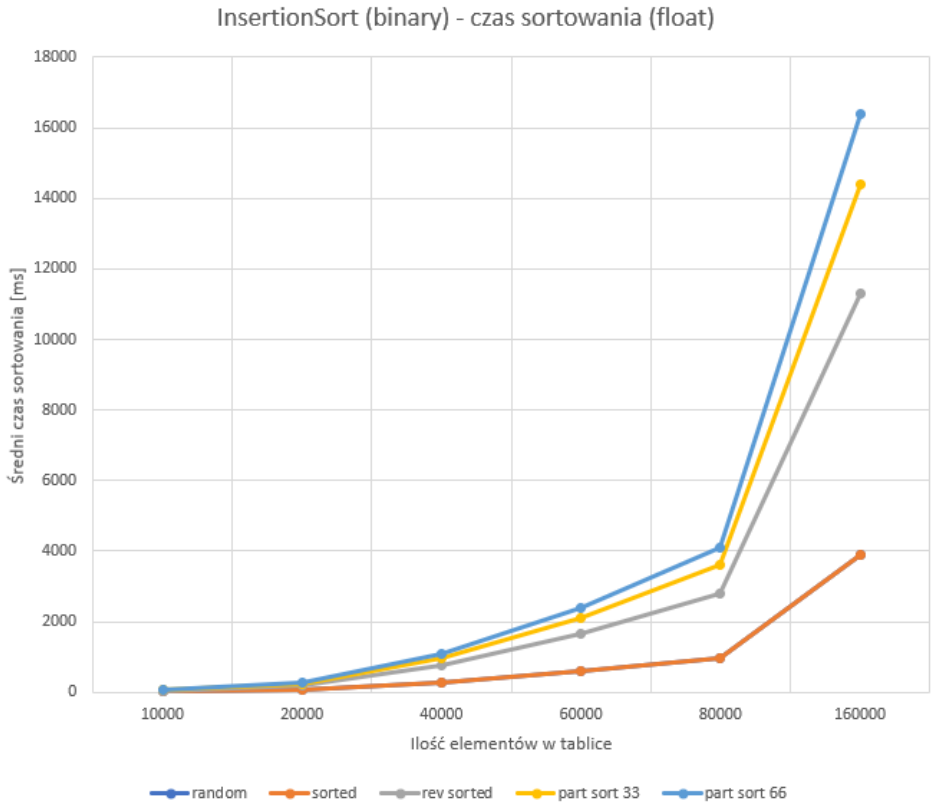
Rysunek 4 - Porównanie czasu algorytmu InsertionSort dla liczb całkowitych (int) w funkcji rozmiaru tablicy wejściowej z uwzględnieniem różnych typów tablic wejściowych. Źródło: opracowanie własne.



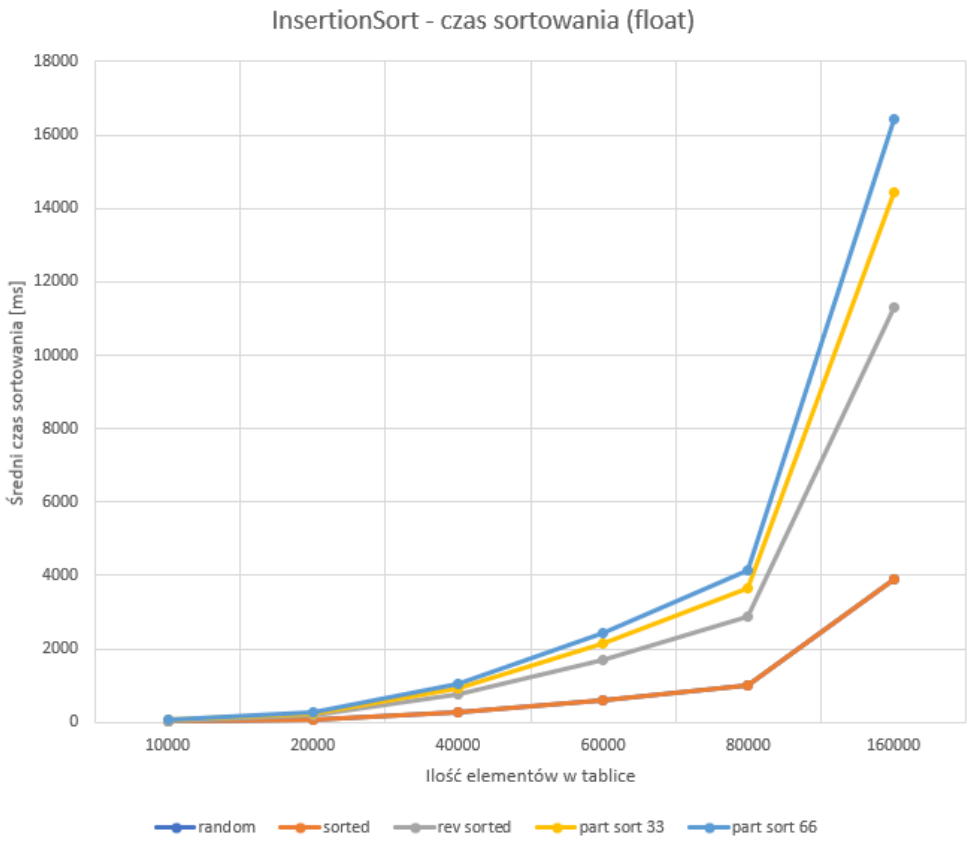
Rysunek 5 - Porównanie czasu algorytmu QuickSort dla liczb zmiennoprzecinkowych (float) w funkcji rozmiaru tablicy wejściowej z uwzględnieniem różnych typów tablic wejściowych. Źródło: opracowanie własne.



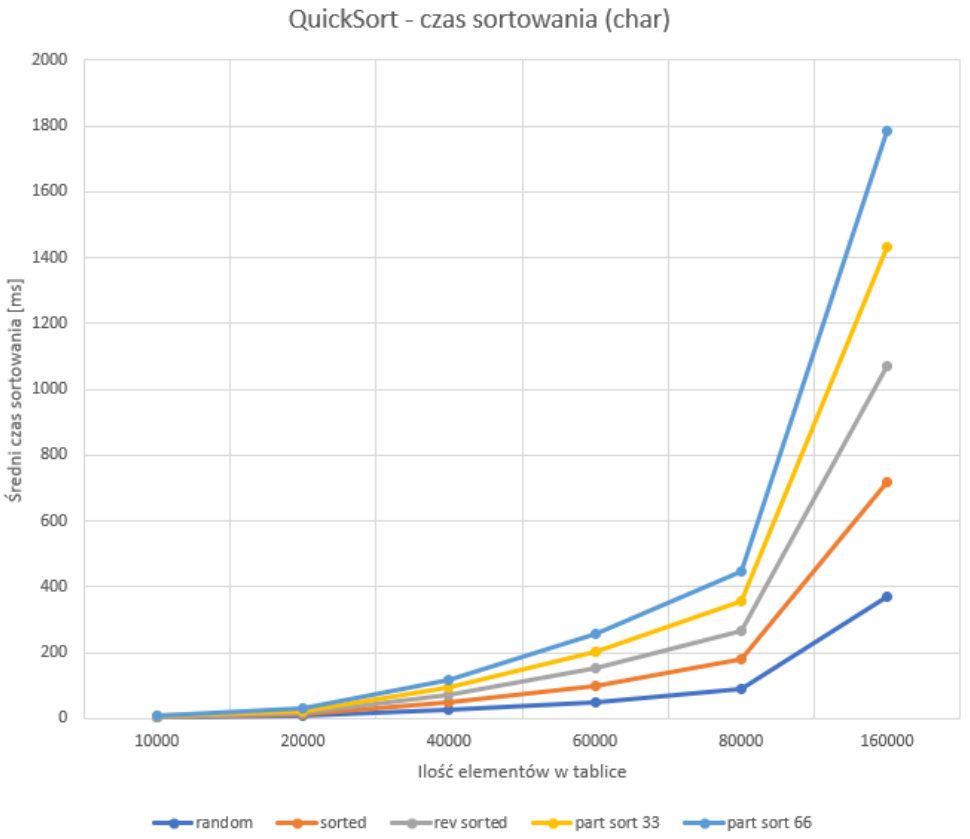
Rysunek 6 - Porównanie czasu algorytmu HeapSort dla liczb zmiennoprzecinkowych (float) w funkcji rozmiaru tablicy wejściowej z uwzględnieniem różnych typów tablic wejściowych. Źródło: opracowanie własne.



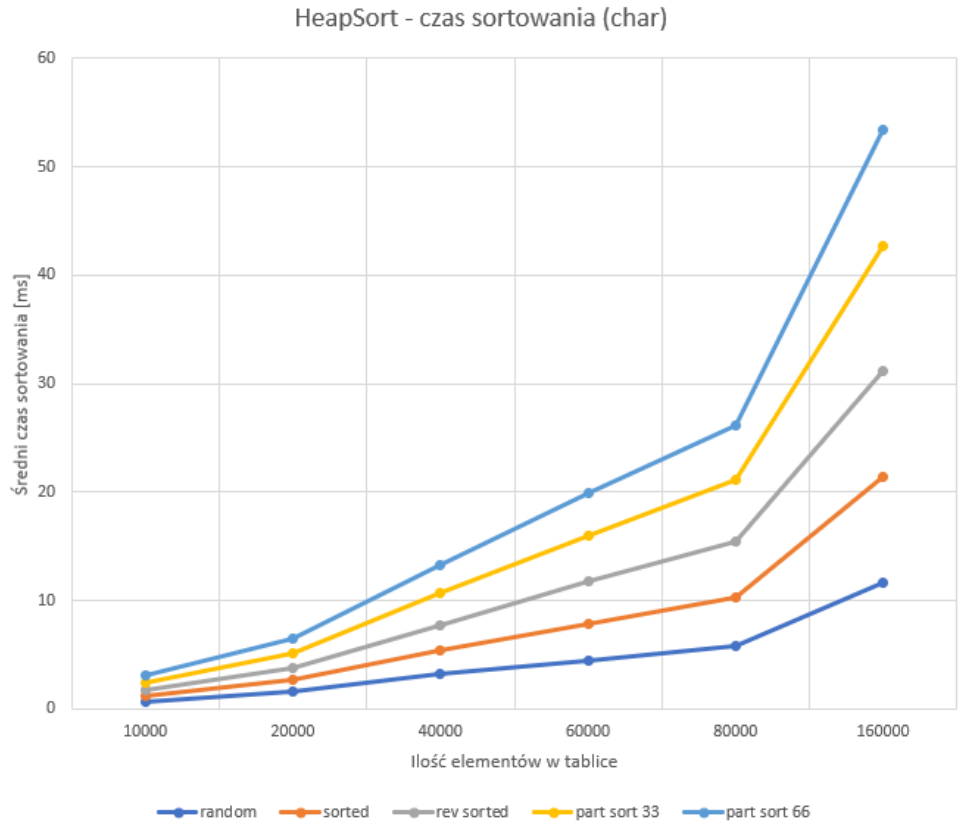
Rysunek 7 - Porównanie czasu algorytmu InsertionSort (Binary) dla liczb zmiennoprzecinkowych (float) w funkcji rozmiaru tablicy wejściowej z uwzględnieniem różnych typów tablic wejściowych. Źródło: opracowanie własne.



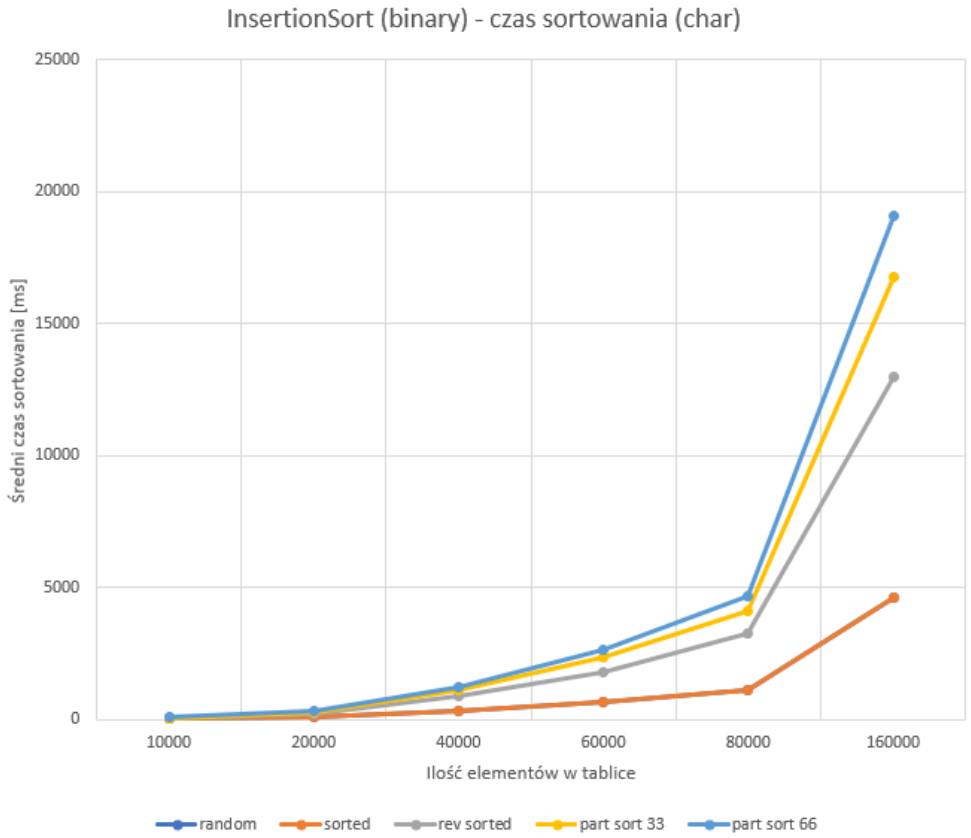
Rysunek 8 - Porównanie czasu algorytmu InsertionSort dla liczb zmiennoprzecinkowych (float) w funkcji rozmiaru tablicy wejściowej z uwzględnieniem różnych typów tablic wejściowych. Źródło: opracowanie własne.



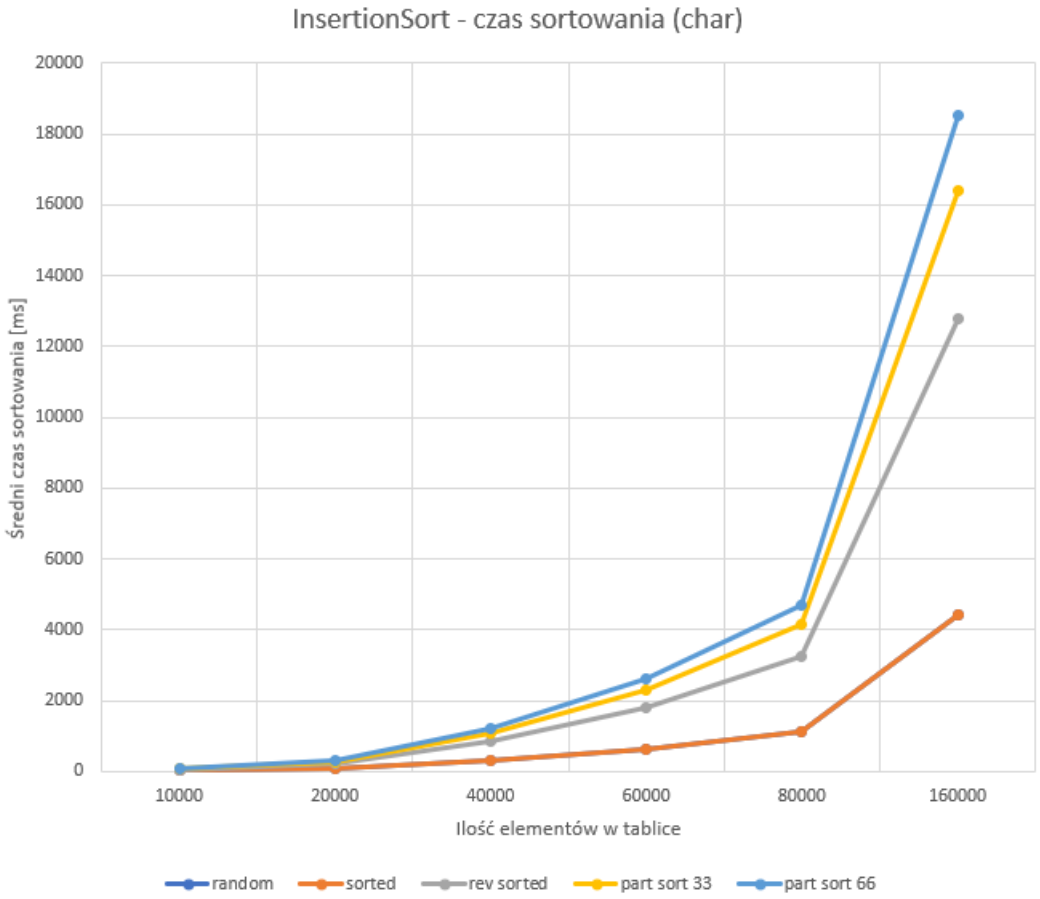
Rysunek 9 - Porównanie czasu algorytmu QuickSort dla znaków (char) w funkcji rozmiaru tablicy wejściowej z uwzględnieniem różnych typów tablic wejściowych. Źródło: opracowanie własne.



Rysunek 10 - Porównanie czasu algorytmu HeapSort dla znaków (char) w funkcji rozmiaru tablicy wejściowej z uwzględnieniem różnych typów tablic wejściowych. Źródło: opracowanie własne.



Rysunek 11 - Porównanie czasu algorytmu InsertionSort (Binary) dla znaków (char) w funkcji rozmiaru tablicy wejściowej z uwzględnieniem różnych typów tablic wejściowych. Źródło: opracowanie własne.



Rysunek 12 - Porównanie czasu algorytmu InsertionSort dla znaków (char) w funkcji rozmiaru tablicy wejściowej z uwzględnieniem różnych typów tablic wejściowych. Źródło: opracowanie własne.

4. Podsumowanie i wnioski

Eksperymenty potwierdziły fundamentalne zasady teorii algorytmów, ale ujawniły również złożoność praktycznych implementacji. Kluczowe wnioski:

- Algorytmy O(n log n) (*Quick Sort, Heap Sort*) są niezastąpione dla dużych zbiorów, ale ich wydajność zależy od implementacyjnych detali, takich jak wybór pivota lub zarządzanie pamięcią.
- Insertion Sort pozostaje użyteczny tylko dla małych lub częściowo uporządkowanych danych, gdzie jego adaptacyjność przekłada się na konkretne korzyści.
- Typ danych ma większy wpływ na wydajność niż często zakładano — różnice sięgają kilkunastokrotności, co wymaga uwzględnienia przy projektowaniu systemów.
- Struktura danych może zarówno przyspieszać (dane posortowane), jak i dramatycznie spowalniać (dane odwrotnie posortowane) działanie algorytmów, co podkreśla wagę analizy kontekstu zastosowań.
- Wyniki eksperymentów wskazują, że wybór algorytmu sortowania powinien być poprzedzony nie tylko analizą teoretycznej złożoności, ale także praktycznym testowaniem na reprezentatywnych danych. Nawet niewielkie optymalizacje (np. randomizacja pivota w Quick Sort) mogą znacząco poprawić stabilność algorytmu w rzeczywistych warunkach.

5. Bibliografia

5.1. Literatura

- Cormen, T. H., Algorithms Unlocked, The MIT Press, 2013.
- https://www.geeksforgeeks.org/
- https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions
- https://stackoverflow.com/

5.2. Kod źródłowy

https://github.com/IShnitko/algorithms-and-computational-complexity-1