Analiza wybranych algorytmów sortowania

Samuel Nowak, Bioinformatyka rok I, nr indeksu 145165
Michał Stanoch, Bioinformatyka rok I, nr indeksu 145168

Cele zadania i specyfikacja komputera pomiarowego

Celami zadania były porównanie i analiza szybkości sortowania danych przez cztery rodzaje algorytmów sortujących: sortowanie przez wybieranie (SelectionSort), sortowanie przez wstawianie (InsertionSort), sortowanie szybkie (QuickSort) oraz sortowanie przez kopcowanie (HeapSort) w czterech przypadkach wstępnego ułożenia danych w tablicach: kolejności losowej (Random), wzrastającej (Increasing), malejącej (Decreasing) oraz v – kształtnej (V-Shape). W tym celu należało stworzyć metody i/lub funkcje generujące tablice o danym wstępnym ułożeniu zmiennych a następnie stworzyć metody i/lub funkcje sortujące. W celu łatwej konwersji danych do programów Excel i Word dodano również kod zapisujący wyniki do zewnętrznego pliku data.dat – plik ten generowany jest za każdym razem od nowa i zawiera kompletny zapis widoczny w konsoli podczas działania programu. Przy wykonywaniu zadania wykorzystano standardowy kompilatora środowiska Code::Blocks, a specyfikacja komputera pomiarowego jest podana poniżej:

OS:	Windows 10 64-bit
Procesor:	Intel® Core™ i7 - 4790k @ 4.00GHz (8 CPUs), ~4.0GHz
Pamięć RAM:	12 288 MB

Omówienie doboru liczności elementów w tablicach pomiarowych

Wielkości tablic dobrano tak, aby łatwo można było zaobserwować zmiany długości wykonywania sortowania: większość wielkości tablic to wielokrotności n początkowego: n_0 = 1500 elementów. Stosunek n/n_0 w przypadku n_1 = 3000 wynosi więc 2/1 = 2, n_2 = 7500 wynosi 5/1 = 5, przez n_4 = 15000, gdzie stosunek wynosi 10/1 = 10, aż do n_9 = 30000, ze stosunkiem $n9/n0$ wynoszącym 20/1 = 20. Dodatkowo zastosowano inne wielkości tablic, np. n_7 = 17501 w celu pokazania prawidłowego działania kodu generującego zmienne w tablicach we wszystkich kategoriach wstępnego ułożenia zmiennych w przypadku wykorzystania dużej liczby nieparzystej (wielokrotności naturalne n_0 byłyby liczbami parzystymi, co miało szczególne znaczenie przy metodzie generującej rozłożenie v – kształtne, jako że metoda ta generuje zmienne w dwóch kierunkach: od początku i od końca tablicy oraz przy ewaluacji wygenerowanych tą metodą tablic). Ze względu na fakt, iż wielkość kroku między kolejnymi zmiennymi nie miała znaczenia (do momentu przekroczenia maksymalnej możliwej wartości rodzaju zmiennej *int* = +/- 2,147,483,647), tablice wypełniono wartościami większymi/ mniejszymi o +/- 1 (poza metodą fill_random, która generuje liczby losowe). Maksymalną wartość n_9 = 30 000 wybrano ze względu na błędy oraz terminacje programu pomiarowego przy ilościach elementów w tablicach większych niż ok. 42 500 w przypadku algorytmu QuickSort. Terminacje te spowodowane są zapewne rekurencyjną budową algorytmu w wykorzystanym do pomiaru sposobie podziału danych przez algorytm: przez ostatni element oraz spowodowaną taką budową dużą

złożonością pamięciową algorytmu. Dodatkowym czynnikiem wpływającym negatywnie mogła być wstępnie przygotowana architektura programu: QuickSort standardowo oczekuje trzech argumentów: tablicy do posortowania oraz indeksów wyższego i niższego, jednak architektura przygotowanego wcześniej programu umożliwiała podanie najpierw tylko dwóch argumentów do funkcji. W celu dodania trzeciego argumentu należało więc wywołać właściwą funkcję (w naszym przypadku: quick_sort_2), która z definicji oczekiwała trzech argumentów, co najprawdopodobniej wyolbrzymiło złe wyniki algorytmu w najgorszym dla niego przypadku Increasing i reszcie poza Random. Przy pomiarach funkcja quick_sort_2 podział tworzyła w oparciu o ostatni element, co również negatywnie wpłynęło na efektywność algorytmu QuickSort.

Omówienie wyników zadania

Wyniki zapisane z dokładnością do $1*10^{-4}$ sekundy zawarto w Tabeli 1., zaś Wykres 1. obrazuje uzyskane pomiary czasu sortowania danych w określonych kategoriach wstępnego ułożenia danych w tablicach. Ze względu na ogromną rozbieżność rzędów wielkości uzyskanych pomiarów (najwolniejszy algorytm w swym najgorszym przypadku: QuickSort w kategorii ułożenia danych Increasing z 30 000 elementów w tablicy wykonał sortowanie w ciągu 1.8580 s., zaś najszybsze algorytmy, jak choćby InsertionSort również w przypadku Increasing dokonywały sortowania w czasie krótszym niż $1*10^{-4}$ sekundy; rzędy wielkości w tym przypadku różniły się więc o ponad 10^4), niemożliwe było czytelniejsze zaprezentowanie danych na Wykresie 1.

Algorytm SelectionSort o złożoności czasowej $O(n^2)$ i pamięciowej O(1): Czas sortowania rósł zgodnie z założeniem - kwadratowo. Widać to dokładnie przy stosunku *n4*/*n0* - elementów jest 10 razy więcej, a czas sortowania rośnie 100-krotnie. Obyło się także bez większych odchyleń w zależności od startowego ułożenia wartości w tabeli.

Algorytm InsertionSort o średniej złożoności czasowej $O(n^2)$ i (w najgorszym przypadku) pamięciowej O(n): Czas sortowania rósł - mimo drobnych odchyleń- zgodnie z założeniem, czyli kwadratowo, co widać przy stosunku *n4*/*n0* - elementów jest 10 razy więcej, a czas sortowania rośnie 100-krotnie. InsertionSort, mimo że teoretycznie ma taką samą złożoność co SelectionSort, poradził sobie lepiej w niektórych przypadkach:

1. Dla wypełnienia Random i V-shape InsertionSort wypadł dużo lepiej przy największych wartościach - dla ostatniej badanej próby osiągnął czas niemal dwukrotnie lepszy.
2. Dla wypełnienia Increasing InsertionSort miał zerowy czas- jest to związanie z budową algorytmu, który w tym wypadku po pierwszym sprawdzeniu danych zakończył pracę.

Algorytm QuickSort o średniej złożoności czasowej O(n log n) i (w najgorszym przypadku) pamięciowej O(log n): Czas sortowania wypadł najlepiej dla wypełnienia losowego - dawał wyniki wielokrotnie lepsze wyniki niż chociażby SelectionSort i InsertionSort. Dla pozostałych wypełnień wyniki nie były już takie dobre: dla wypełnienia V-shape, Increasing i Decreasing czas wypadł gorzej niż w SelectionSort oraz InsertionSort. Zauważalne jest, że przy wypełnieniach Increasing, Decreasing i V-shape czas sortowania rósł kwadratowo (złożoność $O(n^2)$, co było spowodowane wykorzystanym rodzajem elementu rozdzielającego: ostatniego elementu tablicy.

Algorytm HeapSort o średniej złożoności czasowej O(n log n) i pamięciowym (w najgorszym przypadku) O(n): Czas sortowania rósł w przybliżeniu logarytmicznie. Najgorzej algorytm ten poradził

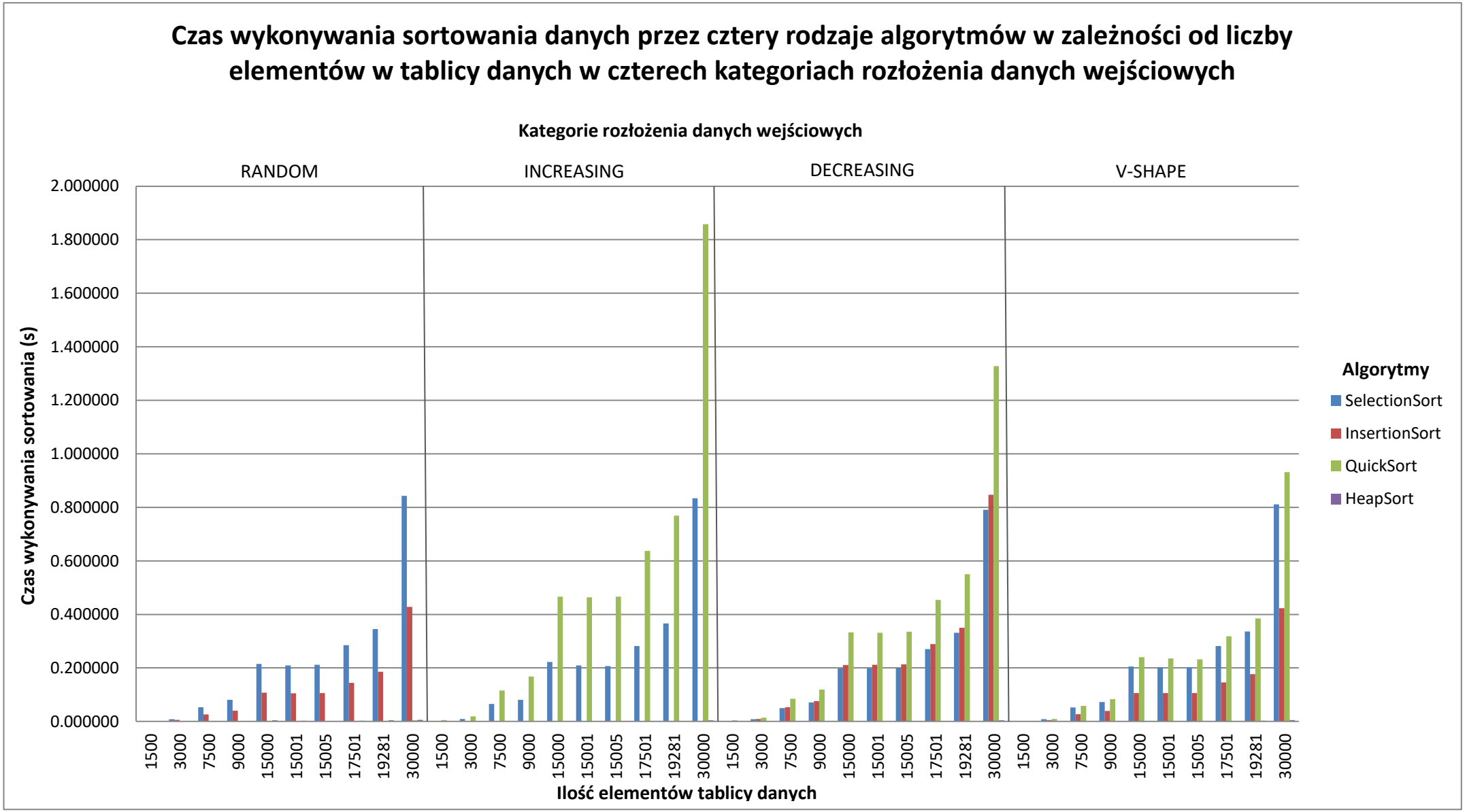
sobie z sortowaniem w kategorii Random - tylko w tym sortowaniu wypadł gorzej niż QuickSort. Można dzięki temu zauważyć że w zasadzie algorytm HeapSort jest w praktyce wolniejszy niż QuickSort, ale wypada lepiej dla “złośliwych danych”.

Wyniki eksperymentu można podsumować następująco: ze wszystkich czterech algorytmów najgorzej wypadł SelectionSort - był najwolniejszy, ale stały: z każdym wypełnieniem radził sobie bardzo podobnie. InsertionSort uplasował się zaraz po nim: jego czas był nieznacznie lepszy, do tego dobrze poradził sobie z wypełnieniem Increasing. Na pierwszym miejscu można uplasować ex aequo QuickSort oraz HeapSort. Mimo że QucikSort wypadł czasowo lepiej dla wartości random, to różnica ta była niewielka, a do tego HeapSort wykazał się dużą odpornością na niewygodne wypełnienia, z czym z kolei QuickSort nie radził sobie dobrze.

Tabela wartości czasowych i wykres (na następnej stronie)

Ilość elementów w tablicy	Rodzaje algorytmów															
	SelectionSort				InsertionSort				QuickSort				HeapSort			
	Random	Increasing	Decreasing	V-Shape	Random	Increasing	Decreasing	V-Shape	Random	Increasing	Decreasing	V-Shape	Random	Increasing	Decreasing	V-Shape
n_0 = 1500	0.0020	0.0020	0.0020	0.0020	0.0020	0.0000	0.0020	0.0010	0.0000	0.0050	0.0040	0.0020	0.0010	0.0000	0.0000	0.0000
n_1 = 3000	0.0080	0.0090	0.0080	0.0080	0.0060	0.0000	0.0090	0.0050	0.0000	0.0190	0.0140	0.0090	0.0010	0.0020	0.0000	0.0000
n_2 = 7500	0.0530	0.0650	0.0500	0.0520	0.0260	0.0000	0.0530	0.0270	0.0010	0.1160	0.0850	0.0580	0.0010	0.0000	0.0010	0.0010
n_3 = 9000	0.0810	0.0810	0.0710	0.0730	0.0400	0.0000	0.0760	0.0390	0.0010	0.1680	0.1190	0.0830	0.0020	0.0020	0.0020	0.0020
n_4 = 15000	0.2150	0.2220	0.1980	0.2050	0.1080	0.0000	0.2110	0.1060	0.0010	0.4660	0.3330	0.2400	0.0040	0.0020	0.0020	0.0020
n_5 = 15001	0.2090	0.2090	0.1980	0.2030	0.1050	0.0000	0.2120	0.1060	0.0010	0.4640	0.3310	0.2350	0.0030	0.0020	0.0020	0.0020
n_6 = 15005	0.2120	0.2070	0.2000	0.2030	0.1060	0.0000	0.2130	0.1060	0.0010	0.4660	0.3350	0.2320	0.0020	0.0020	0.0020	0.0020
n_7 = 17501	0.2850	0.2820	0.2700	0.2820	0.1440	0.0000	0.2890	0.1460	0.0010	0.6370	0.4540	0.3180	0.0030	0.0020	0.0020	0.0020
n_8 = 19281	0.3450	0.3660	0.3310	0.3360	0.1860	0.0000	0.3500	0.1770	0.0020	0.7690	0.5500	0.3850	0.0040	0.0020	0.0020	0.0030
n_9 = 30000	0.8430	0.8340	0.7910	0.8110	0.4280	0.0000	0.8470	0.4230	0.0040	1.8580	1.3280	0.9320	0.0060	0.0040	0.0040	0.0050

Tabela 1. Czas wykonywania sortowania danych (podany w sekundach) przez cztery rodzaje algorytmów w zależności od liczby elementów w tablicy danych w czterech kategoriach rozłożenia danych wejściowych. Wyniki rzędu 0.0000, np. w kolumnie InsertionSort Increasing, oznaczają, że sortowanie przebiegło pomyślnie, lecz z czasem mniejszym niż $1*10^{-4}$ sekundy.



Wykres 1. Czas wykonywania sortowania danych (podany w sekundach) przez cztery rodzaje algorytmów w zależności od liczby elementów w tablicy danych w czterech kategoriach rozłożenia danych wejściowych. Algorytmy sortujące bardzo szybko w danym przypadku ułożenia danych (np. InsertionSort w przypadku Increasing) są ledwie widoczne przy osi X, jako że wykonują swoją pracę w czasie szybszym niż $1*10^{-4}$ s.