Отчет по лабораторной работе №3 по курсу "Анализ алгоритмов" по теме "Сортировки "

Студент: Барсуков Н.М. ИУ7-56
Преподаватель: Волкова Л.Л., Строганов Ю.В.

Содержание

Введение

Для решения многих задач удобно сначала упорядочить данные по определённому признаку, чтобы ускорить поиск некоторого нужного объекта. К примеру, в энциклопедиях статьи распределены по тому, какой стоит буква, с которой начинается название статьи, в алфавите. Перегруппирование заданного множества объектов называется сортировкой. Существует множество методов сортировки, которые различаются по сложности и эффективности, как временной, так и ёмкостной, в зависимости от выборки данных. По причине необходимости экономиии ресурса памяти и времени актуальной является задача выявления наиболее подходящего для конкретной задачи алгоритма сортировки.

Проведение данной работы необходимо для анализа алгоритмов сортировки и их сравнения.

Целью данной работы является изучение реализации и трудозатратности сортировки слиянием, пирамидальной сортировки и плавной сортировки.

Задачами данной работы является ознакомление с тремя сортировками, получение практических навыков их реализации, изучение оценки трудозатратности сортировок, проведение экспериментального сравнения временных характеристик сортировок.

1. Аналитическая часть

В данном разделе описаны задачи, необходимые для решения поставленной цели. Детально рассмотрены выбранные алгоритмы сортировки данных.

1.1. Постановка задачи:

Цель: изучить и реализовать следующие алгоритмы сортировки данных:

1) пирамидальная;

2) слияниями;

3) плавная.

Для реализации поставленной цели необходимо выполнить следующией поставленные задачи:

1) изучить выбранные алгоритмы;

2) реализовать указанные выше алгоритмы;

3) выбрать модель трудоемкости;

4) расчитать трудоемкость;

5) замерить время работы алгоритмов;

6) сравнить результаты замеров;

7) сделать вывод.

Реализовать алгоритмы сортировки:

1) сортировка слиянием (mergesort);

2) пирамидальная сортировка (heapsort);

3) плавная сортивка (smoothsort).

1.2. Описание алгоритмов

1.2.1. Сортировка слияниями

Сортировка слиянием - алгоритм сортировки, который упорядочивает списки в определённом порядке. Массив разделяется на подзадачи, где под подзадачей понимается часть исходного массива. [5]

Данная сортировка обладает одинаковой сложностью O(n*log(n)) при любом наборе данных, причём требует дополнительной памяти, совпадающей по размеру с размером исходного массива. [5]

1.2.2. Пирамидальная сортировка

Пирамидальная сортировка - алгоритм, предложенный Дж.Уильямсом в 1964 году. Пирамидальная сортировка работает с бинарным сортирующим деревом поиска, называемым иначе двоичной кучей (такой структурой двоичного дерева, для которой выполняются условия, что значение в любой вершине не меньше значений её потомков, глубина листьев различается не более чем на 1 слой и что последний слой заполняется слева направо без дырок). [6]

Данная сортировка обладает одинаковой сложностью O(n*log(n)) при любом наборе данных, причём не требует дополнительной памяти, поскольку сортирует массив на месте. [6]

1.2.3. Плавная сортировка

Плавная сортировка - разновидность пирамидальной сортировки, разработанная Э.Дейкстрой в 1981 году. Отличается от пирамидальной сортировки тем, что у пирамидальной сортировки одинаковая сложность (O(n*log(n))) на любом наборе данных, в то время как у плавной сортировки сложность приближается к O(n) при частично отсортированных данных. Также, в отличие от пирамидальной сортировки, использует не двоичную кучу, а её аналог, где число потомков определяется последовательностью чисел Леонардо. [4]

Числа Леонардо - последовательность, элементы которой задаются по формуле [3]

$$L(n) = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ L(n-1) + L(n-2) + 1 & n > 1 \end{cases} \quad (1)$$

2. Конструкторская часть

В данном разделе приводятся логика алгоритмов и блок-схемы их работы.

2.1. Логика алгоритмов

2.1.1. Сортировка слияниями

Алгоритм можно описать следующим образом [1]:

1) разделить исходный массив пополам (или примерно пополам в случае с нечётной длиной);

2) сортировать каждую из получившихся частей с помощью сортировки слиянием (то есть разделять вплоть до длины по одному элементу);

3) соединить разделённые на первом шаге части массива с помощью сортировки слиянием в целый массив. Для этого на каждом шаге взять меньший из первых элементов обоих подмассивов и записать в результирующий массив, переходя к следующим элементам. Когда один из подмассивов закончился, добавить оставшиеся элементы другого, поскольку они уже были отсортированы между собой.

2.1.2. Пирамидальная сортировка

Алгоритм пирамидальной сортировки можно подразделить на две части:

1) выстраивание первых элементов в виде сортирующего дерева;

2) перенос корня дерева (максимальное число) в отсортированную часть;

3) повторить с п.1 до тех пор, пока не будут отсортированы все элементы.

[6]

2.1.3. Плавная сортировка

Алгоритм плавной сортировки:

1) добавляя по одному, строится множество куч Леонардо для каждого элемента;

2) по очереди извлекаются максимальные элементы из построенной структуры (максимумом всегда будет корень самой правой кучи), структура восстанавливается необходимым образом.

[2]

2.2. Разработка алгоритмов

2.2.1. Блок схемы алгоритмов

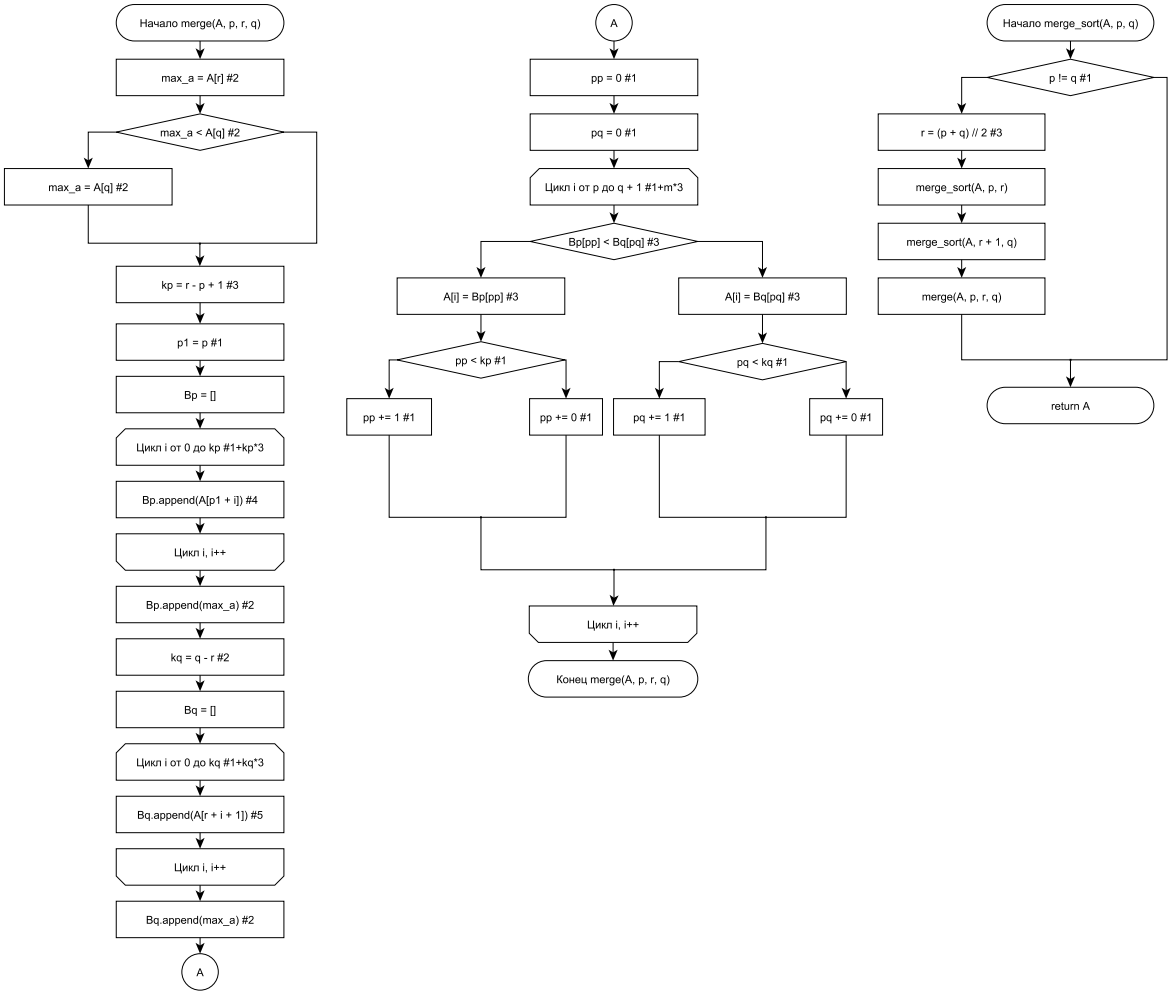

Рис. 1. Сортировка слиянием

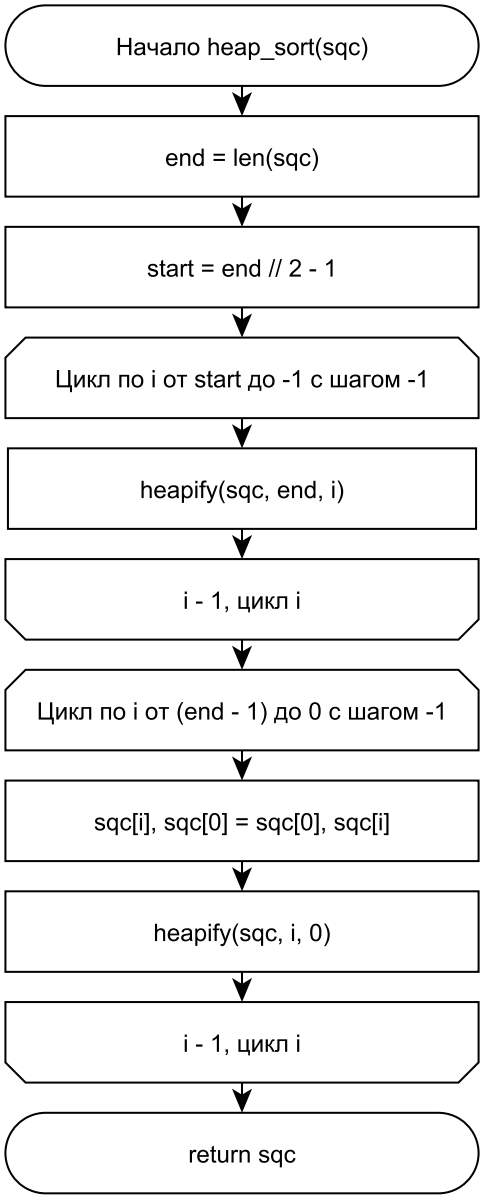

Рис. 2. Пирамидальная сортировка

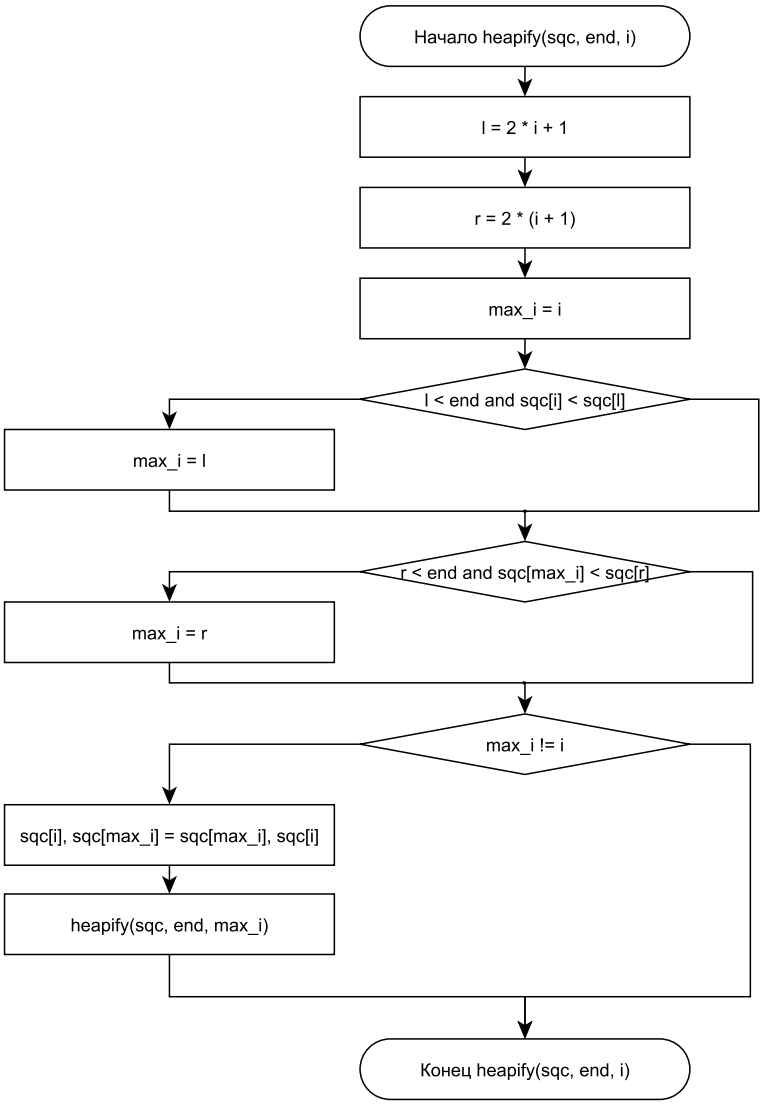

Рис. 3. Пирамидальная сортировка

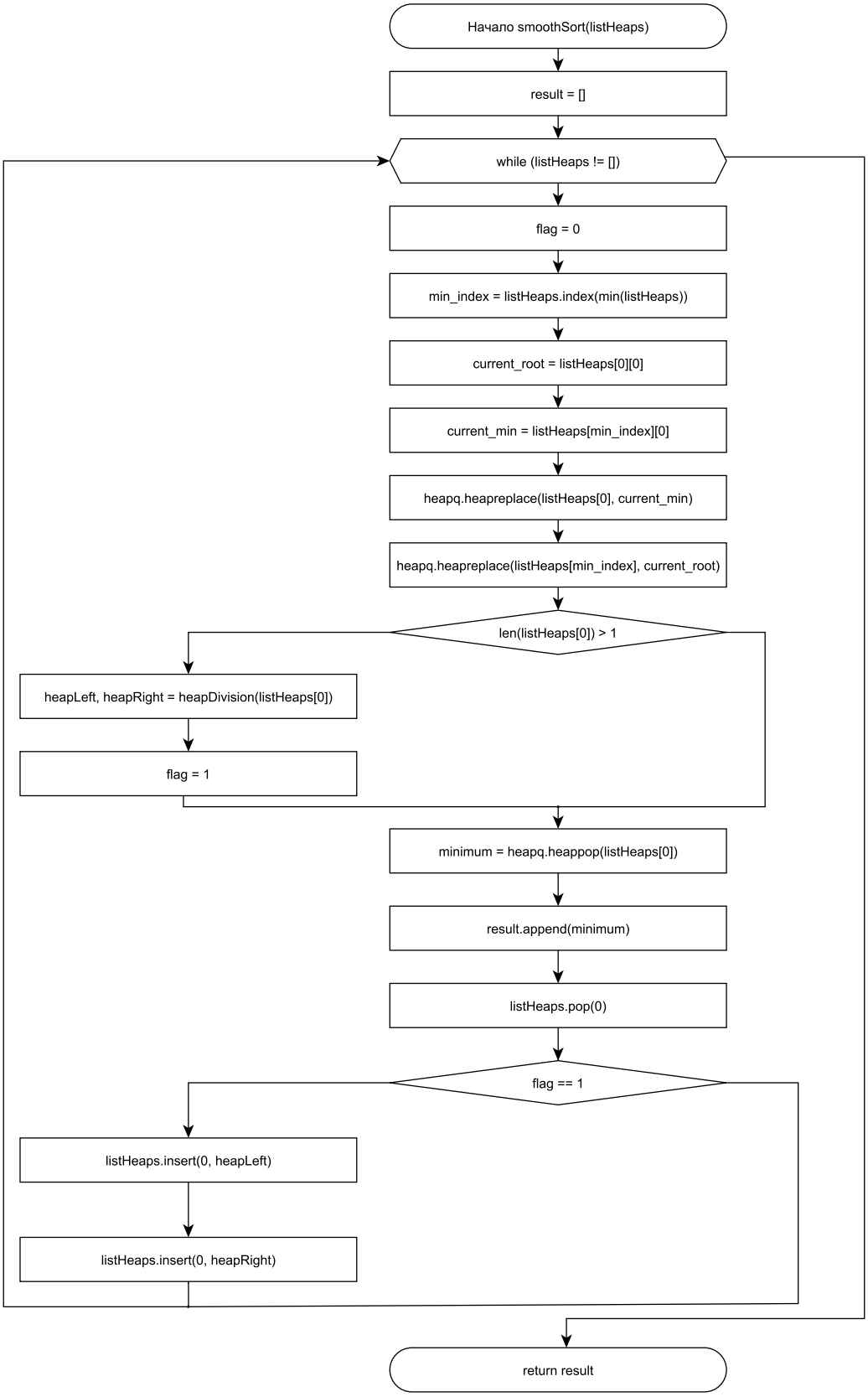

Рис. 4. Плавная сортировка

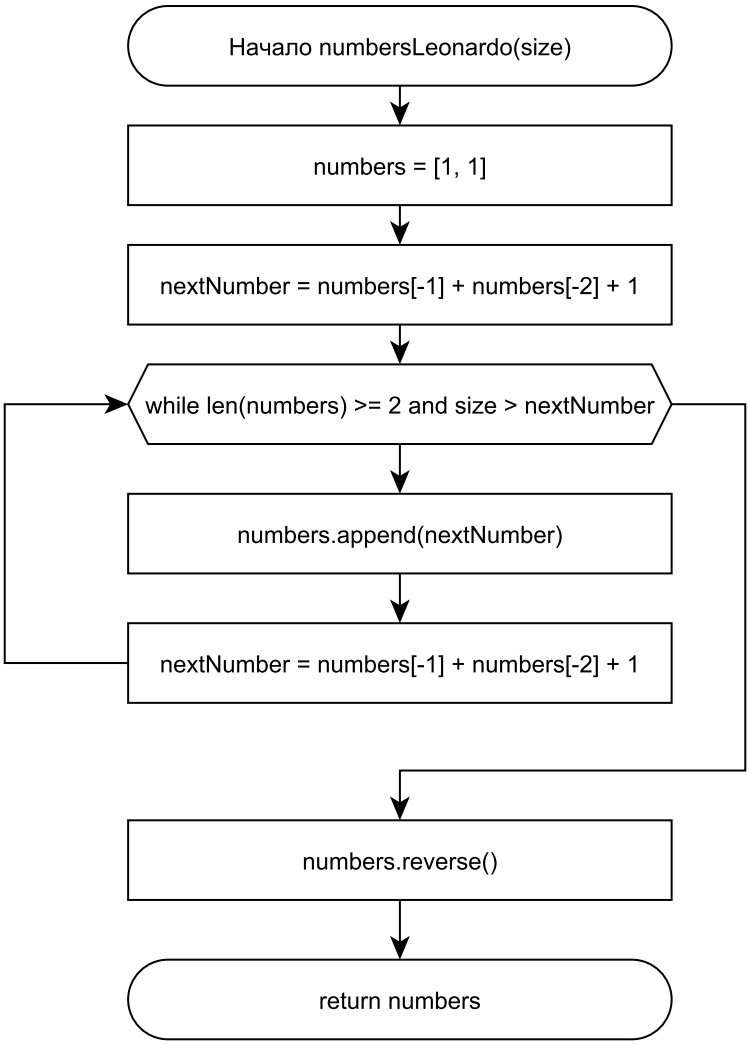

Рис. 5. Плавная сортировка

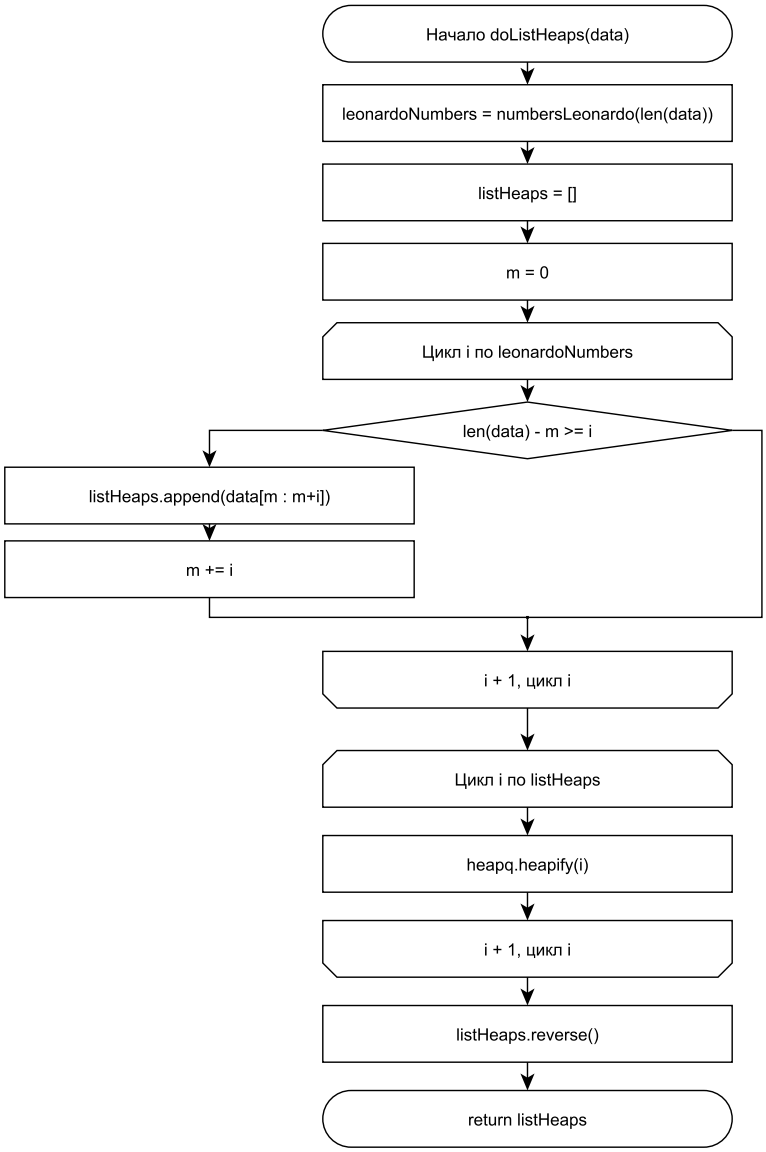

Рис. 6. Плавная сортировка

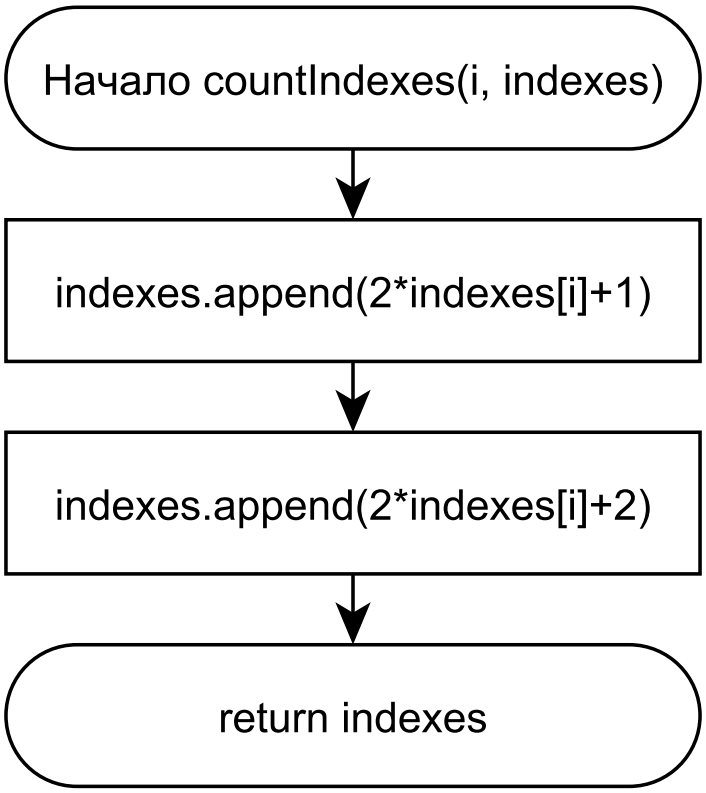

Рис. 7. Плавная сортировка

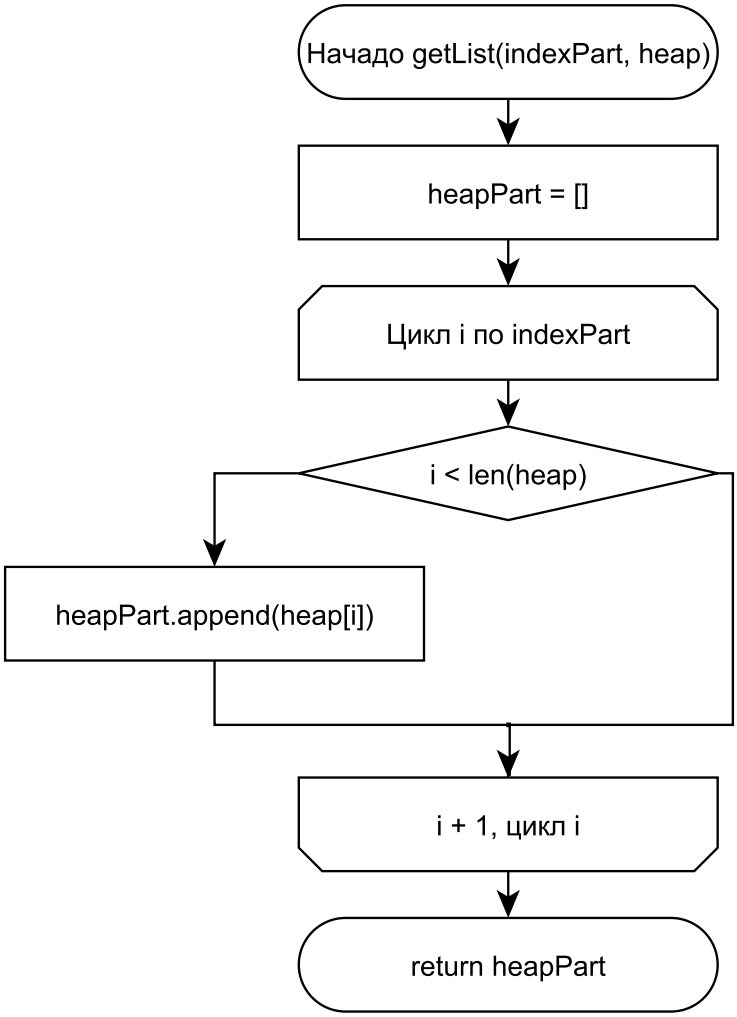

Рис. 8. Плавная сортировка

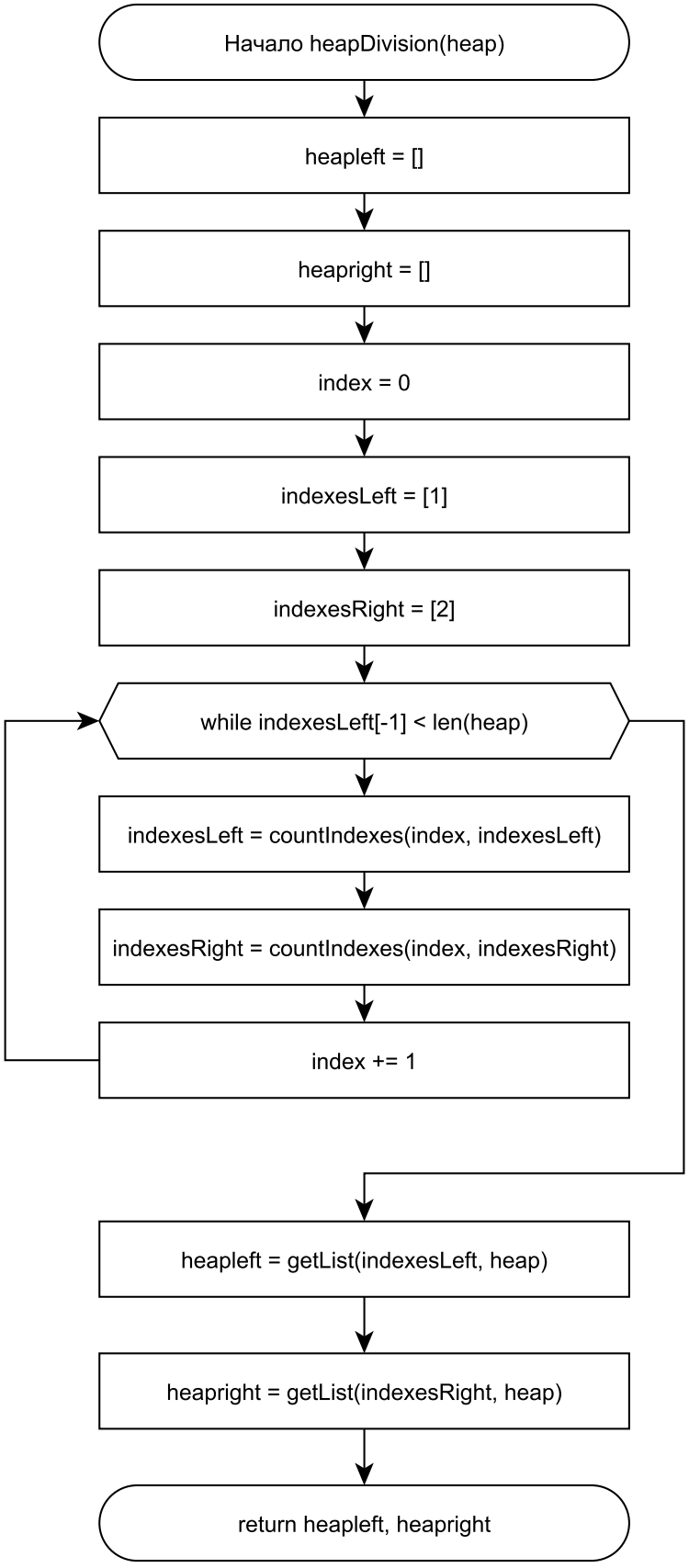

Рис. 9. Плавная сортировка

2.3. Расчет сложности алгоритмов

Расчёт трудоёмкости алгоритма выполняется для алгоритма сортировки слиянием с использованием следующей модели трудоёмкости:

1) единичная трудоёмкость:
   +, −, ∗, /, //, остаток от деления, <, <=, =>, >, ==, =, ! =, [], + =, − =, ∗ =, / =, ++, −−, ., len(), взятие части связного списка (работа со срезами в ЯП Python);

2) C сложность цикла | Python:
   а) for (int i = 0; i < N; i++): for i in range(N)
   COMP(цикла) = 2 + N*(2 + COMP(тела))
   б) for (int i = 0; i < N+1; i++): for i in range(N+1)
   COMP(цикла) = 3 + N*(3 + COMP(тела))

Трудоёмкось алгоритма практически не зависит от данных. В блоке условия if Bp[pp] < Bq[pq] содержится одинаковое число операций, следовательно, выбор ветви не повлияет на трудоёмкость.

Трудоёмкость данного алгоритма для объединённого массива, имеющего длину

$$m = k * p + k * q = q - p + 1.$$

Строка max_a = A[q] выполняется в среднем для половины обращений. Трудоёмкость алгоритма слияния отсортированных массивов как функция длины массива результата:

$$f^{-}_{merge}(m) = 2{+}2{+}1{+}3{+}2{+}1{+}3{*}kp{+}4{*}kp{+}3{+}2{+}1{+}3{*}kq{+}4{*}kq{+}3{+}1{+}1{+}1{+}3{*}m{+}m{*}(3{+}5) \quad (2)$$

$$f^{-}_{merge}(m) = 11 * m + 7 * (kp + kq) + 23 = 18 * m + 23 \quad (3)$$

Проведём анализ алгоритма методом подсчёта дерева рекурсий. Метод подсчёта дерева рекурсий - один из методов, применяемых для временной и ёмкостной трудоёмкости рекурсивных алгоритмов. Итоговая сумма базовых операций формируется с помощью вершин дерева рекурсии. Суммарные ресурсы, требуемые рекурсивной реализацией алгоритма, могут быть раздеены на ресурсы, собственно связанные с решением задачи, и ресурсы, необходимые для организации рекурсии. Если можно определить ресурсные затраты в каждой вершине дерева, то, суммируя, можно получить ресурсную функцию алгоритма в целом, что, собственно, и является методом подсчёта дерева рекурсии. Первым этапом такого подхода является исследование дерева рекурсии, на основе чего строятся функции-характеристики. [8]

Случай 1. Длина массива - степень двойки.
Длина входа

$$n = 2^k, k = log_2(n). \tag{4}$$

Массив делится ровно пополам, после чего продолжает делиться рекурсивно до тех пор, пока в каждом подмассиве не останется по 1 элементу. В данном случае получается дерево рекурсий, которое представляет собой бинарное дерево глубины k, содержащиее числое листьев, равное n.

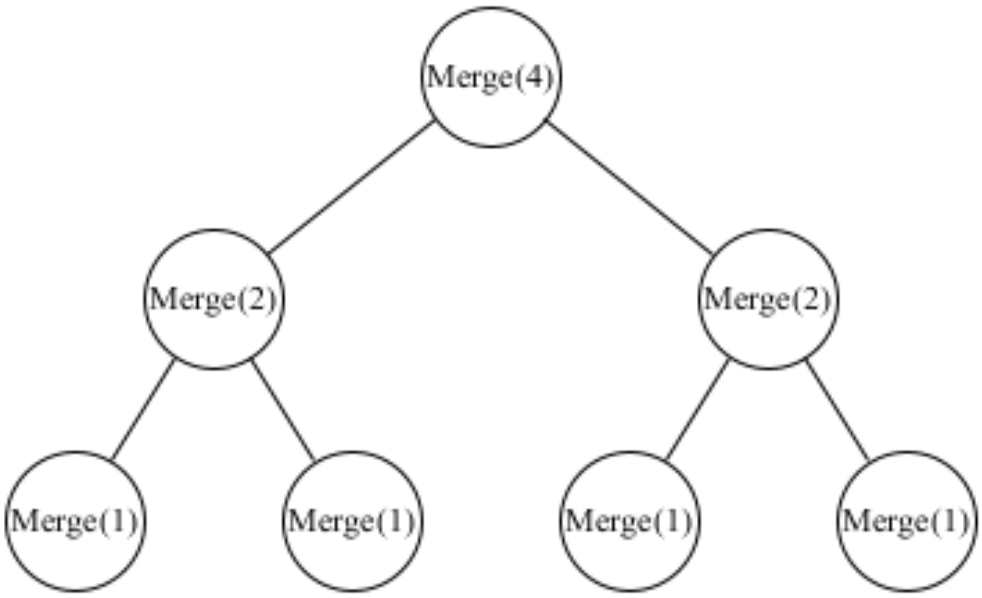

Рис. 10. Бинарное дерево сортировки слиянием на примере массива длины 4

Общее количество вершин подобного бинарно дерева R(n) задаётся с помощью формулы суммы геометрической прогрессии:

$$R(n) = 1 + 2 + ... + 2^k = 2 * 2^k - 1 = 2 * n - 1 \tag{5}$$

Поскольку полное бинарное дерево содержит

$$n = 2^k \tag{6}$$

листьев, то

$$R_L(n) = n, \tag{7}$$

количество внутренних вершин, порождающих рекурсию, равно

$$R_v(n) = n - 1. \tag{8}$$

Вызов данного алгоритма для сортировки массива длины n порождает дерево рекурсии со следующими характеристиками:

$$R(n) = 2*n-1, R_L(n) = n, R_v(n) = n-1, H_R(n) = 1+log_2(n), B_L(n) = R_L(n)/R(n) = n/(2*n-1), \tag{9}$$

где Hr(n) - высота, k - уровни, Rl(n) - количество листьев, Bl(n) - относительная ширина нижнего уровня в дереве рекукрсии.

Определим трудоёмкость функции merge_sort(l) на один вызов fr(1) в соответствии с методом подсчёта вершин дерева рекурсии. Функция имеет три параметра (p = 3), в стеке сохраняются значения четырёх регистров (r = 4), ни одно значение не возвращается через имя функции (f = 0), а функция имеет одну локальную переменную r (l = 1), то в результате получено:

$$f_R(1) = 2 * (3 + 4 + 0 + 1 + 1) = 18, \tag{10}$$

следовательно, с учётом предыдущей формулы,

$$f_R(n) = R(n) * f_R(1) = (2 * n - 1) * 18 = 36 * n - 18. \tag{11}$$

Трудоёмкость останова рекурсии включает в себя одно сравнение, таким образом

$$f_{CL}(1) = 1,$$

следовательно,

$$f_{CL}(n) = R_L(n) * f_{CL}(1) = n * 1 = n. \tag{12}$$

Во всех внутренних вершинах дерева (фрагмент рекурсивного вызова) трудоёмкость включает в себя подготовку рекурсивного вызова, вызов и возврат функции

$$merge(A, p, r, q) - f_{merge\_sort}(v),$$

и трудоёмкость выполнения функции

$$merge(A, p, r, q) - f_{merge}(v).$$

Трудоёкость во внутренней вершине fcv(v) и суммарную трудоёмкость внутренних вершин fcv(n) можно представить в виде следующих сумм:

$$f_{CV}(v) = f_{merge\_sort}(v) + f_{merge}(v), f_{CV}(n) = f_{CVmerge\_sort}(n) + f_{CVmerge}(n). \tag{13}$$

Вычисление значения fmerge_sort(v): выполняется сравнение, вычисление середины длины, прибавление единицы (r+1) и передача управления merge(A, p, r, q) с сохранением значения переменной r (l=1), после чего управление возвращается обратно, т.е.

$$f_{merge\_sort}(v) = 1 + 3 + 1 + 2 * (4 + 4 + 0 + 1 + 1) = 25. \tag{14}$$

Сумма по всем внутренним вершинам:

$$f_{CVmergesort}(n) = R_v(n) * f_{mergesort}(v) = (n-1) * 25 = 25 * n - 25. \quad (15)$$

Введём функцию

$$g(v_j) = f_{merge}(v_j), \quad (16)$$

являющуюся трудоёмкостью слияния в вершине vj. Для рассматриваемого случая на фиксированном уровне рекурсивного дерева слиянию подвергаются массивы одинаковой длины. Поскольку одинаковые слагаемые, связанные одним уровнем, могут быть объединены:

$$\sum_{j=1}^{R_v(n)} g(v_j). \quad (17)$$

Согласно формуле , трудоёмкость алгоритма слияния для массива длины m составляет

$$18 * m + 23,$$

причём алгоритм вызывается

$$R_v(n) = n - 1$$

раз с разными длинами объединяемых фрагментов массива на каждом уровне дерева. Следовательно, значения g(vj) имеют вид

$$g(v_j) \quad : \quad \begin{cases} g(v_1) = 18 * n + 23; \\ g(v_2) = g(v_3) = 18 * (n/2) + 23; \\ g(v_4) = g(v_5) = g(v_6) = g(v_7) = 18 * (n/4) + 23; \\ \quad ... \end{cases} \quad (18)$$

В результате суммирования g по всем внутренним вершинами дерева:

$$f_{CVmerge}(n) = \sum_{j=1}^{R_v(n)} g(v_j) = 23 * R_v(n) + 18 * n + 2 * 18 * (n/2) + 4 * 18 * (n/4) + ... \quad (19)$$

Учитывая, что таким образом обрабатываются все уровни дерева, кроме последнего, который не содержит внутренних вершин, т.е. k уровней рекурсивного дерева с номерами от 0 до k - 1, получаем:

$$f_{CVmerge}(n) = 18 * n * k + 23 * (n-1) = 18 * n * log_2(n) + 23 * n - 23. \quad (20)$$

Окончательный вид функции трудоёмкости для алгоритма сортировки слиянием в случае

$$n = 2^k$$

будет выглядеть следующим образом:

$$f_A^-(n) = f_R(n) + f_{CL}(n) + f_{CVmerge\_sort}(n) + f_{CVmerge}(n) \quad (21)$$

$$f_A^-(n) = 36*n-18+n+25*n-25+18*n*log_2(n)+23*n-23 = 18*n*log_2(n)+85*n-66. \quad (22)$$

Доля операций обслуживания дерева рекурсии составляет

$$F_R(n) = f_R(n)/f_A(n) = (36 * n - 18)/(18 * n * log_2(n) + 85 * n - 66). \quad (23)$$

Значения

$$F_R(n)$$

медленно убывают с ростом длины массива.

Функция трудоёмкости в среднем

$$f_A^-(n)$$

по реультату оценки главного порядка обладает сложностью

$$O(n * log_2(n)). \quad (24)$$

Такой же сложностью обладают и лучший и худший случаи. Они могут быть оценены исходя из того, что каждый вызов функции слияния при вычислении заглушки либо выполняет, либо пропускает две операции. Слияние выполняется для каждой внутренней вершины дерева.

Лучший случай:

$$f_A^{best}(n) = 18 * n * log_2(n) + 84 * n - 67. \quad (25)$$

Худший случай:

$$f_A^{worst}(n) = 18 * n * log_2(n) + 86 * n - 65. \quad (26)$$

Случай 2. Длина входа

$$2^{k-1} < n < 2^k, k = |log_2(n)|.$$

Алгоритм получает на входе массив из n элементов, делит его на две равные части при чётном n или на части, отличающиеся на единицу, при нечётном n. Деление продолжается рекурсивно до тех пор, пока алгоритм не дойдёт до единичных элементов массива. В результате работы алгоритма получается бинарное дерево рекурсий, содержащее k уровней, из которых только k - 1 уровней являются полными. Данное неполное дерево можно охарактеризовать следующими формулами, совпадающими с формулами для полного дерева:

$$R(n) = 2 * n - 1, R_L(n) = n, R_v(n) = n - 1, H_R(n) = 1 + log_2(n), B_L(n) = n/(2 * n - 1), \quad (27)$$

Поскольку формулы для характеристик дерева не изменились, связанные с ним формулы трудоёмкости компонентов остаются в силе. Изменения в случае 2 касаются трудоёмкости слияния во внутренних вершинах

$$f_{CVmerge}(n) = \sum_{j=1}^{R_v(n)} g(v_j) \quad (28)$$

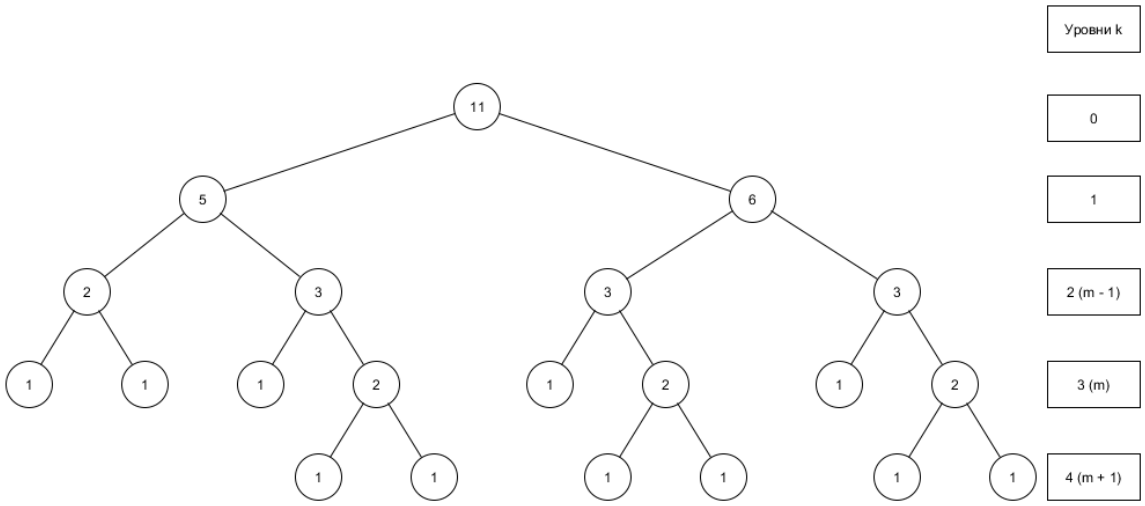

Рис. 11. Бинарное дерево сортировки слиянием на примере массива длины 11

Введём нумерацию внутренних вершин по уровням дерева и обозначим через

$$(n_j)^k$$

длину фрагмента массива, обрабатываемого в j-ой внутренней вершине k-го уровня, а через

$$(v_j)^k$$

обозначим саму эту вершину. С учётом того, что

$$g((v_j)^k) = 18 * n_j^k + 23, \tag{29}$$

искомая сумма может быть записана следующим образом:

$$f_{CVmerge}(n) = \sum_{j=1}^{R_v(n)} (18 * n_j^k + 23) = 23 * R_v(n) + \sum_{j=1}^{R_v(n)} 18 * n_j^k. \tag{30}$$

Дерево рекурсии является полным, за исключением последнего уровня. Пусть

$$m = log_2(n) \tag{31}$$

является номером последнего полного уровня, тогда предыдущий уровень m - 1 не содержит листьев, а последний (m + 1)-й уровень содержит только листья. На уровне m могут находиться как листья, так и внутренние вершины. Поскольку дерево является бинарным, а на уровне m + 1 находятся только листья, то

$$(n_j)^m = 2.$$

Поскольку увеличение n на единицу приводит к созданию новой внутренней вершины, а для полного бинарного дерева все вершины последнего уровня являются листьями, то на уровне m будет находиться

$$n - 2^m$$

внутренних вершин. Исходя из этого, мы можем переписать формулу для

$$f_{CVmerge}(n)$$

в виде двух сумм: суммы по всем m - 1 уровням, содержащим только внутренние вершины, и суммы по внутренним вершинам уровня m:

$$\sum_{j=1}^{R_v(n)} 18 * n_j^k = \sum_{k=0}^{m-1} \sum_{j=1}^{2^k} 18 * n_j^k + \sum_{j=1}^{n-2^m} 18 * n_j^m. \tag{32}$$

Поскольку сумма объединённых фрагментов массива на полных уровнях рекурсивного дерева всегда равна полной длине массива n, а

$$(n_j)^m = 2,$$

то

$$\sum_{j=1}^{R_v(n)} 18 * n_j^k = 18 * n * m + 18 * (n - 2^m) * 2. \tag{33}$$

$$f_{CVmerge}(n) = 23 * R_v(n) + 18 * n * m + 18 * (n - 2^m) * 2 \tag{34}$$

Поскольку

$$m = log_2(n), R_v(n) = n - 1,$$

то

$$f_{CVmerge}(n) = 18 * n * log_2(n) + 59 * n - 36 * 2^{log_2(n)} - 23. \tag{35}$$

Трудоёмкость алгоритма сортировки в среднем для случая, где длина массива не равна степеням двойки:

$$f_A^-(n) = f_R(n) + f_{CL}(n) + f_{CVmerge\_sort}(n) + f_{CVmerge}(n) \tag{36}$$

$$f_A^-(n) = 36*n-18+n+25*n-25+f_{CVmerge}(n) = 18*n*log_2(n)+121*n-36*2^{log_2(n)}-66. \tag{37}$$

Как и в случае 1, разница между полученной трудоёмкостью в среднем и худшем и лучшем случаями равна n - 1.

Аналогично случаю 1, по реультату оценки главного порядка трудоёмкости среднего, лучшего и худшего случаев обладают сложностью

$$O(n * log_2(n)). \tag{38}$$

3. Технологическая часть

В данном разделе представлены средства реализации задачи и листинг кода.

3.1. Требования к программному обеспечению

Программа должна сортировать заданный массив одним из следующих методов:

1) сортировка слиянием;

2) пирамидальная сортировка;

3) плавная сортировка.

В программе должен быть реализован замер времени работы каждой сортировки на выборке размера 100..1000 для отсортированных значений, обратно отсортированных и для случайной выборки (средние значения 10 замеров сортировок на каждой выборке записываются в файлы для всех алгоритмов сортировки). Замеры производились для массивов со случайной выборкой элементов.

Минимальные системные требования: PC с операционной системой Windows версии XP/Vista/7/8/10. Требуются устройства ввода: клавиатура, мышь.

3.2. Средства реализации

Для выполнения данной лабораторной работы был выбран язык программирования (ЯП) Python (3) ввиду его простоты и наглядности. Замеры времени производились посредством возможностей модуля time ЯП Python, в частности, с помощью perf_counter.

Для генерации тестовых данных на массивах выборки 1000..10000 в данной лабораторной работе используются возможности модуля ЯП random, в частности, с помощью randint.

3.3. Листинг кода

Программа сортировки слиянием (*merge_sort.py*) :

```
def merge(left, right):
    result = []
    i, j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
```

```
    result += left[i:]
    result += right[j:]
    return result

# сортировка слиянием
# l - массив
def mergesort(l):
    if len(l) < 2:
        return l
    middle = len(l) // 2
    left = mergesort(l[:middle])
    right = mergesort(l[middle:])
    return merge(left, right)

#print(mergesort([9, 3, 1, 0, 2]))

#A - массив, p, r, q - индексы
#Bp, Bq - дополнительные массивы, куда выполняется
# копирование отсортированных частей
# Алгоритм слияния (Merge) отсортированных фрагментов массива A,
# расположенных в позициях между (p и r) и (r+1 и q).
# Алгоритм использует доп.массивы,
# в конец которых помещаются заглушки.
# Сначала выполняется копирование отсортированных
# частей в Bp и Bq,
# затем объединённый массив формируется непосредственно
# в массиве A между индексами p и q.

def merge(A, p, r, q):
    # формирование заглушки
    max_a = A[r]
    if max_a < A[q]:
        max_a = A[q]

    # копирование в массивы Bp, Bq
    kp = r - p + 1
    p1 = p

    Bp = []
    for i in range(kp):
        Bp.append(A[p1 + i])
    Bp.append(max_a) #заглушка

    kq = q - r
```

```
    Bq = []
    for i in range(kq):
        Bq.append(A[r + i + 1])
    Bq.append(max_a) #заглушка

    # слияние частей
    pp = 0
    pq = 0 #инициализация указателей
    for i in range(p, q + 1):
        if Bp[pp] < Bq[pq]:
            A[i] = Bp[pp]
            pp += 1 if pp < kp else 0
        else:
            A[i] = Bq[pq]
            pq += 1 if pq < kq else 0

def merge_sort(A, p, q):
    if p != q:
        r = (p + q) // 2
        merge_sort(A, p, r)
        merge_sort(A, r + 1, q)

        merge(A, p, r, q)

    return A
```

Программа пирамидальной сортировки (*heap_sort.py*) :

```
def heapify(sqc, end, i):
    l = 2 * i + 1
    r = 2 * (i + 1)
    max_i = i
    if l < end and sqc[i] < sqc[l]:
        max_i = l
    if r < end and sqc[max_i] < sqc[r]:
        max_i = r
    if max_i != i:
        sqc[i], sqc[max_i] = sqc[max_i], sqc[i]
        heapify(sqc, end, max_i)

# пирамидальная сортировка
# sqc - массив
def heap_sort(sqc):
```

```
    end = len(sqc)
    start = end // 2 - 1
    for i in range(start, -1, -1):
        heapify(sqc, end, i)
    for i in range(end - 1, 0, -1):
        sqc[i], sqc[0] = sqc[0], sqc[i]
        heapify(sqc, i, 0)

    return sqc

#sqc = [2, 7, 1, -2, 56, 5, 3]
#print(heap_sort(sqc[:]))
#print(sqc)
```

Программа плавной сортировки (*smooth_sort.py*) :

```
import heapq

# формирование списка чисел Леонардо, являющихся размерами куч
# size - размер входного массива для плавной сортировки
# возвращает список чисел Леонардо
def numbersLeonardo(size):
    numbers = [1, 1] # начальные элементы для последовательности чисел Леонардо
    nextNumber = numbers[-1] + numbers[-2] + 1
    while len(numbers) >= 2 and size > nextNumber:
        numbers.append(nextNumber)
        nextNumber = numbers[-1] + numbers[-2] + 1
    numbers.reverse()
    return numbers

# формирование списка куч по числам Леонардо
# data - входной массив данных
# возвращает выходной список с кучами
def doListHeaps(data):
    # формируем список чисел Леонардо для входной последовательности
    leonardoNumbers = numbersLeonardo(len(data))
    # формируем список куч
    listHeaps = [] # финальный список куч
    m = 0 # хвост предыдущей части и начало следующей
    for i in leonardoNumbers:
        if len(data) - m >= i:
            # если оставшаяся нераспределённая часть входного массива данных
    # больше или равна очередному числу Леонардо
            listHeaps.append(data[m : m+i])
```

```
            # переходим к оставшейся нераспределённой части
            m += i
    # восстанавливаем свойство кучи для каждой кучи
    for i in listHeaps:
        heapq.heapify(i)
    # так как кучи неубывающие, конечный результат будет заполняться
    # с начала - минимального элемента
    # до максимального элемента последовательности, то меняем порядок
    # куч на обратный
    listHeaps.reverse()
    return listHeaps

# формирование списка элементов по заданным индексам:
# i - индекс, потомки которого ищутся
# indexes - список индексов
def countIndexes(i, indexes):
    indexes.append(2*indexes[i]+1)
    indexes.append(2*indexes[i]+2)

    return indexes

# формирование подкучи из заданного списка индексов и исходной кучи
# indexPart - список индексов
# heapPart - найденная подкуча
def getList(indexPart, heap):
    heapPart = []
    for i in indexPart:
        if i < len(heap):
            heapPart.append(heap[i])

    return heapPart

# деление кучи на левые и правые подкучи
# heap - куча для деления
# возвращается кортеж из левой и правой подкучи
def heapDivision(heap):
    heapleft = []
    heapright = []
    index = 0
    indexesLeft = [1] # список индексов для элементов левой подкучи
    indexesRight = [2] # список индексов для элементов правой подкучи
    while indexesLeft[-1] < len(heap):
        # исходя из логики построения куч, левая подкуча никогда не будет
        # меньше правой
```

```
            # считаем индексы для левой подкучи
            indexesLeft = countIndexes(index, indexesLeft)

            # считаем индексы для правой подкучи
            indexesRight = countIndexes(index, indexesRight)

            index += 1

        # составляем списки левой и правой подкуч
        heapleft = getList(indexesLeft, heap)
        heapright = getList(indexesRight, heap)

        return heapleft, heapright

# плавная сортировка
# listHeaps - кучи
# возвращает отсортированную последовательность данных
def smoothSort(listHeaps):
    result = []
    while (listHeaps != []):
        # чтобы не писать пустые подкучи
        flag = 0
        # находим минимальный элемент среди корней куч
        min_index = listHeaps.index(min(listHeaps)) # индекс кучи
        # с минимальным корнем
        # меняем его местами с корнем первой кучи
        # запомним корень текущей кучи
        current_root = listHeaps[0][0]
        # и минимальный элемент
        current_min = listHeaps[min_index][0]
        heapq.heapreplace(listHeaps[0], current_min)
        heapq.heapreplace(listHeaps[min_index], current_root)
        # т.к. корень первой кучи будет в дальнейшем удален, размер кучи
        # уменьшится на 1 -> образуются две кучи из его левого и
        # правого поддерева
        if len(listHeaps[0]) > 1:
            heapLeft, heapRight = heapDivision(listHeaps[0])
            flag = 1
        # удаляем корень первой кучи - это минимальный элемент
        # из всех возможных
        minimum = heapq.heappop(listHeaps[0])
        # ставим его в конечную последовательность чисел
        result.append(minimum)
```

```
        # удалим первый элемент списка и вставим его ранее
        # полученные поддеревья
        listHeaps.pop(0)
        # добавим две получившиеся кучи в начало всей
        # последовательности куч
        if flag == 1:
            listHeaps.insert(0, heapLeft)
            listHeaps.insert(0, heapRight)
    return result

#print(smoothSort(doListHeaps([9, 3, 1, 0, 2])))
```

Основная программа, решающая задачу (main.py):

```
import random
import time
import heap_sort
import merge_sort
import smooth_sort

# Тесты работоспособности
#def print_arr(arr):
#    print('\nОтсортированный массив:')
#    for i in arr:
#        print(i, end=' ')

#arr = list(map(int, input('Массив для сортировки: ').split()))
#heap_sort.heap_sort(arr)
#arr = merge_sort.mergesort(arr)
#arr = smooth_sort.smoothSort(smooth_sort.doListHeaps(arr))
#print_arr(arr)
# конец тестов работоспособности

arr_ms = []
arr_hs = []
arr_ss = []
T1 = open("T1.txt", 'a')
T2 = open("T2.txt", 'a')
T3 = open("T3.txt", 'a')
for i in range(100, 1001, 100):
    s1, s2, s3 = 0, 0, 0
    for j in range(10):
        arr_ms = [random.randint(-10000, 10000) for k in range(i)]
        arr_hs = [random.randint(-10000, 10000) for k in range(i)]
```

```
        arr_ss = [random.randint(-10000, 10000) for k in range(i)]
        t1_start = time.perf_counter()
        arr_ms = merge_sort.mergesort(arr_ms)
        t1 = time.perf_counter() - t1_start
        s1 += t1
        t2_start = time.perf_counter()
        heap_sort.heap_sort(arr_hs)
        t2 = time.perf_counter() - t2_start
        s2 += t2
        t3_start = time.perf_counter()
        smooth_sort.smoothSort(smooth_sort.doListHeaps(arr_ss))
        t3 = time.perf_counter() - t3_start
        s3 += t3
    T1.write(str(s1/10))
    T1.write("\n")
    T2.write(str(s2/10))
    T2.write("\n")
    T3.write(str(s3/10))
    T3.write("\n")
T1.close()
T2.close()
T3.close()

arr_ms = []
arr_hs = []
arr_ss = []
T4 = open("T4.txt", 'a')
T5 = open("T5.txt", 'a')
T6 = open("T6.txt", 'a')
for i in range(100, 1001, 100):
    s1, s2, s3 = 0, 0, 0
    for j in range(10):
        arr_ms = [k for k in range(i)]
        arr_hs = [k for k in range(i)]
        arr_ss = [k for k in range(i)]
        t1_start = time.perf_counter()
        arr_ms = merge_sort.mergesort(arr_ms)
        t1 = time.perf_counter() - t1_start
        s1 += t1
        t2_start = time.perf_counter()
        heap_sort.heap_sort(arr_hs)
        t2 = time.perf_counter() - t2_start
        s2 += t2
        t3_start = time.perf_counter()
```

```
        smooth_sort.smoothSort(smooth_sort.doListHeaps(arr_ss))
        t3 = time.perf_counter() - t3_start
        s3 += t3
    T4.write(str(s1/10))
    T4.write("\n")
    T5.write(str(s2/10))
    T5.write("\n")
    T6.write(str(s3/10))
    T6.write("\n")
T4.close()
T5.close()
T6.close()

arr_ms = []
arr_hs = []
arr_ss = []
T7 = open("T7.txt", 'a')
T8 = open("T8.txt", 'a')
T9 = open("T9.txt", 'a')
for i in range(100, 1001, 100):
    s1, s2, s3 = 0, 0, 0
    for j in range(10):
        arr_ms = [k for k in range(i, 0, -1)]
        arr_hs = [k for k in range(i, 0, -1)]
        arr_ss = [k for k in range(i, 0, -1)]
        t1_start = time.perf_counter()
        arr_ms = merge_sort.mergesort(arr_ms)
        t1 = time.perf_counter() - t1_start
        s1 += t1
        t2_start = time.perf_counter()
        heap_sort.heap_sort(arr_hs)
        t2 = time.perf_counter() - t2_start
        s2 += t2
        t3_start = time.perf_counter()
        smooth_sort.smoothSort(smooth_sort.doListHeaps(arr_ss))
        t3 = time.perf_counter() - t3_start
        s3 += t3
    T7.write(str(s1/10))
    T7.write("\n")
    T8.write(str(s2/10))
    T8.write("\n")
    T9.write(str(s3/10))
    T9.write("\n")
T7.close()
```

```
T8.close()
T9.close()
```

4. Экспериментальная часть

В данном разделе приводятся примеры работы, графики зависимости времени работы алгоритмов, а также расчёт трудоёмкости алгоритма сортировки слиянием.

4.1. Примеры работы

В качестве тестовых случаев рассматриваются следующие:

1) упорядоченный массив;
2) массив, содержащий в себе одинаковые значения;
3) обратно упорядоченный массив;
4) массив случайных неупорядоченных элементов;
5) пустой массив;
6) массив, содержащий в себе одно значение.

Поскольку все сортировки отработали корректно, они дают одинаковые результаты работы на каждом из примеров.

Таблица 1. Пример работы 1

3	4	5	6	7	8
3	4	5	6	7	8

Таблица 2. Пример работы 2

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

Таблица 3. Пример работы 3

7	5	3	1
1	3	5	7

4.2. Постановка эксперимента

Таблица 4. Пример работы 4

5	3	2	6	19	0	-2
-2	0	2	3	5	6	19

Таблица 5. Пример работы 5

λ
λ

Время работы алгоритмов сортировки на случайных выборках представлены на графике 12

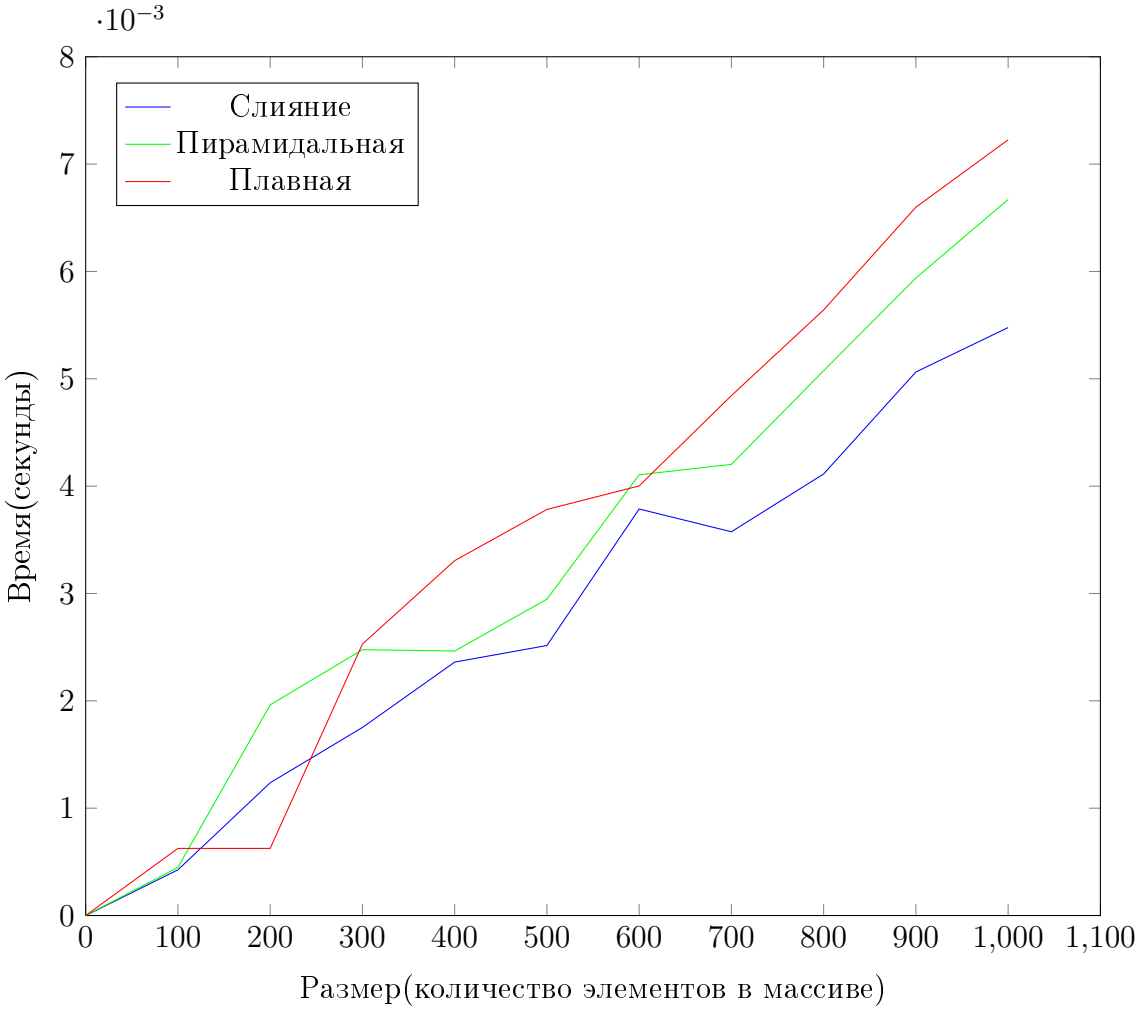

Рис. 12. График зависимости времени выполнения алгоритма от размера массива

Таблица 6. Пример работы 6

1
1

Время работы алгоритмов сортировки на упорядоченных массивах представлены на графике 13

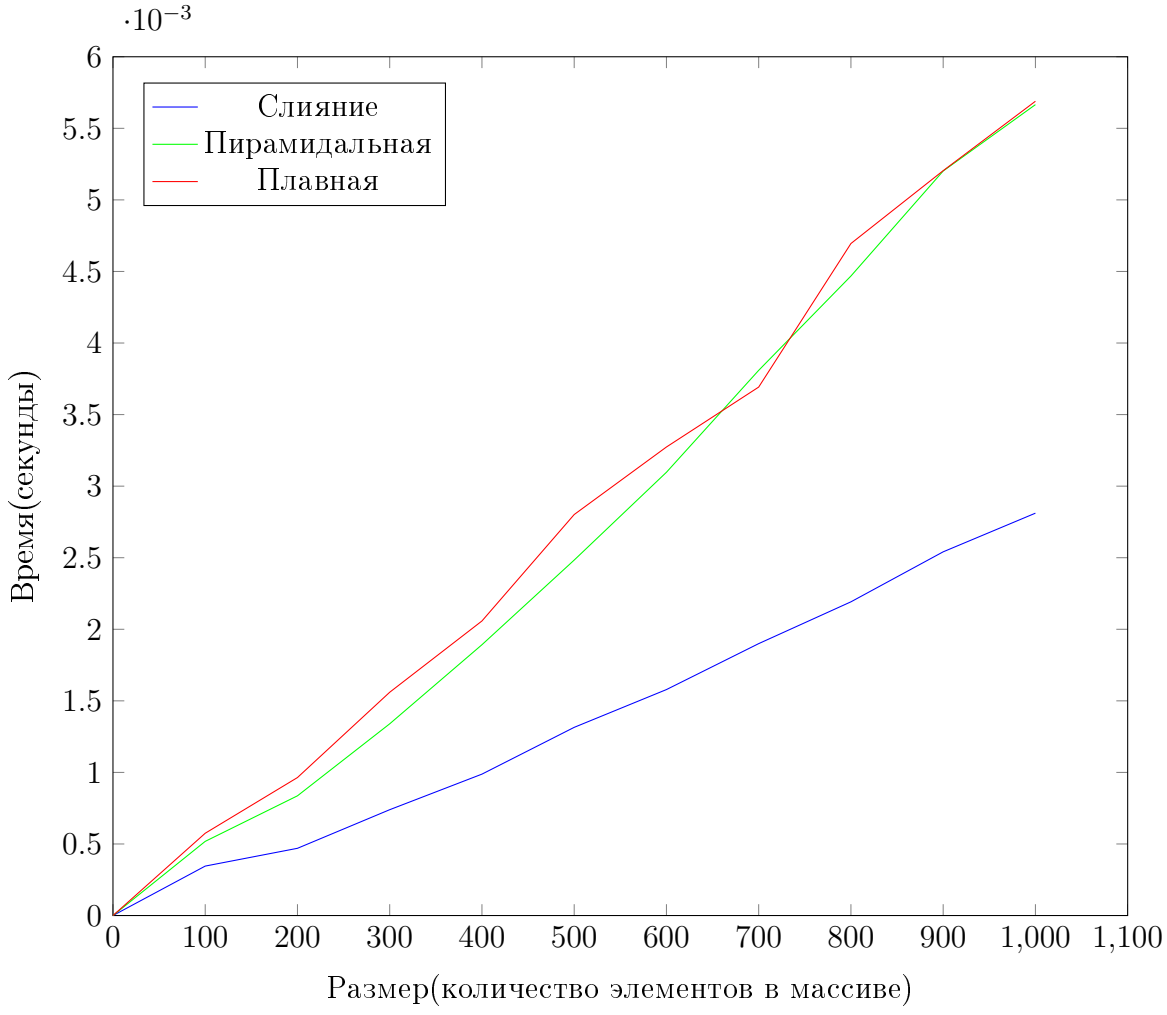

Рис. 13. График зависимости времени выполнения алгоритма от размера массива

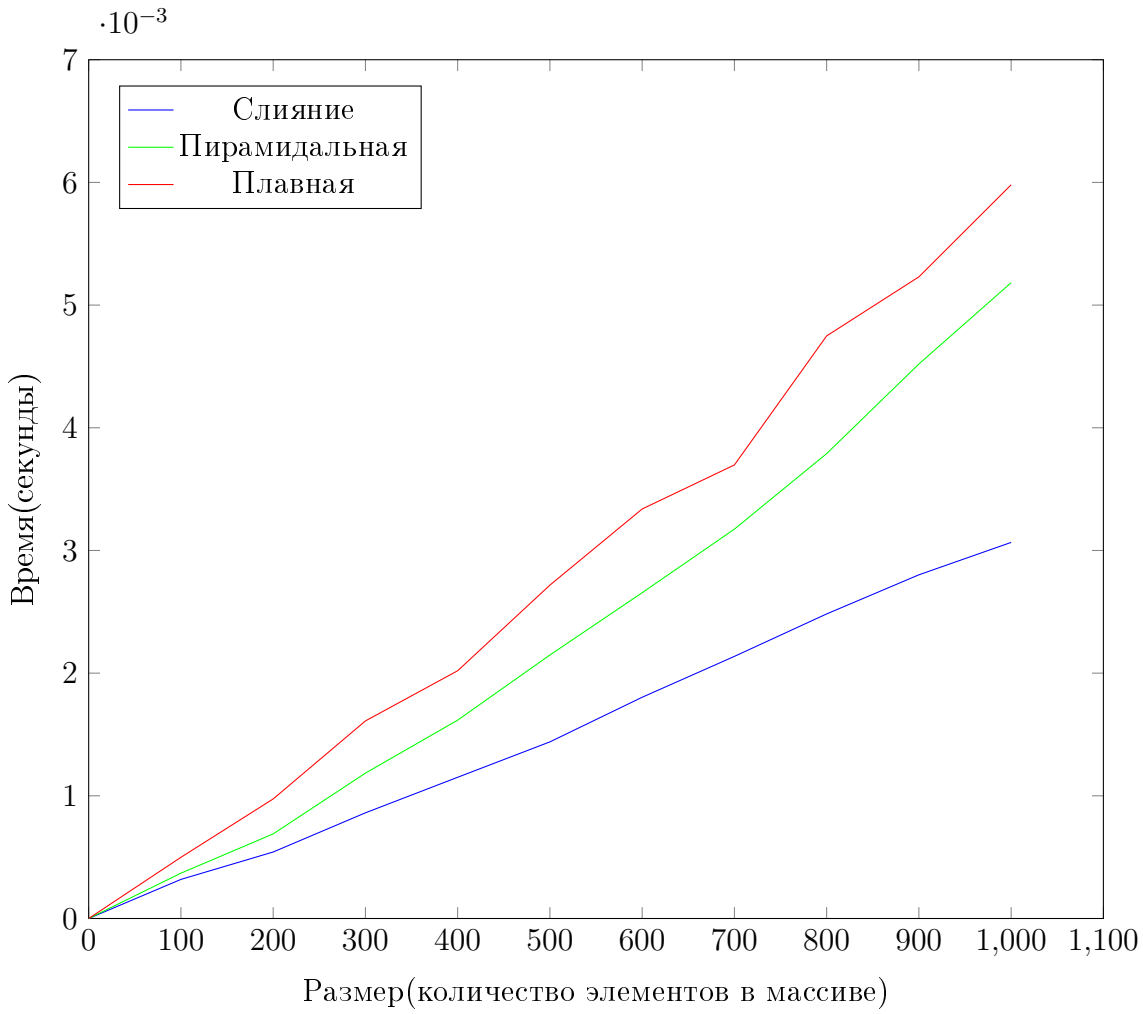

Рис. 14. График зависимости времени выполнения алгоритма от размера массива

Время работы алгоритмов сортировки на обратно отсортированных массивах представлены на графике 14

4.3. Сравнительный анализ на материале экспериментальных данных

Согласно Дж.Макконнеллу, трудоёмкость лучшего и худшего случаев пирамидальной сортировки равна $O(n \cdot \log(n))$, трудоёмкость лучшего и хдшего случаев сортировки слиянием - $O(n \cdot \log(n))$. Трудоёмкость плавной сортировки в худшем случае совпадает с худшим случаем пирамидальной сортировки, в лучшем - стремится к $O(n)$. [7]

Расчитанное значение трудоёмкости сортировки слиянием совпало с результатом, представленным в книге Дж.Макконелла, трудоёмкость сортировки слиянием в лучшем и худшем случаях равна $O(n \cdot \log(n)))$.

Экспериментально было получено, что для конкретной реализации на упорядоченных и на обратно отсортированных массивах сортировка слиянием показывает лучшее время, чем пирамидальная и плавная сортировки (в среднем на 0.0025 секунды, что соответствует 16%) при одинаковой трудоёмкости $O(n \cdot \log(n))$, в то время как пирамидальная и плавная сортировки различаются между собой в среднем на 1%. Сортировка слиянием показывает примерно равное время как на отсортированных, так и на хаотичных выборках. На случайной выборке для конкретной реализации разница между сортировкой слиянием и пирамидальной сортировкой составляет 8%, между пирамидальной и плавной - 7%, между сортировкой слиянием и плавной сортировкой - 15%.

Заключение

Во время выполнения работы было ознакомление с алгоритмами сортировки (слиянием, обладающая трудоёмкостью $O(n \cdot \log(n))$, пирамидальная сортировка с трудоёмкостью $O(n \cdot \log(n))$ и плавная сортировка с трудоёмкостью $O(n \cdot \log(n))$ в общем случае и трудоёмкостью, стремящейся к $O(n)$ для частично отсортированного массива), были получены практические навыки их реализации, изучены оценки трудозатратности на примере сортировки слиянием, было проведено экспериментальное сравнение временных характеристик сортировок на трёх типах выборок.

В результате расчётов было доказано, что сортировка слиянием в лучшем и в худшем случаях имеет трудоёмкость $O(n \cdot \log(n))$.

Список литературы

[1] Сортировка слиянием. – URL: *http* : *//sortings.github.io/sort$_t$ypes/merge.html*

[2] Smoothsort Demystified. – URL: *http* : *//www.keithschwarz.com/smoothsort/*

[3] Smooth sort, an alternative for sorting in situ. – URL: *http* : *//www.enterag.ch/hartwig/order/smoothsort.pdf*

[4] Плавная сортировка (Smooth_sort). – URL: *http* : *//cppalgo.blogspot.com*/2010/10/*smoothsort.html*

[5] Алгоритмы и структуры данных для начинающих: сортировка. – URL: *https* : *//tproger.ru/translations/sorting − for − beginners/*

[6] Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы. Построение и анализ // 2-е издание. - 2005. - Глава 6. - С.178-197. // URL: *http* : *//mexalib.com/view*/28453

[7] Макконелл Дж. Анализ алгоритмов. Вводный курс // 2-е издание. - 2004. - Глава 3. - С.92-105.

[8] Головешкин В.А., Ульянов М.В. Теория рекурсии для программистов // 2006. - Глава 6. - С.172-190.