Практическая работа №2

Тема: «Алгоритмы сортировки»

Цель работы: изучить алгоритмы сортировки

Сортировка простыми обменами, сортиро́вка пузырько́м (англ. bubble sort) — простой алгоритм сортировки. Для понимания и реализации этот алгоритм — простейший, но эффективен он лишь для небольших массивов. Сложность алгоритма: $O(n^2)$.

Алгоритм считается учебным и практически не применяется вне учебной литературы, вместо него на практике применяются более эффективные алгоритмы сортировки. В то же время метод сортировки обменами лежит в основе некоторых более совершенных алгоритмов, таких как шейкерная сортировка, пирамидальная сортировка и быстрая сортировка.

Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются N-1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива («всплывает» до нужной позиции, как пузырёк в воде — отсюда и название алгоритма).

Сложность: $O(n^2)$.

Листинг сортировки пузырьком на языке Python приведен ниже в Листинге 1. Зависимость времения сортировки от количества элементов представлена на рисунке 1, диаграмма деятельности для алгоритма представлена на рисунке 2.

Листинг 1. «Пузырек».

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

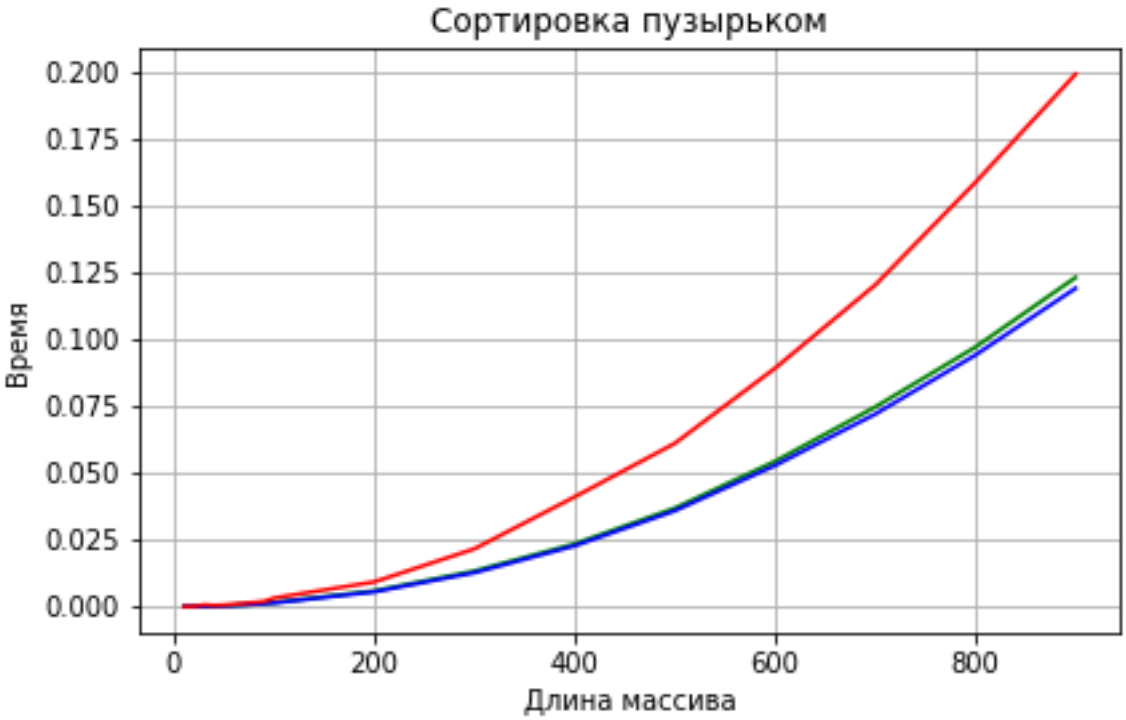

Рисунок 1 - Зависимость времени от количества элементов

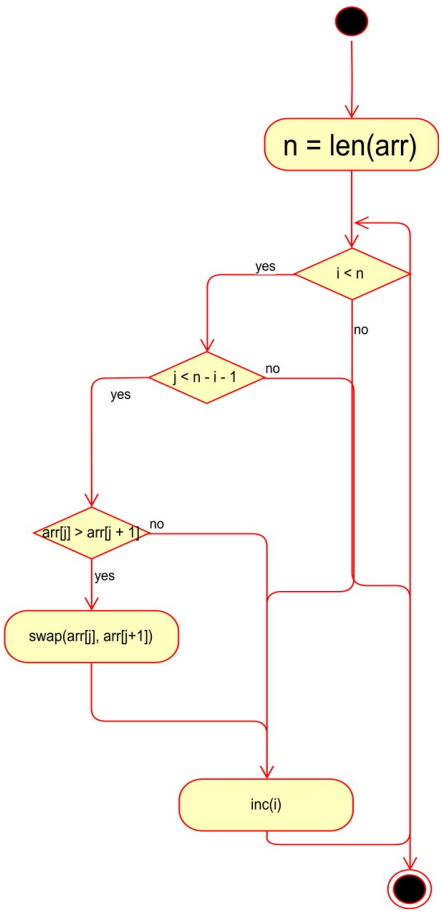

Рисунок 2 - Диаграмма деятельности

Гномья сортировка (англ. *Gnome sort*) — алгоритм сортировки, похожий на сортировку вставками, но в отличие от последней перед вставкой на нужное место происходит серия обменов, как в сортировке пузырьком. Название происходит от предполагаемого поведения садовых гномов при сортировке линии садовых горшков.

> Гномья сортировка основана на технике, используемой обычным голландским садовым гномом (нидерл. *tuinkabouter*). Это метод, которым садовый гном сортирует линию цветочных горшков. По существу он смотрит на текущий и предыдущий садовые горшки: если они в правильном порядке, он шагает на один горшок вперёд, иначе он меняет их местами и шагает на один горшок назад. Граничные условия: если нет предыдущего горшка, он шагает вперёд; если нет следующего горшка, он закончил.
> Дик Грун

Алгоритм концептуально простой, не требует вложенных циклов. Время работы . На практике алгоритм может работать так же быстро, как и сортировка вставками.

Алгоритм находит первое место, где два соседних элемента стоят в неправильном порядке и меняет их местами. Он пользуется тем фактом, что обмен может породить новую пару, стоящую в неправильном порядке, только до или после переставленных элементов. Он не допускает, что элементы после текущей позиции отсортированы, таким образом, нужно только проверить позицию до переставленных элементов.

Листинг гномьей сортировки приведен в листинге 2, зависимость времени выполнения от количества элементов представлено на рисунке 3, диаграмма деятельности представлена на рисунке 4.

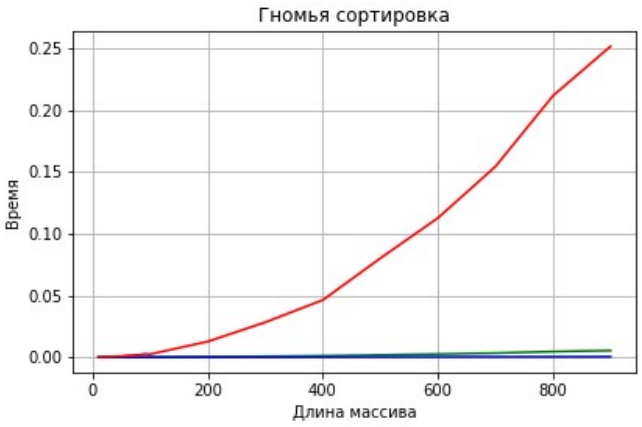

Рисунок 3 - Зависимость времени от количества элементов

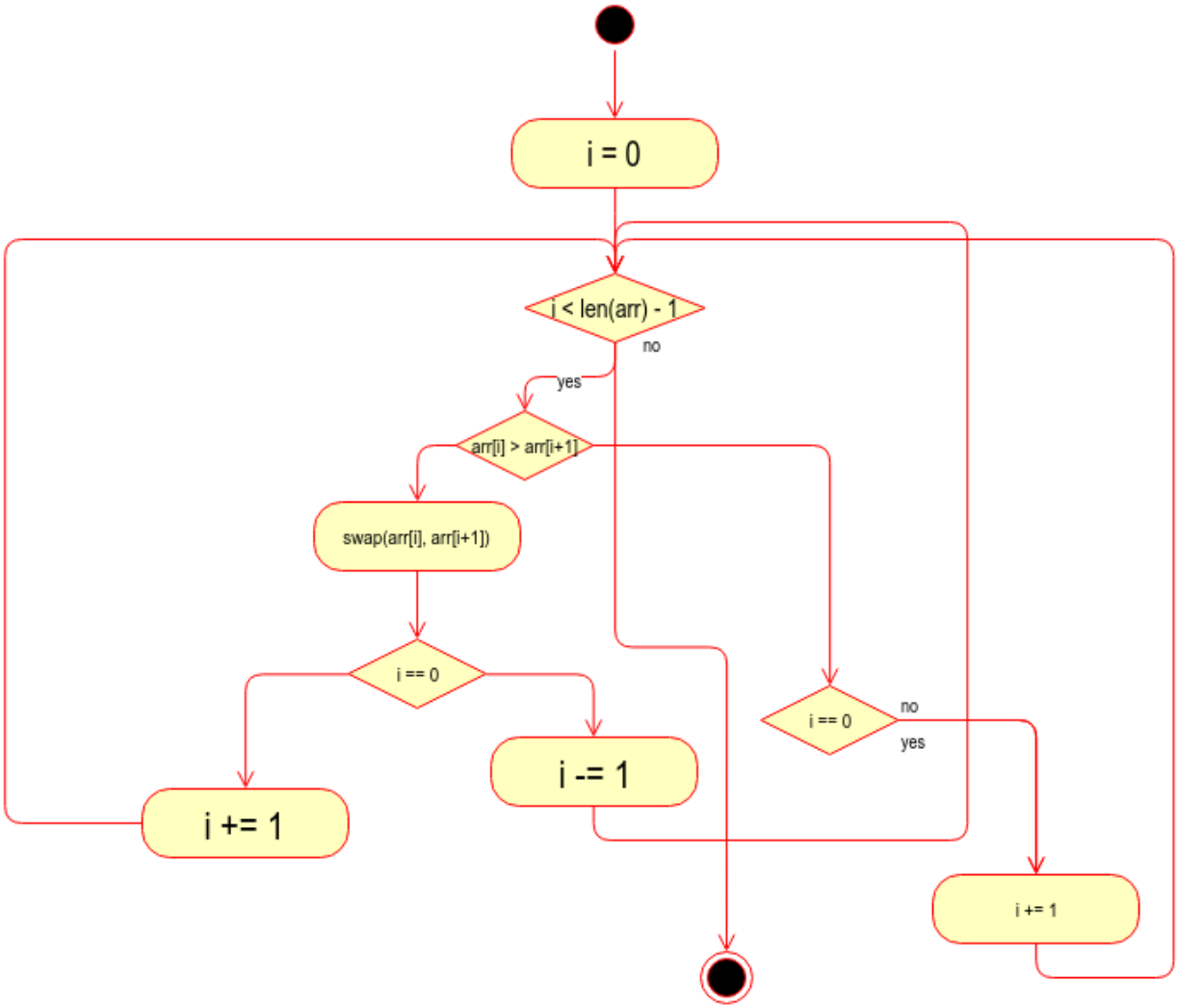

Рисунок 4 - Диаграмма деятельности.

Листинг 2. Гномья сортировка.

```
def gnome_sort(arr):
    i = 0
    while i < len(arr)-1:
        if arr[i] > arr[i+1]:
            arr[i], arr[i+1] = arr[i+1], arr[i]
            if i == 0:
                i += 1
            else:
                i -= 1
        elif i == 0:
            i += 1
        else:
            i += 1
    return arr
```

Сортировка перемешиванием, или Шейкерная сортировка, или двунаправленная (англ. Cocktail sort) — разновидность пузырьковой сортировки. Анализируя метод пузырьковой сортировки, можно отметить два обстоятельства.

Во-первых, если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, её можно исключить из рассмотрения.

Во-вторых, при движении от конца массива к началу минимальный элемент «всплывает» на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо.

Эти две идеи приводят к следующим модификациям в методе пузырьковой сортировки. Границы рабочей части массива (то есть части массива, где происходит движение) устанавливаются в месте последнего обмена на каждой итерации. Массив просматривается поочередно справа налево и слева направо.

Листинг шейкерной сортировки на языке Python представлен в Листинге 3, зависимость времени сортировки от времени представлена на рисунке 5, диаграма деятельности на рисунке 6.

Листинг 3. Коктельная сортиро́вка.

```
def coctail_sort(arr):
    left = 0
    right = len(arr) - 1
    while left < right:
        for i in range(left, right, +1):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
        right -= 1

        for i in range(right, left, -1):
            if arr[i - 1] > arr[i]:
                arr[i - 1], arr[i] = arr[i], arr[i - 1]
        left += 1
return arr
```

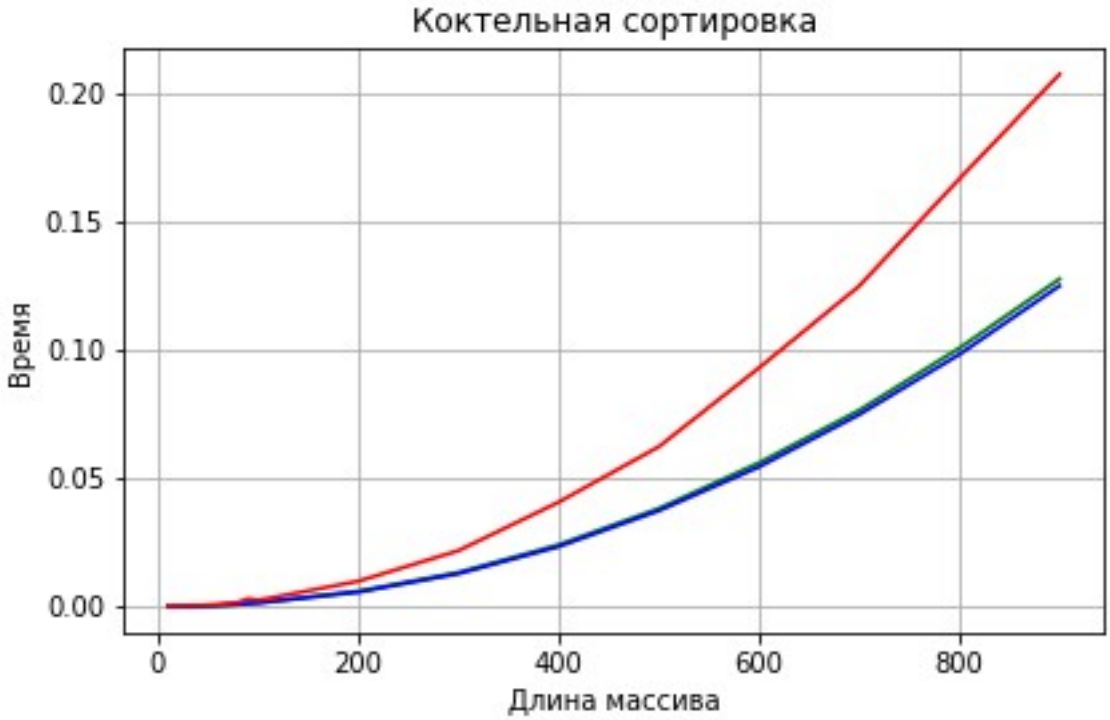

Рисунок 5 - Зависимость времени сортировки от количества элементов

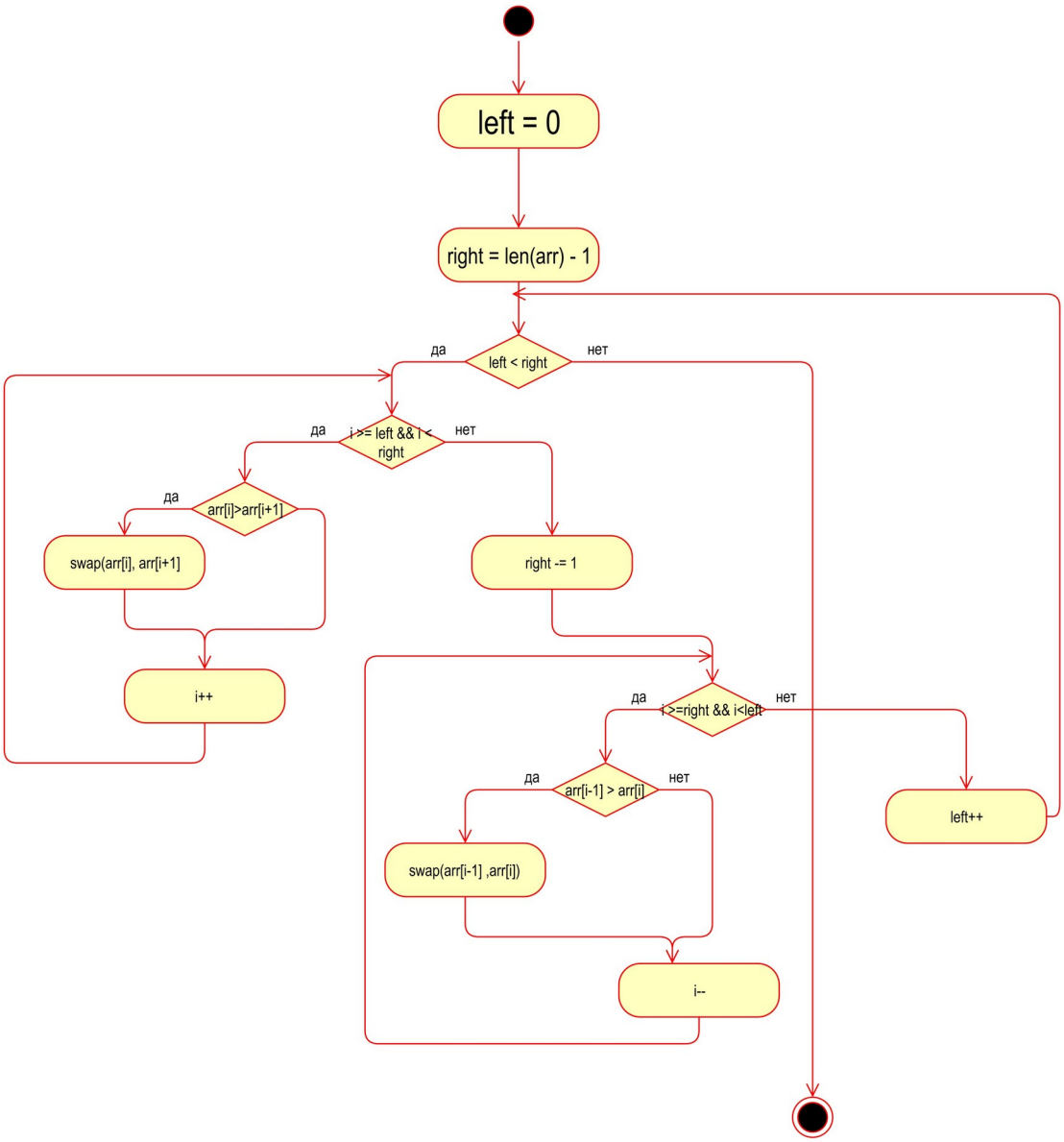

Рисунок 6 - Диаграмма деятельности

Сортировка вставками (англ. Insertion sort) — алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов. Вычислительная сложность — $0(n^2)$.

Время выполнения алгоритма зависит от входных данных: чем большее множество нужно отсортировать, тем большее время потребуется для выполнения сортировки. Также на время выполнения влияет исходная упорядоченность массива. Время работы алгоритма для различных входных данных одинакового размера зависит от элементарных операций, или шагов, которые потребуется выполнить.

Временная сложность алгоритма — $0(n^2)$. Однако, из-за константных множителей и членов более низкого порядка алгоритм с более высоким порядком роста может выполняться для небольших входных данных быстрее, чем алгоритм с более низким порядком роста.

Зависимость времени выполнения от количества элементов приведена на рисунке 7, диаграмма деятельности на рисунке 8, листинг в листинге 4.

Листинг 4. Сортировка вставками.

```
def insertion_sort(arr):
    for j in range(1, len(arr)):
        key = arr[j]
        i = j - 1
        while i > -1 and arr[i] > key:
            arr[i+1] = arr[i]
            i -= 1
        arr[i + 1] = key
    return arr
```

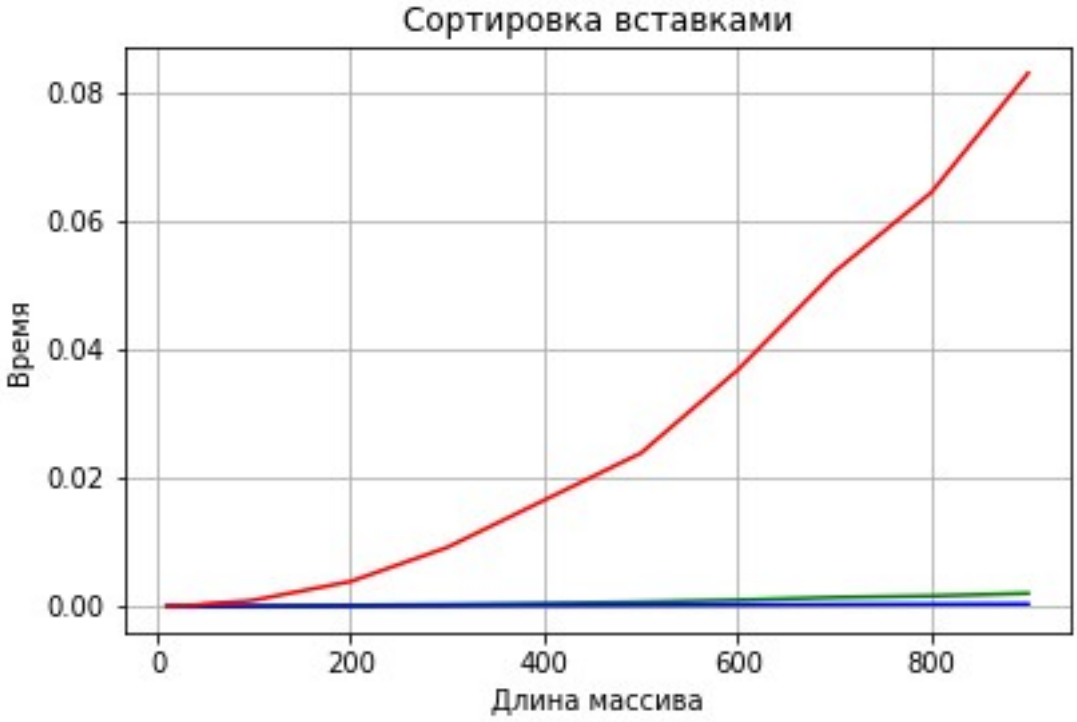

Рисунок 7 - Зависимость времени выполнения от количества элементов

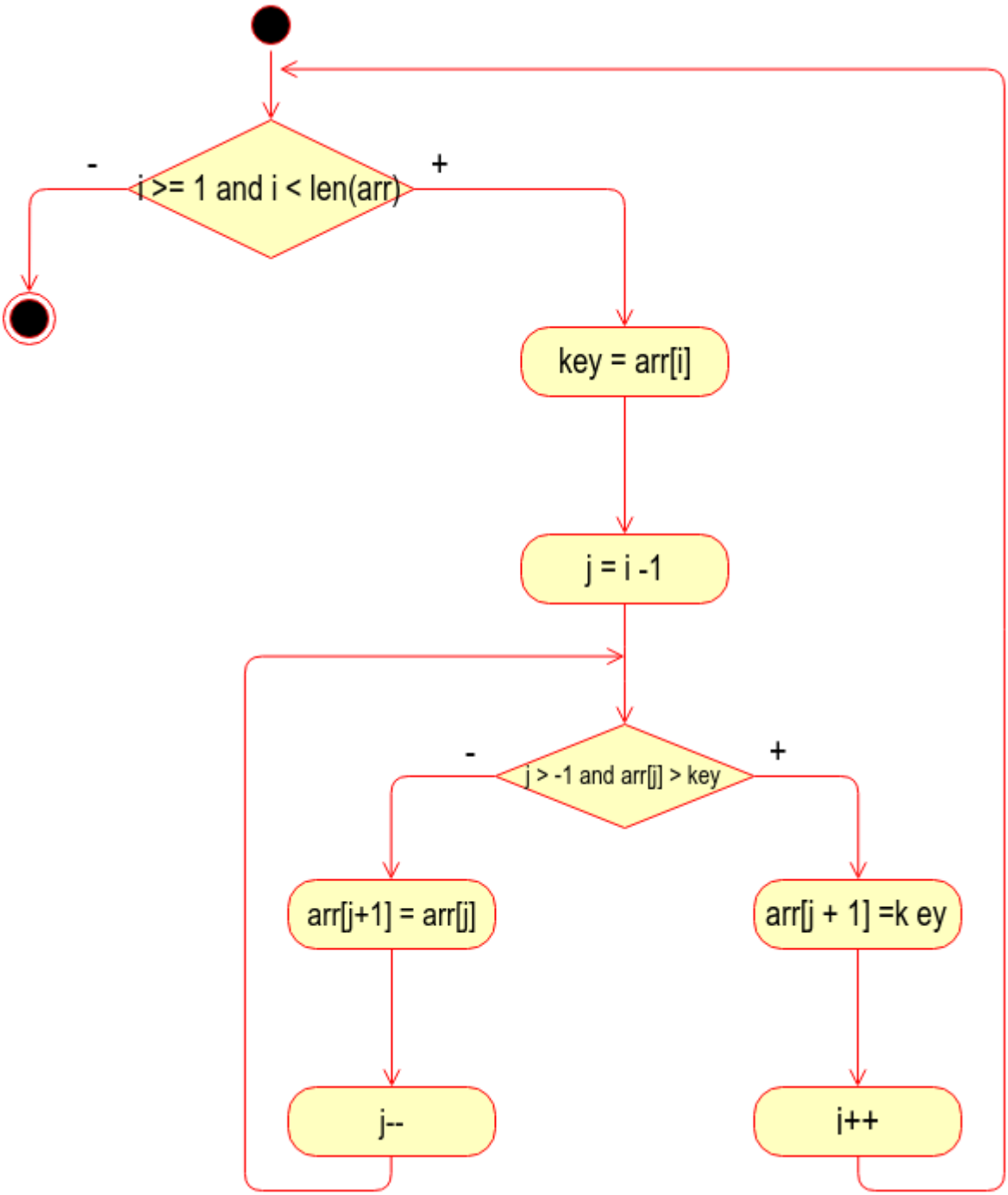

Рисунок 8 - Диаграмма деятельности

Сортировка слиянием (англ. merge sort) — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

Алгоритм был изобретён Джоном фон Нейманом в 1945 году.

В приведённом алгоритме на C++-подобном языке используется проверка на равенство двух сравниваемых элементов подмассивов с отдельным блоком обработки в случае равенства. Отдельная проверка на равенство удваивает число сравнений, что усложняет код программы. Вместо отдельной проверки на равенство и отдельного блока обработки в случае равенства можно использовать две проверки if(L <= R) и if(L >= R), что почти вдвое уменьшает код программы.

Число проверок можно сократить вдвое убрав проверку if(L >= R). При этом, в случае равенства L и R, L запишется в Out в первой итерации, а R - во второй. Этот вариант будет работать эффективно, если в исходном массиве повторяющиеся элементы не будут преобладать над остальными элементами.

Время работы алгоритма порядка O(n * log n) при отсутствии деградации на неудачных случаях, которая является больным местом быстрой сортировки (тоже алгоритм порядка O(n * log n), но только для среднего случая). Расход памяти выше, чем для быстрой сортировки, при намного более благоприятном паттерне выделения памяти — возможно выделение одного региона памяти с самого начала и отсутствие выделения при дальнейшем исполнении.

Листинг представлен в листинге 5, зависимость времени выполнения от количества элементов на рисунке 9, диаграмма деятельности на рисунке 10.

Листинг 5.Сортировка слиянием.

```
def merge_sort(arr):
    n = len(arr)
    if n < 2:
        return arr
    l = merge_sort(arr[:n // 2])
    r = merge_sort(arr[n // 2:n])
    i = j = 0
    res = []
    while i < len(l) or j < len(r):
        if not i < len(l):
            res.append(r[j])
            j += 1
        elif not j < len(r):
            res.append(l[i])
            i += 1
        elif l[i] < r[j]:
            res.append(l[i])
            i += 1
        else:
```

```
            res.append(r[j])
            j += 1
    return res
```

Рисунок 9 - Зависимость времени выполнения от количества элементов

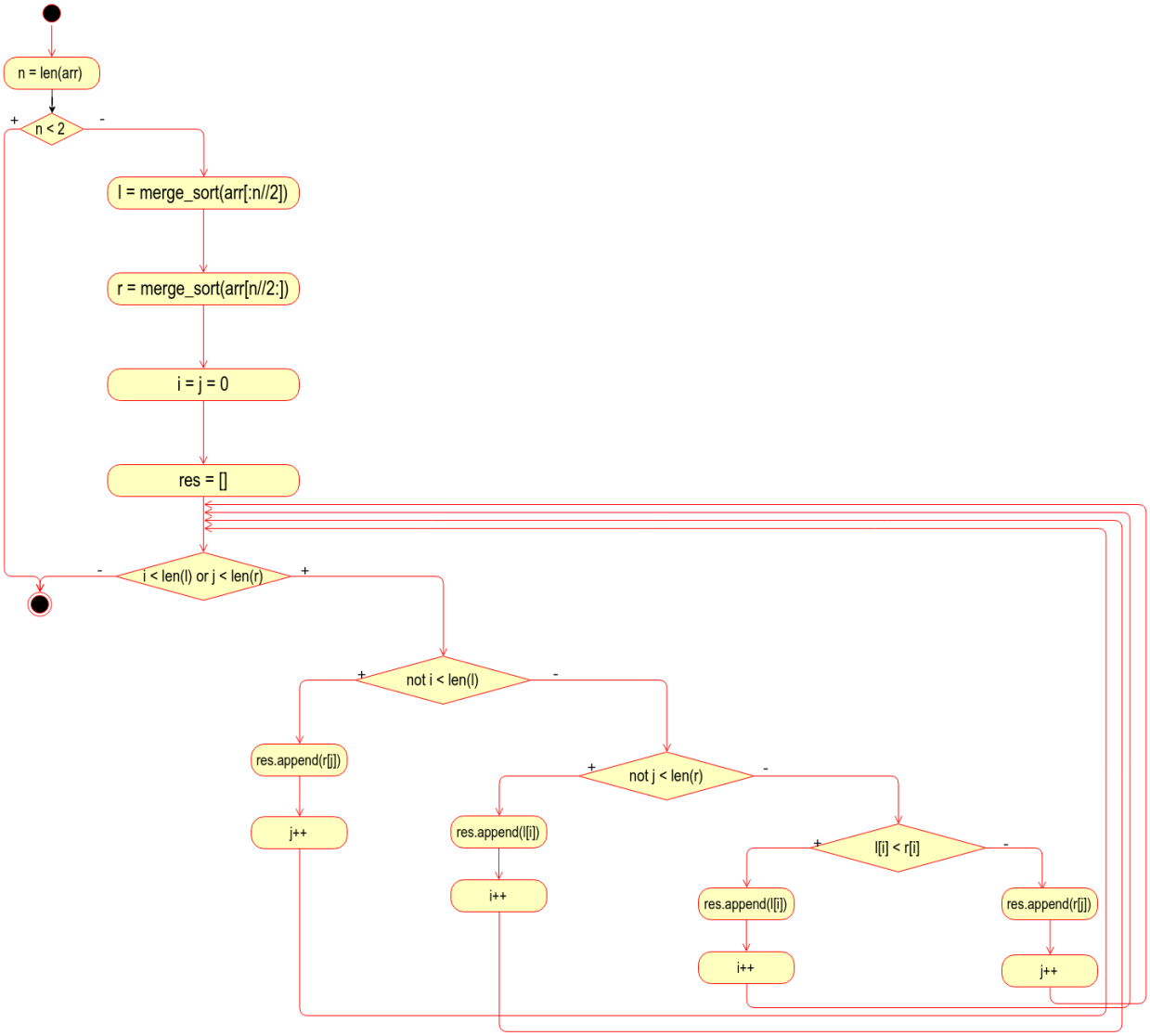

Рисунок 10 - Диаграмма деятельности

Сортировка выбором (Selection sort) — алгоритм сортировки. Может быть как устойчивый, так и неустойчивый. На массиве из n элементов имеет время выполнения в худшем, среднем и лучшем случае $\Theta(n^2)$, предполагая что сравнения делаются за постоянное время.

Наихудший случай:

Число сравнений в теле цикла равно (N-1)*N/2.

Число сравнений в заголовках циклов (N-1)*N/2.

Число сравнений перед операцией обмена N-1.

Суммарное число сравнений N2−1.

Число обменов N-1.

Наилучший случай:

Время сортировки 10000 коротких целых чисел на одном и том же программно-аппаратном комплексе сортировкой выбором составило ≈40сек., а ещё более улучшенной сортировкой пузырьком ≈30сек.

Пирамидальная сортировка сильно улучшает базовый алгоритм, используя структуру данных «куча» для ускорения нахождения и удаления минимального элемента.

Существует также двунаправленный вариант сортировки методом выбора, в котором на каждом проходе отыскиваются и устанавливаются на свои места и минимальное, и максимальное значения.

Листинг представлен в листинге 6, зависимость времени выполнения от количества элементов в рисунке 11, диаграмма деятельности на рисунке 12.

Листинг 6. Сортировка выбором.

```
def selection_sort(arr):
    for i in range(len(arr)):
        min_indx = minimum(arr, i, len(arr))
        if min_indx != i:
            arr[i], arr[min_indx] = arr[min_indx], arr[i]
    return arr
```

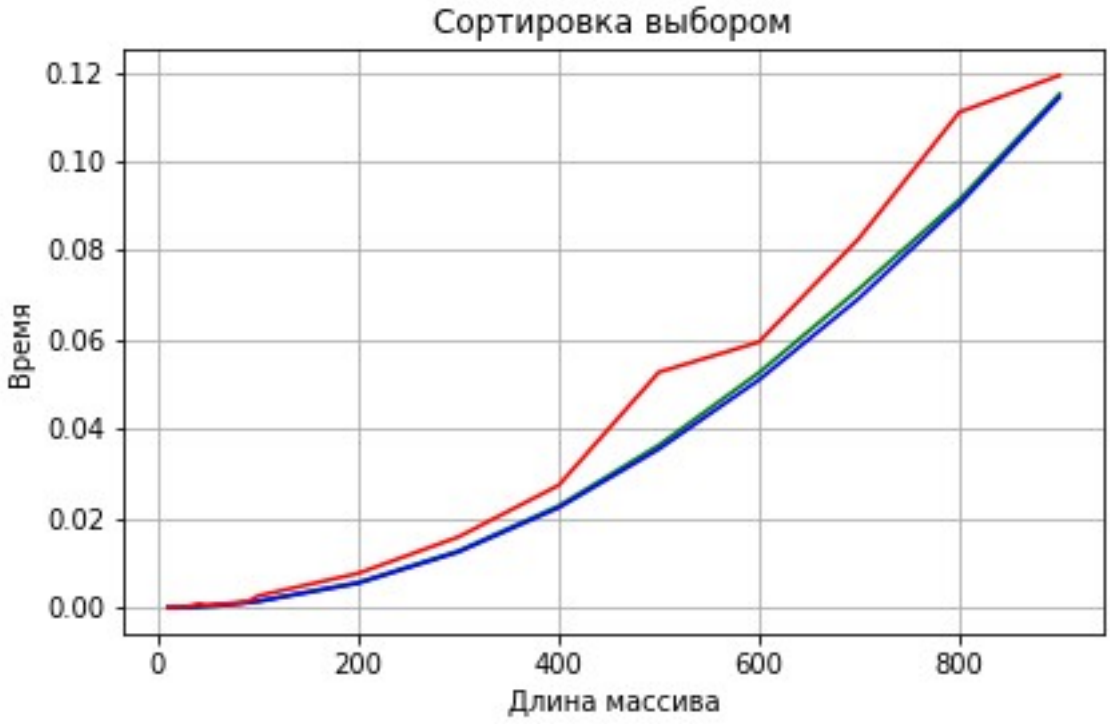

Рисунок 11 - Зависимость времени выполнения от числа элементов

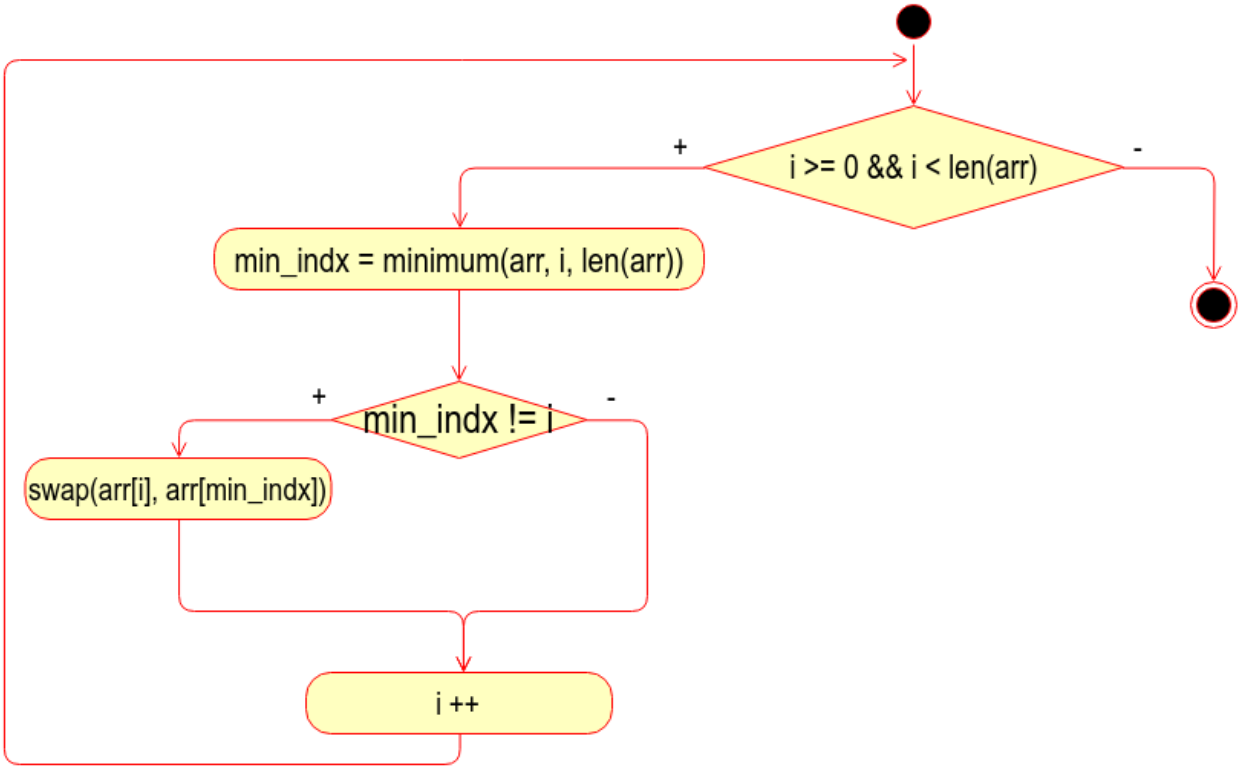

Рисунок 12 - Диаграмма деятельности

Сортировка расчёской (англ. comb sort) — это довольно упрощённый алгоритм сортировки, изначально спроектированный Влодзимежом Добосевичем в 1980 г. Позднее он был переоткрыт и популяризован в статье Стивена Лэйси и Ричарда Бокса в журнале Byte Magazine в апреле 1991 г[1]. Сортировка расчёской улучшает сортировку пузырьком, и конкурирует с алгоритмами, подобными быстрой сортировке. Основная идея — устранить черепах, или маленькие значения в конце списка, которые крайне замедляют сортировку пузырьком (кролики, большие значения в начале списка, не представляют проблемы для сортировки пузырьком).

В сортировке пузырьком, когда сравниваются два элемента, промежуток (расстояние друг от друга) равен 1. Основная идея сортировки расчёской в том, что этот промежуток может быть гораздо больше, чем единица (сортировка Шелла также основана на этой идее, но она является модификацией сортировки вставками, а не сортировки пузырьком).

В «пузырьке», «шейкере» и «чёт-нечете» при переборе массива сравниваются соседние элементы. Основная идея «расчёски» в том, чтобы первоначально брать достаточно большое расстояние между сравниваемыми элементами и по мере упорядочивания массива сужать это расстояние вплоть до минимального. Таким образом, мы как бы причёсываем массив, постепенно разглаживая на всё более аккуратные пряди. Первоначальный разрыв между сравниваемыми элементами лучше брать с учётом

специальной величины, называемой фактором уменьшения, оптимальное значение которой равно примерно 1,247[источник не указан 267 дней]. Сначала расстояние между элементами равно размеру массива, разделённого на фактор уменьшения (результат округляется до ближайшего целого). Затем, пройдя массив с этим шагом, необходимо поделить шаг на фактор уменьшения и пройти по списку вновь. Так продолжается до тех пор, пока разность индексов не достигнет единицы. В этом случае массив досортировывается обычным пузырьком.

Оптимальное значение фактора уменьшения $1{,}247 = \frac{1}{1 - e^{-\phi}}$.

Листинг приведен в Листинге 7, зависимость времени выполнения от количества элементов на рисунке 13, диаграмма деятельности на рисунке 14.

Листинг 7. Сортировка расческой.

```
def combsort(arr):
    n = len(arr)
    width = (n * 10 // 13) if n > 1 else 0
    while width:
        if 8 < width < 11:
            width = 11
        swapped = False
        for i in range(n - width):
            if arr[i + width] < arr[i]:
                arr[i], arr[i + width] = arr[i + width],
arr[i]
                swapped = True
        width = (width * 10 // 13) or swapped
    return arr
```

Рисунок 13 - Зависимость времени выполнения от количества элементов

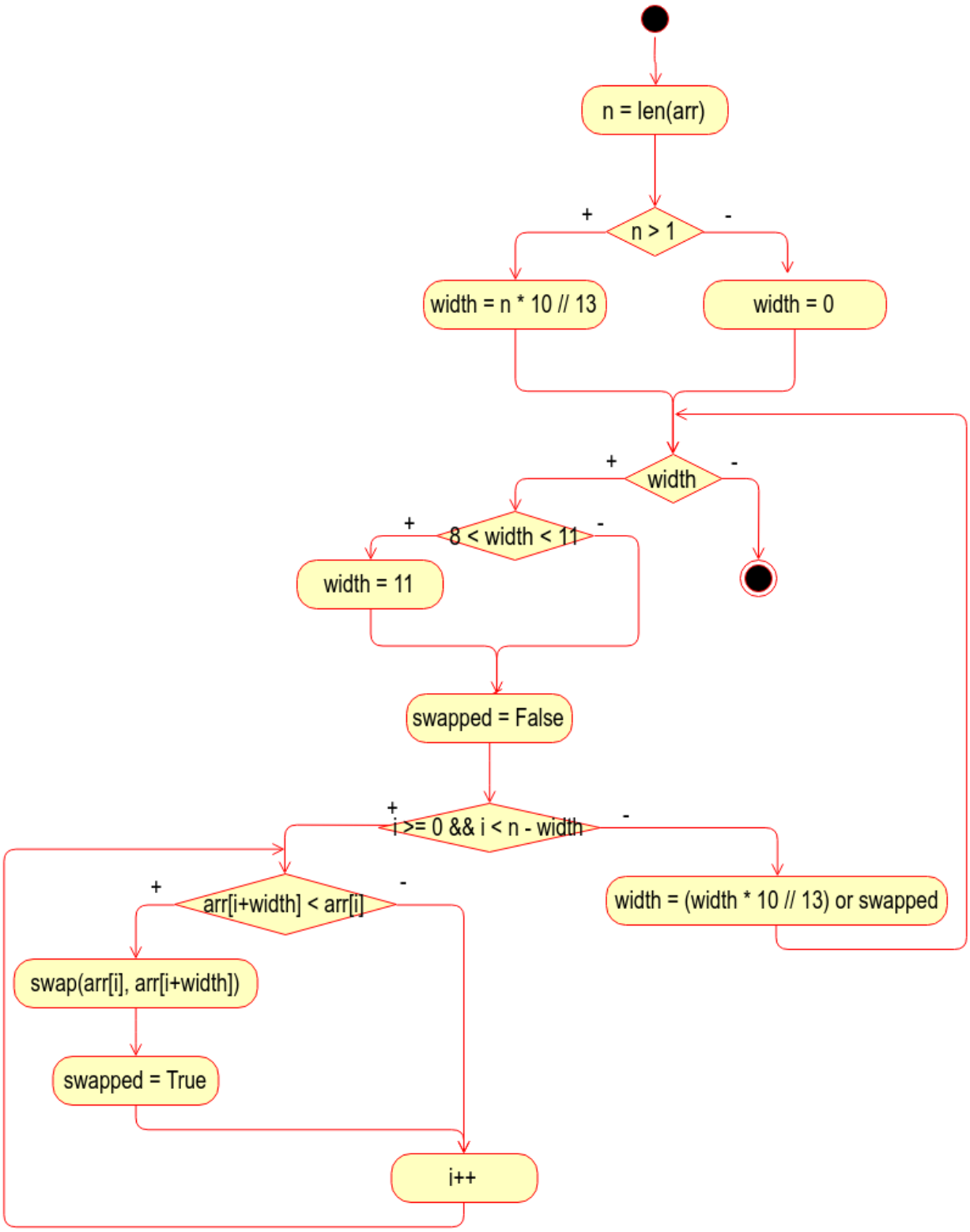

Рисунок 14 - Диаграмма деятельности

Быстрая сортировка, сортировка Хоара (англ. quicksort), часто называемая qsort (по имени в стандартной библиотеке языка Си) — алгоритм сортировки, разработанный английским информатиком Чарльзом Хоаром во время его работы в МГУ в 1960 году.

Один из самых быстрых известных универсальных алгоритмов сортировки массивов: в среднем O (n log n) обменов при упорядочении n элементов; из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками.

Ясно, что операция разделения массива на две части относительно опорного элемента занимает время $O(\log_2 n)$. . Поскольку все операции разделения, проделываемые на одной глубине рекурсии, обрабатывают разные части исходного массива, размер которого постоянен, суммарно на каждом уровне рекурсии потребуется также O(n) операций. Следовательно, общая сложность алгоритма определяется лишь количеством разделений, то есть глубиной рекурсии. Глубина рекурсии, в свою очередь, зависит от сочетания входных данных и способа определения опорного элемента.

В наиболее сбалансированном варианте при каждой операции разделения массив делится на две одинаковые (плюс-минус один элемент) части, следовательно, максимальная глубина рекурсии, при которой размеры обрабатываемых подмассивов достигнут 1, составит $\log_2 n$. В результате количество сравнений, совершаемых быстрой сортировкой, было бы равно значению рекурсивного выражения $C_n = 2C_{n/2}+n$, что дает общую сложность алгоритма $O(n \log_2 n)$.

Алгоритм сортировки представлен в Листинге 8, зависимость времени сортиро́вки от количества элементов на рисунке 15, диаграмма деятельности на рисунке 16.

Листинг 8. Быстрая сортиро́вка.

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        q = random.choice(arr)
        s_nums = []
        m_nums = []
        e_nums = []
        for n in arr:
            if n < q:
                s_nums.append(n)
            elif n > q:
                m_nums.append(n)
            else:
                e_nums.append(n)
              return quicksort(s_nums) + e_nums + quicksort(m_nums)
```

Рисунок 15 - Зависимость времени сортировки от количества элементов

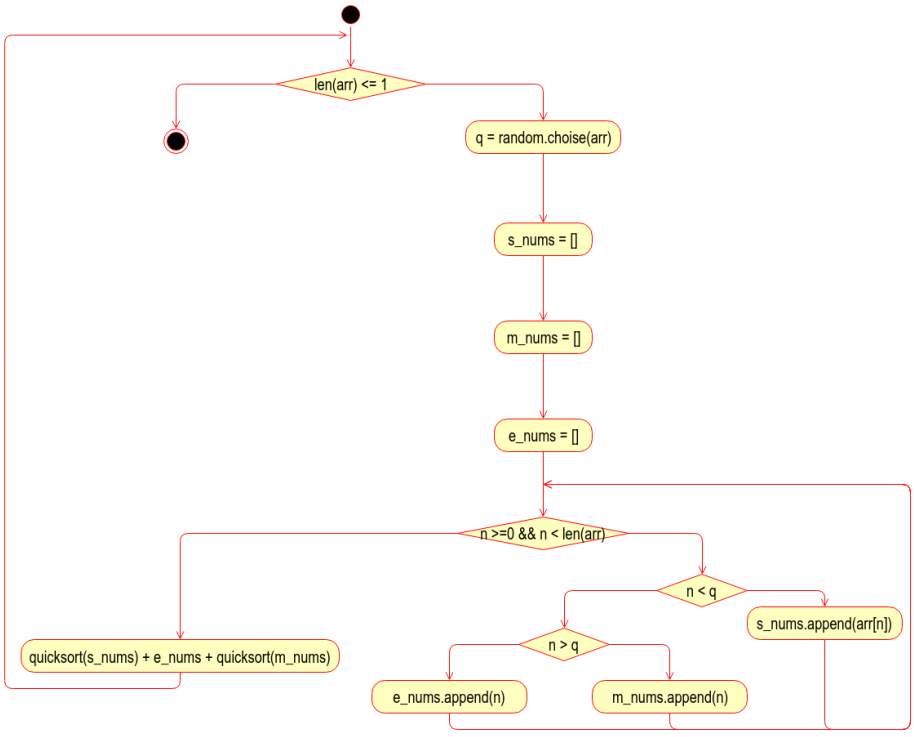

Рисунок 16 - Диаграмма деятельности

Вывод: в ходе выполнения практической работы была изучена работа сортиро́вка массива различными методами.