Porównanie zdolności sieci do implementacji najbardziej podstawowych algorytmów w javie

Każdy prompt to rozpoczęcie nowej konwersacji z siecią. Wszystkie public static void main rozbudowałem o mierzenie czasu wykonania kodu w nich.

Badanie 18.10.2025

**Algorytm sortujący**

**Badanie bez użycia dodatkowych technik**

Zdecydowałem się na możliwie najbardziej lakoniczny opis, chcąc zachęcić sieci do własnej inicjatywy.

Write me optimal sorting algorithm in java.

Żadna sieć nie napisała programu od razu gotowego do uruchomienia. W tym celu sprecyzuję im, aby program był od razu gotowy do uruchomienia w pliku main.java. Dodatkowo usunę fragment o „optymalnym” algorytmie.

Write me a sorting algorithm in java ready to be executed in Main.java file.

Wszystkie sieci zwróciły kody gotowe do uruchomienia. Wszystkie też dostarczyły swoje zestawy danych do sortowania zawierające wyłącznie liczby naturalne.

* ChatGPT zastosował sortowanie przez scalanie z wykorzystaniem 7 liczb. Czas wykonania kodu wyniósł 3,97 ms.
* DeepSeek zastosował szereg różnych algorytmów sortujących: zaimplementował sortowanie bąbelkowe, sortowanie przez scalanie i sortowanie szybkie; również wybrał 7 liczb do posortowania. Dodatkowo przedstawił pojedyncze sortowanie dużej tablicy zawierającej 15 liczb. Nieproszony, dokonał pomiaru czasu wykonania każdej z zaimplementowanych metod, a także zaimplementował metody do tworzenia losowych zbiorów liczb. Według tych wyliczeń, posortowanie tablicy liczącej 1000 elementów zajęło sortowaniu szybkiemu ~0 ms, sortowaniu przez scalanie ~1 ms, a wbudowana metoda Arrays.sort() również ok ~0 ms. Wykonanie całego kodu zajęło 24,2137 ms. W celu lepszego porównania, zdecydowałem się wziąć fragment odpowiedzialny za szybkie i podmienić nim funkcję w kodzie ChatGPT. Czas wykonania kodu wyniósł 3,85 ms.
* W przeciwieństwie do dwóch poprzednich sieci, Claude postawił na implementację algorytmu sortowania szybkiego. Użył w tym celu większej tablicy niż pozostałe sieci – zaproponowana przez niego miała 10 liczb, a wykonie całego kodu zajęło aż 14,64 ms. Ponownie w celu lepszego porównania, zdecydowałem się na przeklejenie fragmentu odpowiedzialnego za sortowanie do kodu ChataGPT. Czas wykonania kodu wyniósł 4,18 ms.

Podsumowanie: Mimo braku zachęty z mojej strony, sieci postawiły na możliwie najefektywniejsze algorytmy, nie na najprostsze do implementacji, przy czym warto zauważyć, że tylko dwie sieci postawiły na algorytm sortowania szybkiego, prezentowany jako najbliższy optimum. W ramach dodatkowego badania, stworzyłem 5 losowych tablic po 7 liczb każda. Wyniki wykonania całego programu dla każdej z sieci były następujące: 3,31 ms dla DeepSeek, 3,88 ms dla ChataGPT i 3,62 ms dla Claude. Najbardziej do optimum zbliżył się DeepSeek.

**Badanie z wykorzystaniem dodatkowych technik**

Zapytanie do sieci zostanie rozbudowane o rodzaj IDE, jego wersję, prośbę o wcielenie się w rolę, a także skonstruowanie możliwie najbliższego optimum algorytmu gotowego do uruchomienia.

As a professional Java developer with over ten years of active work, write for me optimal sort algorithm, ready for use in Main.java file. You work in IntelliJ IDEA Community Edition 2025.2.3. If you have any additional questions ask them freely.

Claude i ChatGPT zapytały o dodatkowe informacje. ChatGPT zapytał się o to jak rozumiem znaczenie słowa optymalny, podając kilka przykładów; wskazałem na najbardziej zoptymalizowany pod kątem wydajności algorytm ogólnego zastosowania. Dodatkowo, chciał wiedzieć, w jaką wersję celuję; odparłem że 21.

Claude natomiast zapytał o typ danych przewidzianych do sortowania, jakie są główne kryteria optymalizacji (prędkość dla większości przypadków, zużycie pamięci, stabilność, wydajność na prawie posortowanych danych), średnia wielkość danych, czy sortowanie ma się odbywać w miejscu, dodatkowe wymagania (zapewnienie wielowątkowości, sortowanie obiektów czy prymitywów, specjalistyczne wsparcie komparatora); odpowiedziałem, że sortowane będą liczby całkowite, ma się skupić na średniej prędkości, typowa tablica będzie niewielka (<100 elementów), nie ma znaczenia czy sortowanie będzie w miejscu czy nie i nie mam żadnych dodatkowych wymagań. Dodatkowo, zaproponował mi wykorzystanie wbudowanych rozwiązań; poprosiłem by napisał dla mnie od podstaw.

Kod zwrócony przez DeepSeek był podzielony na dwie klasy mimo moich wcześniejszych próśb. Dodatkowo, z powodu zastosowania w nim w klasie Main metody prywatnej z drugiej ze zwróconych klas. Po ręcznej zamianie tej klasy na publiczną, kod się wykonał w 85,29 ms.

Podobnie jak w poprzednim przypadku, sieć zaimplementowała szereg algorytmów i przedstawiła je dla różnych przypadków, między innymi sortowanie listy obiektów stworzonej przez siebie klasy po konkretnym atrybucie. Zaproponowanymi przez nią metodami były:

* Sortowanie szybkie z dwoma elementami osiowymi,
* Sortowanie Tima,
* Sortowanie przez wstawianie,
* Sortowanie na kopcu.

Następnie sieć z własnej inicjatywy porównała czasy wykonywania zaimplementowanych algorytmów dla różnych przypadków oraz dodatkowo upewniła się, że zwrócone tablice faktycznie zawierają posortowane elementy. Według otrzymanych wyników:

* Dla tablic 50 elementów najlepsze okazało się sortowanie przez wstawianie (0,081 ms),
* Dla tablic 1000 elementów najlepsze okazało się sortowanie na kopcu (0,599 ms),
* Dla tablic 10000 elementów najlepsze okazało się sortowanie Tima (4,729 ms),
* Dla „prawie posortowanej” tablicy 1000 elementów najlepsze okazało się sortowanie na kopcu (0,238 ms).

Dodatkowo, dla każdego porównania przedstawiony został także wynik zaimplementowanej domyślnie Arrays.sort(). Warto zauważyć, że tylko w jednym przypadku osiągnęła ona wynik lepszy niż jakakolwiek z innych zaimplementowanych metod; było to dla ostatniego przypadku („prawie posortowanej” tablicy 1000 elementów)

Kod zwrócony przez ChatGPT zgodnie z założeniami zawierał się w tylko jednej klasie i wykorzystywał zbliżony do sortowania Tima algorytm będący połączeniem sortowania przez scalanie i sortowania przez wstawianie. Wykonanie całego kodu zajęło 96,22 ms, a posortowana została tablica 100.000 losowych liczb całkowitych; według zaprezentowanego na koniec wyniku czas jej posortowania wyniósł 54, 069 ms.

Kod wygenerowany przez Claude także zawierał się w jednej klasie, a wykorzystywał sortowanie przez wstawianie jako swój podstawowy algorytm; jego wykonanie zajęło 6,05 ms dla kilku niewielkich tablic (największa zawierała 7 elementów).

W celu porównania jakości otrzymanych kodów zdecydowałem się je sprawdzić na dziesięciu tysiącach tabel zawierających po 70 nieposortowanych liczb naturalnych. Razem z uruchomieniem programu oraz wygenerowaniem tablic, czasy wyszły następujące:

* Średnio 179,72 ms dla sortowania hybrydowego opracowanego przez ChatGPT,
* Średnio 116,76 ms dla sortowania przez wstawianie opracowanego przez DeepSeek (zdecydowałem się na ten algorytm, ponieważ sama sieć zalecała go w komentarzach jako dobry do niewielkich tablic),
* Średnio 81,54 ms dla sortowania przez wstawianie opracowanego przez Claude.

**Porównanie**

Postanowiłem porównać te wyniki z otrzymanymi z kodów wygenerowanych w pierwszej próbie celem stwierdzenia obecności bądź braku poprawy jakości kodu. W tym porównaniu otrzymałem następujące wyniki:

* Średnio 108,94 ms dla sortowania przez scalanie opracowanego przez ChatGPT,
* Średnio 52,10 ms dla sortowania szybkiego opracowanego przez DeepSeek.
* Średnio 52,75 ms dla sortowania szybkiego opracowanego przez Claude.

Ku mojemu ogromnemu zdziwieniu, żadna sieć nie dostarczyła kodu lepszej jakości w drugiej próbie. Możliwe, że powodem tego były niesprecyzowane wymagania dotyczące rozmiaru sortowanych danych, które są kluczowe dla wielu algorytmów; tablice 70 elementów są bardzo małe.

**Rozwinięcie wymagań technicznych w prompcie**

Mając to na uwadze rozczarowujące wyniki poprzedniej próby, postanowiłem ponownie zredagować prompt, tym razem przykładając większą wagę do wymagań czysto technicznych działania samego algorytmu. Skonstruowałem następujący prompt:

As a professional Java developer with over ten years of active work, write for me optimal sort algorithm to sort a table of 70 random generated integers, ready for use in Main.java file. Your algorithm should focus mostly on execution speed. You work in IntelliJ IDEA Community Edition 2025.2.3 and use Java 21. If you have any additional questions ask them freely.

Ponownie ChatGPT oraz Claude zadały mi dodatkowe pytania; DeepSeek od razu przystąpił do pracy. ChatGPT chciał się dowiedzieć, czy możliwe jest wykorzystanie Arrays.sort(), czy sortowanie ma być w miejscu oraz czy losowane zbiory liczb mogą zawierać duplikaty (postanowił także wygenerować fragment kodu odpowiedzialny za losowanie); odparłem że implementacja ma być od podstaw, nie ma znaczenia czy sortowanie będzie w miejscu czy nie oraz, że losowane zbiory mogą zawierać duplikaty. Claude spytał czy będzie to jednorazowe sortowanie, czy będzie to wiele sortowań po sobie, jaki jest zakres sortowanych liczb, czy istotne jest zużycie pamięci, czy ma przeprowadzić testy porównawcze; odparłem, że sortowanie będzie wielokrotne dla różnych zbiorów, zakres sortowanych liczb będzie się zawierać od 0 do 100, nie ma znacznie zużycie pamięci oraz ma nie przeprowadzać żadnych testów tylko zaprezentować jedną, najlepszą implementację.

ChatGPT postawił na sortowanie szybkie z dwoma elementami osiowymi; dostarczony przez niego kod wykonał się w 6,19 ms. DeepSeek zaprezentował dwa algorytmy - sortowanie szybkie z dwoma elementami osiowymi oraz sortowanie przez zliczanie – a także dokonał ich porównania z Arrays.sort(); według wyliczeń sortowanie przez zliczanie było znacznie szybsze od dwóch pozostałych implementacji (52 µs w porównaniu do 0,89 ms dla sortowania szybkiego i 0,39 ms dla Arrays.sort()); wykonanie całego kodu zajęło 25,38 ms. Claude zaimplementował także sortowanie przez zliczanie; jego kod wykonał się 6,71 ms. Poddałem otrzymane algorytmy analogicznej próbie jak poprzednie. Otrzymałem następujące wyniki:

* Średnio 51,17 ms dla sortowania szybkiego z dwoma elementami osiowymi opracowanego przez ChatGPT.
* Średnio 42,24 ms dla sortowania przez zliczanie opracowanego przez DeepSeek,
* Średnio 38,98 ms dla sortowania przez zliczanie opracowanego przez Claude.

Badanie 21.10.2025r.

**Kolejka priorytetowa**

**Badanie bez użycia dodatkowych technik**

Write me a lifo queue in java ready to be executed in Main.java file.

Wszystkie sieci zwróciły kody gotowe do uruchomienia.

ChatGPT stworzył prostą klasę poza klasą Main, użył LinkedList jako podstawowej konstrukcji i wyposażył ją w metody do usuwania ostatniego dodanego elementu (pop), dodawania nowych (push), podejrzenia znajdującego się na szczycie (peek), sprawdzenia czy kolejka jest pusta (isEmpty) oraz ilości elementów znajdujących się w niej (size). Przedstawił na kilku liczbach zasady działania zaimplementowanych przez siebie metod; kod wykonał się w 16,74 ms.

DeepSeek analogicznie stworzył nową klasę poza Main; w zbliżonej do ChataGTP manierze, wykorzystał ArrayList i rozbudował dostępne metody o konstruktor (LifoQueue), usuwającą wszystkie znajdujące się w niej elementy (clear) oraz zamieniającą kolejkę na łańcuch znaków (toString). Zapewne chcąc zaprezentować większą elastyczność, podał także odpowiednio więcej przykładów; kod wykonał się w 28,10 ms.

Claude postąpił podobnie do ChataGPT – stworzył nową klasę, najważniejszy element opierając o LinkedList; stworzył również taki sam zestaw metod, uzupełniając go jednak o kilka metod, które już pojawiły się w DeepSeeku - metodę zamieniającą kolejkę na łańcuch znaków (toString) oraz konstruktor. Tak samo jak pozostałe sieci, zaprezentował także kilka przykładów użycia; kod wykonał się w 14,62 ms.

Warto tutaj wspomnieć, że wszystkie stworzone kolejki zostały pomyślane jako działające dla wszystkich rodzajów przekazywanych danych.

W ramach porównania jakości implementacji, do każdej z kolejek zostało włożone, a następnie wyjęte 100 tysięcy liczb naturalnych od 0 do 99.999.

* 1163,56 ms zajęło wykonanie kodu ChatGPT (przy każdym dodaniu elementu i jego usunięciu wypisywał informację o tym na konsolę; po usunięciu tych komunikatów czas wykonania spadł do 14,50 ms);
* 16,01 ms zajęło wykonanie kodu DeepSeek;
* 14,92 ms zajęło wykonanie kodu Claude.

W tym badaniu żadna z sieci nie wyróżniła się specjalnie. Najlepsze wyniki bezwzględne osiągnął Claude, a po drobnych poprawkach na prowadzenie wysunął się ChatGPT. Z ciekawości sprawdziłem, ile na tych samych implementacjach będzie wykonywała się podobna operacja oparta na 10 milionach powtórzeń.

* 1723,33 ms zajęło wykonanie kodu ChatGPT (po usunięciu każdorazowych wpisów na konsolę);
* 492,48 ms zajęło wykonanie kodu DeepSeek;
* 1686,75 zajęło wykonanie kodu Claude.

Jak widać powyżej, dla większej ilości liczb implementacja zaproponowana przez DeepSeek okazała się znacznie bardziej wydajna.

Badanie 26.10.2025r.

**Badanie z wykorzystaniem dodatkowych technik**

Podobnie jak w poprzednim badaniu, postaram się wykorzystać techniki zalecane w interakcjach z sieciami neuronowymi, bez jednoczesnego podawania charakteru danych, na których będą operować.

As a professional Java developer with over ten years of active work, write for me optimal lifo queue algorithm ready for use in Main.java file. Your algorithm should focus mostly on execution speed. You work in IntelliJ IDEA Community Edition 2025.2.3 and use Java 21. If you have any additional questions ask them freely.

Ponownie jak w poprzednich badaniach DeepSeek nie zadał żadnych dodatkowych pytań. ChatGPT zapytał o bezpieczeństwo dla rozwiązań wielowątkowych, mieć ograniczone czy nieograniczone zasoby pamięciowe, na jakich typach danych będzie funkcjonować, a także czy ma być zaimplementowana od podstaw, czy może wykorzystywać takie elementy jak ArrayDeque, czy LinkedList; odparłem, że nie musi się obawiać o sytuacje wielowątkowe, rozmiar kolejki powinien być z góry ograniczony, kolejka będzie wykorzystywana do liczb całkowitych, już zaimplementowane elementy mogą być wykorzystywane i powinien się skupić na prędkości wykonania. Claude także zadał pytania, jednak nie czekając na moją odpowiedź, przystąpił do implementacji, tworząc pewne założenia. Zapytał o bezpieczeństwo dla rozwiązań wielowątkowych (założył, że będzie stosowana w sytuacjach jednowątkowych), spodziewaną ilość elementów znajdujących się w niej (nie napisał, co założył), typ danych przetwarzanych (założył że kolejka ma być generyczna), czy kolejka ma mieć zawczasu zaalokowaną pamięć aby działać szybciej (postawił na rozwiązanie pośrednie, pozwalając jej na wzrost rozmiaru z jednoczesnym wcześniejszym zarezerwowaniem miejsca), czy jakieś inne operacje poza dołożeniem oraz usunięciem z kolejki będą konieczne (założył, że poza nimi przydadzą się podejrzenie pierwszego elementu oraz sprawdzenie, czy kolejka jest pusta); odparłem, że nie musi się przejmować odpornością na sytuacje wielowątkowe, powinien się spodziewać użycia dla ponad 10 tysięcy elementów, wykorzystywana będzie do liczb całkowitych, powinien się skupić przede wszystkim na prędkości przetwarzania oraz jedynie operacje dodawania do kolejki oraz usuwania z niej są niezbędne.

Kod zaproponowany przez ChatGPT wykorzystywał jako podstawową strukturę do przechowywania danych tablicę, której rozmiar był jedynym parametrem konstruktora i zawierał niewielkie demo dla 5 liczb; wykonał się w 1,64 ms. Kod DeepSeek był zdecydowanie najbardziej rozbudowany.