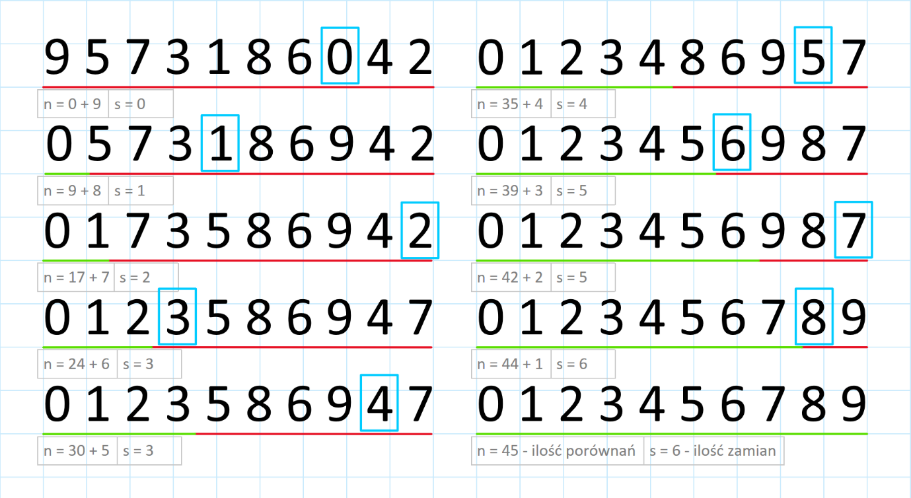
# Wstęp i opis algorytmów

W tym rozdziale opisuję główne celi projektu oraz wykorzystane algorytmy dla ich implementacji.

Głównym zadaniem projektu było porównanie działania dwóch algorytmów sortowania, a mianowicie: sortowania przez wybieranie oraz sortowania kopcowego. Drugorzędnym zadaniem było napisanie algorytmów odczytu danych z plików i generowania danych testowych o różnej złożoności sortowania dla każdego algorytmu.

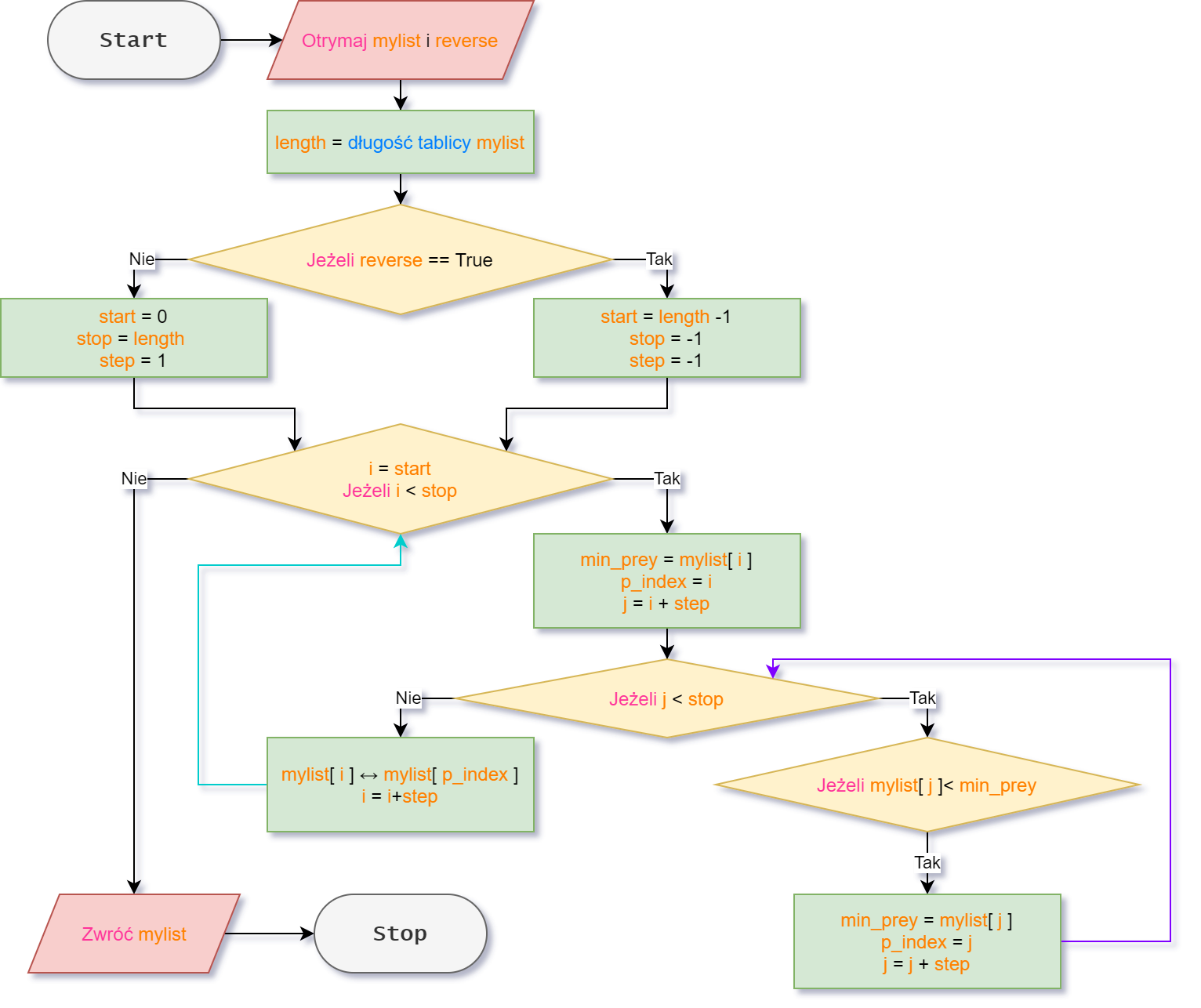
## Sortowanie przez wybieranie

Sortowanie przez wybieranie to niestabilny porównujący algorytm sortowania na miejscu. Ma złożoność czasową , co oznacza, że jest nieefektywnym w przypadku dużych list. Taki algorytm charakteryzuje się prostotą i ma przewagę wydajnościową nad bardziej złożonymi algorytmami w pewnych sytuacjach, zwłaszcza gdy pamięć pomocnicza jest ograniczona.

Algorytm dzieli listę wejściową na dwie partycji: posortowaną i nieposortowaną. Początkowo posortowana część jest pusta, a nieposortowana partycja zajmuje całą listę wejściową. Algorytm jest wykonywany poprzez znalezienie najmniejszego elementu w niesortowanej części, zamianę go na pierwszy nieposortowany element i przesunięcie podziału listy. Działanie algorytmu krok po kroku pokazano na rysunku.

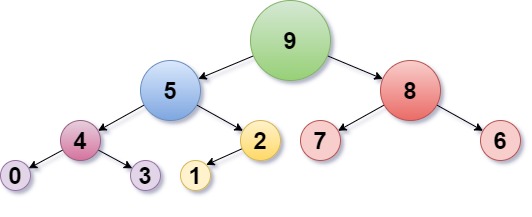
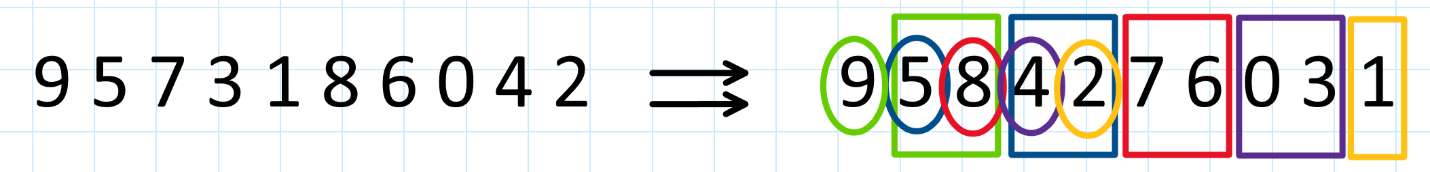
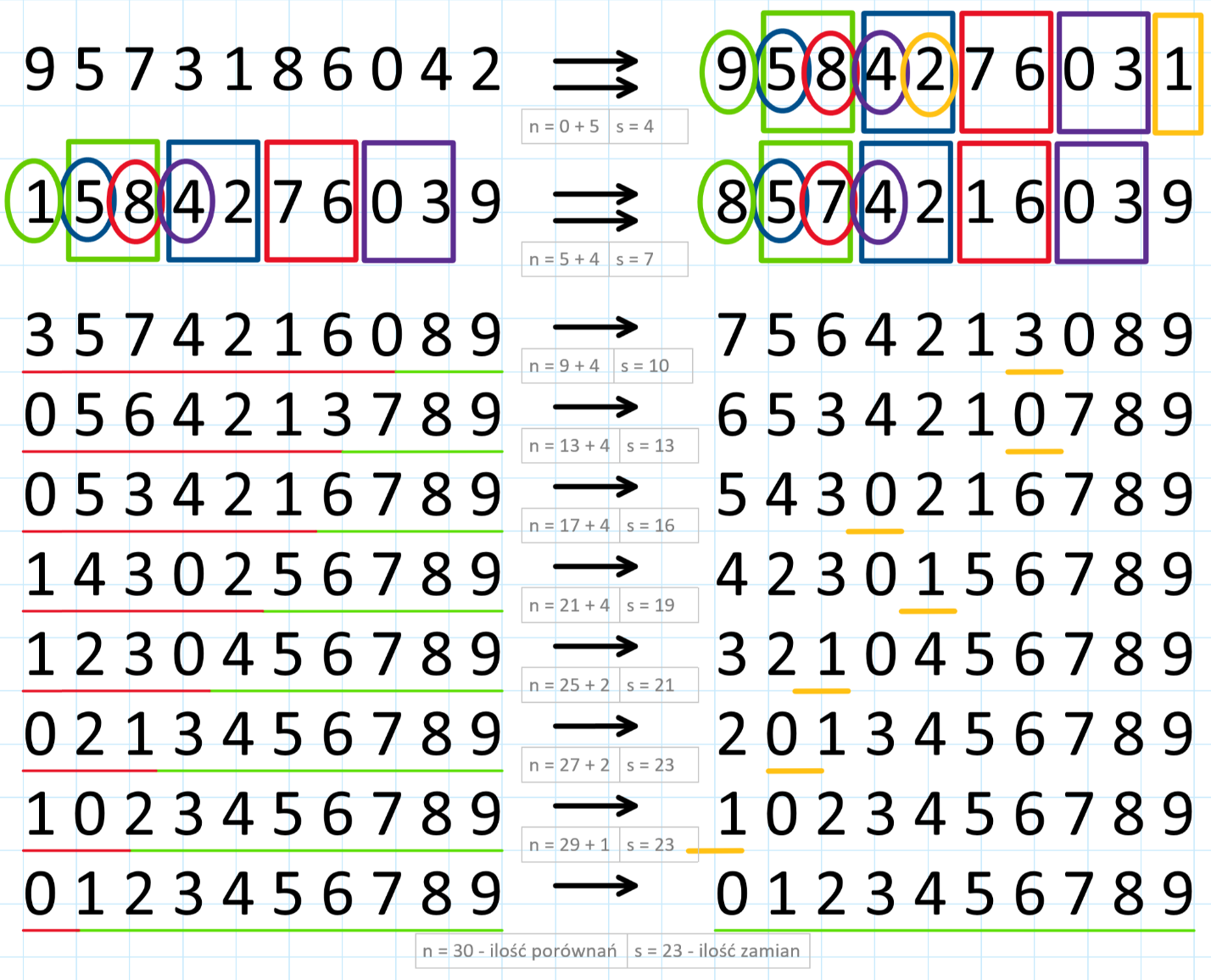
Pseudokod algorytmu sortowania przez wybieranie:

1. Otrzymaj = mylist i reverse
2. length = długośc tablicy mylist
3. Jeżeli reverse == True wykonuj K04:K06
4. | start = 0
5. | stop = length - 1
6. | step = 1
7. Inaczej wykonuj K08:K10
8. | start = length - 1
9. | stop = 0
10. | step = -1
11. Dla i = start, start + step, start + 2\*step, start + 3\*step, … , stop: wykonuj K12:K18
12. | min\_prey = mylist[ i ]
13. | p\_index = i
14. | Dla j = i + step, i + 2\*step, i + 3\*step, … , stop: wykonuj K15:K17
15. | | Jeżeli mylist [ i ] < min\_prey wykonuj K16:K17
16. | | | min\_prey = mylist[ j ]
17. | | | p\_index = j
18. | mylist[ i ] ↔ mylist[ p\_index ]
19. Zwróć mylist
20. Zakończ

Schemat blokowy tego algorytmu podano na rysunku.

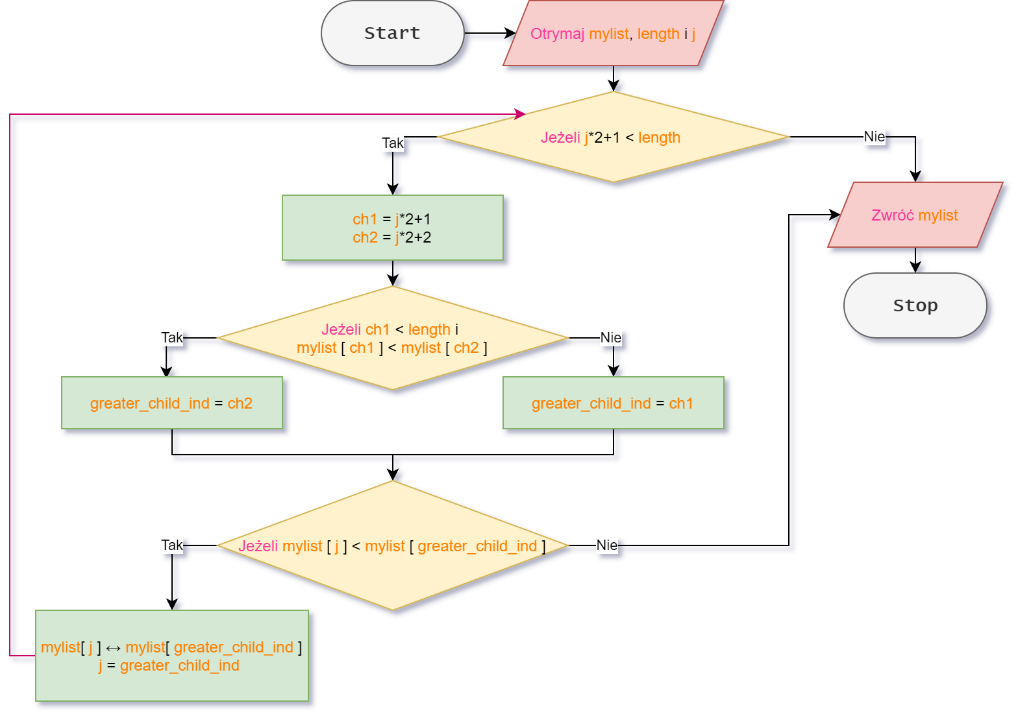
Największą zaletą takiego algorytmu jest minimalna możliwa liczba zamian elementów (w najgorszym przypadku ). Mimo to efektywność czasowa sortowania według wyboru jest kwadratowa, więc istnieje wiele algorytmów, które mają mniejszą złożoność czasową.

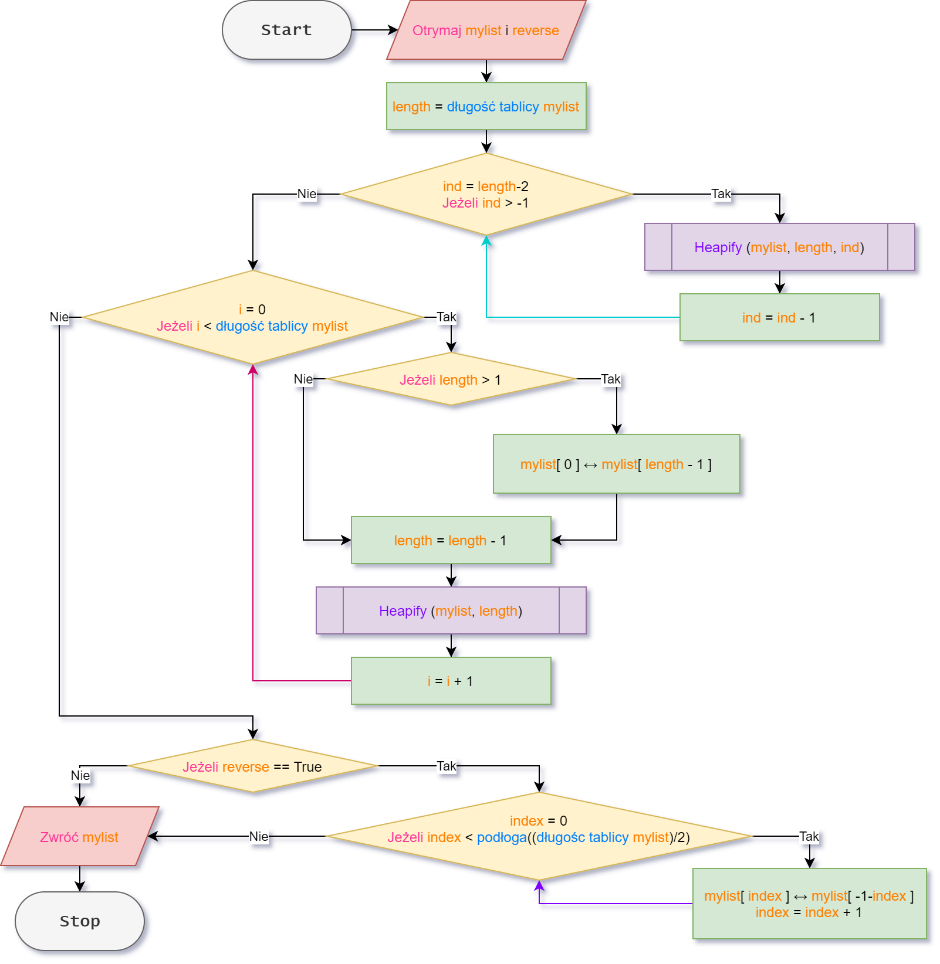
## Sortowanie przez kopcowanie

Jednym z takich algorytmów jest sortowanie przez kopcowanie, które zostało wynalezione przez J. Williamsa w 1964 roku. Podobnie do algorytmu sortowania przez wybieranie, kopcowy algorytm sortuje na miejscu metodą porównań, nie jest stabilnym i analogicznie dzieli dane wejściowe na posortowaną i nieposortowaną partycji. Sortowanie kopcowe wybiera największy element z części niesortowanej i wstawiając go do posortowanej części, iteracyjnie zmniejsza nieposortowaną partycję. W przeciwieństwie do sortowania przez wybieranie, sortowanie kopcowe nie skanuje liniowo cały nieposortowany obszar. Ten algorytm obsługuje nieposortowaną część w postaci kopca (rysunek), aby szybko znaleźć największy element, który zawsze będzie znajdować się na górze. Ale dla roboty takiego algorytmu dane wejściowe muszą być ułożone w postaci kopca. Takie początkowe przygotowanie danych wejściowych wykonuje algorytm Heapify, sprawdzając czy potomki danego elementu są mniejsze od niego. Przykład kopca z rysunku w postaci tablicy podano na rysunku. Działanie algorytmu krok po kroku pokazano na rysunku.

Sortowanie kopcowe ma pesymistyczną złożoność czasową , co jest nawet lepsze niż szybkie sortowanie (), chociaż w praktyce działa wolniej (oczekiwana złożoność obu algorytmów jest taka sama i wynosi ).

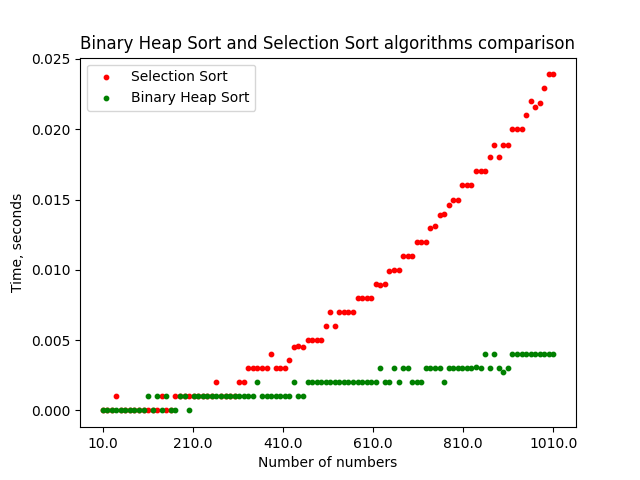
Pseudokod algorytmu tworzenia kopca:

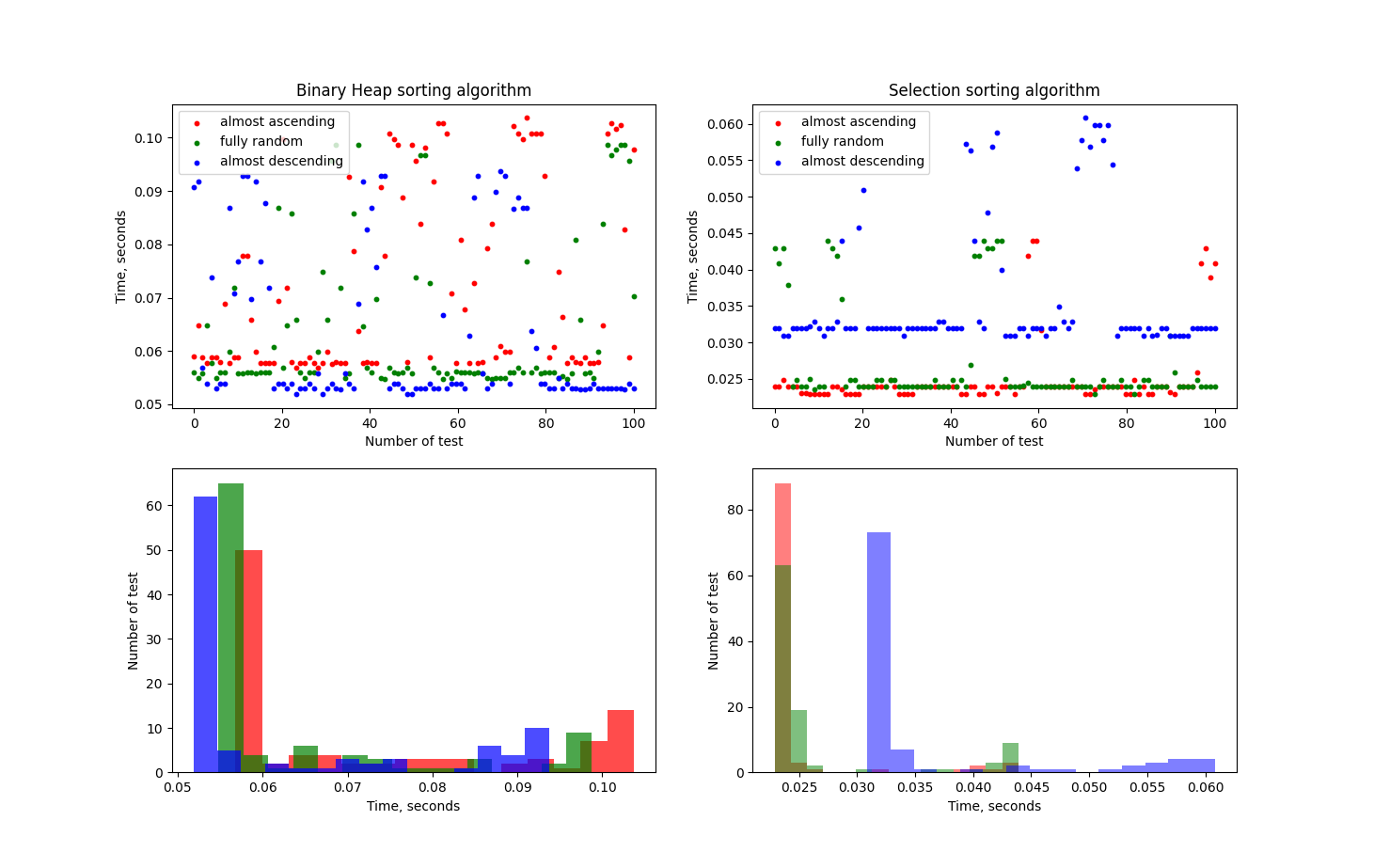
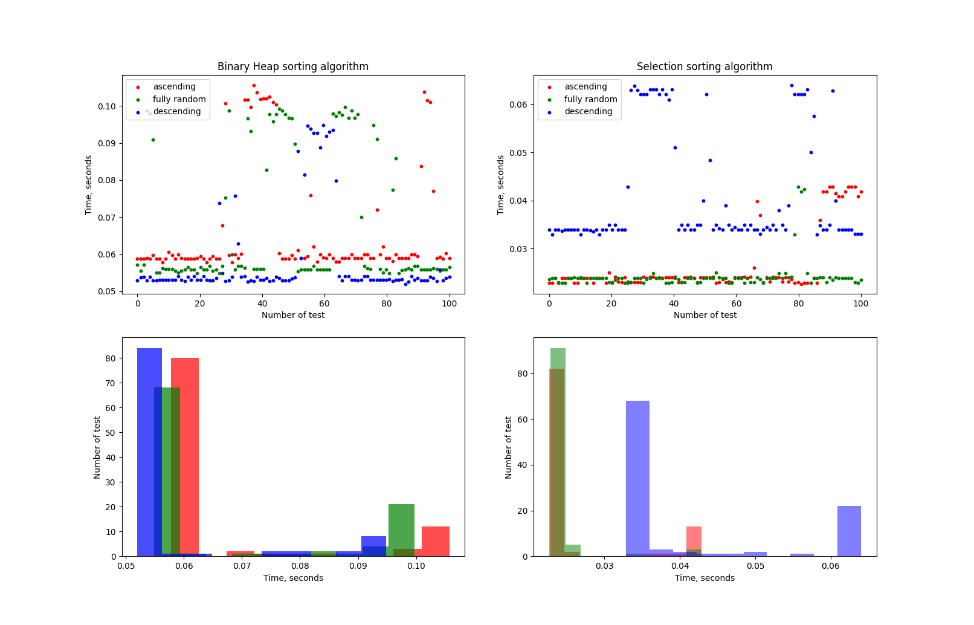
1. Otrzymaj = mylist, length i j
2. Dopóki j\*2+1 < length wykonuj K03:K13
3. | ch1 = j\*2+1
4. | ch2 = j\*2+2
5. | Jeżeli ch1 < length i mylist [ ch1 ] < mylist [ ch2 ] wykonuj K06
6. | | greater\_child\_ind = ch2
7. | Inaczej wykonuj K08
8. | | greater\_child\_ind = ch2
9. | Jeżeli mylist [ j ] < mylist [ greater\_child\_ind ] wykonuj K10:K11
10. | | mylist [ j ] ↔ mylist [ greater\_child\_ind ]
11. | | j = greater\_child\_ind
12. | Inaczej wykonuj K13
13. | | break
14. Zwróć mylist
15. Zakończ

Pseudokod algorytmu sortowania przez kopcowanie:

1. Otrzymaj = mylist i reverse
2. length = długośc tablicy mylist
3. Dla ind = length - 2, length - 3, length - 4, … , 0: wykonuj K04
4. | mylist = tworzenie kopca (mylist, length, ind)
5. Dla i = 0, 1, 2, … , długośc tablicy mylist - 1: wykonuj K12:K18
6. | Jeżeli length > 1 wykonuj K06
7. | | mylist [ 0 ] ↔ mylist [ length - 1 ]
8. | length = length – 1
9. | mylist = tworzenie kopca (mylist, length, 0)
10. Jeżeli reverse == True wykonuj K04:K06
11. | Dla index = 0, 1, 2, … , podłoga((długośc tablicy mylist)/2)- 1: wykonuj K12
12. | | mylist [ index ] ↔ mylist [ długośc tablicy mylist - index - 1 ]
13. Zwróć mylist
14. Zakończ

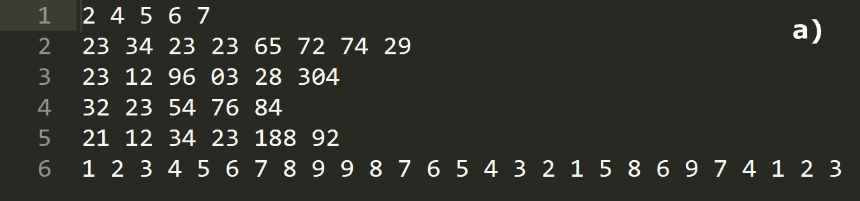
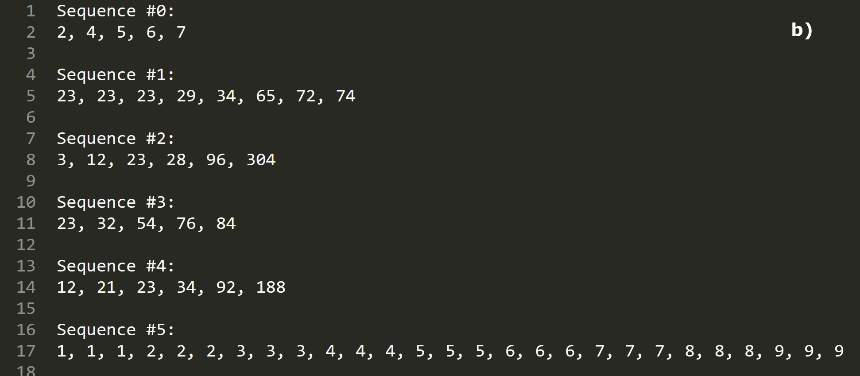
# Porównanie algorytmów

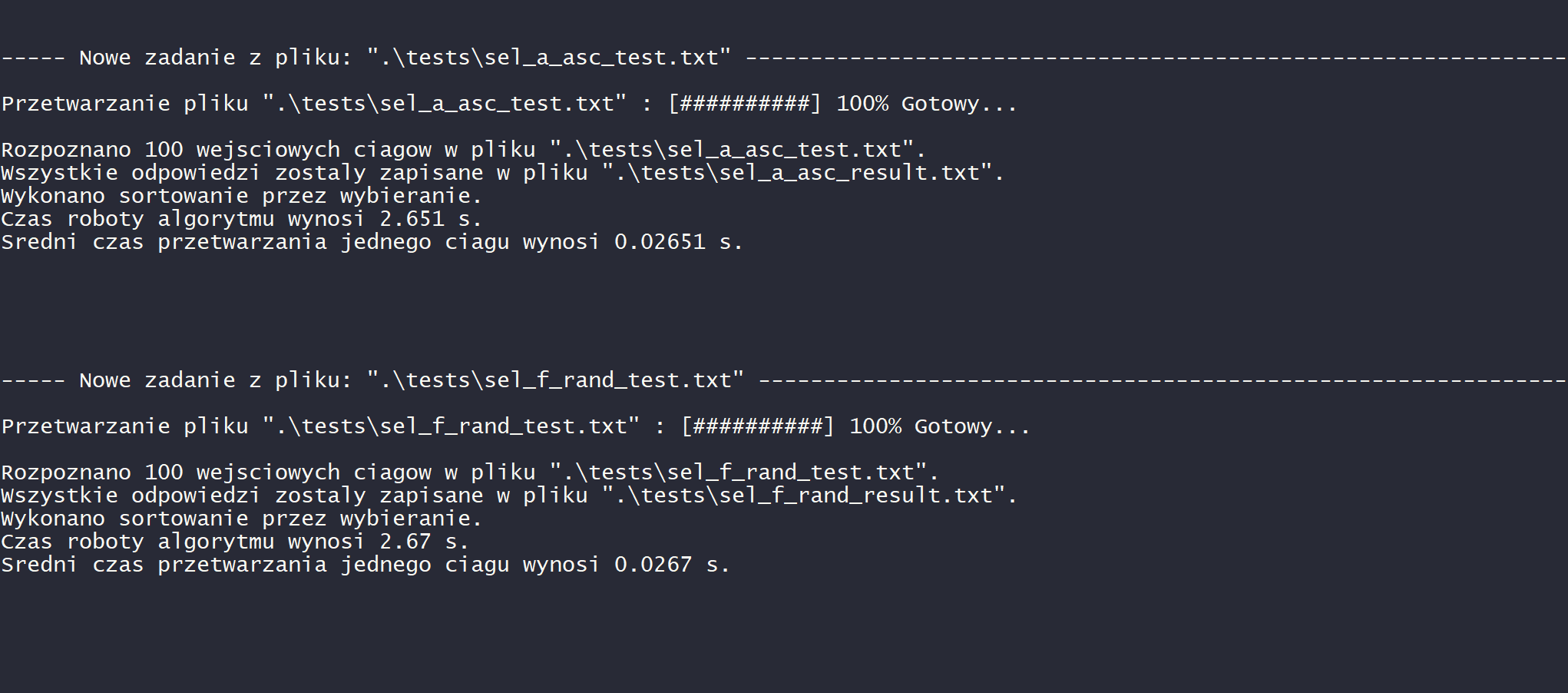
Jak już było umówiono w poprzednim rozdziale, złożoność obliczeniowa algorytmu sortowania przez wybór zawsze wynosi , a sortowania przez kopcowanie - . Od razu widać przewagę drugiego algorytmu nad pierwszym, co potwierdzają dane o ilości porównań na rysunkach. Dla pełnego przekonania w poprawności takich danych wykonano niektóre testy algorytmów i otrzymano wyniki opatrzone na rysunku. Na wykresie bardzo dobrze widać, że sortowanie przez wybieranie już po czterystu liczbach zaczyna działać znacznie wolniej w porównaniu do sortowanie przez kopcowanie.

Mimo to, że oba algorytmy sortowania są dość niezależne od rozkładu danych wejściowych (optymistyczna i pesymistyczna złożoność obliczeniowa obu algorytmów nie różni się od oczekiwanej), wciąż dane wejściowe można przygotować w taki sposób, że sortowanie będzie działać wolniej lub szybciej. Przygotowałem zbiór danych wejściowych, które już są prawie posortowane (większość elementów znajdują się na swoich pozycjach) rosnąco lub malejąco i otrzymałem wyniki pokazane na rysunku. Widać, że około 60 procent ciągów wejściowych, które były posortowane prawie malejąco algorytm sortowania przez kopcowanie posortował najszybciej, gdy algorytm przez wybieranie posortował najdłużej. Dla sortowanie kopcowego taki wzór wynika z tego, że utworzyć kopic jest prościej dla danych, które są posortowane malejąco. Natomiast dla algorytmu przez wybieranie to wynika z komand K16 i K17 w pseudokodzie. Idąc po ciągu liczb, spotykając każdego razu liczbę mniejszą, zapisujemy jej indeks, co zwiększa czas roboty programu. Uzupełnić ten pomysł można patrząc na ciągi liczb posortowanych rosnąco. Oczywistym jest pomysł, że zrobić kopiec z ciągu rosnącego jest dość trudno. Natomiast znaleźć najmniejszy element można prawie z pierwszego razu, dlatego nie jest potrzebnym przepisywanie indeksu każdego razu.

Jeżeli dane są posortowane idealnie, to wyżej omówione właściwości są jeszcze lepiej widoczne. Wykresy takiego zbioru testów pokazany na rysunku.

# Dokumentacja z doświadczeń

Algorytm wczytywania plików, który znajduje się w pliku „file\_handling.py”, przyjmuje wszystkie ciągi liczbowe separowane spacją, przecinkiem lub średnikiem. W przypadku więcej niż jednego ciągu, należy ich oddzielić nową linią. Na rysunku podano przykład danych wejściowych i wyników otrzymanych po uruchomieniu programu.

Po uruchomieniu każdego sortowania w konsoli są podawane niektóre dane o zadaniu, które oblicza się w tej chwili. Na przykład dla użytkownika mogą być użytecznymi dane o czasie wykonania sortowania, lub który algorytm jest używany. Przykładowe dane z konsoli są podane na rysunku.

# Wnioski

W danym projekcie przeanalizowałem dwa algorytmy sortowania ciągów liczbowych o złożonościach i , a mianowicie sortowanie przez wybieranie i sortowanie przez kopcowanie.

Wynikiem przerobionej pracy został program, który ma możliwości:

* wczytywania danych wejściowe z różnych plików tekstowych oraz plików „\*.csv”;
* tworzenia danych wejściowych w postaci ciągu losowego lub ciągów o różnym poziomie optymistyczności/pesymistyczności;
* wypisywania posortowanych ciągów do plików;
* demonstracji złożoności algorytmów sortowania;
* demonstracja zależności czasu sortowania od wyglądu danych wejściowych.

Także były utworzone schematy blokowe, pseudokody i szczegółowe schematy działania obu algorytmów, wykresy porównania złożoności czasowej algorytmów i zależności czasu sortowania od pliku wejściowego.