Algorytmy i złożoność obliczeniowa – Projekt Sprawozdanie

Jan Napieralski

273036

09.04.2024r

Zadanie projektowe nr 1	
Badanie efektywności wybranych algorytmów sortowania ze względu na złożoność obliczeniową	
Prowadzący kurs	Dr. Inż. Zbigniew Buchalski
Termin zajęć	Wtorki Nieparzyste 7:30-9:15
Termin oddania	16.04.2024r

Wprowadzenie

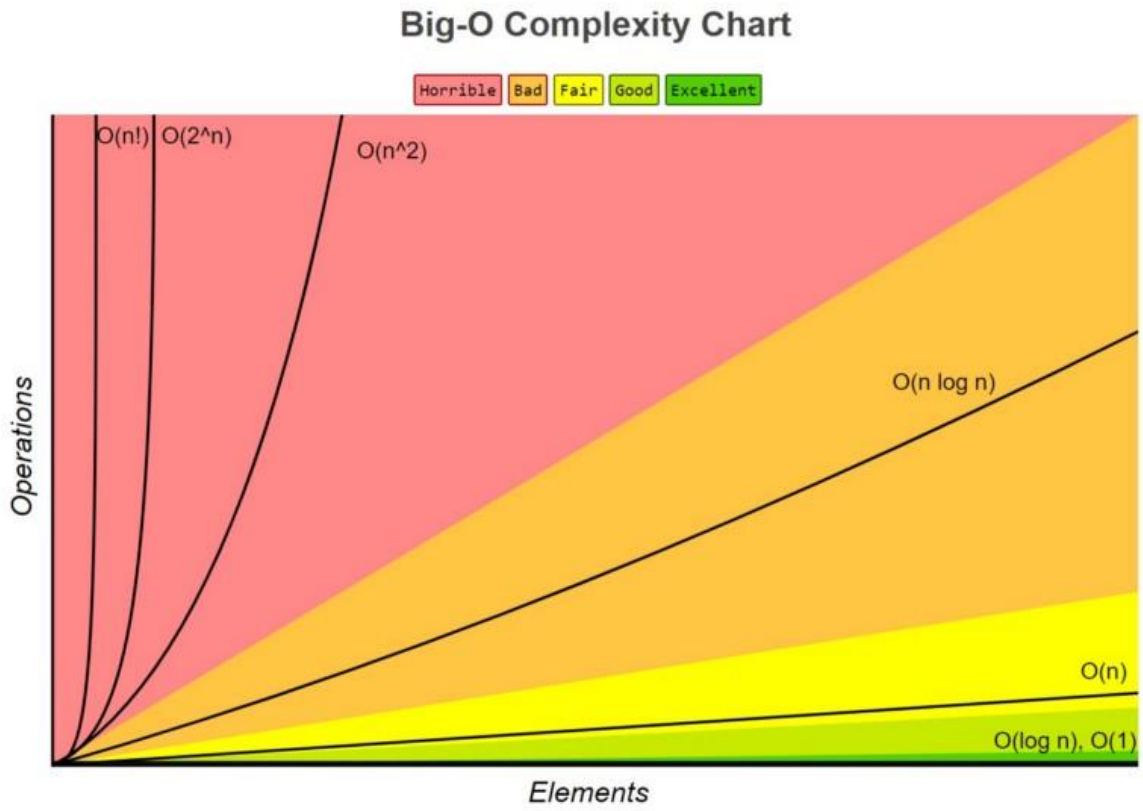
Zadaniem projektowym było wybranie jednego z predefiniowanych przez prowadzących zbiorów algorytmów sortowań, ich realizacja na podstawowych typach danych int, char, float oraz zaprojektowanie i przeprowadzenie eksperymentu weryfikacji złożoności obliczeniowej podanej w literaturze z wynikami pomiarów algorytmów na kilku przypadkach tablic. Wybrane przeze mnie algorytmy sortowania to:

- Sortowanie przez wstawianie (eng. Insertion Sort)
- Sortowanie przez kopcowanie (eng. Heap Sort)
- Sortowanie szybkie (eng. Quick sort)
- Sortowanie Shella (eng. Shell Sort)

Notacją zastosowaną do przedstawienia złożoności obliczeniowej algorytmów jest notacja dużego O. Jest ona matematyczną notacją używaną do opisywania górnych ograniczeń (złożoności) algorytmów i funkcji. Jest często stosowana w analizie algorytmów, aby określić, jak długo (czasowo lub pamięciowo) algorytm będzie działał w zależności od wielkości danych wejściowych.

Formalnie, jeśli f(n) i g(n) są dwiema funkcjami określonymi dla wszystkich dodatnich liczb całkowitych n, to mówimy, że f(n) jest O(g(n)), oznaczane jako f(n) = O(g(n)), jeśli istnieją stałe dodatnie c i n0 takie, że dla wszystkich n > n0, funkcja f(n) jest ograniczona od góry przez funkcję c * g(n).

W skrócie, notacja dużego O mówi nam, jak szybko wzrasta funkcja (czasowo lub pamięciowo) w najgorszym przypadku, gdy n (rozmiar danych) dąży do nieskończoności. Daje nam to informacje o złożoności algorytmów w ich najgorszym przypadku, co pomaga w porównywaniu i ocenie efektywności algorytmów w zależności od wielkości danych wejściowych.



Zestawienie efektywności różnych algorytmów o złożoności dużego O

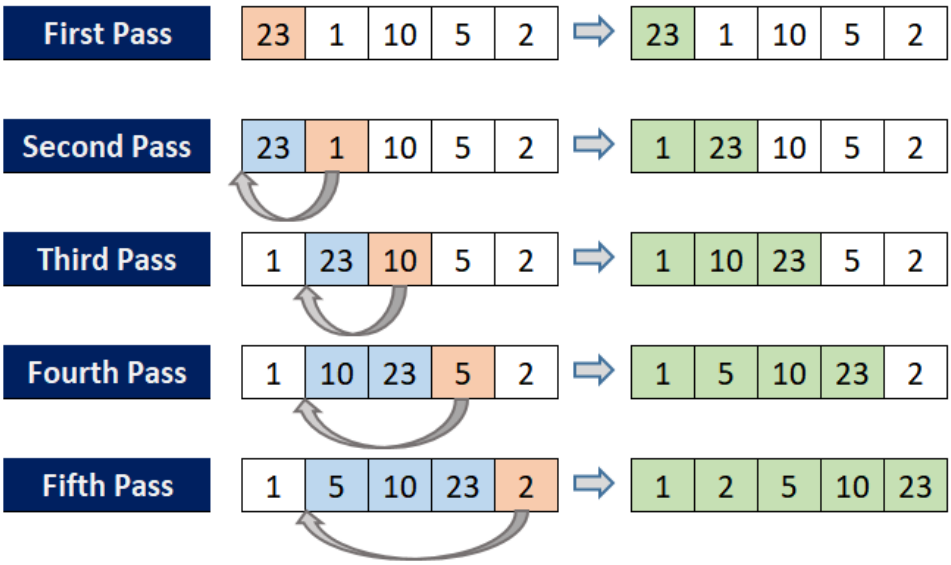
https://cdn-media-1.freecodecamp.org/images/1*KfZYFUT2OKfjekJlCeYvuQ.jpeg

Jak widać to na grafice powyżej, algorytmy o złożoności powyżej $O(n\ log\ n)$ są bardzo złe, ponieważ bardzo szybko wzrasta ich czas wykonania dla większych danych. W praktyce algorytmy powyżej złożoności $O(n^2)$ są uznawane jako nieefektywne i nie powinny być używane w większości przypadków. Wskazane jest wtedy znalezienie alternatywy

Sortowanie przez wstawianie

Najlepszy przypadek	Przypadek średni	Przypadek najgorszy
$O(n)$	$O(n^2)$	$O(n^2)$

Ta metoda sortowania opiera się na bardzo prostym mechanizmie. Przechodzimy po kolei poprzez elementy tablicy. Dla każdego rozpatrywanego elementu sprawdzamy czy poprzedni element nie jest większy od obecnego. Jeśli jest, to zamieniamy kolejnością te dwa elementy. Wykonujemy takie porównania dopóki obecnie rozpatrywany element nie znajdzie się we właściwym miejscu. Zgodnie z literaturą, ten typ sortowania jest o wiele gorszy dla dużych zbiorów danych od innych np. sortowania szybkiego, ale ma on również swoje zalety. Jest bardzo prosty do zaimplementowania, efektywny dla małego zbioru danych, jest to algorytm stabilny



Graficzne przedstawienie sortowania poprzez wstawianie

*https://miro.medium.com/v2/resize:fit:765/0*1zi2XtjiLXa3LYZh.PNG*

Sortowanie przez kopcowanie

Najlepszy przypadek	Przypadek średni	Przypadek najgorszy
$O(n\ log\ n)$	$O(n\ log\ n)$	$O(n\ log\ n)$

Sortowanie przez kopcowanie wykorzystuje strukturę danych o nazwie kopiec. Jest to prawie pełne drzewo binarne, w którym wartość rodzica jest nie mniejsza od wartości jego maksymalnie dwóch potomków. Dzięki tej własności gwarantujemy, że w prawidłowo utworzonym kopcu, element maksymalny to jego korzeń, czyli pierwszy element. Sortowanie polega na podanym algorytmie:

1. Utwórz kopiec z podanej tablicy (O(log n))
2. Zamień korzeń kopca z ostatnim elementem kopca (O(1))
3. Usuń ostatni element kopca który jest teraz na prawidłowym miejscu (O(1))
4. Odtwórz kopiec (O(log n))
5. Wróć do punktu 2 jeśli kopiec ma więcej niż jeden element (O (n))

Ponieważ musimy powtarzać czynności 2-4 dla każdego elementu tablicy a w tej pętli najbardziej kosztowna operacja ma złożoność O (log n), to algorytm ma złożoność O(n log n). Należy zauważyć bardzo ciekawą właściwość tego sortowania, bo poprzez modyfikacje tablicy w strukturę kopca, algorytm ten jest nieczuły na stopień posortowania tablicy, czyli ma taką samą złożoność obliczeniową dla dowolnego przypadku

Sortowanie szybkie

Najlepszy przypadek	Przypadek średni	Przypadek najgorszy
$O(n\ log\ n)$	$O(n\ log\ n)$	$O(n^2)$

Sortowanie szybkie jest przykładem algorytmu z rodziny „dziel i zwyciężaj”. Dzieli on problem sortowania na mniejsze podproblemy, które są prostsze do rozwiązania, za pomocą mechanizmu rekurencji. Unikalnym pojęciem dla tego sortowania jest pivot. Będzie to dowolnie wybrany element z tablicy według którego będziemy ją sortowali. Algorytm przedstawia się następująco:

1. Wybierz element który będzie służył za pivota
2. Ustaw indeks lewy i prawy na skrajne indeksy rozważanej tablicy
3. Dopóki element z indeksem prawym jest większy od pivota, zmniejszaj indeks prawy
4. Dopóki element z indeksem lewym jest mniejszy od pivota, zwiększaj indeks lewy
5. Jeżeli indeks lewy i prawy nie minęły się, to zamień elementy pod ich indeksami ze sobą, zwiększ indeks lewy, zmniejsz indeks prawy
6. Jeżeli indeks lewy i prawy minęły się to zwróć indeks prawy
7. Podziel tablicę na dwie części według zwróconego indeksu prawego i wykonaj dla nich operację sortowania (od kroku 2)
8. Wykonuj tak długo jak nie natrafisz na podtablicę złożoną z 0 lub 1 elementu

Po każdym przejściu do kroku 7 mamy zależność, że elementy mniejsze od pivota w obecnie sortowana podtablica są od niego ustawione na lewo, większe zaś na prawo. Stosowanie takiej taktyki dla coraz mniejszych tablic w konsekwencji posortuje nam wszystkie dane. Ten algorytm jest dość unikalny ze względu na pivota, ponieważ może się okazywać, że dla różnych przypadków tablic inne taktyki ustalania pivota mogą wpłynąć na czas sortowania np. Zgodnie z literaturą wybieranie „skrajnego” lewego lub prawego elementu w tablicy już posortowanej daje nam złożoność $O(n^2)$. Z wad tego typu algorytmu można wymienić samą rekurencję, to znaczy, że nie jest on przystosowany do pracy na architekturach o małym rozmiarze stosu, gdyż narażamy się na błędy jego przepełnienia. Jest on za to dość prosty w implementacji i stabilny.

Sortowanie Shella

Najlepszy przypadek	Przypadek średni	Przypadek najgorszy (Shella)
$O(n)$	$O\left(n^{\frac{3}{2}}\right)$	$O(n^2)$

Najlepszy przypadek	Przypadek średni	Przypadek najgorszy (Lazarusa)
$O(n)$	$O(n\,log^2 n)$	$O\left(n^{\frac{3}{2}}\right)$

Sortowanie Shella zostało wymyślone w 1959 przez Donalda Shella. Można je nazwać uogólnieniem sortowania poprzez wstawianie. Osiąga się to poprzez zdefiniowanie zmiennej często określanej mianem odległości oraz sposobu jej zmniejszania. Dopóki nasza odległość będzie większa od zera, wykonujemy sortowanie poprzez wstawianie na wszystkich elementach oddalonych od siebie o obecną odległość. Po wykonaniu tej czynności zmniejszamy odległość. Dzięki temu stopniowo można przenosić elementy w docelowe miejsce szybciej niż przez konwencjonalne sortowanie poprzez wstawianie, które nie jest efektywne dla dużych zbiorów danych. Algorytm ten jest bardzo ciekawy, ponieważ jego złożoność pesymistyczna i średnia zależą od sposobu wyboru i zmniejszania odległości. Dla oryginalnego ciągu zaproponowanego przez Shella, sortowanie jest średnio szybsze niż $O(n^2)$, a przez dziesiątki lat, naukowcy wynajdywali sposoby żeby polepszyć jego złożoność czasową

Wyraz ogólny ciągu $(k \geqslant 1)$	Konkretne odstępy	Rząd złożoności pesymistycznej	Autor i rok publikacji
$\lfloor N/2^k \rfloor$	$\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \ldots, 1$	$\Theta(N^2)$ [gdy $N = 2^p$]	Shell, 1959[1]
$2\lfloor N/2^{k+1} \rfloor + 1$	$2\left\lfloor \frac{N}{4} \right\rfloor + 1, \ldots, 3, 1$	$\Theta(N^{3/2})$	Frank, Lazarus, 1960[2]
$2^k - 1$	$1, 3, 7, 15, 31, 63, \ldots$	$\Theta(N^{3/2})$	Hibbard, 1963[3]
$2^k + 1$, na początku 1	$1, 3, 5, 9, 17, 33, 65, \ldots$	$\Theta(N^{3/2})$	Papiernow, Stasiewicz, 1965[4]
kolejne liczby postaci $2^p 3^q$	$1, 2, 3, 4, 6, 8, 9, 12, \ldots$	$\Theta(N \log^2 N)$	Pratt, 1971[5]
$(3^k - 1)/2$, nie większe niż $\lceil N/3 \rceil$	$1, 4, 13, 40, 121, \ldots$	$\Theta(N^{3/2})$	Knuth, 1973[6]
$\prod_{\substack{0 \leqslant q < r \\ q \neq (r^2+r)/2-k}} a_q$, gdzie $r = \left\lfloor \sqrt{2k + \sqrt{2k}} \right\rfloor$, $a_q = \min\{n \in \mathbb{N} : n \geqslant (5/2)^{q+1},$ $\forall p : 0 \leqslant p < q \Rightarrow \gcd(a_p, n) = 1\}$	$1, 3, 7, 21, 48, 112, \ldots$	$O\left(N^{1+\sqrt{\frac{8\ln(5/2)}{\ln(N)}}}\right)$	Incerpi, Sedgewick, 1985[7]
$4^k + 3 \cdot 2^{k-1} + 1$, na początku 1	$1, 8, 23, 77, 281, \ldots$	$O(N^{4/3})$	Sedgewick, 1986[8]
$9(4^{k-1} - 2^{k-1}) + 1, 4^{k+1} - 6 \cdot 2^k + 1$	$1, 5, 19, 41, 109, \ldots$	$O(N^{4/3})$	Sedgewick, 1986[8]
$h_k = \max\{\lfloor 5h_{k-1}/11 \rfloor, 1\}, h_0 = N$	$\left\lfloor \frac{5N}{11} \right\rfloor, \left\lfloor \frac{5}{11} \left\lfloor \frac{5N}{11} \right\rfloor \right\rfloor, \ldots, 1$	?	Gonnet, Baeza-Yates, 1991[9]
$\lceil 1{,}8 \cdot 2{,}25^{k-1} - 0{,}8 \rceil$	$1, 4, 9, 20, 46, 103, \ldots$	?	Tokuda, 1992[10]
nieznany	$1, 4, 10, 23, 57, 132, 301, 701$	?	Ciura, 2001[11]

Porównanie obliczania odległości przez lata

https://pl.wikipedia.org/wiki/Sortowanie_Shella

Ostatnie 3 rekordy są o tyle ciekawe, ponieważ chociaż ich autorom nie udało się wykazać konkretnej złożoności obliczeniowej przez notację dużego O, to według ich wielokrotnych testów, opracowane przez nich ciągi zmniejszania odległości są lepsze od poprzednich. W tym sprawozdaniu algorytm Shella został zaimplementowany przy użyciu oryginalnie zaproponowanego sposobu zmniejszania odległości oraz sposobu Lazarusa

Plan eksperymentu

W badaniu złożoności obliczeniowej podanych algorytmów, przygotowałem ich implementację w języku C++, posługując się klasami szablonowymi przyjmującymi jako typ danych int, float lub char.

Każdy algorytm został reprezentowany poprzez klasę z implementacją jego sposobu sortowania, które jako podstawowe argumenty przyjmowało wskaźnik do pierwszego elementu i rozmiar tablicy. W wyniku operacji sortowania, modyfikowany został podany oryginał danych.

W każdej wymienionej klasie zaimplementowana została również metodę na mierzenie czasu wykonania sortowania. Wykonuje ona operację sortowania zadaną ilość razy na przekazanej tablicy mierząc czas sortowania poprzez użycie biblioteki std::chrono oraz podaje wynik jako średnią arytmetyczną zmierzonych czasów

Dla każdego algorytmu przygotowałem 5 scenariuszy ułożenia elementów w tablicy:

- Tablica z elementami losowymi
- Tablica posortowana rosnąco
- Tablica posortowana malejąco
- Tablica posortowana rosnąco w 33%, reszta losowo
- Tablica posortowana rosnąco w 66%, reszta losowo

Dla każdego scenariusza przeprowadziłem pomiary na zadanych wielkościach tablicy:

- 1 000 000 elementów
- 2 000 000 elementów
- 5 000 000 elementów
- 10 000 000 elementów
- 15 000 000 elementów

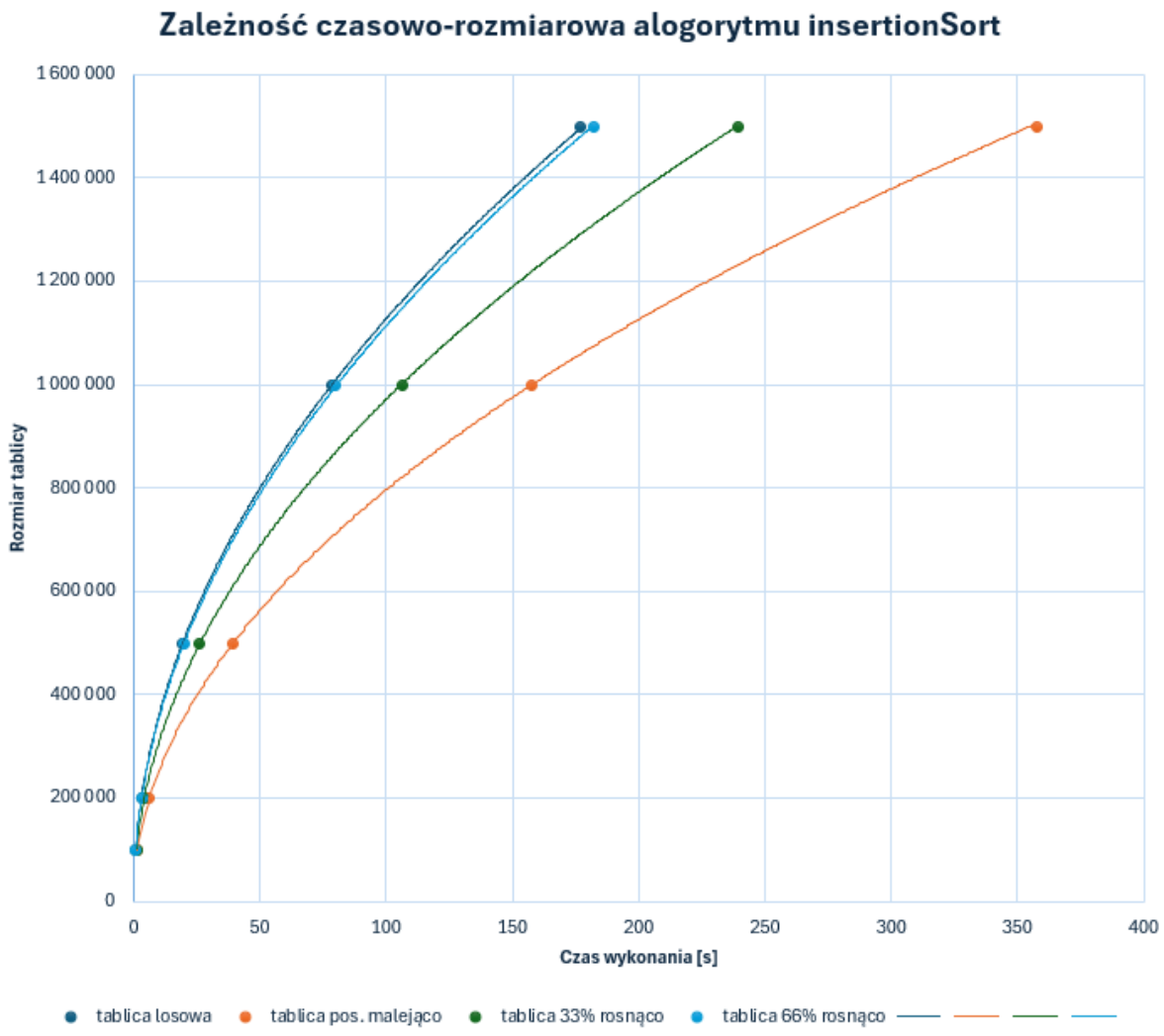
Jako efekt sortowania przyjąłem tablicę z elementami rosnącymi, podstawowym typem danych dla przeprowadzonych pomiarów był typ stało-liczbowy integer z wyjątkiem testu wpływu typu danych na długość sortowania wyszczególnionego w sprawozdaniu poniżej. Dla każdego pomiaru zostało automatycznie zweryfikowane czy tablica jest posortowana

Przebieg eksperymentu

Sortowanie poprzez wstawianie

Sortowanie przez wstawianie		
Stan tablicy	Czas sortowania [s]	Rozmiar
losowa	0,743489	100 000
losowa	3,08583	200 000
losowa	19,5656	500 000
losowa	78,4652	1 000 000
losowa	176,594	1 500 000
posortowana rosnąco	0,000226	1 000 000
posortowana rosnąco	0,000365	2 000 000
posortowana rosnąco	0,510801	5 000 000
posortowana rosnąco	1,071590	10 000 000
posortowana rosnąco	1,65062	15 000 000
posortowana malejąco	1,57974	100 000
posortowana malejąco	6,303140	200 000
posortowana malejąco	38,966600	500 000
posortowana malejąco	157,872000	1 000 000
posortowana malejąco	357,32300	1 500 000
0.33% posortowane rosnąco	1,04932	100 000
0.33% posortowane rosnąco	4,196990	200 000
0.33% posortowane rosnąco	26,179400	500 000
0.33% posortowane rosnąco	106,629000	1 000 000
0.33% posortowane rosnąco	239,08100	1 500 000
0.66% posortowane rosnąco	0,787809	100 000
0.66% posortowane rosnąco	3,225800	200 000
0.66% posortowane rosnąco	20,071800	500 000
0.66% posortowane rosnąco	79,932400	1 000 000
0.66% posortowane rosnąco	182,35900	1 500 000

Wyniki pomiarów przeprowadzonych na algorytmie insertionSort



Wyniki pomiarów przeprowadzonych na algorytmie insertionSort w postaci wykresu

Wnioski

Ze względu na bardzo wolny czas wykonywania algorytmu dla standardowych wielkości tablic podanych w tym sprawozdaniu, zdecydowałem się zmniejszyć wielkości tablic dla każdego przypadku oprócz najlepszego 10-krotnie. Tak wolne działanie algorytmu dla dużej ilości liczb oddalonych od siebie jest zgodne z literaturą

Pomimo, że złożoność obliczeniowa dla podanych przypadków średnich i najgorszego są wszystkie równe $O(n^2)$, to stopień posortowania tablicy ma widoczny wpływ na szybkość działania algorytmu. Dla sortowania poprzez wstawianie najgorszym przypadkiem jest tablica posortowana odwrotnie niż sortuje algorytm, czyli tutaj posortowana malejąco. Jak widzimy z wykresu, czas wykonania jest wtedy najdłuższy, by potem poprawiać się w ramach stopnia poprawnego, rosnącego posortowania tablicy. Jest tutaj prosta zależność, im bardziej posortowana jest

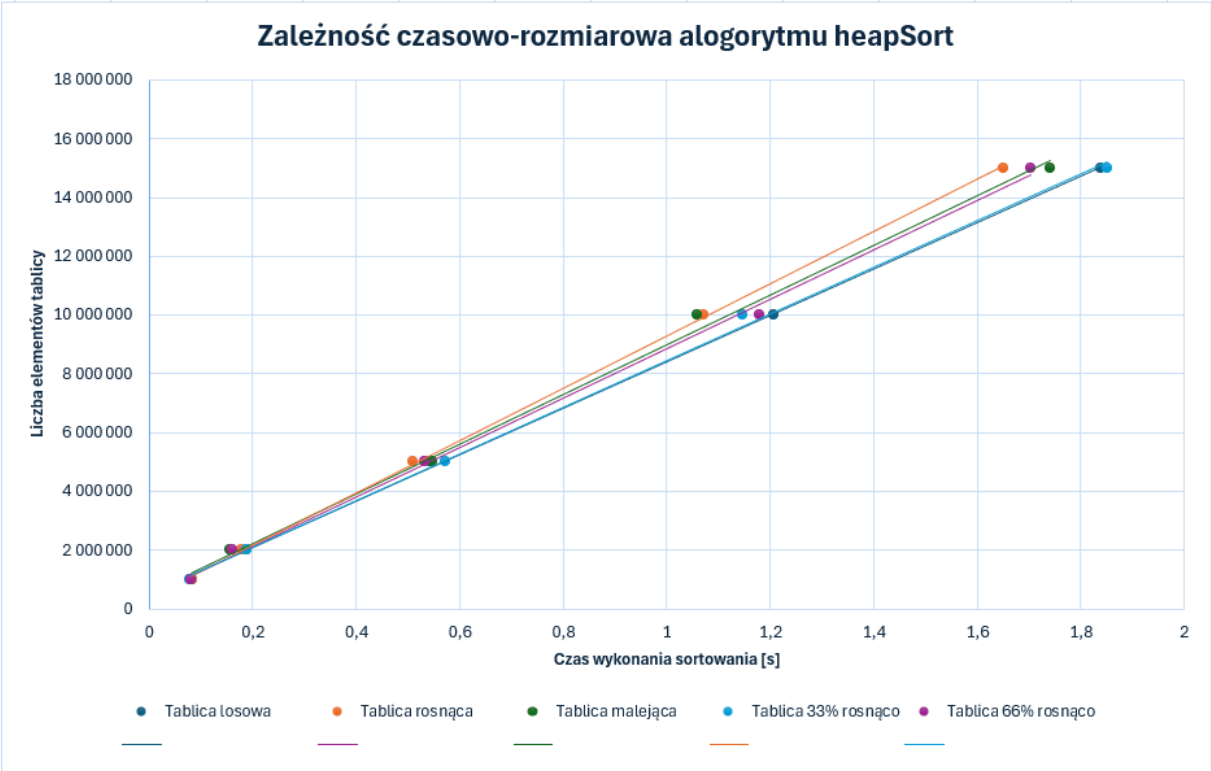
tablica, tym szybciej działa sortowanie. Dla stopnia posortowania bliskiego 66% algorytm jest tak samo wydajny czasowo co dla losowej tablicy. Dzieję się tak, ponieważ pomimo początkowego „zysku” jakim jest część posortowana, wstawianie potem na właściwe miejsca elementów jest tak kosztowne i długotrwałe, że przeważa ten zysk i następuje średnie zrównanie czasowe do sortowania losowej tablicy

Przypadek najlepszy tzn. tablica już posortowana wywołuje tak szybkie przejście algorytmu, że zdecydowałem się nie nanosić pomiarów na wykres żeby zachować jego czytelność

Sortowanie poprzez kopcowanie

Sortowanie przez kopcowanie		
Stan tablicy	Czas sortowania [s]	Rozmiar
losowa	0,0845178	1 000 000
losowa	0,189071	2 000 000
losowa	0,542306	5 000 000
losowa	1,20745	10 000 000
losowa	1,83983	15 000 000
posortowana rosnąco	0,082626	1 000 000
posortowana rosnąco	0,179483	2 000 000
posortowana rosnąco	0,510801	5 000 000
posortowana rosnąco	1,071590	10 000 000
posortowana rosnąco	1,65062	15 000 000
posortowana malejąco	0.0805713	1 000 000
posortowana malejąco	0,155393	2 000 000
posortowana malejąco	0,547610	5 000 000
posortowana malejąco	1,059850	10 000 000
posortowana malejąco	1,74013	15 000 000
0.33% posortowane rosnąco	0.0785168	1 000 000
0.33% posortowane rosnąco	0,189203	2 000 000
0.33% posortowane rosnąco	0,571630	5 000 000
0.33% posortowane rosnąco	1,146590	10 000 000
0.33% posortowane rosnąco	1,85092	15 000 000
0.66% posortowane rosnąco	0.0816702	1 000 000
0.66% posortowane rosnąco	0,160203	2 000 000
0.66% posortowane rosnąco	0,531890	5 000 000
0.66% posortowane rosnąco	1,179049	10 000 000
0.66% posortowane rosnąco	1,70376	15 000 000

Wyniki pomiarów przeprowadzonych na algorytmie heapSort



Wyniki pomiarów przeprowadzonych na algorytmie heapSort w postaci wykresu

Wnioski

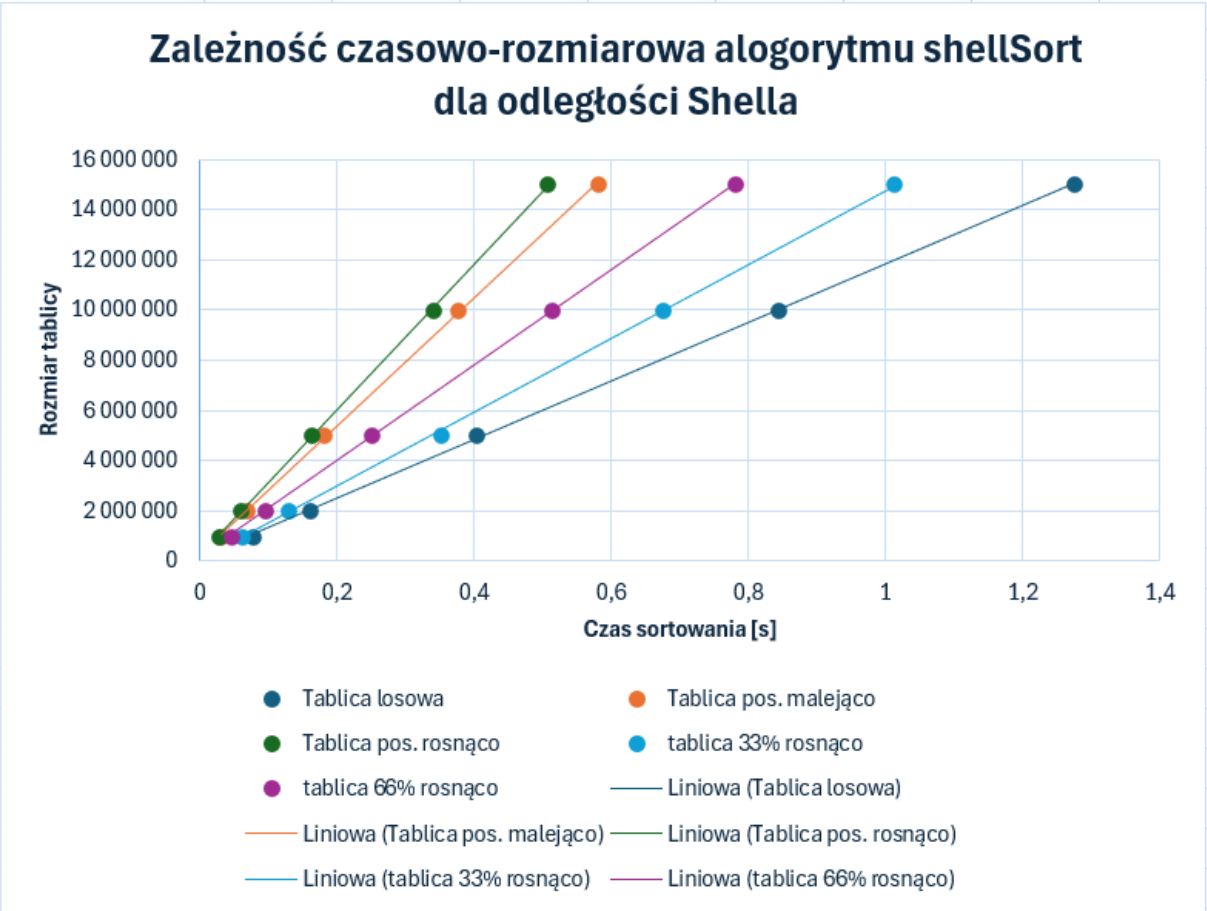
Dla powyższych pomiarów można zauważyć kilka ciekawych rzeczy. Po pierwsze, to jak wygląda tablica przed sortowaniem, nie ma znaczenia dla szybkości realizacji algorytmu. Są co prawda drobne różnice, ale można je potraktować jako błędy pomiarowe, lub statystyczne. Zgadza się to z logiką oraz literaturą, ponieważ przed podjęciem pierwszego kroku sortowania, „niszczymy” oryginalną strukturę danej tablicy tworząc w to miejsce kopiec, więc już na wstępnym etapie, porządkujemy swoje dane, ujednolicając czas wykonania sortowania dla każdego przypadku.

Pomimo bardzo bliskiej aproksymacji funkcją liniową, zmierzona złożoność obliczeniowa jest spodziewaną. Dzieje się tak dlatego, że złożoność $O(n\ log\ n)$ jest bardzo zbliżona liniowej i dla tak małej ilości pomiarów nie jesteśmy w stanie odróżnić jej od liniowej

Sortowanie Shella

Sortowanie Shella (odległość Shella)		
Stan tablicy	Czas sortowania	Rozmiar
losowa	0,0766572	1 000 000
losowa	0,16052	2 000 000
losowa	0,402193	5 000 000
losowa	0,843276	10 000 000
losowa	1,27465	15 000 000
posortowana rosnąco	0,0280267	1 000 000
posortowana rosnąco	0,0589838	2 000 000
posortowana rosnąco	0,163388	5 000 000
posortowana rosnąco	0,340091	10 000 000
posortowana rosnąco	0,50713	15 000 000
posortowana malejąco	0,0311309	1 000 000
posortowana malejąco	0,0676138	2 000 000
posortowana malejąco	0,18128	5 000 000
posortowana malejąco	0,375517	10 000 000
posortowana malejąco	0,58131	15 000 000
0.33% posortowane rosnąco	0,0605451	1 000 000
0.33% posortowane rosnąco	0,128235	2 000 000
0.33% posortowane rosnąco	0,350982	5 000 000
0.33% posortowane rosnąco	0,675089	10 000 000
0.33% posortowane rosnąco	1,01217	15 000 000
0.66% posortowane rosnąco	0,0454683	1 000 000
0.66% posortowane rosnąco	0,0958381	2 000 000
0.66% posortowane rosnąco	0,249773	5 000 000
0.66% posortowane rosnąco	0,512892	10 000 000
0.66% posortowane rosnąco	0,780594	15 000 000

Wyniki pomiarów przeprowadzonych na algorytmie shellSort



Wyniki pomiarów przeprowadzonych na algorytmie shellSort w postaci wykresu

Wnioski

Sortowanie Shella nawet dla oryginalnie zaproponowanego obliczania odległości wykazuje się bardzo szybkim działaniem. Ogranicza on wadę sortowania przez wstawianie jakim jest długie wykonanie dla dużych ilości danych poprzez podział tablicy. Ze względu na swoją średnią złożoność $O\left(n^{\frac{3}{2}}\right)$ można aproksymować złożoność funkcją liniową o ostrym charakterze (duży współczynnik kierunkowy)

Jak widać na wykresie, dla tego algorytmu przypadkiem najgorszym nie jest tablica posortowana odwrotnie. Przypadkiem najgorszym w tym przypadku byłaby specjalnie spreparowana tablica, która jest posortowana odwrotnie względem zmniejszającej się odległości. Ze względu na ograniczony czas nie zdążyłem przygotować takich danych

Algorytm jest tym szybszy w jakim stopniu tablica jest już posortowana. Wynika z faktu mniejszej ilości wstawień i porównań, które musi zrobić znajdujący się w nim algorytm sortowania przez wstawianie

Sortowanie Shella (odległość Lazarusa)		
Stan tablicy	Czas sortowania	Rozmiar
losowa	0,0743231	1 000 000
losowa	0,152463	2 000 000
losowa	0,398835	5 000 000
losowa	0,818557	10 000 000
losowa	1,22955	15 000 000
posortowana rosnąco	0,0297713	1 000 000
posortowana rosnąco	0,0643145	2 000 000
posortowana rosnąco	0,173738	5 000 000
posortowana rosnąco	0,362668	10 000 000
posortowana rosnąco	0,537868	15 000 000
posortowana malejąco	0,0333991	1 000 000
posortowana malejąco	0,0711949	2 000 000
posortowana malejąco	0,191723	5 000 000
posortowana malejąco	0,400567	10 000 000
posortowana malejąco	0,600762	15 000 000
0.33% posortowane rosnąco	0,060222	1 000 000
0.33% posortowane rosnąco	0,125726	2 000 000
0.33% posortowane rosnąco	0,330205	5 000 000
0.33% posortowane rosnąco	0,672045	10 000 000
0.33% posortowane rosnąco	1,01894	15 000 000
0.66% posortowane rosnąco	0,0468534	1 000 000
0.66% posortowane rosnąco	0,0987284	2 000 000
0.66% posortowane rosnąco	0,261549	5 000 000
0.66% posortowane rosnąco	0,536516	10 000 000
0.66% posortowane rosnąco	0,798183	15 000 000

Wyniki pomiarów przeprowadzonych na algorytmie shellSort z odległością Lazariusa

Wyniki pomiarów przeprowadzonych na algorytmie shellSort w postaci wykresu

Wnioski

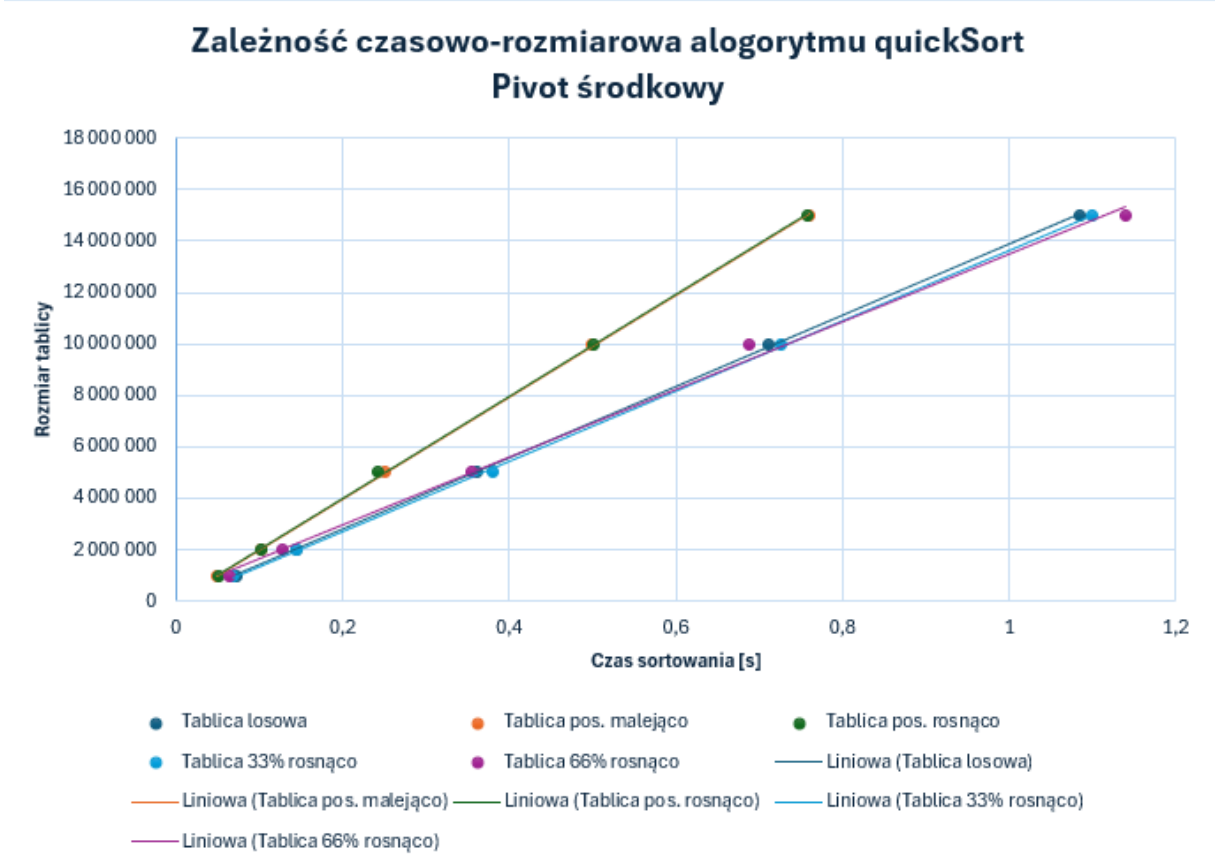
Sortowanie Shella dla zmniejszania odległości opracowanej przez Franka Lazarusa jest szybsze od oryginalnie zaproponowanego. Widzimy tutaj głównie przyspieszenie w obliczaniu losowej tablicy, dla reszty przypadków polepszenie lub pogorszenie czasu jest w zakresie błędu pomiarowego

Sortowanie szybkie

Ze względu na zdecydowanie różne zachowywanie się algorytmu sortowania szybkiego ze względu na strategię umieszczania pivota, zdecydowałem się przeprowadzić 4 pełne analizy wydajności dla 4 strategii: umieszczanie pivota w skrajnie lewym miejscu, skrajnie prawym, środkowym oraz za każdym razem losując jego miejsce

Sortowanie szybkie - pivot środkowy		
Stan tablicy	Czas sortowania	Rozmiar
losowa	0,072162	1 000 000
losowa	0,144618	2 000 000
losowa	0,361072	5 000 000
losowa	0,711608	10 000 000
losowa	1,0835	15 000 000
posortowana rosnąco	0,0513222	1 000 000
posortowana rosnąco	0,102267	2 000 000
posortowana rosnąco	0,24204	5 000 000
posortowana rosnąco	0,501616	10 000 000
posortowana rosnąco	0,758086	15 000 000
posortowana malejąco	0,0499818	1 000 000
posortowana malejąco	0,10294	2 000 000
posortowana malejąco	0,250542	5 000 000
posortowana malejąco	0,499696	10 000 000
posortowana malejąco	0,759347	15 000 000
0.33% posortowane rosnąco	0,0679193	1 000 000
0.33% posortowane rosnąco	0,145515	2 000 000
0.33% posortowane rosnąco	0,379973	5 000 000
0.33% posortowane rosnąco	0,726897	10 000 000
0.33% posortowane rosnąco	1,09856	15 000 000
0.66% posortowane rosnąco	0,064339	1 000 000
0.66% posortowane rosnąco	0,127927	2 000 000
0.66% posortowane rosnąco	0,353489	5 000 000
0.66% posortowane rosnąco	0,68795	10 000 000
0.66% posortowane rosnąco	1,13999	15 000 000

Wyniki pomiarów przeprowadzonych na algorytmie quickSort z środkowym pivotem



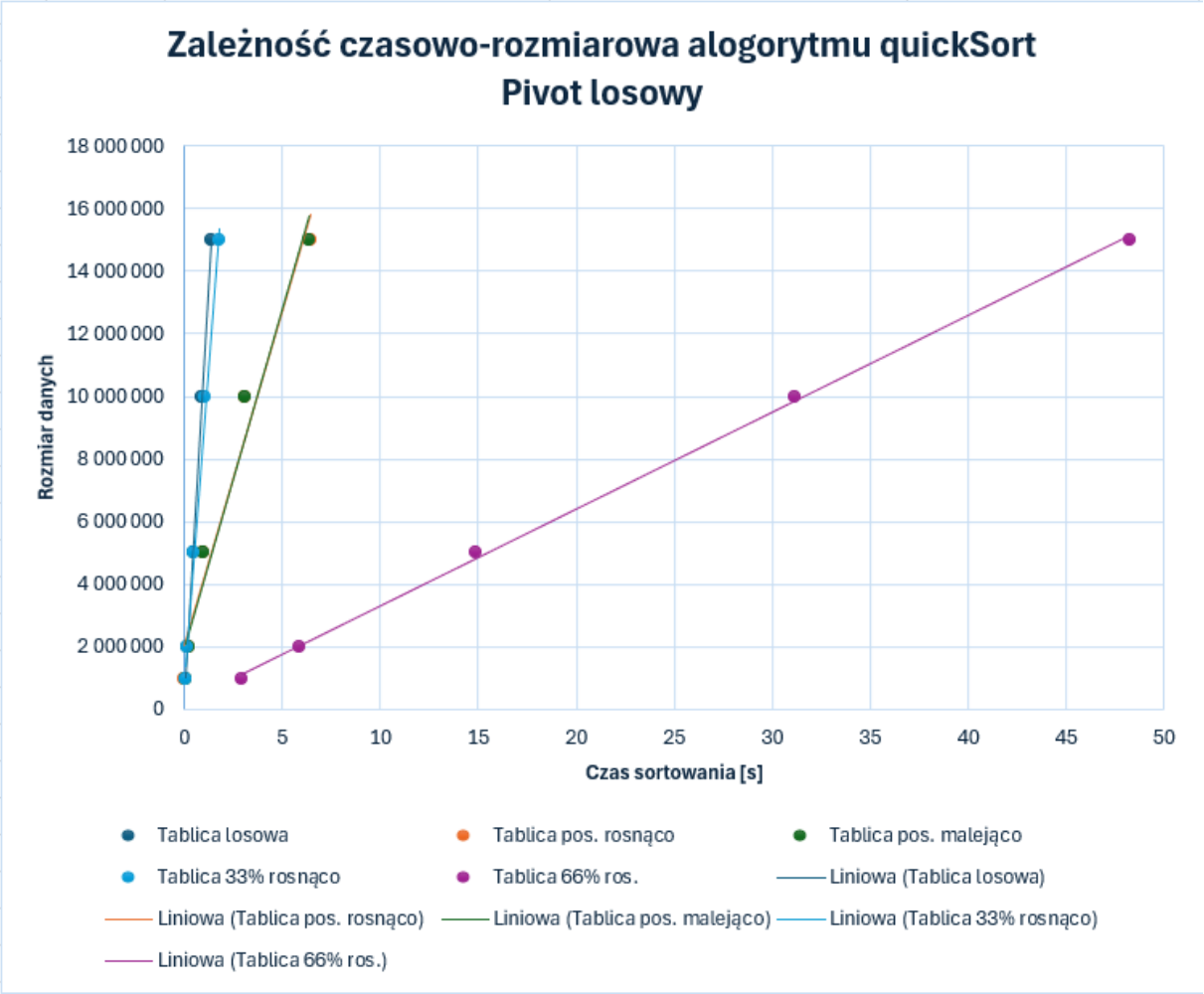
Wyniki pomiarów przeprowadzonych na algorytmie quickSort z środkowym pivotem w postaci wykresu

Wnioski

Dla pivota środkowego quickSort zachowuje się bardzo stabilnie, niezależnie od uporządkowania danych. Widoczne są spadki wydajności jeśli będziemy sortowali częściowo posortowaną lub w całości sortowaną tablicę względem algorytmu, lecz są to relatywnie małe odstępstwa od przypadków losowych i całościowo posortowanych

Sortowanie szybkie - pivot losowy		
Stan tablicy	Czas sortowania	Rozmiar
losowa	0,09081	1 000 000
losowa	0,17881	2 000 000
losowa	0,44555	5 000 000
losowa	0,89482	10 000 000
losowa	1,34050	15 000 000
posortowana rosnąco	0,0123509	1 000 000
posortowana rosnąco	0,216052	2 000 000
posortowana rosnąco	0,954269	5 000 000
posortowana rosnąco	3,07874	10 000 000
posortowana rosnąco	6,44891	15 000 000
posortowana malejąco	0,0878069	1 000 000
posortowana malejąco	0,220661	2 000 000
posortowana malejąco	0,961992	5 000 000
posortowana malejąco	3,11646	10 000 000
posortowana malejąco	6,38633	15 000 000
0.33% posortowane rosnąco	0,0853835	1 000 000
0.33% posortowane rosnąco	0,177936	2 000 000
0.33% posortowane rosnąco	0,478984	5 000 000
0.33% posortowane rosnąco	1,06232	10 000 000
0.33% posortowane rosnąco	1,77188	15 000 000
0.66% posortowane rosnąco	2,93372	1 000 000
0.66% posortowane rosnąco	5,86895	2 000 000
0.66% posortowane rosnąco	14,9073	5 000 000
0.66% posortowane rosnąco	31,1493	10 000 000
0.66% posortowane rosnąco	48,2922	15 000 000

Wyniki pomiarów przeprowadzonych na algorytmie quickSort z losowym pivotem



Wyniki pomiarów przeprowadzonych na algorytmie quickSort z losowym pivotem w postaci wykresu

Wnioski

Dla pivota losowego za każdym partycjonowaniem zauważamy interesującą rzecz. Im bardziej posortowana jest tabela, wyłączając przypadek całkowicie posortowany, tym zauważamy znaczące wydłużenie działania. Dla tablicy posortowanej w 66% straty czasu działania algorytmu są ogromne, a problem pogłębia się w jeszcze większym tępie, dla części posortowanej większej od 66%

Sortowanie szybkie - pivot skrajnie lewy		
Stan tablicy	Czas sortowania	Rozmiar
losowa	0,0729987	1 000 000
losowa	0,145554	2 000 000
losowa	0,36327	5 000 000
losowa	0,714903	10 000 000
losowa	1,10064	15 000 000
posortowana rosnąco	118,799	1 000 000
posortowana rosnąco	491077	2 000 000
posortowana rosnąco	3681,7	5 000 000
posortowana rosnąco	?	10 000 000
posortowana rosnąco	?	15 000 000
posortowana malejąco	121,88	1 000 000
posortowana malejąco	501,793	2 000 000
posortowana malejąco	3735,43	5 000 000
posortowana malejąco	?	10 000 000
posortowana malejąco	?	15 000 000
0.33% posortowane rosnąco	0,00076675	10000
0.33% posortowane rosnąco	0,00590389	20000
0.33% posortowane rosnąco	0,032787	50000
0.33% posortowane rosnąco	0,124836	100000
0.33% posortowane rosnąco	0,281325	150000
0.66% posortowane rosnąco	0,00224274	10000
0.66% posortowane rosnąco	0,0067137	20000
0.66% posortowane rosnąco	0,0633022	50000
0.66% posortowane rosnąco	0,252271	100000
0.66% posortowane rosnąco	0,564691	150000

Wyniki pomiarów przeprowadzonych na algorytmie quickSort z skrajnie lewym pivotem

Wyniki pomiarów przeprowadzonych na algorytmie quickSort z skrajnie lewym pivotem w postaci wykresu

Wnioski

Dla tej strategii ustawiania pivota napotkałem dużo trudności. Przypadek tablicy losowej jest złożonościowo i czasowo bardzo zbliżony do pivota losowego oraz środkowego, natomiast przy napotkaniu tablicy w jakimkolwiek stopni posortowanym i to nawet w tą samą stronę w jaką algorytm porządkuje liczby, stajemy przed złożonością potęgową $O(n^2)$, czyli najgorszymi przypadkami dla tego algorytmu. Dzieje się tak, ponieważ ze względu na sortowanie, niemal wszystkie elementy trafiają do jednej partycji

Jeśli weźmiemy tablicę w całości posortowaną w dowolną stronę, algorytm zajmuje katastrofalnie dużo czasu. Ilość ta jest tak duża, że zmuszony byłem nie brać średniej z pomiarów oraz zaprzestać prób dla dwóch największych testowanych rozmiarów, ponieważ szacunkowe czasy wykonania byłyby liczone w dniach, a nie sekundach pracy

W przypadku tablic częściowo posortowanych byłem zmuszony drastycznie obniżyć liczbę elementów w testowanych tablic z powodu błędów przepełnień stosu. Ponieważ w swojej klasycznej implementacji, algorytm sortowania szybkiego jest rekurencyjny, możliwa jest tylko skończona liczba wywołań funkcji do momentu alokacji jej za dużej części. Sposobem na łagodzenie tego problemu jest zmiana implementacji rekurencyjnej na iteracyjną lub włączenie do algorytmu korzystania z sortowania poprzez wstawianie dla małych podtablic. Nie skorygowałem mojego kodu tymi rozwiązaniami, ponieważ uznałem napotkane błędy wydajnościowe za pouczające do umieszczenia i omówienia w sprawozdaniu, a także uznałem, że zbytnio ingerowałyby w standardowy algorytm quickSorta prezentowany w większości materiałów zapoznawczych z nim

Podsumowując, ze względu na swoją wysoką niewydajność biorąc pod uwagę czas wykonania algorytmu oraz w niektórych przypadkach możliwość przepełnienia stosu, wybieranie pivota skrajnie lewego nie jest dobrą strategią

Sortowanie szybkie - pivot skrajnie prawy		
Stan tablicy	Czas sortowania	Rozmiar
losowa	0,071005	1 000 000
losowa	0,142509	2 000 000
losowa	0,354665	5 000 000
losowa	0,710157	10 000 000
losowa	1,07464	15 000 000
posortowana rosnąco	0,00016092	1000
posortowana rosnąco	0,00055405	2000
posortowana rosnąco	0,0030994	5000
posortowana rosnąco	0,0115365	10000
posortowana rosnąco	0,0271177	15000
posortowana malejąco	0,00014778	1000000
posortowana malejąco	0,00053865	2000000
posortowana malejąco	0,003172	5000000
posortowana malejąco	0,0125769	100000000
posortowana malejąco	0,0252367	150000000
0.33% posortowane rosnąco	0,00074294	10000
0.33% posortowane rosnąco	0,00152953	20000
0.33% posortowane rosnąco	0,0149197	50000
0.33% posortowane rosnąco	0,00774097	100000
0.33% posortowane rosnąco	0,0412847	150000
0.66% posortowane rosnąco	~0	1000
0.66% posortowane rosnąco	0,00015607	2000
0.66% posortowane rosnąco	0,00058046	5000
0.66% posortowane rosnąco	0,00149913	10000
0.66% posortowane rosnąco	0,0034768	15000

Wyniki pomiarów przeprowadzonych na algorytmie quickSort z skrajnie prawym pivotem

Wnioski

Umieszczanie pivota jako skrajnie prawy element ma bardzo podobne wady jak skrajnie lewego elementu. Dla tablicy losowej zachowuje się tak jak inne warianty quickSorta, natomiast każdy inny przypadek ma złożoność $O(n^2)$ i powoduje czasami jeszcze większe problemy z przepełnieniem stosu, niż poprzednia strategia. Nie należy stosować w programach

Końcowe wnioski na temat sortowania szybkiego

Sposób umieszczania pivota w podtablicach wpływa znacząco na czas działania oraz na interpretację, który przypadek umiejscowienia elementów tablicy jest tym najgorszym, z potęgową złożonością obliczeniową. Z sprawdzonych strategii najlepszą okazuje się umiejscawianie pivota w środku tablicy. Zmniejsza to drastycznie liczbę partycji jaką musimy wykonywać

Chociaż był to najlepszy sprawdzony przypadek z kilku, to istnieje jeszcze jedna strategia warta wspomnienia, mediana z lewego, prawego oraz środkowego indeksu. Podobnie jak wybieranie środkowego elementu zmniejsza to liczbę partycji

Wpływ typu danych na czas sortowania

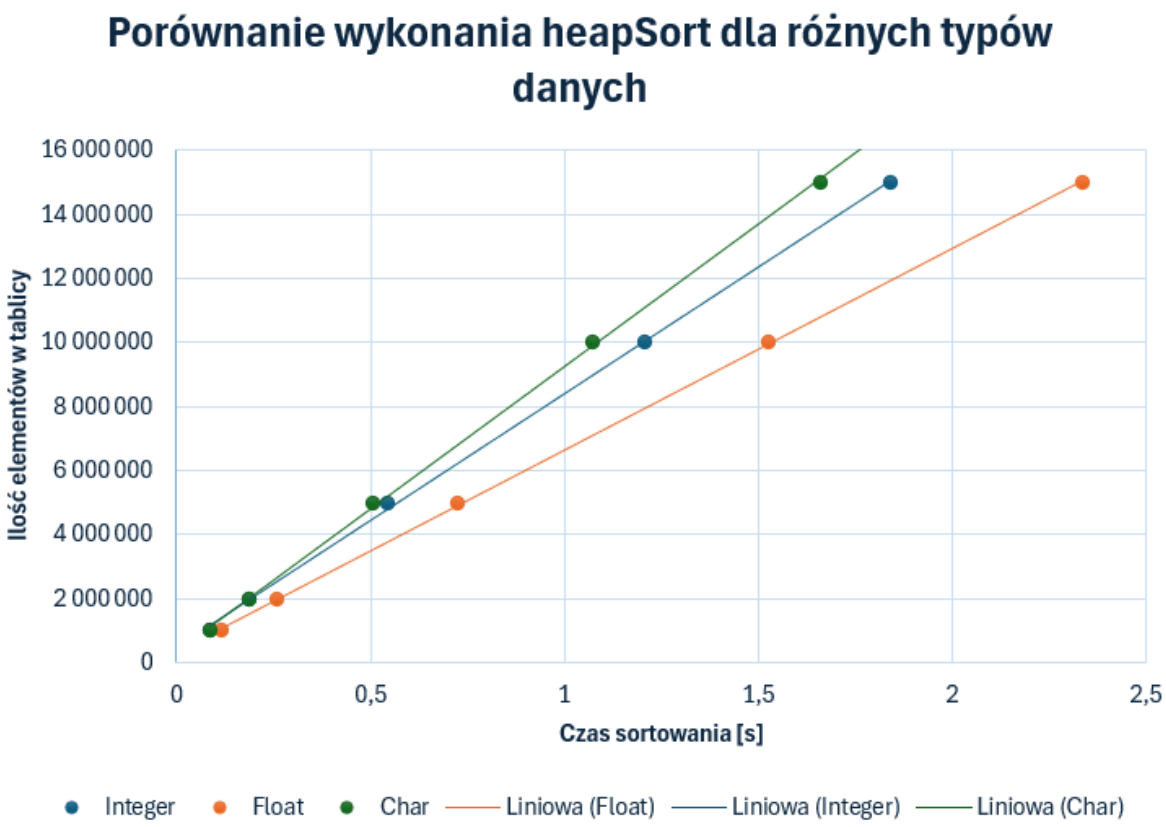
Do realizacji tej części eksperymentu zdecydowałem się wybrać algorytm sortowania przez kopcowanie, ponieważ jego „stabilność” pod względem ułożenia danych efektywnie zrówna wszystkie przypadki podania tablicy do tylko jednego. Dzięki temu wystarczy wykonać testy tylko na jednym przypadku podania danych

Sortowanie danych typu char		
Stan tablicy	Czas sortowania [s]	Rozmiar
losowa	0.0880891	1 000 000
losowa	0,18784	2 000 000
losowa	0,50718	5 000 000
losowa	1,07212	10 000 000
losowa	1,65983	15 000 000

Sortowanie danych typu integer		
Stan tablicy	Czas sortowania [s]	Rozmiar
losowa	0,0845178	1 000 000
losowa	0,189071	2 000 000
losowa	0,542306	5 000 000
losowa	1,20745	10 000 000
losowa	1,83983	15 000 000

Sortowanie danych typu float		
Stan tablicy	Czas sortowania [s]	Rozmiar
losowa	0,114827	1 000 000
losowa	0,257567	2 000 000
losowa	0,724681	5 000 000
losowa	1,52704	10 000 000
losowa	2,33431	15 000 000

Porównanie pomiarów przeprowadzonych na algorytmie heapSort z różnymi typami danych



Porównanie pomiarów przeprowadzonych na algorytmie heapSort z różnymi typami danych w postaci wykresu

Wnioski

Typ danych ma wpływ na czas wykonania sortowania, jednak nie zmienia jego złożoności obliczeniowej, przynajmniej dla podstawowych typów danych int, float oraz char. Sortowanie przez kopcowanie pomimo różnych typów danych ma stale złożoność $O(n\ log\ n)$

Wykonanie jest najszybsze dla typu char, potem int a na końcu float. Wynika to z ilości miejsca zajmowanego przez dane o tych typach (odpowiednio 4 bajty, 1 bajt oraz 4 bajty) jednak nie jest to zależność liniowa. Zmienienie typu danych z int na char nie powoduje przyspieszenia rzędu 2 razy

To, ile miejsca zajmuje dany typ w pamięci to też nie wszystko. Według standardu C++ float oraz int zajmują tyle samo miejsca w pamięci, jednak widzimy między nimi znaczące różnice czasowe. Dzieje się tak ze względu na sposób przechowywania reprezentacji liczb zmiennoprzecinkowych w komputerach. Standard float IEEE 754

wymusza w każdej operacji arytmetycznej oraz logicznej więcej czasu na ich wykonanie niż w zwykłych intach. Widać zatem, że liczy się zarówno zajętość pamięciowa typu danych oraz to, jak efektywnie są na nim przeprowadzane operacje

Wnioski finalne

Analiza podanych algorytmów wykazała znaczące różnice pomiędzy nimi. Na różnych przykładach widać, że pomimo czasami takiej samej złożoności obliczeniowej konkretnych przypadków, czasy ich wykonania dla różnych sortowań mogą mieć znaczące różnice.

Każdy z algorytmów opisanych wyżej ma swoje wady oraz zalety i z tego powodu zawsze musimy przemyśleć jakie przypadki tablic w podanym problemie spotkamy najczęściej i dopiero według tego dobrać odpowiednią metodę.

Porównując czas wykonania algorytmów dla różnych typów danych wykazano, że istotne jest nie tylko samo ułożenie elementów, ale też sposób ich przechowywania i obliczania w programie. O ile typ danych nie wpływa na samą charakterystykę złożoności obliczeniowej, to może wprowadzić istotne różnice czasowe w wykonaniu sortowania

Algorytmy nawet o tej samej złożoności obliczeniowej mogą mieć spore różnice w czasie wykonania (patrz HeapSort i ShellSort). Wynika to z złożoności czasowej algorytmów. Te zaimplementowane rekurencyjnie zużywają więcej pamięci w czasie trwania oraz zajmują więcej czasu do przeprowadzenia kroków. Z tego względu ważna jest analiza czasowa oraz w przypadku użycia rekurencji, upewnienie się czy dla pracy na szacowanych przez nas ilościach danych nie nastąpi przepełnienie stosu

Literatura

Cormen T., Leiserson C.E., Rivest R.L., Stein C., Wprowadzenie do algorytmów, WNT

Banachowski L., Diks K., Rytter W., Algorytmy i struktury danych, PWN