Практическая работа №2

Тема: Алгоритмы сортировки.

Цель работы: Изучить алгоритмы сортировки, построить графики зависимости времени от сортировки массивов различной длины.

Алгоритм сортировки — это алгоритм для упорядочивания элементов в списке. В случае, когда элемент списка имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма.

Алгоритмы сортировки оцениваются по скорости выполнения и эффективности использования памяти:

Время — основной параметр, характеризующий быстродействие алгоритма. Называется также вычислительной сложностью. Для упорядочения важны худшее, среднее и лучшее поведение алгоритма в терминах мощности входного множества A. Если на вход алгоритму подаётся множество A, то обозначим n = |A|. Для типичного алгоритма хорошее поведение — это O(n log n) и плохое поведение — это O(n2). Идеальное поведение для упорядочения — O(n). Алгоритмы сортировки, использующие только абстрактную операцию сравнения ключей всегда нуждаются по меньшей мере в сравнениях. Тем не менее, существует алгоритм сортировки Хана (Yijie Han) с вычислительной сложностью O(n log log n log log log n), использующий тот факт, что пространство ключей ограничено (он чрезвычайно сложен, а за O-обозначением скрывается весьма большой коэффициент, что делает невозможным его применение в повседневной практике). Также существует понятие сортирующих сетей. Предполагая, что можно одновременно (например, при параллельном вычислении) проводить несколько сравнений, можно отсортировать n чисел за O(log2 n) операций. При этом число n должно быть заранее известно;

Память — ряд алгоритмов требует выделения дополнительной памяти под

временное хранение данных. Как правило, эти алгоритмы требуют O(log n) памяти. При оценке не учитывается место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы (так как всё это потребляет O(1)). Алгоритмы сортировки, не потребляющие дополнительной памяти, относят к сортировкам на месте.

Сортировка выбором

Алгоритм сортировки выбором заключается в поиске на необработанном срезе массива или списка минимального значения и в дальнейшем обмене этого значения с первым элементом необработанного среза. На следующем шаге необработанный срез уменьшается на один элемент.

1. Найти наименьшее значение в списке.

2. Записать его в начало списка, а первый элемент - на место, где раньше стоял наименьший.

3. Снова найти наименьший элемент в списке. При этом в поиске не участвует первый элемент.

4. Второй минимум поместить на второе место списка. Второй элемент при этом перемещается на освободившееся место.

5. Продолжать выполнять поиск и обмен, пока не будет достигнут конец списка.

Алгоритм сортировки выбором:

```
#Сортировка выбором:
def select_sort(nums):
    for i in range(len(nums) - 1):
        m = i
        j = i + 1
        while j < len(nums):
            if nums[j] < nums[m]:
                m = j
            j = j + 1
        nums[i], nums[m] = nums[m], nums[i]
    return nums
```

Диаграмма деятельности для сортировки выбором представлена на рисунке 1.

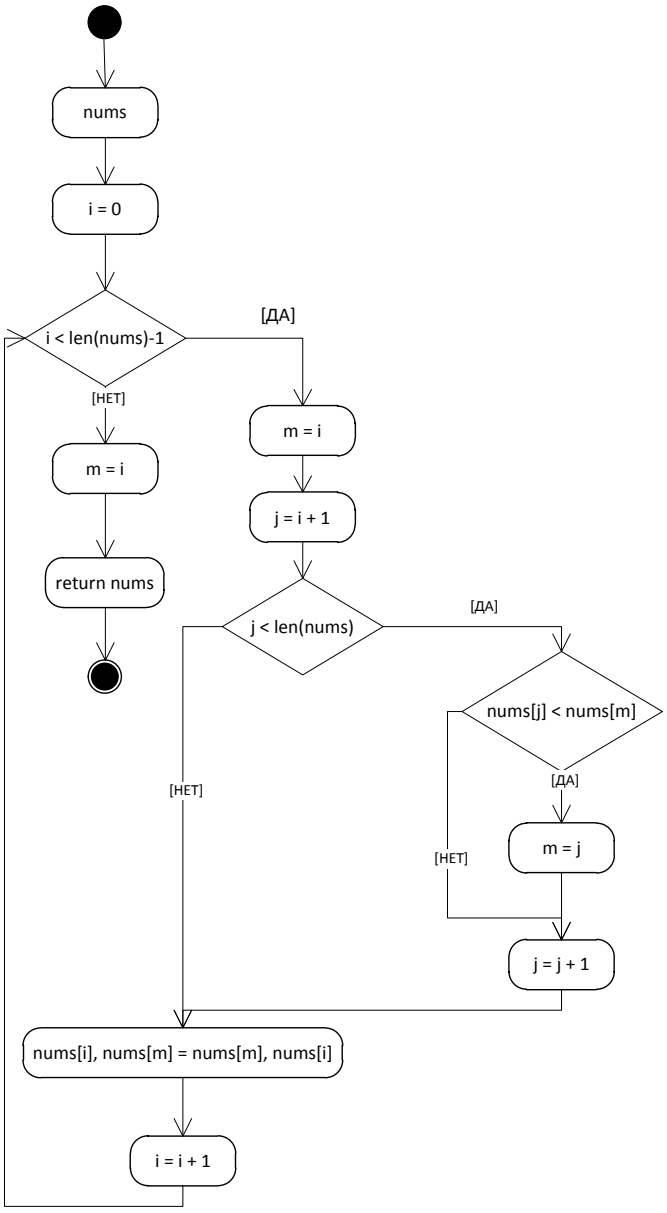

Рисунок 1 – Диаграмма деятельности для сортировки выбором.

Сортировка вставками.

Как и сортировка выборкой, этот алгоритм сегментирует список на две части: отсортированную и неотсортированную. Алгоритм перебирает второй сегмент и вставляет текущий элемент в правильную позицию первого сегмента.

Алгоритм. Предполагается, что первый элемент списка отсортирован. Переходим к следующему элементу, обозначим его x. Если x больше первого, оставляем его на своём месте. Если он меньше, копируем его на вторую позицию, а x устанавливаем как первый элемент. Переходя к другим элементам несортированного сегмента, перемещаем более крупные элементы в отсортированном сегменте вверх по списку, пока не встретим элемент меньше x

или не дойдём до конца списка. В первом случае x помещается на правильную позицию.

Алгоритм сортировки вставками:

```
#Сортировка вставкой:
#iti - элемент для вставки
def insert_sort(nums):
    for i in range(1, len(nums)):
        iti = nums[i]
        j = i - 1
        while j >= 0 and nums[j] > iti:
            nums[j + 1] = nums[j]
            j -= 1
        nums[j + 1] = iti
    return nums
```

Диаграмма деятельности для сортировки вставками представлена на рисунке 2.

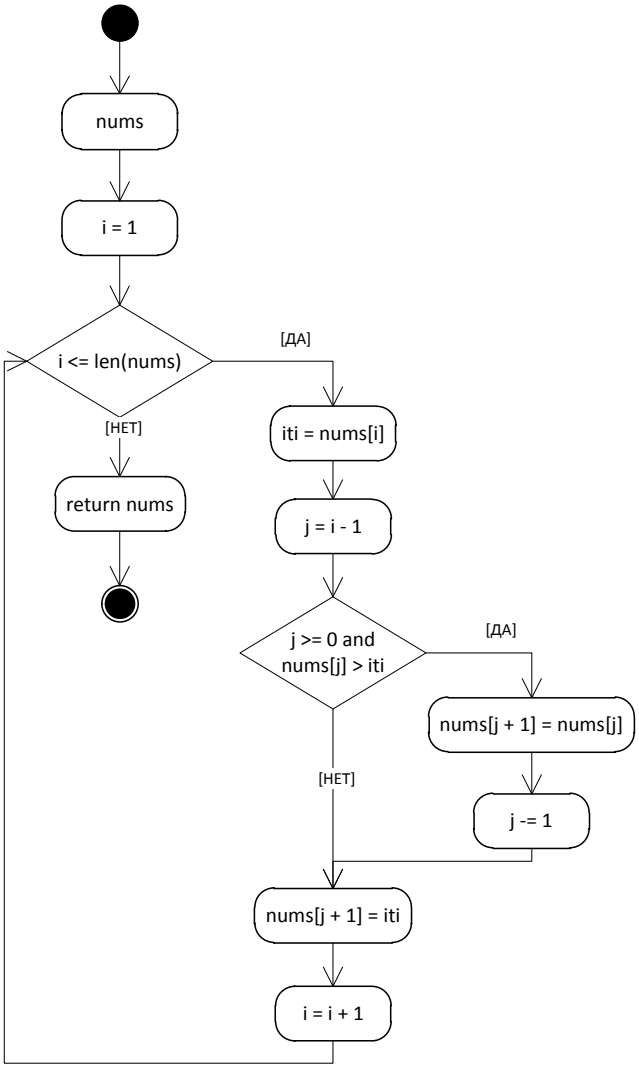

Рисунок 2 – Диаграмма деятельности для сортировки вставками.

Сортировка Шелла.

Сортировка Шелла – алгоритм сортировки, являющийся усовершенствованным вариантом сортировки вставками. Идея метода Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга. Иными словами – это сортировка вставками с предварительными «грубыми» проходами. Аналогичный метод усовершенствования пузырьковой сортировки называется сортировка расчёской.

При сортировке Шелла сначала сравниваются и сортируются между собой значения, стоящие один от другого на некотором расстоянии d. После этого процедура повторяется для некоторых меньших значений d, а завершается сортировка Шелла упорядочиванием элементов при d=1 (то есть обычной сортировкой вставками). Эффективность сортировки Шелла в определённых случаях обеспечивается тем, что элементы «быстрее» встают на свои места (в простых методах сортировки, например, пузырьковой, каждая перестановка двух элементов уменьшает количество инверсий в списке максимум на 1, а при сортировке Шелла это число может быть больше).

Невзирая на то, что сортировка Шелла во многих случаях медленнее, чем быстрая сортировка, она имеет ряд преимуществ:

– отсутствие потребности в памяти под стек;

– отсутствие деградации при неудачных наборах данных – быстрая сортировка легко деградирует до $O(n^2)$, что хуже, чем худшее гарантированное время для сортировки Шелла.

Алгоритм сортировки Шелла:

```
#Сортировка Шелла:
def shell_sort(nums):
    o = len(nums) // 2
    while o != 0:
        for id, el in enumerate(nums):
            while id >= o and nums[id - o] > el:
                nums[id] = nums[id - o]
                id -= o
            nums[id] = el
        o = 1 if o == 2 else int(o * 5.0 / 11)
    return nums
```

Диаграмма деятельности для сортировки Шелла представлена на рисунке 3

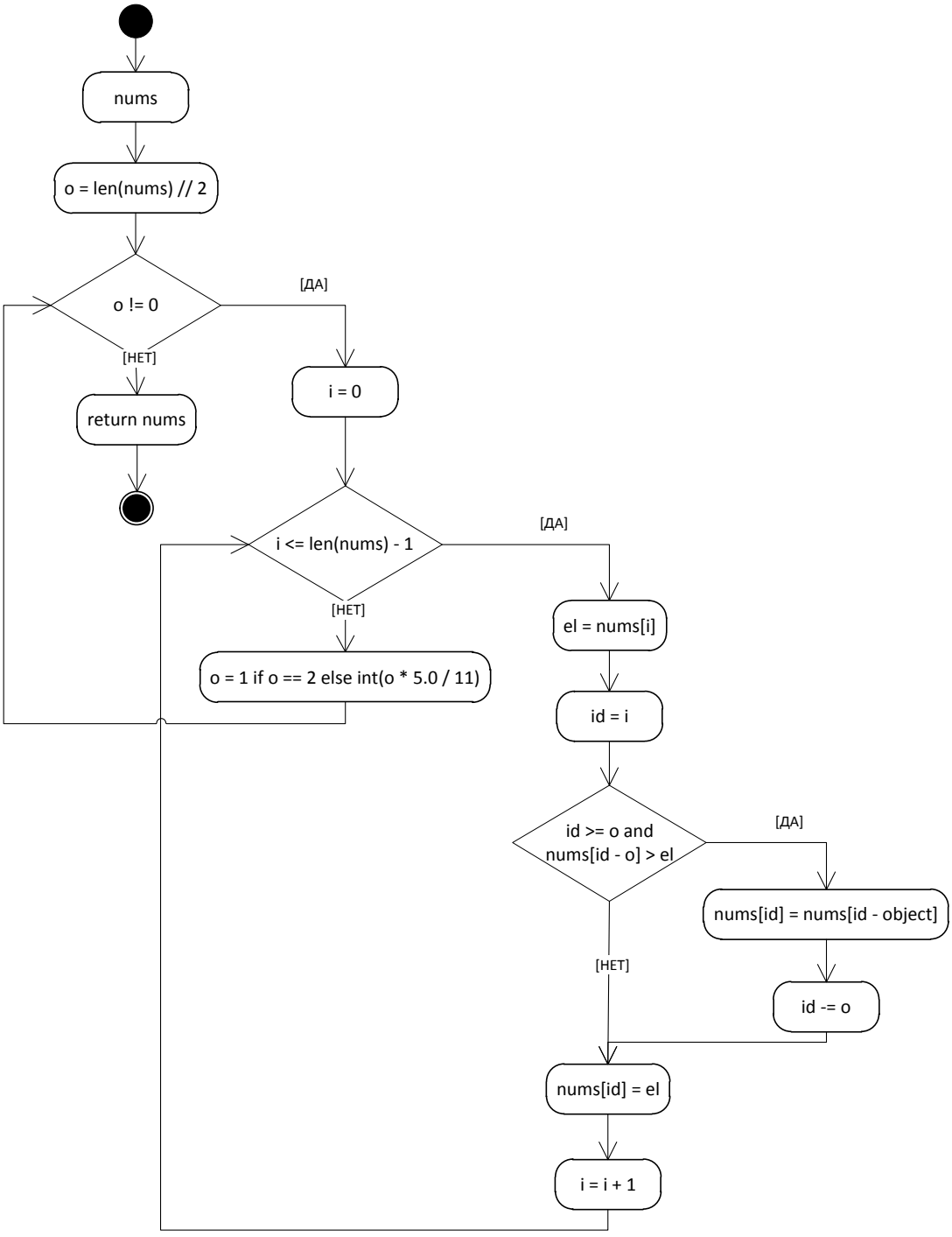

Рисунок 3 – Диаграмма деятельности для сортировки Шелла.

Сортировка пузырьком.

Сортировка пузырьком - это метод сортировки массивов и списков путем последовательного сравнения и обмена соседних элементов, если предшествующий оказывается больше последующего. В процессе выполнения данного алгоритма элементы с большими значениями оказываются в конце

списка, а элементы с меньшими значениями постепенно перемещаются по направлению к началу списка. Образно говоря, тяжелые элементы падают на дно, а легкие медленно всплывают подобно пузырькам воздуха.

В сортировке методом пузырька количество итераций внешнего цикла определяется длинной списка минус единица, так как когда второй элемент становится на свое место, то первый уже однозначно минимальный и находится на своем месте.

Количество итераций внутреннего цикла зависит от номера итерации внешнего цикла, так как конец списка уже отсортирован, и выполнять проход по этим элементам смысла нет.

Алгоритм сортировки пузырьком:

```
#Пузырьковая сортировка:
def bubl_sort(nums):
    n = len(nums)
    for i in range(n-1):
        for j in range(n-i-1):
            if nums[j] > nums[j+1]:
                nums[j], nums[j+1] = nums[j+1], nums[j]
    return nums
```

Диаграмма деятельности для сортировки пузырьком представлена на рисунке 4.

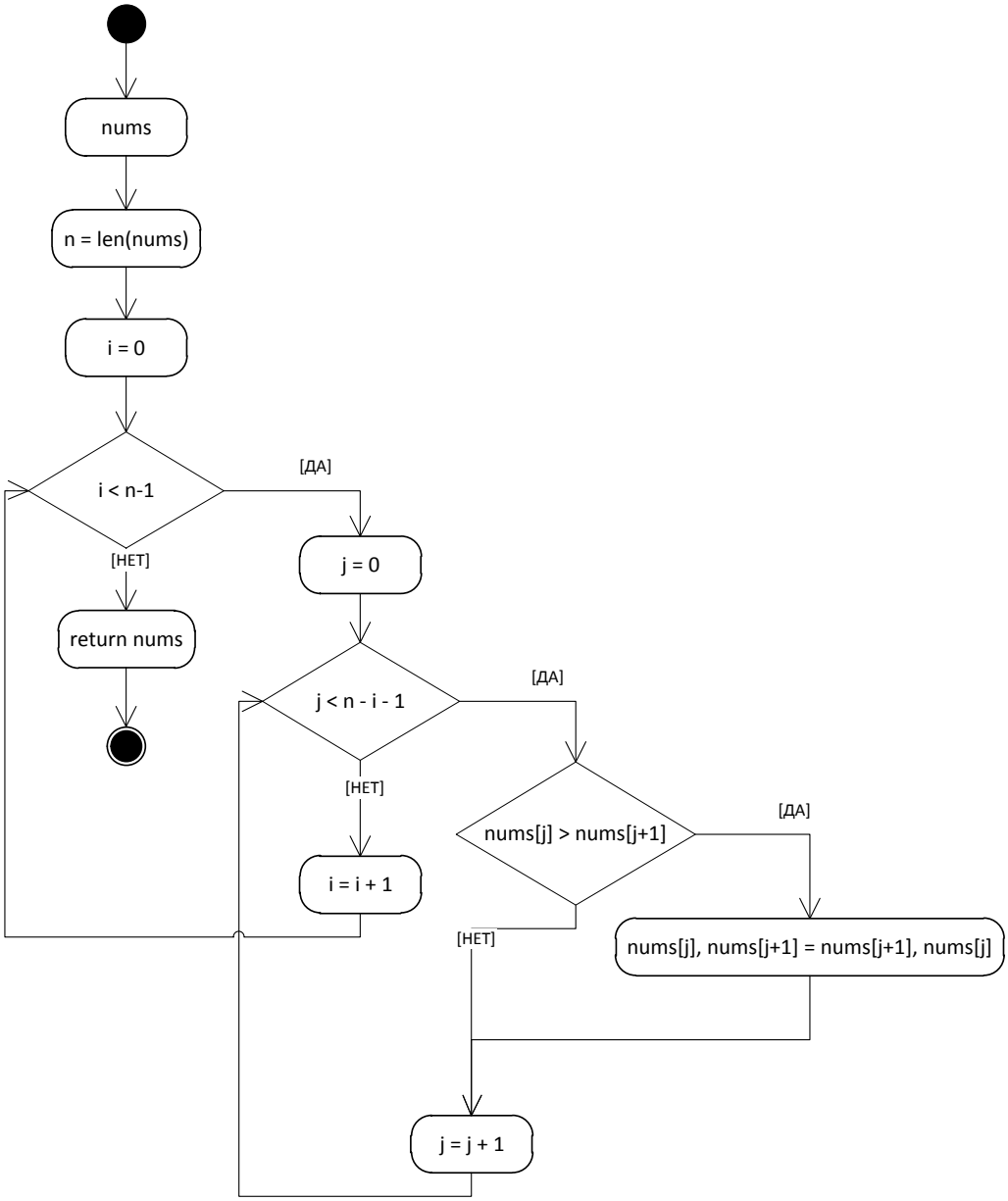

Рисунок 4 – Диаграмма деятельности для сортировки пузырьком.

Быстрая сортировка.

Быстрая сортировка – популярный алгоритм сортировки, который часто используется вместе с сортировкой слиянием. Это алгоритм является хорошим примером эффективного алгоритма сортировки со средней сложностью O(n logn). Часть его популярности еще связана с простотой реализации.

Быстрая сортировка является представителем трех типов алгоритмов: divide and conquer (разделяй и властвуй), in-place (на месте) и unstable (нестабильный).

– Divide and conquer: Быстрая сортировка разбивает массив на меньшие массивы до тех пор, пока он не закончится пустым массивом, или массивом, содержащим только один элемент, и затем все рекурсивно соединяется в сортированный большой массив.

– In place: Быстрая сортировка не создает никаких копий массива или его подмассивов. Однако этот алгоритм требует много стековой памяти для всех рекурсивных вызовов, которые он делает.

– Unstable: стабильный (stable) алгоритм сортировки – это алгоритм, в котором элементы с одинаковым значением отображаются в том же относительном порядке в отсортированном массиве, что и до сортировки массива. Нестабильный алгоритм сортировки не гарантирует этого, это, конечно, может случиться, но не гарантируется. Это может быть важным, когда вы сортируете объекты вместо примитивных типов. Например, представьте, что у вас есть несколько объектов Person одного и того же возраста, например, Дейва в возрасте 21 года и Майка в возрасте 21 года. Если бы вы использовали Quicksort в коллекции, содержащей Дейва и Майка, отсортированных по возрасту, нет гарантии, что Дейв будет приходить раньше Майка каждый раз, когда вы запускаете алгоритм, и наоборот.

Когда мы описываем элементы как «больше» или «меньше», чем другой элемент – это не обязательно означает большие или меньшие целые числа, мы можем отсортировать по любому выбранному нами свойству. К примеру, если у нас есть пользовательский класс Person, и у каждого человека есть имя и возраст, мы можем сортировать по имени (лексикографически) или по возрасту (по возрастанию или по убыванию).

Алгоритм сортировки Хоара:

```
#Быстрая сортировка - метод Хоара:
def xoar_sort(nums):
    if len(nums) <= 1:
        return nums
    else:
        q = random.choice(nums)
        minor_nums = [n for n in nums if n < q]
        equal_nums = [q] * nums.count(q)
        large_nums = [n for n in nums if n > q]
```

```
        return xoar_sort(minor_nums) + equal_nums + xoar_sort(la
rge_nums)
```

Диаграмма деятельности для сортировки Хоара представлена на рисунке 5.

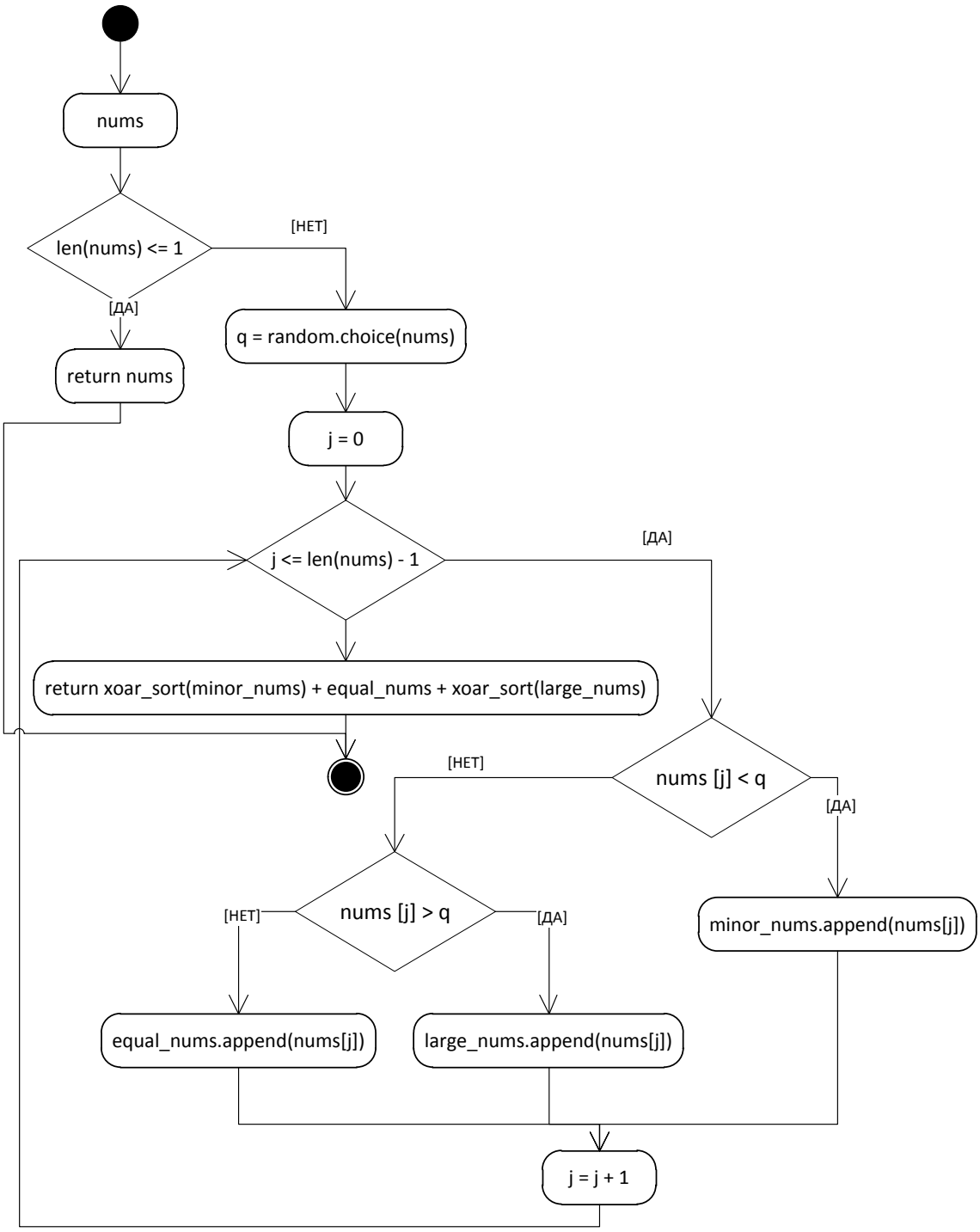

Рисунок 4 – Диаграмма деятельности для сортировки Хоара.

Работоспособность программы при задачи любого количества чисел массива с клавиатуры представлена на рисунке 5.

```
Введите число элементов в массиве (от 1), n = 5
Начальный массив
[71663, 65904, 14883, 17057, 61267]
Отсортированный массив методом выбора
[14883, 17057, 61267, 65904, 71663]
Отсортированный массив методом вставками
[14883, 17057, 61267, 65904, 71663]
Отсортированный массив методом Шейкера
[14883, 17057, 61267, 65904, 71663]
Отсортированный массив методом Шелла
[14883, 17057, 61267, 65904, 71663]
Отсортированный массив методом пузырьков
[14883, 17057, 61267, 65904, 71663]
Отсортированный массив методом Хоар
[14883, 17057, 61267, 65904, 71663]
```

Рисунок 5 – Работоспособность программы.

Для анализа быстроты работы алгоритмов сортировки, были построены гистограммы зависимостей времени (микросекунды) от длины массива. Были взяты четыре массива с размерностями 10, 100, 1000, 10000. Гистограмма, представленная на рисунке 6, отображает зависимость времени от массива с 10 элементами.

Рисунок 6 – Зависимость времени от массива с 10 элементами.

Гистограмма, представленная на рисунке 7, отображает зависимость времени от массива с 100 элементами.

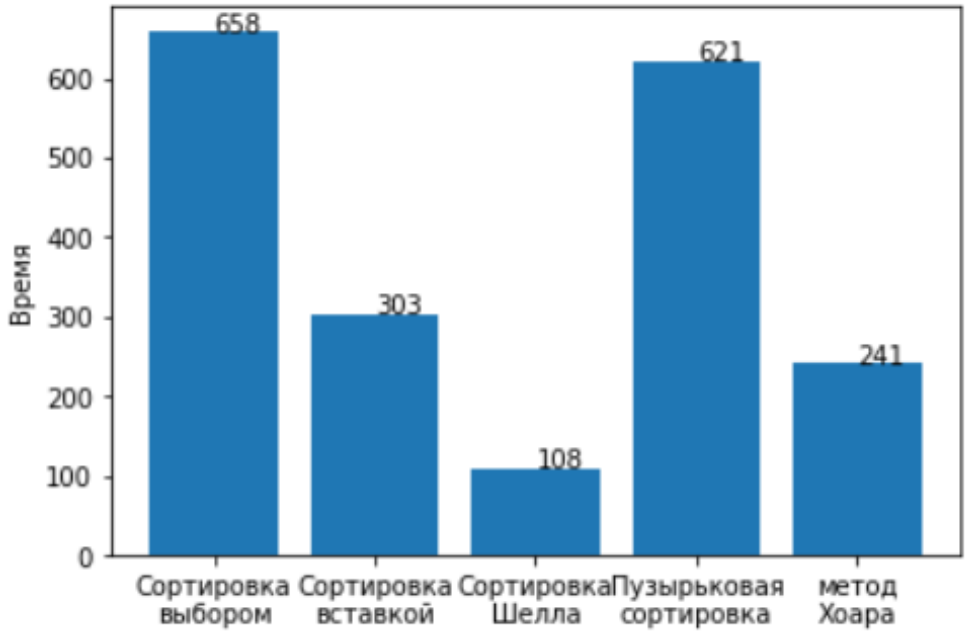

Рисунок 7 – Зависимость времени от массива с 100 элементами.

Гистограмма, представленная на рисунке 8, отображает зависимость времени от массива с 1000 элементами.

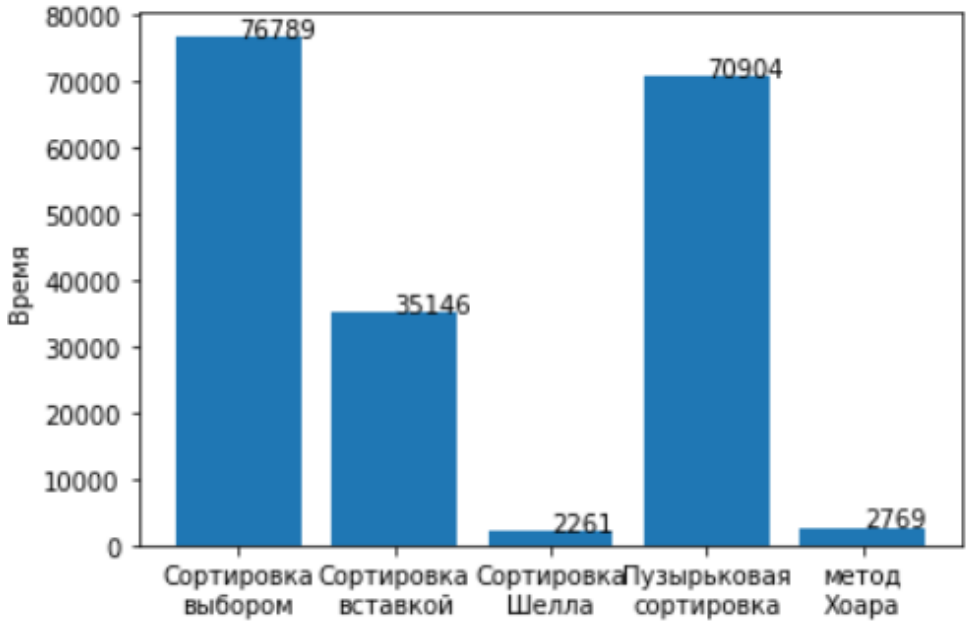

Рисунок 8 – Зависимость времени от массива с 1000 элементами.

Гистограмма, представленная на рисунке 9, отображает зависимость времени от массива с 10000 элементами.

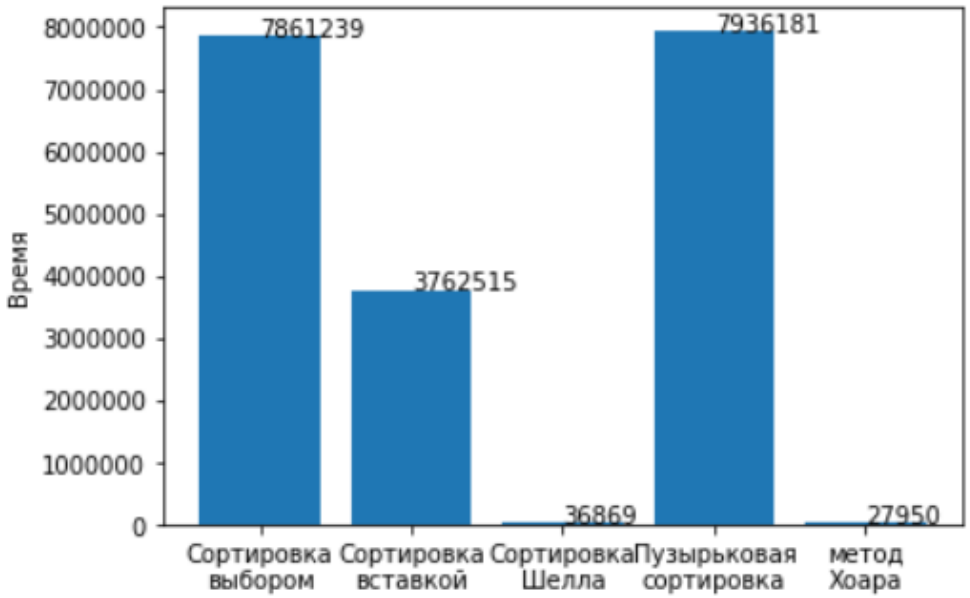

Рисунок 9 – Зависимость времени от массива с 10000 элементами.

Вывод. Мы изучили алгоритмы сортировок (выбором, вставкой, Шелла, пузырьковой, Хоара) и построили графики зависимости времени от длины массива.