Najważniejszy fragment kodu źródłowego

```
1.  def mergeSort(lst):
2.      if len(lst) == 1:
3.          return lst
4.      else:
5.          middle = len(lst) // 2
6.          L = lst[:middle]
7.          R = lst[middle:]
8.          mergeSort(L)
9.          mergeSort(R)
10.         """merging part"""
```

Uzasadnienie

Wybrałem powyższy fragment kodu, ponieważ pomógł mi on zrozumieć rekurencję oraz metodę dziel i zwyciężaj. Najpierw poprzez odwoływanie się do tej samej funkcji sprowadzamy problem do jego wersji podstawowej, gdzie rozwiązanie jest trywialne, czyli w tym przypadku dzielimy całą tablicę na pojedyncze elementy. Następnie systematycznie powracamy do pierwotnego stanu rzeczy rozwiązując problem dla coraz to bardziej skomplikowanej sytuacji (co raz większa ilość danych), co w efekcie prowadzi do rozwiązania początkowego problemu. Jest to klasyczny przykład rekurencji oraz metody dziel i rządź.

Podsumowanie

W przypadku sortowania przez wstawianie przechodzimy przez n elementów i każdy z nich porównujemy z poprzednio wybranymi. To znaczy, że n-ty element porównujemy z n-1 elementami dopóki nie trafi on na właściwe miejsce. Czas działania algorytmu jest sumą czasów wykonania poszczególnych instrukcji. Możemy to zatem zapisać jako c(n-1) + c(n-2)+c(n-3)…, gdzie c jest czasem wykonania pojedynczej operacji, a n ilością operacji do wykonania . Otrzymujemy zatem złożoność czasową wynoszącą O(n^2). Tak jest w przypadku średnim oraz pesymistycznym (elementy posortowane w odwrotnej kolejności). W najlepszym przypadku (lista poprawnie posortowana) każdy element porównujemy tylko z jego poprzednikiem, zatem wtedy złożoność czasowa wynosi O(n).

Sortowanie przez wstawanie jest efektywne tylko w przypadku niewielkiej ilości danych lub danych w dużej mierze już poprawnie posortowanych. Jednak w przypadku dużej liczby losowych wartości daje ono bardzo słabe rezultaty.

Ciekawszym przypadkiem jest natomiast sortowanie przez scalanie. Jest to algorytm, który prezentuje metodę dziel i rządź, będąc przy tym prostym w implementacji i jednocześnie wydajnym. Przy wywołaniu algorytmu powstaje drzewo wywołań o długości $\log_2 n$. Na każdym poziomie scalanie tablicy wymaga cn operacji. Zatem czasowa złożoność obliczeniowa tego algorytmu w każdym przypadku wynosi $cn * \log_2 n$.

Dzięki temu ćwiczeniu zrozumiałem jak istotny jest dobór odpowiedniego algorytmu do danej sytuacji. Podczas kiedy dla takiej samej ilości danych, merge sort wykonał wszystkie operacje w przeciągu kilku minut, insertion sort nie był w stanie uporać się z nimi nawet w przeciągu kilku godzin.