Projekt Sortowanie

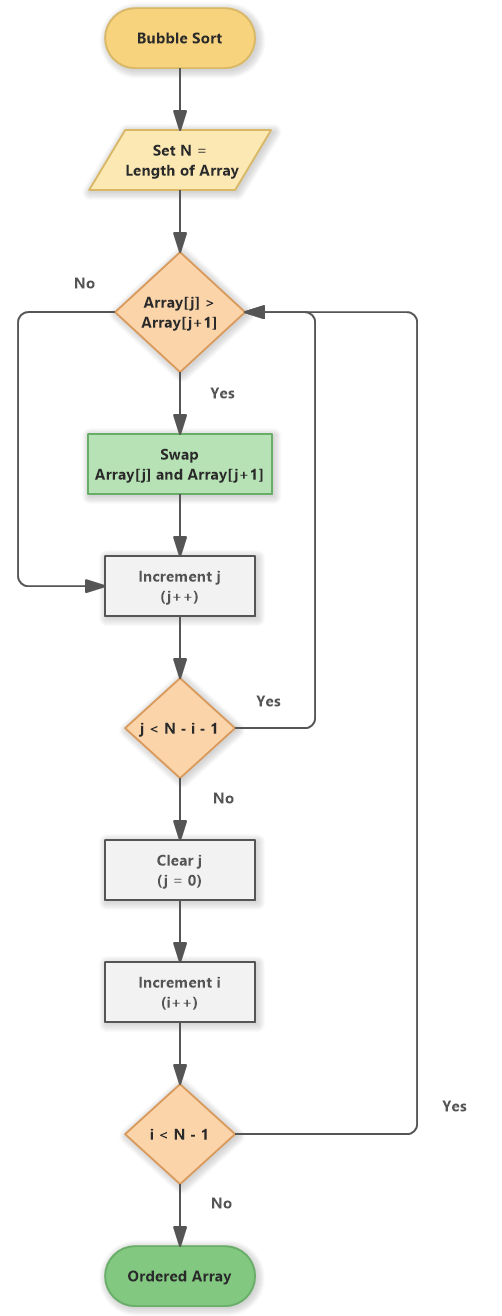
1. Badane sortowania i ich wyjaśnienia

GRUPA 1

1. **Sortowanie bąbelkowe**

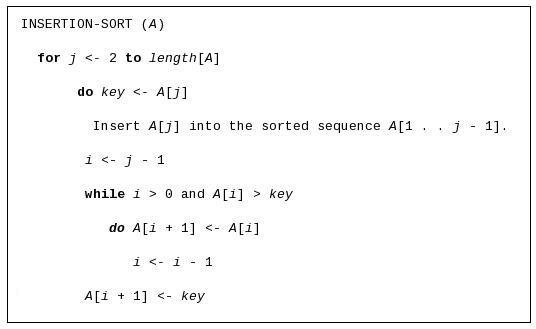
*Komentarz:* Sortowanie bąbelkowe jest jednym z pierwszych i najbardziej intuicyjnych sposobów sortowania, przy jednoczesnym byciu jednym z najwolniejszych możliwych do wyboru (wolniejszy jest np. Bogosort). Stosuje się go głownie w celach demonstracyjnych.

*Intuicja*: to sortowanie opiera się na bardzo prostym rozumowaniu, znaleźć arr[i] i arr[i + 1], porównać ich wartości i jeżeli arr[i] > arr[i + 1] zamienić je. Operacji zamian jest w najgorszym przypadku, odwróconej tablicy.



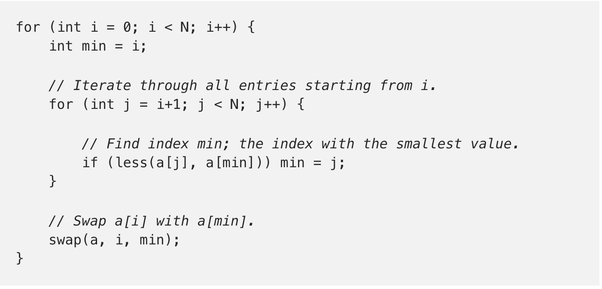
1. **Insert sort (sortowanie przez wstawianie)**

Intuicja: Mając nieposortowaną tablicę wybieramy jej pierwszy element. Następnie powtarzamy te kroki: patrzymy na pierwszy nieposortowany element na prawo od naszej posortowanej tablicy(element wybrany na początku). Porównujemy nieposortowany element z naszą posortowaną tablicą, jeżeli jest mniejszy od któregoś elementu, to zamieniamy go z nim miejscami. Operację powtarzamy dla każdego elementu nieposortowanej tablicy dopóki nie znajdziemy odpowiedniego dla nich miejsca w posortowanej.



1. **Selection sort (sortowanie przez wybieranie)**

Intuicja: Zaczynamy na początku tablicy. Wybieramy element, ustawiamy go jako nasz obecny element a jego wartość jako obecne minimum. Przeszukujemy tablicę do momentu znalezienia elementu od niego mniejszego. Jeżeli takowy znajdziemy, ustawiamy jego wartość jako nowe minimum a cały element jako „obecny element”. Przeszukujemy resztę talicy powtarzając tą operację w wypadku znalezienia jeszcze mniejszej wartości. Na końcu tablicy zamieniamy obecny element z początkiem tablicy. Mamy 1-elementową posortowaną tablicę. Cały algorytm powtarzamy dla zaczynając od każdego elementu tablicy po kolei, za każdym razem znajdując następny element posortowanej tablicy. Na samym końcu całość jest posortowana.

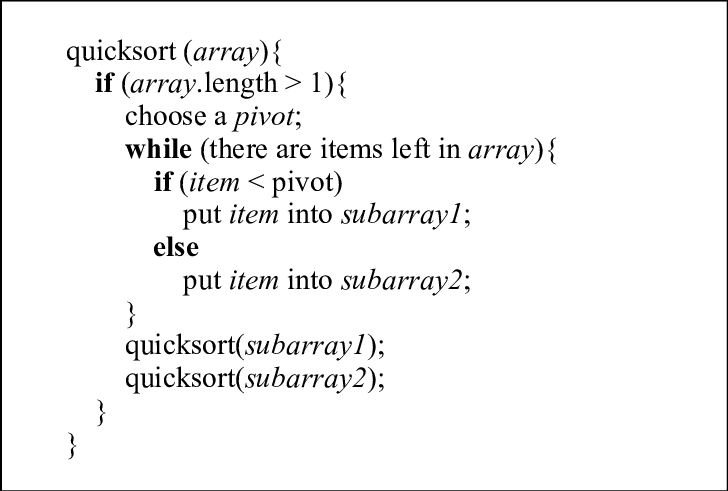


GRUPA B

1. **Quick sort**

*Komentarz*: Sortowanie szybkie jest jednym z najbardziej efektywnych i popularnie używanych sposobów sortowania tablic. Stosuje on, podobnie jak heap sort i merge sort, zasadę „dziel i rządź”, gdzie sortowanie rozbijamy na sortowania coraz to mniejszych tablic na końcu „łącząc” je w jedno.

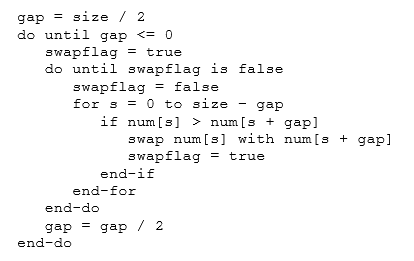
*Intuicja*: Wybieramy z tablicy arr[] ostatni element. Jest on naszym „pivotem”. Porównujemy z nim pozostałe elementy tablicy i rozdzielamy ją na te, które są większe od pivota i te, które są mniejsze. W następnym kroku powtarzamy powtarzamy poprzednią operację dla każdej z powstałej tablicy i nowego dla niej pivota. Dzielenie tablic następuje dopóki nie powstaną tablice 1-elementowe. Gdy otrzymamy takowe, zaczynamy porównywać elementy większych i mniejszych tablic ze sobą ustawiając je w kolejności rosnącej i wstawiając je na odpowiednie miejsca. Po całej operacji tablica zostaje posortowana.



1. **Shell sort**

*Komentarz*: Shell sort, sortowanie shella, jest nazywane i wyjaśniane często jako uogólnienie albo optymalizacja insert sort. Zazwyczaj jest on niego szybsze, ale nie zawsze, nie ma stałej złożoności czasowej, zależy ona od wielkości wybranych odstępów dla danej tablicy. Dla wybranych ręcznie odstępów może działać szybciej lub wolniej od stabilnego algorytmu.

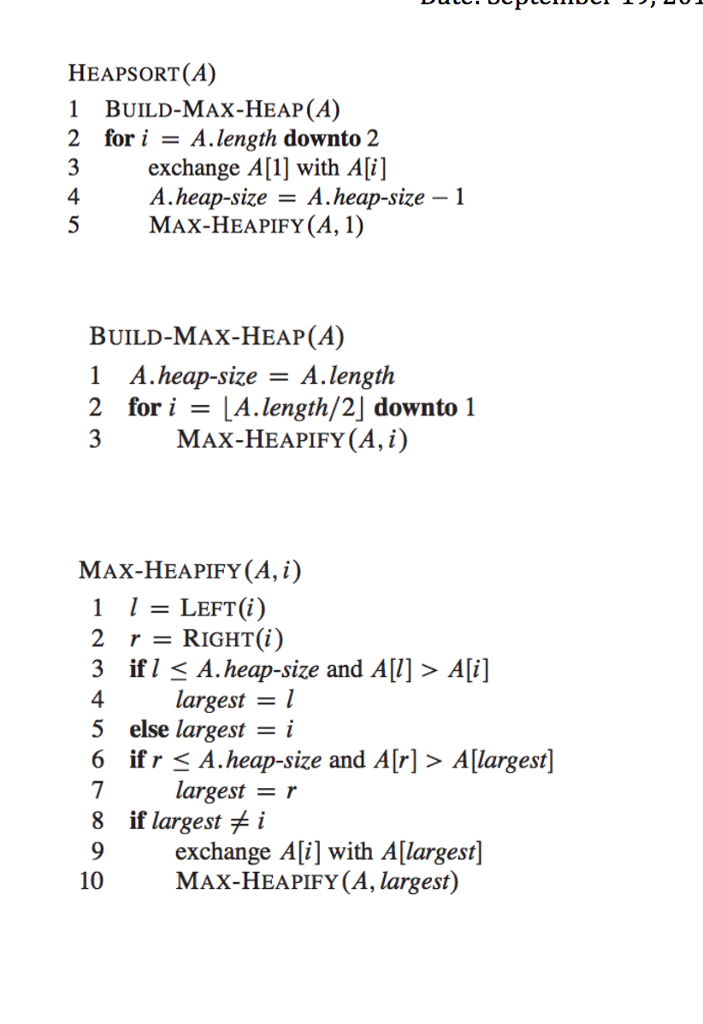
*Intuicja*: Wybieramy odstęp. Odstęp będzie wskazywał które elementy tablicy będziemy ze sobą porównywać. Od wyboru odstępu zależy efektywność działania algorytmu. Dla tablicy n my wybierzemy n/2 jako odstęp. Następnie wybieramy element tablicy od którego zaczynamy, niech to będzie element n/2. Porównujemy go a także każdy następny element z elementami oddalonymi o i\*odstęp, gdzie „i” będzie liczbą dotychczasowych porównań. W każdym przypadku gdzie obecna liczba jest mniejsza od sprawdzanej, zamieniamy je miejscami. Gdy już tak sprawdzimy wszyskie odpowiednie elementy, zmniejszamy odstęp dzieląc go przez 2 i powtarzamy wcześniejszą procedurę. W momencie gdy ostęp osiąga 0, algorym się kończy a tablica pozostaje posortowana.



1. **Heap sort (sortowanie przez kopcowanie)**

*Komentarz*: wraz z opisanym wcześniej algorytmem quicksort, pomimo bycia szybkim, jest niestabilny. Potencjalnie jest szybszy od quicksorta, gdyż jego złożoność czasowa wynosi w najgorszym przypadku n log n, w przeciwieństwie do n^2.

*Intuicja*: Postępujemy rekurencyjnie. Postępowanie w każdym kroku przebiega następująco: utworzenia kopca typu max, zamieniamy element max z końcowym elementem tablicy i usuwamy go z kopca. Po wykonaniu tych trzech części wywołujemy funkcję heapify() która pozwoli nam na ponowne utworzenia właściwego kopca z drzewa binarnego (powstało po zamianie i usunięciu elementu). Te operacje powtarzamy aż do momentu powstania kopca 1-elementowego, wskazującego na posortowanie tablicy.



1. Wyniki i estymacja efektywności użytych algorytmów

*Komentarz*: wyniki dla pierwszej grupy algorytmów są brane z użycia ich na tablicach do 50000 elementów, 40s działania, powyżej tego szybko osiągamy limit 15min za względu użycia bubble sort. O ile dla grupy I wyniki są dość jasne, to dla grupy II bardzo dużym odbiegiem od normy jest shell sort który wyprzedza quickSort prawie dwukrotnie przy każdej wielkości tablicy a także heapsort, który działa o wiele wolniej niż pozostałe dwa algorytmy w tej grupie pomimo spodziewanej złożoności n log n.

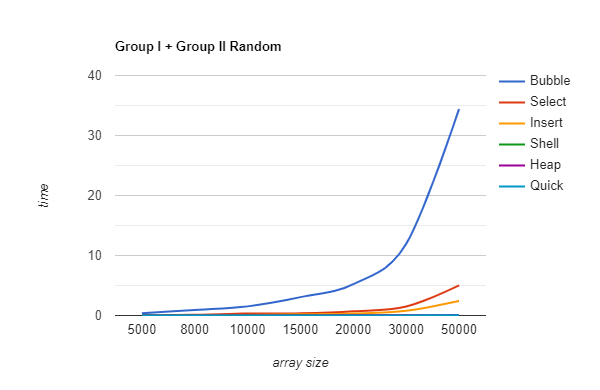
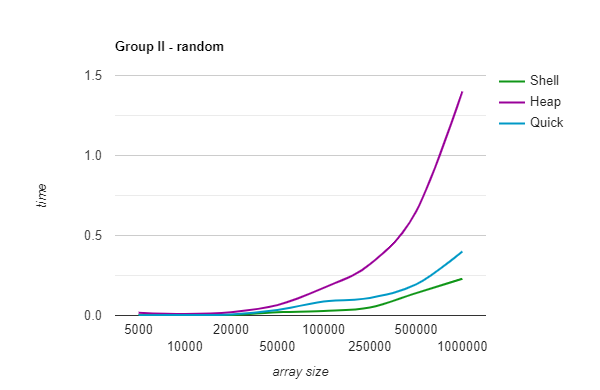
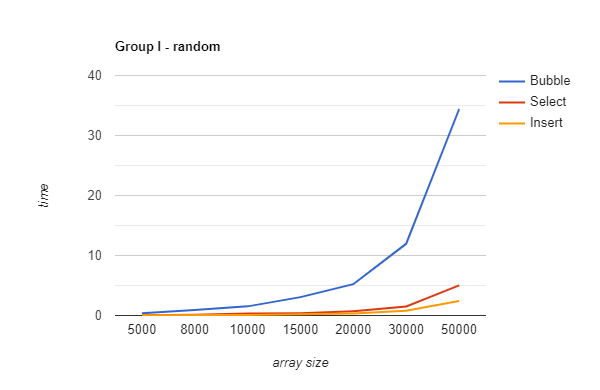
*Potencjalne przyczyny*: Quicksort jest wolniejszy od shellsorta w tym przypadku, a wynikać to może z dużej ilości różnych operacji, stosowania „if” w wielu miejscach a także większej ilości zmiennych i rekurenyjności w przeciwieństwie do iterowanego shellsorta. Jeżeli chodzi o heapsort, potencjalną przyczyną może być nie zoptymalizowana funkcja heapify() z dużą ilością porównań i zmiennych, a także ilość jej wywołań. Inną, prostszą i zgadzającą się z wynikami przyczyną może być pomieszanie wyników heapsort i shellsort, choć nie udało mi się doszukać błędu tego rodzaju w programie

*Alternatywna przyczyna uzyskanych wyników*: Możliwe jest również, że winę ponoszą nie algorytmy same w sobie, ale sposób mierzenia czasu. Podczas tworzenia projektu w VS2017 nie miałem dostępu do „gettimeofday” będącego częścią biblioteki <sys/time.h>. Funkcja time() była zbyt niedokładna jako że zwracała całe sekundy. Alternatywą w tym przypadku była funkcja clock() która zwracała nam czas procesora, różnica początku i końca zmiennych uzyskanych z jej pomocą pokazywała ilość „tików” procesora wykonanych w czasie algorytmu. Dzieląc tą liczbę przez stałą „CLOCKS\_PER\_SEC” powinniśmy uzyskać realny czas sortowania, choć możliwe że ta logika jest błędna. Jeżeli tak, to wyniki przedstawiają czas nie w czasie ludzkim, ale czasie procesora, dlatego też na wykresach czas jest ujęty bez jego miary.

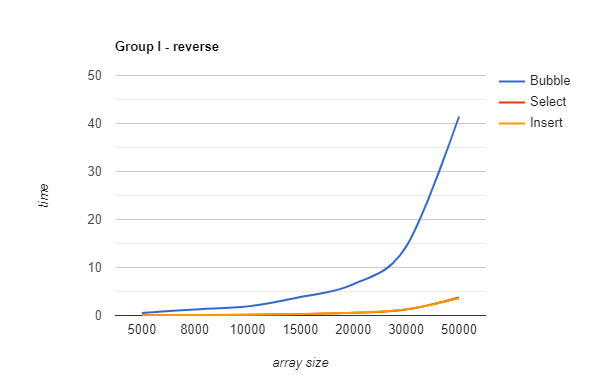
|  |  |  |  |
| --- | --- | --- | --- |
|  | Bubble sort | Selection sort | Insert sort |
| Oczekiwane O(n) | O(n^2) | O(n^2) | O(n^2) |
| Obserwowane O(n) | O(n^2) | O(n^2) | O(n^2) |

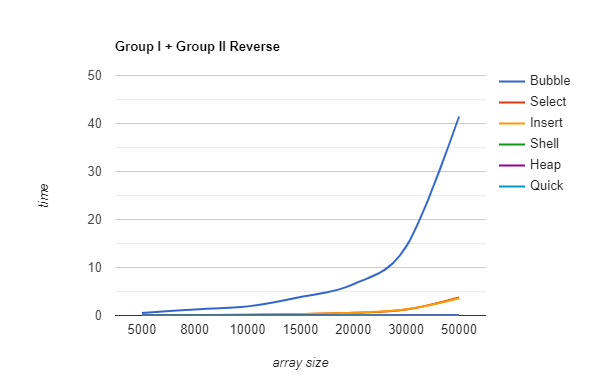
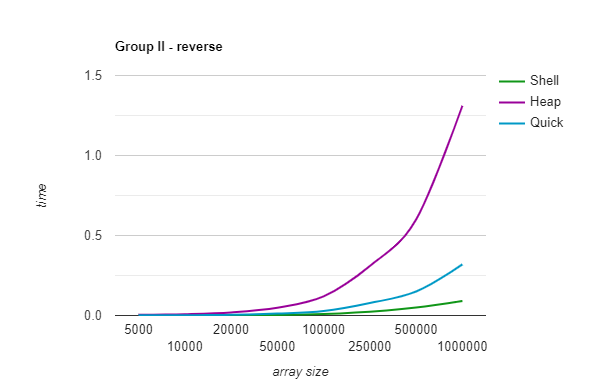
|  |  |  |  |
| --- | --- | --- | --- |
|  | Shell sort | Heap sort | Quick sort |
| Oczekiwane O(n) | O(n^(3/2)) | O(nlogn) | O(nlogn) |
| Obserwowane O(n) | ~O(nlogn) | O(n^3/2) | O(nlogn) |

1. Losowe

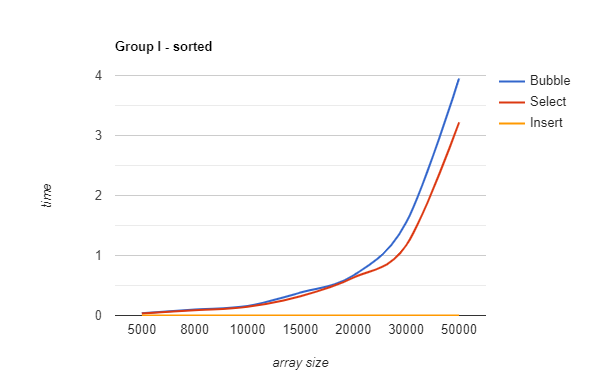


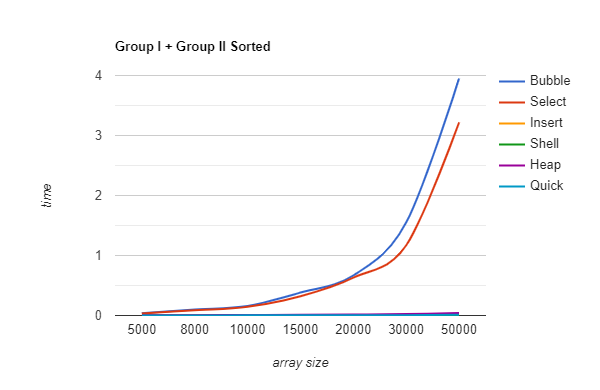
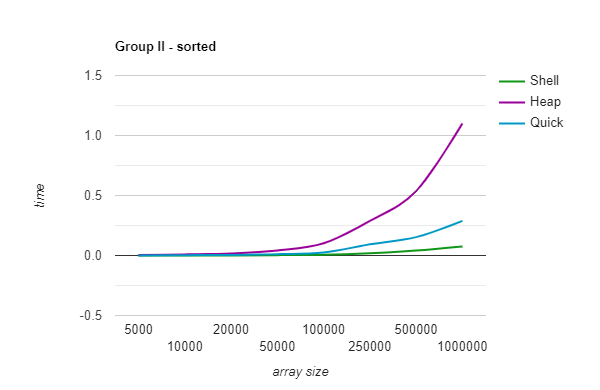
1. Tablica odwrócona





1. Tablica posortowana





Analizując uzyskane wyniki dla konkretnych testów możemy wyciągnąć kilka wniosków:

- Dla każdego rodzaju testu, sortowanej tablicy, algorytmy z grupy I były znacząco wolniejsze od tych należących do grupy II. Jest to spowodowane głównie filozofią algorytmów z obu tych grup; grupa I układa tablicę w sposoby głównie proste, sprawdzające bardzo dużą część tablicy za każdą iteracją. Druga grupa natomiast w swoich sposobach próbuje rozbijać dany im problem na mniejsze, quicksort z mniejszymi posortowanymi tablicami, heapsort z kopcem typu max i shell sort z porównywaniem dopiero co m-tego elementu i zamienianiem go, powoli zmniejszając m.

- Tablica odwrócona spowalnia z algorytmy z pierwszej grupy, poza selectsort, które przyspieszyło i zrównało swoją szybkość z insertsort. W drugiej grupie tablica odwrócona sprawiła przyspieszenie działania każdego z algorytmów względem sortowania elementów losowych.

- Tablica posortowana przyspieszyła działanie wszystkich algorytmów z grupy I, przy czym czas działania insertsort został zmniejszony do 0. W drugiej grupie posortowana tablica nieznacząco przyspieszyła działanie każdego z algorytmów względem ich szybkości sortowania tablicy odwróconej.

- Mierząc szybkość algorytmów z pomocą clock(), shellsort zostaje najszybszym algorytmem z opisanych, na podstawie doświadczeń.

- BubbleSort jest jednoznacznie najwolniejszym algorytmem z opisanych, zawdzięcza swojej prostocie, nieelastyczności i linearności działania, przez co dla większości tablic czas jego działania pozostaje długi i trudny do polepszenia (możliwe przez bardziej posortowaną tablicę).