МЕЖГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Автоматизированные системы управления»

ТЕОРИЯ АЛГОРИТМОВ

Методические рекомендации к лабораторным работам для студентов направления подготовки 09.03.01 «Информатика и вычислительная техника» очной формы обучения

Могилев 2023

УДК 604.43
ББК 32.973-018.1
Т39

Рекомендовано к изданию
учебно-методическим отделом
Белорусско-Российского университета

Одобрено кафедрой «Автоматизированные системы управления» «31» августа 2023 г., протокол № 1

Составитель ст. преподаватель И. А. Беккер

Рецензент канд. техн. наук, доц. В. В. Кутузов

Методические рекомендации к лабораторным работам содержат краткие теоретические сведения, задания, контрольные вопросы. Предназначены для студентов направления подготовки 09.03.01 «Информатика и вычислительная техника» очной формы обучения.

Учебное издание

ТЕОРИЯ АЛГОРИТМОВ

Ответственный за выпуск	А. И. Якимов
Корректор	А. А. Подошевко
Компьютерная верстка	Е. В. Ковалевская

Подписано в печать . Формат 60×84/16. Бумага офсетная. Гарнитура Таймс.
Печать трафаретная. Усл. печ. л. . Уч.-изд. л. . Тираж 21 экз. Заказ №

Издатель и полиграфическое исполнение:
Межгосударственное образовательное учреждение высшего образования
«Белорусско-Российский университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/156 от 07.03.2019.
Пр-т Мира, 43, 212022, г. Могилев.

Содержание

Введение

Дисциплина «Теория алгоритмов» является важным инструментом при изучении различных дисциплин специальности. В результате изучения дисциплины у студентов формируются знания терминов, основных фактов и закономерностей теории алгоритмов, умения анализировать алгоритмы.

Методические рекомендации к лабораторным работам являются неотъемлемой частью комплексного методического обеспечения дисциплины и способствуют развитию способности применять методы теоретического и экспериментального исследования алгоритмов и оценивать временную и емкостную сложность программного обеспечения.

Процесс выполнения и защиты лабораторной работы студентом состоит в том, что следует успешно пройти следующие этапы учебной деятельности:

– выполнить согласно варианту, выданному преподавателем, задание;

– составить отчет с описанием хода работы и результатов, сделать вывод по заданию (требования к отчету представлены в приложении А);

– предъявить разработанное программное обеспечение с тестированием его на различных наборах входных данных;

– продемонстрировать знание теоретического материала применительно к заданию лабораторной работы (знать все термины, свойства, закономерности, встречающиеся в работе; ответить на контрольные вопросы, приведенные в конце лабораторной работы).

1 Лабораторная работа № 1. Анализ алгебраических алгоритмов

Цель работы: научиться для нерекурсивного алгоритма строить точную функцию трудоемкости теоретически и находить ее экспериментально, выполнять ее асимптотическую оценку.

Теоретические сведения

Трудоемкость конструкции «Следование» есть сумма трудоемкостей блоков, следующих друг за другом. Таким образом, для вычисления теоретически трудоемкости f_A алгоритма A суммируют трудоемкости последовательно идущих друг за другом блоков кода.

С учетом договоренности о том, какие операции в алгоритме мы считаем элементарными, трудоемкость строки кода

```
a[1] = a[0] + 2*(1 + itog)
```

будет равна 6.

Основными блоками алгоритма являются «Ветвление» и «Цикл».

Теоретическая трудоемкость конструкции If вычисляется как

$$F_{If} = f_{условия} + f_+ \cdot p_+ + f_- \cdot p_- , \quad (1)$$

где f_+ и f_- – трудоемкости ветвей;

p_+ и p_- – вероятности попадания на конкретную ветвь блока условия.

Проверка условия выполняется в любом случае.

Экспериментальная трудоемкость блока, алгоритма это количество элементарных операций, подсчитанных счетчиком в коде.

Для конструкции If теоретически подсчитанное и полученное экспериментально значение трудоемкости могут отличаться. Дело в том, что теоретически рассчитанная вероятность и вероятность, полученная экспериментально, различаются методом вычисления.

Трудоемкость конструкции For зависит от количества итераций и в общем случае в C# определяется по формуле

$$F_{For} = 2 + 2N + N \cdot f_{тела\ цикла}, \quad (2)$$

где N – количество итераций цикла.

Задание

Напишите программный код, реализующий поставленную задачу с помощью циклов for, while; оптимизируйте код, эффективно используя операторы цикла.

1 Определите точную трудоемкость алгоритма, анализируя псевдокод, оцените ее асимптотически.

2 Экспериментально в программной реализации выведите результат и значение счетчика трудоемкости.

3 Определите время выполнения программного кода.

Варианты

1 В одномерном массиве найти произведение минимального и максимального элемента.

2 Найти сумму минимальных элементов двух одномерных массивов.

3 Найти в векторе сумму всех входящих в него элементов.

4 Определить, есть ли в векторе нулевой элемент.

5 Определить количество единиц в векторе.

6 Найти максимальный элемент над главной диагональю матрицы.

7 Подсчитать количество нечетных чисел над главной диагональю матрицы.

8 Найти минимальный элемент под главной диагональю матрицы.

9 Найти количество четных чисел под главной диагональю матрицы.

10 Найти сумму элементов главной диагонали матрицы.

11 Найти на дополнительной диагонали матрицы наименьший элемент.

12 Выяснить, сколько нулей содержится на дополнительной диагонали матрицы.

13 Найти минимальный элемент в третьем столбце матрицы A(n × 2n).

14 Найти максимальный элемент во второй строке матрицы A(n × (n + 2)).

15 Найти максимальный элемент во второй и третьей строках матрицы A(n × (n + 3)).

16 Найти суммы элементов в первом и четвертом столбцах матрицы A(n × 2n).

17 Найти сумму элементов в первом и четвертом столбцах матрицы A(n × 2n).

18 Найти разность максимального и минимального элементов в квадратной матрице.

19 Найти количество элементов, кратных 3, в первой строке матрицы A(n × 3n).

20 Суммировать матрицы A(n × 2n) и B(n × 2n).

21 Умножить матрицы A(n × n) и B(n × n).

22 Умножить матрицы A(n × 2n) и B(2n × n).

23 Умножить матрицы A(2n × n) и B(n × 2n).

24 Найти матрицу A + 2B для матриц A(n × 4n) и B(n × 4n).

25 Найти матрицу 2A + 3B для матриц A(n × 2n) и B(n × 2n).

26 Найти матрицу 5A + B для матриц A(3n × n) и B(3n × n).

27 Транспонировать квадратную матрицу.

28 Транспонировать произвольную матрицу.

29 Транспонировать вектор-столбец.

30 Транспонировать вектор-строку.
31 Транспонировать матрицу A(n × (2n + 1)).
32 Для матрицы A(n × m) посчитать количество ненулевых элементов.
33 В матрице C(m × 4m) вычислить сумму неотрицательных элементов.
34 В матрице A((n + 1) × n) посчитать количество нулевых элементов.
35 Отсортируйте одномерный массив по возрастанию.

Контрольные вопросы

1 Классифицируйте алгоритмы в зависимости от поведения функции трудоемкости на различных входных данных.
2 Опишите правила нахождения точной функции трудоемкости по псевдокоду алгоритма.
3 Опишите, как найти трудоемкость цикла в C# с учетом использования операции i++.
4 Дайте определение асимптотическим оценкам (с графической интерпретацией и символьной записью).
5 Почему для конструкции If теоретически подсчитанное и полученное экспериментально значение трудоемкости могут отличаться?
6 Есть ли смысл подбирать значение p для F_{If}?
7 Как встроить в If счетчик трудоемкости?
8 Как встроить счетчик трудоемкости в For?

2 Лабораторная работа № 2. Анализ алгоритмов, вычисляющих значение многочлена по схеме Горнера

Цель работы: сформировать знания и умения по работе с такими структурами данных, как последовательности, умения анализировать алгоритм с помощью функции трудоемкости.

Теоретические сведения

Задать многочлен стандартного вида можно последовательностью его коэффициентов. Предположим, что последовательность числовых элементов находится в одномерном массиве (в реальных программах она также может читаться из файла, из списка).

Отметим следующие составные части алгоритма, вычисляющего функцию на последовательности элементов, например, при суммировании элементов последовательности:

1) инициализация значения функции s для пустой последовательности s : = 0.0;

2) вычисление нового значения функции по прочтению очередного элемента последовательности. Новое значение вычисляется по старому значению и очередному прочитанному элементу. В данном случае это суммирование s : = s + a[i].

Эти две части присутствуют в любом алгоритме, вычисляющем функцию на последовательности.

Пусть дан многочлен

$$p(x) = a_0x^n + a_1x^{n-1} + ... + a_n. \quad (3)$$

Нужно вычислить значение многочлена в точке x_0.

Сначала для удобства рассмотрения возьмем многочлен третьей степени $p(x) = ax^3 + bx^2 + cx + d$. Если не считать d, то у оставшихся слагаемых есть общий множитель x, вынесем его за скобки. В скобках проделаем эти же действия. Получается, что исходный многочлен можно представить в виде $p(x) = ((ax + b)x + c)x + d$ (схема Горнера) и для вычисления значения многочлена третьей степени по схеме Горнера достаточно трех умножений и трех сложений.

Таким образом, в программе потребуется ввести значение x_0, степень многочлена и его коэффициенты. Коэффициенты многочлена удобно ввести в одномерный числовой массив (вектор).

В общем случае многочлен в схеме Горнера представляется в виде

$$p(x) = (... ((a_0x + a_1)x + a_2)x + ... + a_{n-1})x + a_n. \quad (4)$$

Заметим, что начиная с внутренней скобки, вычислительные действия повторяются: надо промежуточное значение многочлена умножить на значение x, а затем прибавить к произведению новый коэффициент.

Задание

Разработать алгоритм по схеме Горнера согласно варианту, реализовать свой алгоритм в программной среде. Выполнить анализ алгоритма: вычислить его трудоемкость теоретически и проверить ее экспериментально, введя в код программы счетчик элементарных операций. Сделать выводы о соответствии значений нескольких пар чисел теоретической и полученной экспериментально трудоемкости для разных входных значений.

Варианты

1 Разработать алгоритм схемы Горнера с использованием арифметического цикла.

2 Используя арифметический цикл с отрицательным шагом, разработать алгоритм схемы Горнера в случае, когда коэффициенты многочлена заданы по возрастанию, а не по убыванию степеней.

3 Выполнить алгоритм схемы Горнера с использованием цикла ПОКА.

Контрольные вопросы

1 Как ввести многочлен в консоль?
2 Какие типы и структуры данных использовались в программе?
3 Что называют трудоемкостью алгоритмов?
4 Какие операции относят к элементарным?

3 Лабораторная работа № 3. Сравнение алгоритмов сортировки

Цель работы: сформировать знания и умения организации сортировки с выполнением анализа алгоритма с учетом рассмотрения лучшего, среднего, худшего случаев.

Теоретические сведения

Рассмотрим несколько простых методов, называемых прямыми, где требуется порядка n^2 сравнений элементов.

***Сортировка вставками Insertion sort* (*с помощью прямого включения*)**

Первый прямой метод, который будем рассматривать, называется сортировкой **вставками**, т. к. на i-м этапе i-й элемент A[i] вставляется в нужную позицию среди элементов A[1], A[2], ..., A[i – 1], которые уже упорядочены. После этой вставки первые i элементов будут упорядочены.

При сортировке элементов массива вставками массив фактически делится на каждом шаге на две части: отсортированную, в которую упорядоченно вставляются элементы – a_1, …, a_{i-1} и еще не отсортированную – a_i, …, a_n. В начале работы в качестве отсортированной части берем только один первый элемент, а в качестве неотсортированной – все остальные. На каждом шаге, начиная с i = 2, из неотсортированной последовательности извлекается i-й элемент и вставляется в отсортированную так, чтобы не нарушить в ней упорядоченности элементов.

Каждый шаг алгоритма включает четыре действия.

1 Взять i-й элемент массива (он же является первым элементом в неотсортированной части) и сохранить его в дополнительной переменной.

2 Найти позицию j в отсортированной части массива, куда будет вставлен i-й элемент, значение которого теперь хранится в дополнительной переменной. Вставка этого элемента не должна нарушить упорядоченности элементов отсортированной части массива.

3 Сдвинуть элементы массива с i – 1 позиции по j-ю на один элемент вправо, чтобы освободить найденную позицию для вставки.

4 Вставить взятый элемент в найденную j-ю позицию.

Сказанное можно записать в виде псевдокода (псевдопрограммы):

```
for i:= 2 to n do
  переместить A[i] на позицию j ≤ i такую, что
  A[i] < A[k] для j < k < i  и
  либо A[i] > A[j – 1], либо j = 1
```

Чтобы сделать процесс перемещения элемента A[i] более простым, можно ввести элемент A[0], чье значение ключа будет меньше значения ключа любого элемента A[1], ..., A[n]. Если такую константу нельзя применить, то при вставке A[i] в позицию j – 1 надо проверить, не будет ли j = 1, если нет, тогда нужно сравнивать элемент A[i] (который сейчас находится в позиции j) с элементом A[j –1]. Описанный алгоритм показан в листинге 1. Стандартная процедура swap используется для перестановки элементов местами.

Листинг 1. Сортировка вставками

```
1   A[0].key := -∞;
2     for i:= 2 to n do begin
3       j:= i;
4     while A[j] < A[j – 1] do begin
5       swap(A[j], A[j – 1]);
6       j:= j – 1
7     end;
8     end.
```

Число сравнений C_i при i-м проходе в худшем случае равно i – 1, в лучшем – единице. Если считать, что все перестановки из n элементов равновероятны, то среднее число сравнений i/2.

Число перестановок $M_i = C_i + 2$.

Общее число сравнений и число перестановок таковы:

$C_{min} = n - 1$; $M_{min} = 3(n - 1)$;

$C_{ave} = (n^2 + n - 2)/4$; $M_{ave} = (n^2 + 9n - 10)/4$;

$C_{max} = (n^2 + n - 4)/4$; $M_{max} = (n^2 + 3n - 4)/2$.

Данный алгоритм показывает наилучшие результаты работы в случае уже упорядоченной исходной части массива, наихудшие – когда элементы первоначально расположены в обратном порядке. Он эффективен на неболь-ших наборах данных, на наборах данных до десятков элементов может оказаться лучшим; также он эффективен на наборах данных, которые уже частично отсортированы.

Это устойчивый алгоритм сортировки (не меняет взаимного расположения равных элементов). Если элемент состоит только из одного ключа (т. е. значения, по которому и производится сортировка), то устойчивость сортировки не важна, но при сортировке записей (например, с полями Фамилия и Имя) в поддержании исходного порядка может быть смысл, т. к. равные по ключу элементы различны.

Устойчивыми алгоритмами сортировки также являются пузырьковая, слиянием.

Неустойчивые алгоритмы сортировки: быстрая сортировка Хоара, пирамидальная, Шелла.

Сортировка выбором имеет как устойчивые, так и неустойчивые реализации.

Устойчивость сортировки всегда может быть достигнута путём удлине-

ния исходных ключей, если включить в них информацию об первоначальном порядке значений.

***Сортировка выбором* (*Selection sort*)**

При сортировке с помощью **прямого выбора** массив также делится на две части: отсортированную последовательность – a_1, …, a_{i-1} и «исходную» – a_i, …, a_n. Происходит поиск одного элемента из исходной последовательности, который обладает наименьшим (наибольшим) значением ключа, и уже найденный элемент помещается в конец готовой последовательности.

На момент начала сортировки методом прямого выбора готовая последовательность считается пустой, соответственно, исходная последовательность включает в себя все элементы массива.

Алгоритм сортировки с помощью прямого выбора.

1 Из всего массива выбирается элемент с наименьшим значением ключа.

2 Он меняется местами с первым элементом a_1.

3 Затем этот процесс повторяется с оставшимися n – 1 элементами, n – 2 элементами и т. д. до тех пор, пока не останется один элемент с наибольшим значением.

На i-м этапе сортировки **выбором** выбирается запись с наименьшим ключом среди записей A[i], ..., A[n] и меняется местами с записью A[i]. В результате после i-го этапа все записи A[1], ..., A[i] будут упорядочены.

Сортировку выбором можно описать на псевдокоде следующим образом:

```
for i:= 1 to n – 1 do
  выбрать среди A[i], …, A[n] элемент с наименьшим
  ключом и поменять его местами с A[i];
```

Листинг 2. Сортировка посредством выбора

```
1   var lowkey: keytype; {текущий наименьший ключ,
    найденный при проходе по элементам A[i], … A[n]}
2   lowindex: integer; {позиция элемента с ключом
    lowkey}
3   begin
4     for i:= 1 to n – 1 do begin
5       lowindex: = i;
6       lowkey:= A[i].key;
7       for j:= i + 1 to n do begin
8       {сравнение ключей с текущим ключом lowkey}
9         if A[j].key < lowkey then begin
10          lowkey:= A[j].key;
11          lowindex:= j
12        end;
13      swap(A[i], A[lowindex])
14      end;
15    end;
16  end.
```

При работе данного алгоритма число сравнений элементов C не зависит от начального порядка элементов. $C = (n^2 – n)/2$.

Лучший случай: $M_{min} = 3(n – l)$.

Худший случай: $M_{max} = n^2/4 + 3(n – 1)$.

В общем случае алгоритм с прямым выбором предпочтительнее алгоритма прямого включения. Однако если элементы вначале упорядочены (почти упорядочены), алгоритм с прямым включением выполнит сортировку быстрее.

Сортировка «пузырьком»

Самым простым методом сортировки является **метод «пузырька»**. Он относится к методам прямого обмена, сравнивая и меняя местами соседние элементы.

Чтобы описать основную идею этого метода, представим, что записи, подлежащие сортировке, хранятся в массиве, расположенном вертикально. Записи с малыми значениями ключевого поля более «легкие» и «всплывают» вверх наподобие пузырька.

При первом проходе вдоль массива, начиная проход снизу, берем первую нижнюю запись массива и ее ключ поочередно сравниваем с ключами последующих записей. Если встречается запись с более «тяжелым» ключом, то эти записи меняются местами. При встрече с записью с более «легким» ключом эта запись становится эталоном для сравнения, и все последующие записи сравниваются с этим новым ключом. В результате запись с наименьшим значением ключа окажется в самом верху массива.

Во время второго прохода вдоль массива находится запись со вторым по величине ключом, которая помещается под записью, найденной при первом проходе массива, т. е. на вторую сверху позицию, и т. д.

Отметим, что во время второго и последующих проходов вдоль массива нет необходимости просматривать записи, найденные за предыдущие проходы, т. к. они имеют ключи меньшие, чем у оставшихся записей. Другими словами, во время i-го прохода не проверяются записи, стоящие на позициях выше i (листинг 3).

Листинг 3. Алгоритм «пузырька»

```
1  for i:= 1 to n – 1 do begin
2     for j:= 2 to i + 1 do begin
3        if A[j].key < A[j – 1].key then
4          swap(A[j], A[j – 1]);
5     end;
6  end.
```

Алгоритм прямого обмена основывается на сравнении и перестановке пары соседних элементов и продолжении этого процесса до тех пор, пока не будут упорядочены все элементы.

В пузырьковой сортировке легкие элементы быстро всплывают, а легкие – медленно тонут, потому что цикл сравнения продвигается от конца массива к началу и «подталкивает» вверх легкие элементы. А если двигаться от начала к концу, то легкие элементы станут медленно всплывать, а тяжелые быстро тонуть (цикл сравнения продвигается к концу массива и «тащит» с собой тяжелые элементы).

Число сравнений в алгоритме сортировки «пузырьком» $C = (n^2 - n)/2$, а минимальное и максимальное число перестановок элементов следующее:

$$M_{min} = 0;$$

$$M_{max} = 3(n^2 - n)/4.$$

Вычислительная сложность этого алгоритма: худшее время $O(n^2)$, среднее время $\Theta(n^2)$, лучшее время $\Omega(n)$.

Шейкерная сортировка

Пример пузырьковой сортировки отражает асимметрию работы алгоритма: легкий «пузырек» всплывает сразу – за один проход, а тяжелый тонет очень медленно – за один проход на одну позицию.

Так, массив **10 22 33 55 70 90 1** с помощью сортировки «пузырьком» будет упорядочен за один проход, а для массива **90 1 10 22 33 55 70** потребуется шесть проходов.

Это наводит на мысль о следующем улучшении: менять направление просмотра на каждом последующем проходе. Алгоритм, реализующий такой подход, называется шейкерной сортировкой.

Такой подход не позволяет уменьшить вычислительную сложность по сравнению с пузырьковой сортировкой и вычислительная сложность здесь такая же: худшее время $O(n^2)$, среднее время $\Theta(n^2)$, лучшее время $\Omega(n)$.

Шейкерная сортировка с успехом используется лишь в тех случаях, когда известно, что элементы почти упорядочены, что на практике бывает весьма редко.

Анализ показывает, что «обменная» сортировка и ее усовершенствования фактически оказываются хуже сортировок с помощью включений и с помощью выбора.

Сортировка расческой

Сортировка расческой улучшает сортировку пузырьком, и конкурирует с алгоритмами, подобными быстрой сортировке. Основная идея – устранить маленькие значения в конце списка, которые крайне замедляют сортировку пузырьком (интересно, что большие значения в начале списка не представляют проблемы для сортировки пузырьком).

В сортировке пузырьком, когда сравниваются два элемента, расстояние между ними (разность индексов) равно 1. Основная идея сортировки расческой в том, что этот промежуток может быть гораздо больше, чем единица.

Сначала расстояние между элементами максимально, и равно размеру массива минус один. Затем, пройдя массив с этим шагом, необходимо поделить шаг на фактор уменьшения (его оптимальное значение 1,247) и пройти по списку вновь. Так продолжается до тех пор, пока разность индексов не достигнет единицы. В этом случае сравниваются соседние элементы, как и в сортировке пузырьком, но такая итерация будет только одна.

Эта сортировка является неустойчивой, т. е. она может менять порядок одинаковых элементов.

Такой подход позволяет снизить вычислительную сложность алгоритма: худшее время $O(n^2)$, лучшее время $\Omega(n \log n)$.

Сортировка Шелла (*Shell sort*)

Сортировка Шелла также основана на идее увеличить расстояние между сравниваемыми элементами, но она является модификацией сортировки вставками, а не сортировки пузырьком.

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, а завершается сортировка Шелла упорядочиванием элементов при d = 1 (т. е. 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).

Среднее время работы алгоритма зависит от длин промежутков, на которых будут находиться сортируемые элементы исходного массива размера N на каждом шаге алгоритма. Существует несколько подходов к выбору этих значений.

Первоначально использованная Шеллом последовательность промежутков:

$$d_{нач} = n/2;\ d_i = d_{i-1}/2;\ d_{конеч} = 1.$$

В худшем случае сложность такого алгоритма составит $O(n^2)$.

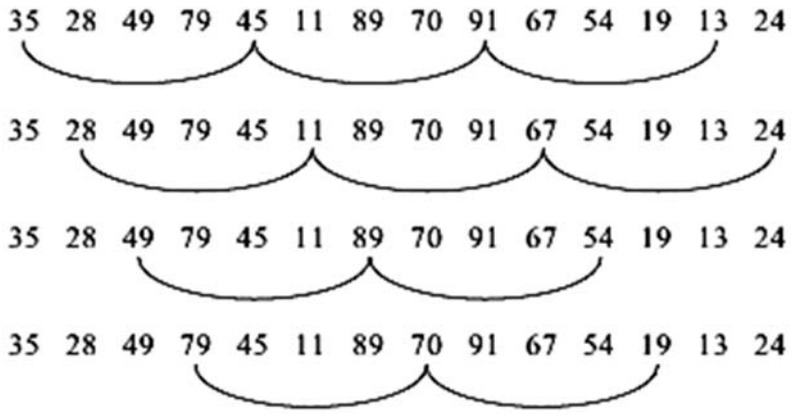
Эмпирическая последовательность Циура с $d \in \{1, 4, 10, 23, 57, 132, 301, 701, 1750\}$ является одной из лучших для сортировки массива ёмкостью приблизительно до 4000 элементов.

Среднее расстояние, на которое должен продвигаться каждый из n элементов во время сортировки, равно n/3 позиций. Это число является целью в поиске более эффективных методов сортировки.

Рассмотрим работу алгоритма на примере следующего массива:

35 28 49 79 45 11 89 70 91 67 54 19 13 24.

Сначала отдельно сгруппируем элементы, отстоящие друг от друга на расстоянии d = 4. Таких групп будет четыре. Элементы, принадлежащие одной группе, объединены дугами.

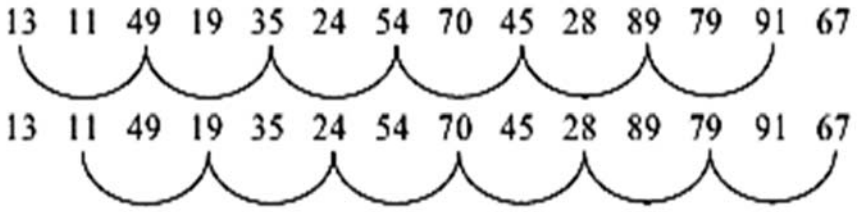


Следующим шагом выполняется сортировка внутри каждой из групп методом прямого включения. Сначала сортируются элементы **35, 45, 91, 13**, затем **28, 11, 67, 24**, следом **49, 89, 54** и, наконец, **79, 70, 19**. В результате получаем массив

13 11 49 19 35 24 54 70 45 28 89 79 91 67.

Такой процесс называется четвертной сортировкой.

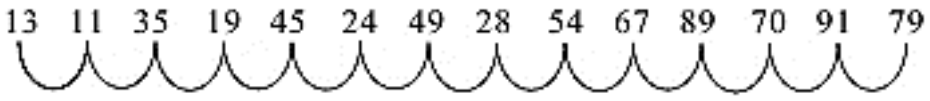
Следующим проходом элементы группируются так, что теперь в одну группу входят элементы, отстоящие друг от друг на две позиции. Таких групп две: **13, 49, 35, 54, 45, 89, 91** и **11, 19, 24, 70, 28, 79, 67.**



Вновь в каждой группе выполняется сортировка с помощью прямого включения. Это называется двойной сортировкой, ее результатом будет массив:

***13* 11 *35* 19 *45* 24 *49* 28 *54* 67 *89* 70 *91* 79.**

И, наконец, на третьем проходе идет обычная или одинарная сортировка с помощью прямого включения.



Результатом работы алгоритма сортировки Шелла является отсортированный массив:

11 13 19 24 28 35 45 49 54 67 70 79 89 91.

Расстояния в группах можно уменьшать по-разному, лишь бы последнее было единичным, ведь в самом плохом случае последний проход и сделает всю работу. Однако совсем не очевидно, что такой прием «уменьшающихся расстояний» может дать лучшие результаты, если расстояния не будут степенями двойки.

При выполнении лабораторной работы рекомендуется начальный шаг взять равным $h_{нач} = n/3$ (здесь n – количество элементов массива) и уменьшать его на каждом проходе вдвое:

$$h_{i-1} = h_i/2;\ h_{конеч} = 1.$$

Математический анализ показывает, что для сортировки n элементов методом Шелла затраты пропорциональны $\mathbf{n^{1,2}}$, что значительно лучше $\mathbf{n^2}$, необходимых для методов прямой сортировки.

Сравнение различных алгоритмов сортировки

Эффективность различных алгоритмов сортировки массивов, состоящих из n сортируемых элементов, проведем по двум критериям (таблица 1): числу необходимых сравнений элементов C и числу перестановок элементов M.

Таблица 1

Метод	Количество сравнений C и перестановок M		
	минимальное	среднее	максимальное
Прямое включение	$C_{min} = n - 1$ $M_{min} = 3(n - 1)$	$C_{ave} = (n^2 + n - 2)/ 4$ $M_{ave} = (n^2 + 9n - 10)/4$	$C_{max} = (n^2 + n - 4)/4$ $M_{max} = (n^2 + 3n - 4)/2$
Прямой выбор	$C_{min} = (n^2 - n)/2$ $M_{min} = 3(n - 1)$	$C_{ave} = (n^2 - n)/2$ $M_{ave} = n * (\ln n + 0{,}57)$	$C_{max} = (n^2 - n)/2$ $M_{max} = n^2/4 + 3(n - 1)$
Прямой обмен	$C_{min} = (n^2 - n)/2$ $M_{min} = 0$	$C_{ave} = (n^2 - n)/2$ $M_{ave} = (n^2 - n) * 0{,}75$	$C_{max} = (n^2 - n)/2$ $M_{max} = 1{,}5(n^2 - n)$

Для усовершенствованных методов нет простых и точных формул. Существенно, что в случае сортировки Шелла вычислительные затраты составляют $c*n^{1.2}$, в то время как для прямых методов – $c*n^2$. Очевидно преимущество сортировки Шелла как улучшенного метода по сравнению с прямыми методами сортировки.

Пузырьковая сортировка определенно наихудшая из всех сравниваемых. Ее усовершенствованная версия – шейкерная сортировка – продолжает

оставаться плохой по сравнению с прямым включением и прямым выбором (за исключением уже упорядоченного массива).

Задание

1 Реализовать в программе два алгоритма сортировки из указанных ниже. Сравнить эффективность реализованных алгоритмов по числу перестановок. Сравнить в программе время выполнения каждого из методов:

а) сортировка с помощью прямого включения;

б) сортировка с помощью прямого выбора при помощи поиска минимального элемента;

в) сортировка с помощью прямого выбора при помощи поиска одновременно минимального и максимального элементов;

г) сортировка «пузырьком»;

д) шейкерная сортировка;

е) сортировка расческой;

ж) сортировка Шелла.

2 Дан линейный массив из n целых чисел. Упорядочить его:

а) сортировкой «пузырьком» в порядке возрастания модулей элементов;

б) переставив все нулевые элементы в конец массива (порядок ненулевых элементов может быть произвольным);

в) так, чтобы все четные числа стояли в начале массива, а нечетные – в конце (порядок четных (нечетных) элементов между собой может быть произвольным);

г) переставив все нулевые элементы в конец массива (порядок следования элементов должен сохраниться);

д) так, чтобы все положительные числа стояли в начале массива, а отрицательные – в конце (порядок отрицательных (положительных) элементов между собой может быть произвольным);

е) чтобы все положительные числа стояли в начале массива, а отрицательные – в конце (порядок следования элементов должен сохраниться).

3 Дана матрица A_{nxm}. Найти:

а) номера строк, элементы которых расположены в возрастающем порядке;

б) номера столбцов, элементы которых расположены в убывающем порядке.

4 Дан одномерный массив из n целых чисел, упорядоченных в порядке возрастания. Необходимо некоторое число M вставить в данный массив, не нарушая его упорядоченность.

5 Задан числовой вектор из n элементов. Требуется получить в порядке возрастания все различные числа-элементы вектора.

4 Лабораторная работа № 4. Анализ рекурсивных алгоритмов

Цель работы: проанализировать сложность рекурсивного алгоритма сортировки вставками.

Теоретические сведения

Время работы рекурсивных алгоритмов складывается из:

1) времени D(n) разбиения исходной задачи на подзадачи (это действие выполняется на входе в рекурсию);

2) $T(n_1) +\ldots+ T(n_k)$ – времени решения каждой подзадачи в отдельности (количество подзадач k определяется числом рекурсивных вызовов);

3) S(n) – времени на соединение полученных решений (это действие выполняется на выходе из рекурсии).

Рассмотрим рекурсивный алгоритм сортировки слиянием.

Листинг

```
Type OdMas=Array[1..1000] Of Integer;
Var A : OdMas;
    n : Integer;
…
Procedure Sli(l, s, r : Integer);
Var B : OdMas;
    i, j, k : Integer;
Begin
k:=l; i:=l; j:=s; B:=A;
Repeat
 If A[i]<A[j] Then Begin B[k]:=A[i]; Inc(i) End
               Else Begin B[k]:=A[j]; Inc(j) End;
  Inc(k);
 Until (i>s) Or (j=r);
 If i>s Then
   For i:=j To r Do Begin B[k]:=A[i]; Inc(k) End
  Else For j:=i To s Do Begin B[k]:=A[j]; Inc(k) End;
 A:=B
End;
Procedure Sort_Sli(l, r : Integer);
Var s : Integer;
Begin
 If l<r Then Begin
  s:=(l+r) Div 2;
  Sort_Sli(l, s);
Sort_Sli(s+1, r);
  Sli(l, s, r)
 End
End;
Begin {основная программа}
 …
 Sort_Sli(1, n);
 …
End.
```

Процедура Sl (процедура слияния отсортированных частей массива) – нерекурсивная процедура, следовательно она анализируется ранее рассмотренным методом. Временная сложность данной процедуры – Θ(n).

Процедура Sort_Sl (основная процедура сортировки слиянием) – рекурсивная процедура. Определим временную сложность данной процедуры.

Очевидно, что если массив состоит из одного элемента (n = 1), то выполняется только проверка условия (l < r), следовательно время работы пропорционально единице T(n) = Θ(1).

Если же в массиве более одного элемента (n > 1), то выполняются следующие действия:

```
      If l<r Then Begin
        s:=(l+r) Div 2;     {1 - Действия  на  входе
в  рекурсию, разбиение  исходной  задачи
на подзадачи}

         Sort_Sli(l, s);
        Sort_Sli(s+1, r);   {2 - два рекурсивных  вызова}

        Sli(l, s, r);       {3 - действия  на  выходе
из  рекурсии}
      End
```

Здесь инструкция кода 1 – это разбиение исходной задачи на подзадачи, т. е. D(n) = Θ(1).

Инструкция кода 2 означает, что за два рекурсивных вызова задача разбивается на две подзадачи равной длины:

$$T(n/2) + T(n/2) = 2 \cdot T(n/2). \quad (5)$$

Инструкция кода 3 – выполненное на выходе из рекурсии соединение полученных решений (процедура слияния), т. е.

$$S(n) = \Theta(n). \quad (6)$$

Таким образом, время работы данной процедуры определяется соотношением:

$$T(n) = D(n) + T(n/2) + T(n/2) + S(n) = 2 \cdot T(n/2) + \Theta(1) + \Theta(n). \quad (7)$$

Так как сложность Θ(1) меньше сложности Θ(n), то с учетом подбора констант пропорциональности слагаемое Θ(1) можно отбросить (фактически внести под знак Θ(n)).

Таким образом, время работы сортировки слиянием определяется рекуррентным соотношением:

$$T(n) = \Theta(1), \text{ если } n = 1;$$

$$T(n) = 2\,T(0{,}5n) + \Theta(1), \text{ если } n > 1. \quad (8)$$

Рассмотрим метод подстановки решения рекуррентных соотношений.

Его идея заключается в нахождении (угадывании) функции f(n) такой, что для любого n ($n \geq 1$):

$$T(n) \leq f(n). \quad (9)$$

Для подбора функции выполняются следующие действия:

1) определяется вид функции f(n) с предположением, что она зависит от некоторых, пока неопределенных, параметров;

2) подбираются значения параметров так, чтобы для всех $n \geq 1$ выполнялось $T(n) \leq f(n)$;

3) доказывается по математической индукции правильность подбора функции и значений параметров.

Так как на каждом этапе рассматриваемая часть массива сокращается в 2 раза, а при слиянии выполняются действия, время которых пропорционально n, т. е. $S(n) = \Theta(n)$, то следует предполагать, что f(n) – логарифмическая функция.

Пусть $f(n) = a \cdot n \cdot \log_2 n$, где a пока неизвестно. Должно выполняться неравенство $T(n) \leq f(n)$ при любом натуральном n. Проверим его для первого натурального числа $n = 1$.

$T(n) = \Theta(1)$, т. е. существует константа c_1, такая, что при любом натуральном n справедливо неравенство $T(n) \leq c_1$.

$$f(1) = a \cdot 1 \cdot \log_2 1 = 0.$$

Таким образом, вид функции подобран неверно, т. к. не выполняется неравенство $T(1) \leq f(1)$. Подбирая функцию, добавим еще один параметр b как слагаемое:

$$f(n) = a \cdot n \cdot \log_2 n + b. \quad (10)$$

Докажем выполнение неравенства при $n = 1$:

$$f(1) = a \cdot 1 \cdot \log_2 1 + b = b \geq T(1) = c_1 \text{ при } b \geq c_1. \quad (11)$$

Докажем методом математической индукции справедливость для любого натурального n.

1 При $n = 1$ справедливость неравенства доказана выше – база индукции.

2 Предположим, что для всех $k < n$ выполняется индуктивное предположение – неравенство

$$T(k) \leq a \cdot k \cdot \log_2 k + b. \quad (12)$$

3 Докажем его справедливость для n.

Так как $n > 1$, то из рекуррентного соотношения следует, что

$$T(n) \leq 2 \cdot T(n/2) + c_2 \cdot n. \quad (13)$$

Поскольку n/2 < n, то для n/2 выполняется индуктивное предположение и при $a \geq c_2 + b$

$$T(n) \leq 2\cdot(a\cdot(n/2)\cdot\log_2(n/2) + b) + c_2\cdot n = a\cdot n\cdot\log_2 n - a\cdot n + c_2\cdot n + 2\cdot b \leq a\cdot n\cdot\log_2 n + b. \quad (14)$$

Таким образом, при $b = c_1$, $a = c_1+c_2$ для любого натурального n выполняется неравенство

$$T(n) \leq (c_1+c_2)\cdot n\cdot\log_2 n + c_1, \quad (15)$$

т. е. T(n) имеет верхнюю оценку $O(n\cdot\log_2 n)$.

Сложность рекурсивной сортировки слияниями как по числу сравнений, так и по числу перемещений элементов массива допускает оценку $\Theta(n \log n)$.

Задание

Сортировку вставкой можно представить в виде рекурсивной последовательности следующим образом: чтобы отсортировать массив A[1, …, n], сначала нужно выполнить сортировку массива A[1, …, n – 1], после чего в этот отсортированный массив помещается элемент A[n].

1 Реализуйте рекурсивную сортировку вставками с замерами времени работы и подсчетом трудоемкости на различных входах (20, 50, 100, 200, 500, 1000 элементов).

2 Запишите рекуррентное уравнение для времени работы описанной рекурсивной версии алгоритма сортировки вставкой.

Контрольные вопросы

1 Какой алгоритм называют рекурсивным?

2 В чем отличие в оценивании сложности рекурсивных и нерекурсивных алгоритмов?

3 Опишите метод подстановки решения рекуррентных уравнений (соотношений).

4 Как ведет себя трудоемкость рекурсивной сортировки вставками при росте времени?

5 Продемонстрируйте последовательность вызовов рекурсивного метода, опускаясь до дна рекурсии.

6 Изобразите дерево рекурсивных вызовов программы при n = 5.

7 В чем состоит метод математической индукции?

8 Что называют временной сложностью алгоритмов?

9 Что означает запись $T(n) = \Theta(1)$?

Список литературы

1 **Белов, В.** Алгоритмы и структуры данных / В. Белов, В. Чистякова. – Москва : КУРС ; ИНФРА-М, 2020. – 240 с.

2 **Вайнштейн, Ю. В.** Математическая логика и теория алгоритмов : учебное пособие / Ю. В. Вайнштейн, Т. Г. Пенькова, В. И. Вайнштейн. – Красноярск : Сиб. федер. ун-т, 2019. – 110 с.

3 **Судоплатов, С. В.** Математическая логика и теория алгоритмов : учебник и практикум для академ. бакалавриата / С. В. Судоплатов. – 5-е изд., стер. – Москва : Юрайт, 2019. – 255 с.

Приложение А
(обязательное)

Общие требования к отчету

Отчет по лабораторным работам № 1–4 принимается в печатном виде.

Печатный отчет должен содержать стандартные составные части.

1 Титульный лист (рисунок А.1) с указанием следующих реквизитов: название учреждения образования, название закрепленной за дисциплиной кафедры, номер и название лабораторной работы, название дисциплины, вариант, кто выполнил лабораторную работу (ФИО, группа), кто проверяет работу (должность, ФИО), место и дата составления отчета.

2 Цель работы и постановка задачи.

3 Выполненное задание согласно варианту: код программы, реализующей данный алгоритм, с необходимыми комментариями.

4 Скриншоты с входными и выходными данными. Обычно программа тестируется на нескольких вариантах входных данных для проверки ее корректности.

5 Выводы по теме лабораторной работы.

Отчет оформляется шрифтом гарнитуры Times New Roman, кегль 12–14 пт, межстрочный интервал – полуторный, абзацный отступ – 1–1,25 см.

Страницы должны быть пронумерованы вверху посередине. Титульный лист при нумерации считается, но не нумеруется.

МЕЖГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Автоматизированные системы управления»

ТЕОРИЯ АЛГОРИТМОВ

Лабораторная работа № 1
Анализ алгебраических алгоритмов

Вариант 5

Проверила:
старший преподаватель
Беккер Инга Александровна

Выполнил:
Лосев Владислав Сергеевич,
студент группы АСОИР-231

Могилев 2023

Рисунок А.1 – Титульный лист отчета