**Лабораторная работа №1.**

**1 часть.**

1. Сортировка пузырьком:

def bubble\_sort(arr): # O(n^2)  
 n = len(arr)  
 for i in range(n):  
 for j in range(0, n-i-1):  
 if arr[j] > arr[j+1] :  
 arr[j], arr[j+1] = arr[j+1], arr[j]  
 return arr

Рис.1.1.1 Реализация сортировки обменом

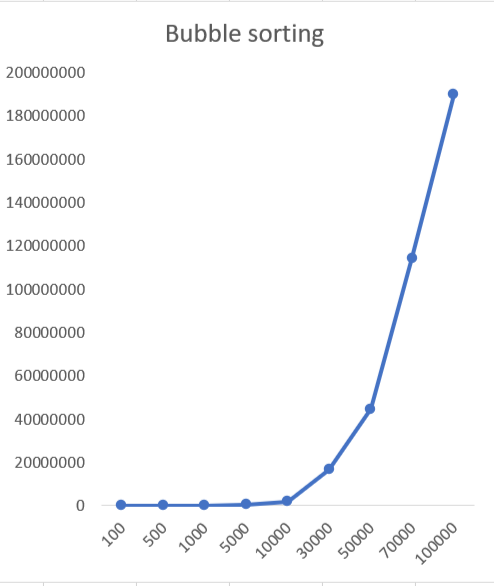
Сложность: O(n²)

Рис.1.1.2 График сложности алгоритма сортировки обменом

1. Сортировка выбором:

def selection\_sort(arr):  
 n = len(arr)  
 for i in range(n):  
 min\_idx = i  
 for j in range(i + 1, n):  
 if arr[j] < arr[min\_idx]:  
 min\_idx = j  
 arr[i], arr[min\_idx] = arr[min\_idx], arr[i]

Рис.1.2.1 Реализация сортировки выбором

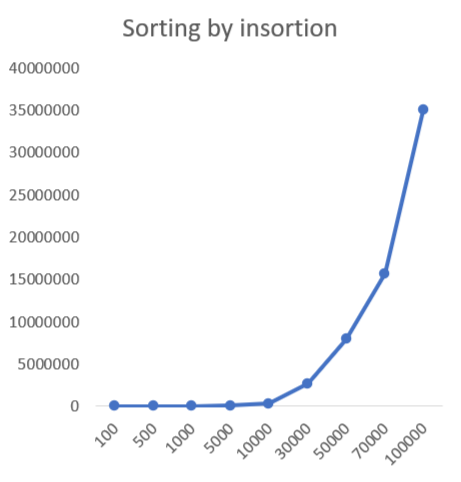
Сложность: O(n²)

Рис.1.2.2. График сложности алгоритма сортировки вставками

1. Сортировка слиянием:

def merge\_sort(arr): # O(n log n)  
 if len(arr) > 1:  
 mid = len(arr)//2  
 L = arr[:mid]  
 R = arr[mid:]  
 merge\_sort(L)  
 merge\_sort(R)  
 i = j = k = 0  
 while i < len(L) and j < len(R):  
 if L[i] < R[j]:  
 arr[k] = L[i]  
 i += 1  
 else:  
 arr[k] = R[j]  
 j += 1  
 k += 1  
 while i < len(L):  
 arr[k] = L[i]  
 i += 1  
 k += 1  
 while j < len(R):  
 arr[k] = R[j]  
 j += 1  
 k += 1  
 return arr

Рис.1.3.1. Реализация сортировки слиянием.

Сложность: O(n log(n))

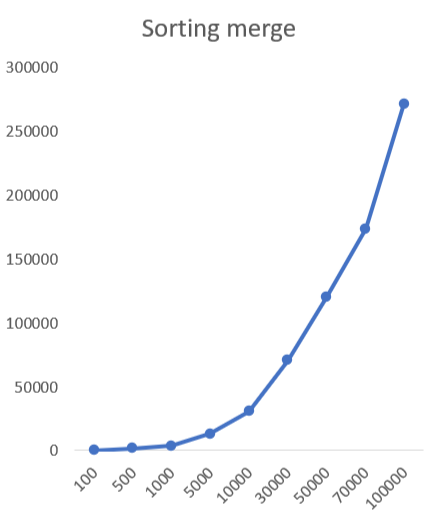


Рис.1.3.2. График сложности алгоритма сортировки вставками.

**2 часть.**

Реализация на языке Python линейного и бинарного поиска в отсортированном массиве целых чисел.

1. Линейный поиск:

def linear\_search(arr, x):  
 for i in range(len(arr)):  
 if arr[i] == x:  
 return i  
 return -1

Рис.2.1.1 Реализация линейного поиска.

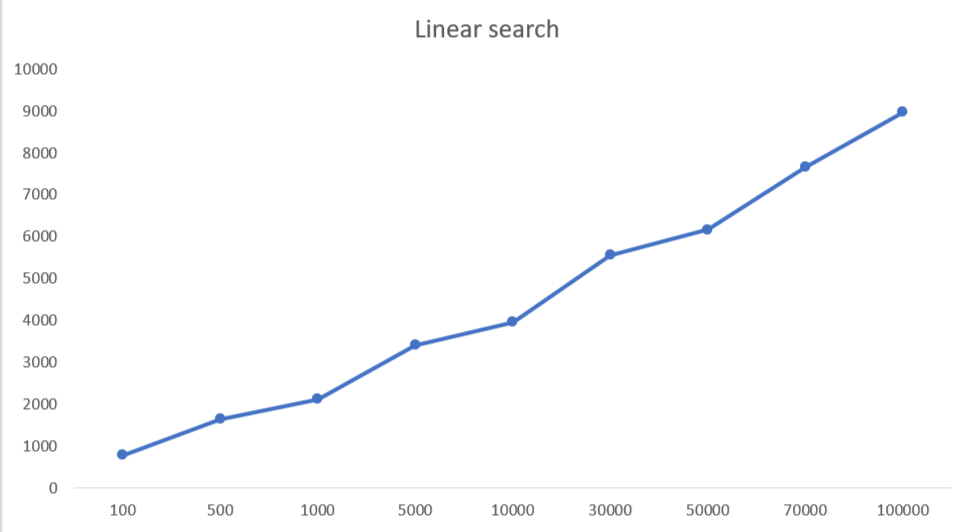
Сложность: O(n) времени, O(1) памяти.

Рис.2.1.2 График сложности линейного поиска по времени.

1. Бинарный поиск:

def binary\_search(arr, x):  
 low = 0  
 high = len(arr) - 1  
 mid = 0  
 while low <= high:  
 mid = (high + low) // 2  
 if arr[mid] < x:  
 low = mid + 1  
 elif arr[mid] > x:  
 high = mid - 1  
 else:  
 return mid  
 return -1

Рис.2.2.1 Реализация бинарного поиска.

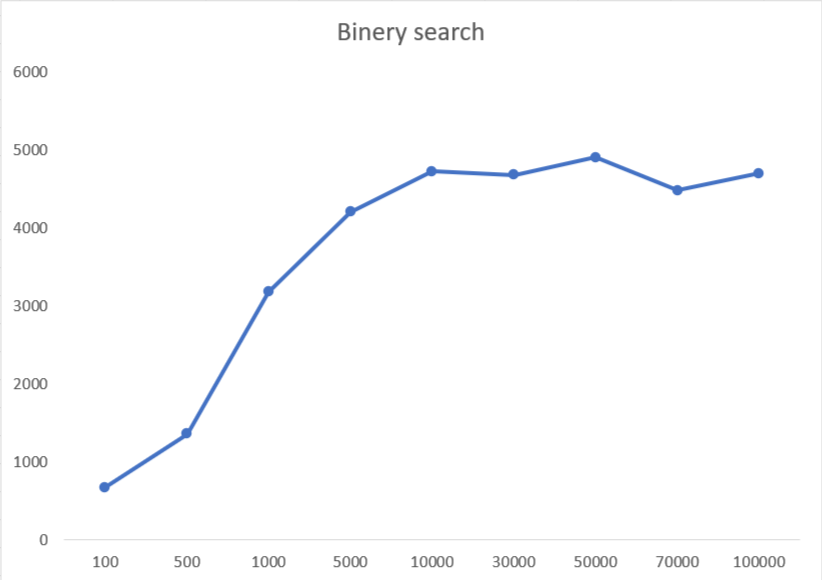
Сложность: O(log(n)) времени, O(1) памяти.

Рис.2.2.2 График сложности бинарного поиска по времени.

**Дополнительно:**

1. Функция для замера времени выполнения:

Код программы:

def measure\_time(func, \*args):  
 start\_time = time.perf\_counter()  
 func(\*args)  
 end\_time = time.perf\_counter()  
 return end\_time - start\_time

1. Измерение времени сортировки: Для каждого размера массива создается случайный массив и измеряется время выполнения трех алгоритмов сортировки: пузырьком, выбором и слиянием. Результаты выводятся в виде таблицы.

Код программы:

print("Сортировка:")  
print(f"{'Размер':<10} {'Пузырьком':<15} {'Выбором':<15} {'Слиянием':<15}")  
for size in sizes:  
 random\_array = [random.randint(0, 10000000) for \_ in range(size)]  
  
 bubble\_time = measure\_time(bubble\_sort, random\_array.copy())  
 selection\_time = measure\_time(selection\_sort, random\_array.copy())  
 merge\_time = measure\_time(merge\_sort, random\_array.copy())  
  
 print(f"{size:<10} {bubble\_time:<15.6f} {selection\_time:<15.6f} {merge\_time:<15.6f}")

1. Измерение времени поиска: для каждого размера массива создается отсортированный массив и измеряется время выполнения линейного и бинарного поиска. Результаты также выводятся в виде таблицы.

Код программы:

print("\nПоиск:")  
print(f"{'Размер':<10} {'Линейный':<15} {'Бинарный':<15}")  
for size in sizes:  
 sorted\_array = sorted([random.randint(0, 10000000) for \_ in range(size)])  
  
 linear\_search\_target = random.choice(sorted\_array)  
 linear\_time = measure\_time(linear\_search, sorted\_array, linear\_search\_target)  
  
 binary\_search\_target = random.choice(sorted\_array)  
 binary\_time = measure\_time(binary\_search, sorted\_array, binary\_search\_target)  
  
 print(f"{size:<10} {linear\_time:<15.6f} {binary\_time:<15.6f}")

**Вывод**: графики, построенные за счёт данных, полученных экспериментально, похожи на зависимости от времени, которые должны получиться.