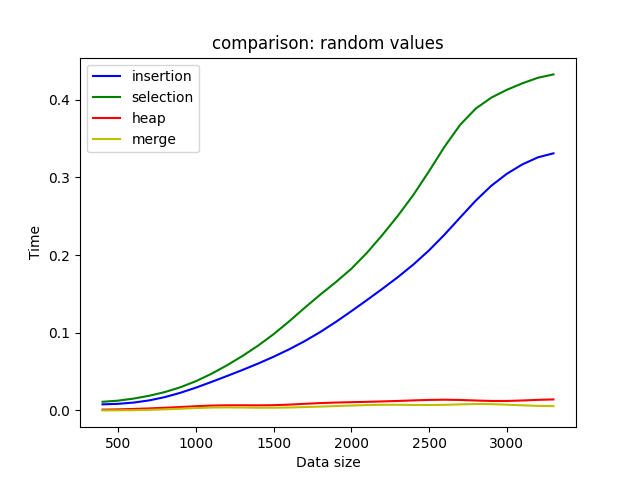
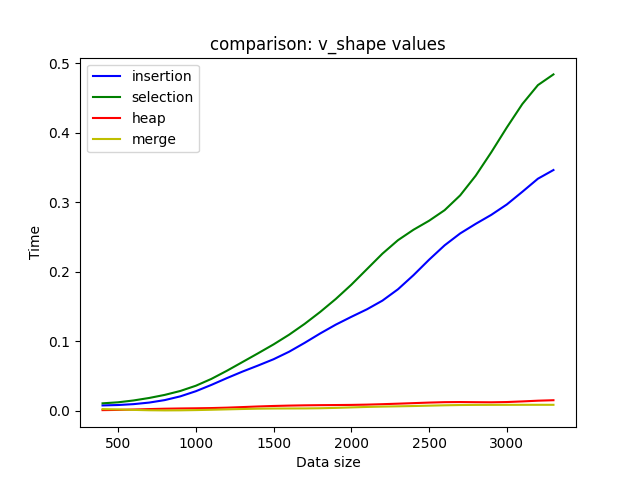
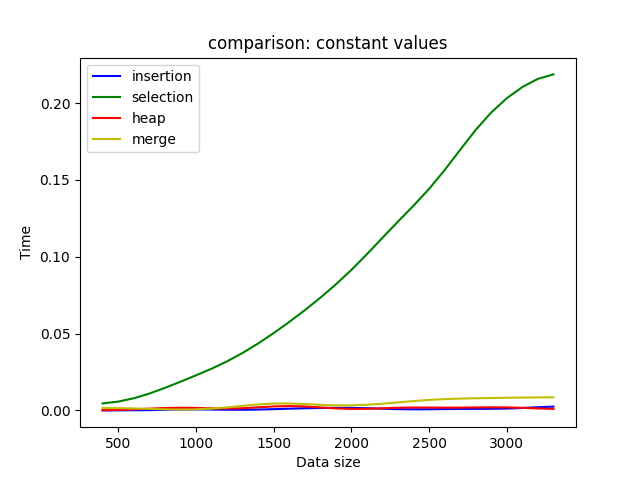
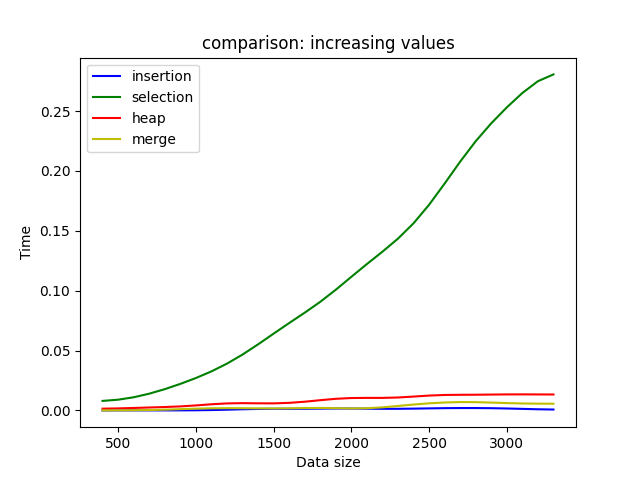
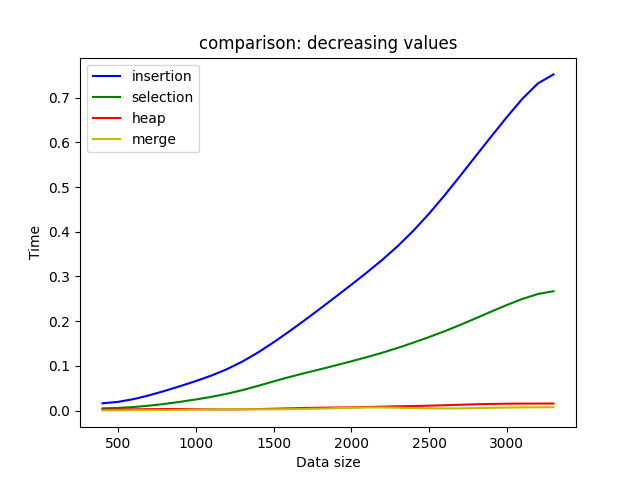
**Implementacja metod sortowania**

Do zbadania złożoności obliczeniowej algorytmów oraz sporządzenia wykresów wykorzystaliśmy język Python – pomiar czasu przeprowadzony był przy pomocy metody proces\_time pochodzącej z biblioteki time. Nieścisłości w pomiarze czasu na wykresach mogą wynikać z błędów wybranej metody pomiarowej.

* 1. **Porównanie złożoności czasowej dla danych losowych **
  2. **Porównanie złożoności czasowej dla danych v-kształtnych **
  3. **Porównanie złożoności czasowej dla danych stałych**
  4. **Porównanie złożoności czasowej dla danych rosnących**
  5. **Porównanie złożoności czasowej dla danych malejących**

**Wnioski:**

Złożoność pamięciowa: wszystkie z przedstawionych algorytmów poza merge sortem działają w miejscu – ich złożoność pamięciowa wynosi O(1). Natomiast dla merge sortu wyniesie ona O(n), co wynika z konieczności utworzenia dodatkowych list podczas podziału wejściowej listy.

Algorytmy heap sort oraz merge sort należą do bardziej efektywnej klasy sortowań o złożoności czasowej O(n logn) dla średniego przypadku, podczas gdy insertion sort oraz selection sort należą do klasy wolniejszej - o złożoności czasowej O(n2). Widoczne jest to na wykresach. Poza przypadkiem, gdy insertion sort przyjmuje złożoność O(n), dla danych posortowanych oraz stałych, algorytmy heap sort i merge sort – niezależnie od danych wejściowych – wykonują się w czasie zdecydowanie krótszym niż selection sort i insertion sort, a ewentualne różnice między nimi oraz między ich działaniem dla różnych typów danych wynikają w dużym stopniu z błędów pomiaru czasu.

1.1 Selection sort jest niewrażliwy na dane wejściowe – zawsze przyjmuje złożoność czasową O(n2) przez konieczność wyszukania najmniejszego elementu w nieposortowanej części listy (czyli przeszukania jej całej). Liczba porównań będzie większa niż w przypadku insertion sortu, dla którego porównanie z każdym elementem listy (części posortowanej) konieczne jest tylko w przypadku, gdy natrafiono na element najmniejszy. Czasami, by ocenić, że element jest na dobrej pozycji, wystarczy jedno porównanie.

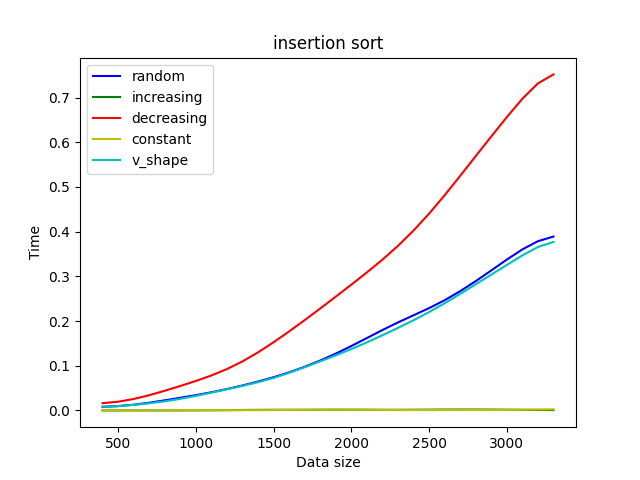
1.2 Analogicznie do 1.1 - przypadek v-kształtny dla insertion sortu jest połączeniem przypadku najgorszego oraz najlepszego, w rezultacie dając przypadek średni (występujący również dla danych losowych).

1.3 Niewrażliwy na dane wejściowe selection sort wykona się w O(n2), podczas gdy insertion sort wykonuje się w czasie liniowym. Wynika to z faktu, że pierwszym krokiem podczas wykonywania insertion sortu jest sprawdzenie, czy element występujący po lewej stronie listy (część uporządkowana) jest większy od tego po prawej i – jeśli to prawda – dokonanie serii zamian elementów. Zamiany ta nie zostają wykonane w przypadku danych uporządkowanych rosnąco oraz stałych, co przyspiesza działanie algorytmu (złożoność O(n)). W przypadku danych stałych złożoność czasowa algorytmu heap sort wynosi O(n) – kopiec z identycznych elementów zachowuje poprawność zarówno na etapie budowy, jak i sortowania (do oceny tego wystarczy po 1 porównaniu), nie zachodzi więc konieczność rekurencyjnego dokonywania zamian elementów i następującego po nim sprawdzania poprawności.

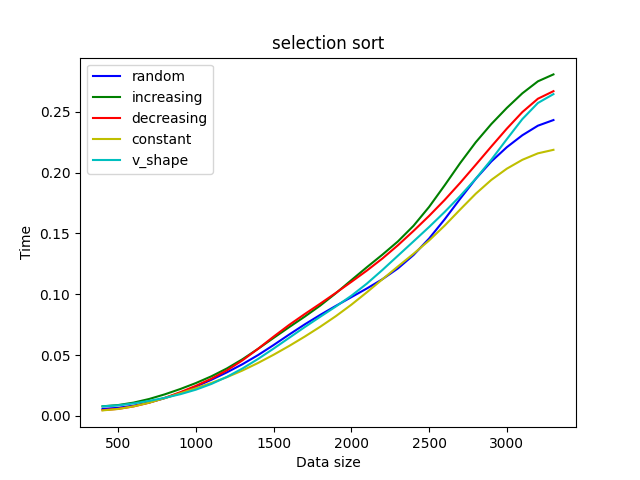
1.4 Analogicznie do poprzedniego przypadku, również dla danych posortowanych insertion sort będzie miał złożoność czasową O(n), selection sort O(n2), merge sort O(n log n), jedyna różnica dotyczyć będzie heap sortu, który wykona się w czasie logarytmicznym, ponieważ na etapie sortowania liczb na podstawie utworzonego drzewa dokonane zostaną rekurencyjne zamiany elementów i sprawdzanie poprawności kopca.

1.5 Algorytmy heap sort, merge sort i selection sort wykonają się w czasie podobnym do przypadku losowego – różnica dla algorytmów o złożoności logarytmicznej jest niezauważalna, a czas wykonania selection short ponownie stały. Wysoki czas wykonania insertion sort wynika z wystąpienia najgorszego przypadku – wykonana zostanie maksymalna liczba zamian elementów. Wynika to z faktu, że algorytm, w celu znalezienia miejsca dla każdego z elementów musi przejść przez całą dotychczas posortowaną listę.

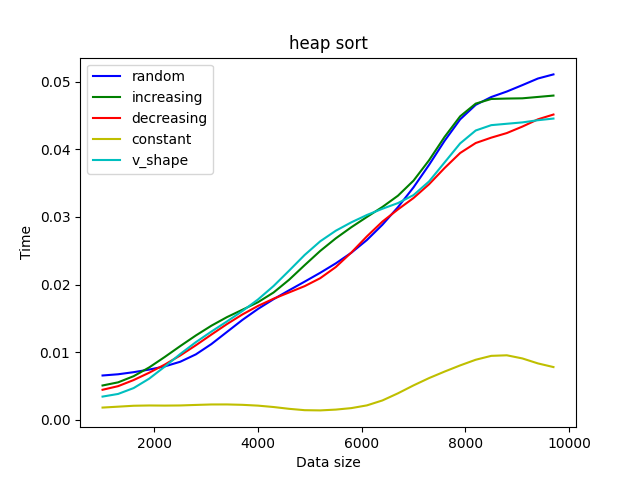
* 1. **Porównanie złożoności czasowej dla algorytmu insertion sort**



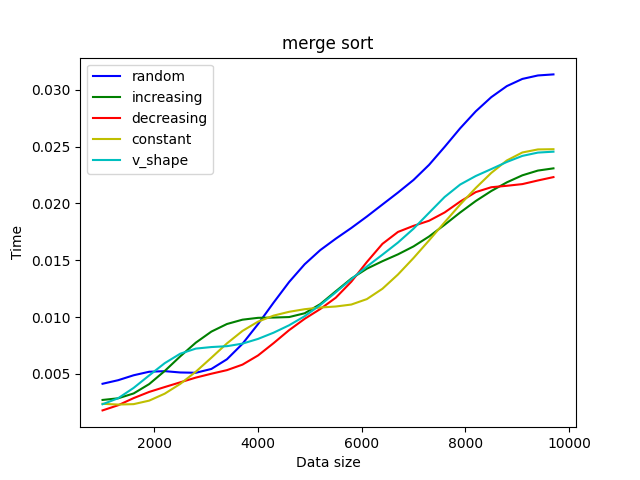
* 1. **Porównanie złożoności czasowej dla algorytmu selection sort**

****

* 1. **Porównanie złożoności czasowej dla algorytmu heap sort**

****

* 1. **Porównanie złożoności czasowej dla algorytmu merge sort**



**Wnioski:**

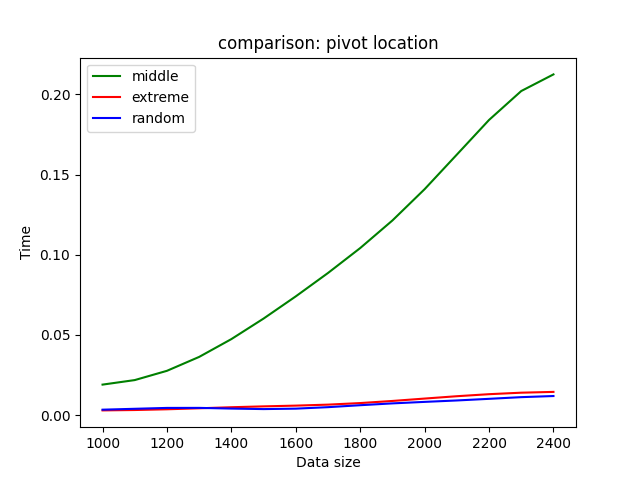
2.1 Przypadek wartości stałych i posortowanych jest optymistyczny, ponieważ algorytm przechodzi przez całą tablicę w poszukiwaniu elementu mniejszego niż element stojący przed nim. Dla listy posortowanej element taki nie występuje, co skutkuje złożonością liniową. Przypadek losowy i v-kształtny są przypadkami średnimi – dla listy v-kształtnej występuje połączenie przypadku najgorszego (listy liczb malejących) oraz najlepszego (liczb posortowanych). Liczba porównań i zamian jest więc podobna do listy wartości liczb wygenerowanych zupełnie losowo. W najgorszym przypadku konieczne jest przejście przez całą uporządkowaną część listy w celu znalezienia pozycji dla pierwszego elementu części nieuporządkowanej. Krok ten musi być powtórzony dla każdego elementu, co skutkuje wysokim czasem wykonania algorytmu dla danych malejących.

2.2 Selection sort wykonuje tą samą liczbę zamian elementów i porównań niezależnie od danych wejściowych – niewielkie różnice pomiaru czasu można zaliczyć jako błędy pomiarowe.

2.3 Heap sort dla większości typów danych wejściowych wykona się ze złożonością czasową O(n log n), wyjątkiem jest przypadek listy liczb stałych. Na etapie tworzenia posortowanej listy z utworzonego wcześniej drzewa (tworzenie kopiec z identycznych wartości odbywa się ze złożonością czasową O(n)) nie dochodzi do zamiany elementów identycznych – zamiany te miałyby natomiast miejsce w pozostałych przypadkach, nie jest również rekurencyjnie sprawdzany warunek poprawności kopca.

2.4 Dodatkowy czas wykonania algorytmu dla losowych danych wynika ze sposobu, w jaki scalane są podlisty. W przypadku, gdy posortowane zostają wszystkie elementy jednej z 2 scalanych podlist pozostała zawartość drugiej jest dopisywana na końcu podlisty wynikowej. Przypadek ten, generujący mniejszą liczbę porównań, występuje rzadziej dla danych losowych, niż dla pozostałych.

**Quick sort z różnym wyborem pivotu dla danych a-kształtnych:**

****

**Wnioski:**

Na wykresie widać znaczącą różnicę w czasie wykonania algorytmu dla pivotu umiejscowionego w środku, w porównaniu do umiejscowienia losowego i skrajnego. Wynika ona z faktu, że dla ciągu a-kształtnego element środkowy jest największy (lub prawie największy), algorytm dzieli więc listę na 2 części: składającą się z pivotu (lub największej wartości listy) oraz pozostałą. Po podziale uzyskuje się podlistę, której środkowy element jest jednym z największych. W rezultacie, mimo niewielu zamian (ich liczba stopniowo rośnie, chociaż maleje liczba elementów do posortowania) zachodzi konieczność wielokrotnego porównywania wszystkich elementów listy z pivotem, a stos wywołań rekurencyjnych osiąga duże rozmiary (ze względu na dysproporcję między rozmiarami podlist – jedna jest zdecydowanie większa), a więc złożoność wzrasta do O(n2). W przypadku pivotu skrajnego algorytm przez kilka początkowych iteracji będzie zachowywać się podobnie, ale w końcu pivot przyjmie wartości zbliżające się bardziej do mediany ciągu, co z kolei doprowadzi do osiągnięcia złożoności logarytmicznej. Losowy wybór pivotu doprowadzi również do przypadku średniego – o złożoności O(n log n).