|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Rodzaje algorytmów** | | | | | | | | | | | | | | | |
|  |
| **Ilość elementów w tablicy** | **SelectionSort** | | | | **InsertionSort** | | | | **QuickSort** | | | | **HeapSort** | | | |
|
| **Random** | **Increasing** | **Decreasing** | **V-Shape** | **Random** | **Increasing** | **Decreasing** | **V-Shape** | **Random** | **Increasing** | **Decreasing** | **V-Shape** | **Random** | **Increasing** | **Decreasing** | **V-Shape** |
| ***n0* = 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 |
| ***n1* = 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 |
| ***n2* = 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 |
| ***n3*= 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 |
| ***n4* = 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 |
| ***n5* = 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 |
| ***n6* = 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 |
| ***n7* = 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 |
| ***n8* = 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 |
| ***n9* = 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. | | | | | | | | | | | | | | | | |
|

**Ilość elementów tablicy danych**

**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\*10-6 s.

Analiza wybranych algorytmów sortowania

# 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 kompilatoa 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: *n0* = 1500 elementów. Stosunek *n*/*n0* w przypadku *n1* = 3000 wynosi więc 2/1 = 2, *n2* = 7500 wynosi 5/1 = 5, przez *n4* = 15000, gdzie stosunek wynosi 10/1 = 10, aż do *n9* = 30000, ze stosunkiem *n9*/*n0* wynoszącym 20/1 = 20. Dodatkowo zastosowano inne wielkości tablic, np. *n7* = 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 *n0* 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 *n9* = 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.

\*\*\* WSPOMNIEĆ O TYM, ŻE PRZY POMIARZE QUICKSORTA WYKORZYSTANO podział na ostatnim elemencie – on ma najgorszą złożoność i dlatego terminował program. TO, ŻE MUSIELIŚMY WYWOŁAĆ FUNKCJĘ W FUNKCJI ŻEBY QUICK SORT W OGÓLE DZIAŁAŁ (funkcja dawała 2 argumenty, a do buick sorta są potrzebne 3) też pewnie wpłynęło na beznadziejne wyniki QuickSorta w najgorszym jego przypadku (Increasing)\*\*\*

# 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 104), niemożliwe było czytelniejsze zaprezentowanie danych na Wykresie 1.

\*\*\* WYNIKI GENERALNIE dobrze obrazują złożoność obliczeniową algorytmów – trzeba opisać, że oczekiwany wzrost jest taki a taki (np. O(n^2) w SelectionSorcie ładnie widać – najczęściej pewnie będzie się korzystać z n4/n0, bo tam jest stosunek = 10 albo n9/n0, bo tam jest 20. Może wypisać te stosunki, oczekiwane czasu i uzyskane rezultaty?) \*\*\*

\*\*\* OPISAĆ FAKT, IŻ NA PRZYKŁADZIE SELECTION SORT ŁADNIE WIDAĆ CZAS rosnący kwadratowo: n4/n0 = 10, a czas jest 100 razy większy \*\*\*

\*\*\* ZWRÓCIĆ uwagę na ewentualne błędy pomiarowe: SelectionSort maleje między n4 a n5, a nie powinien – to pewnie przez randomowe lagi komputera albo błąd w przybliżaniu czasu przez algorytm: to są bardzo małe wartości \*\*\*

\*\*\* MOŻE ZROBIĆ TIER listę tych algorytmów? Najlepszy jest zdecydowanie HeapSort, potem pewnie InsertionSort, a le potem to nie wiem – liczenie tego chyba nie jest potrzebne \*\*\*