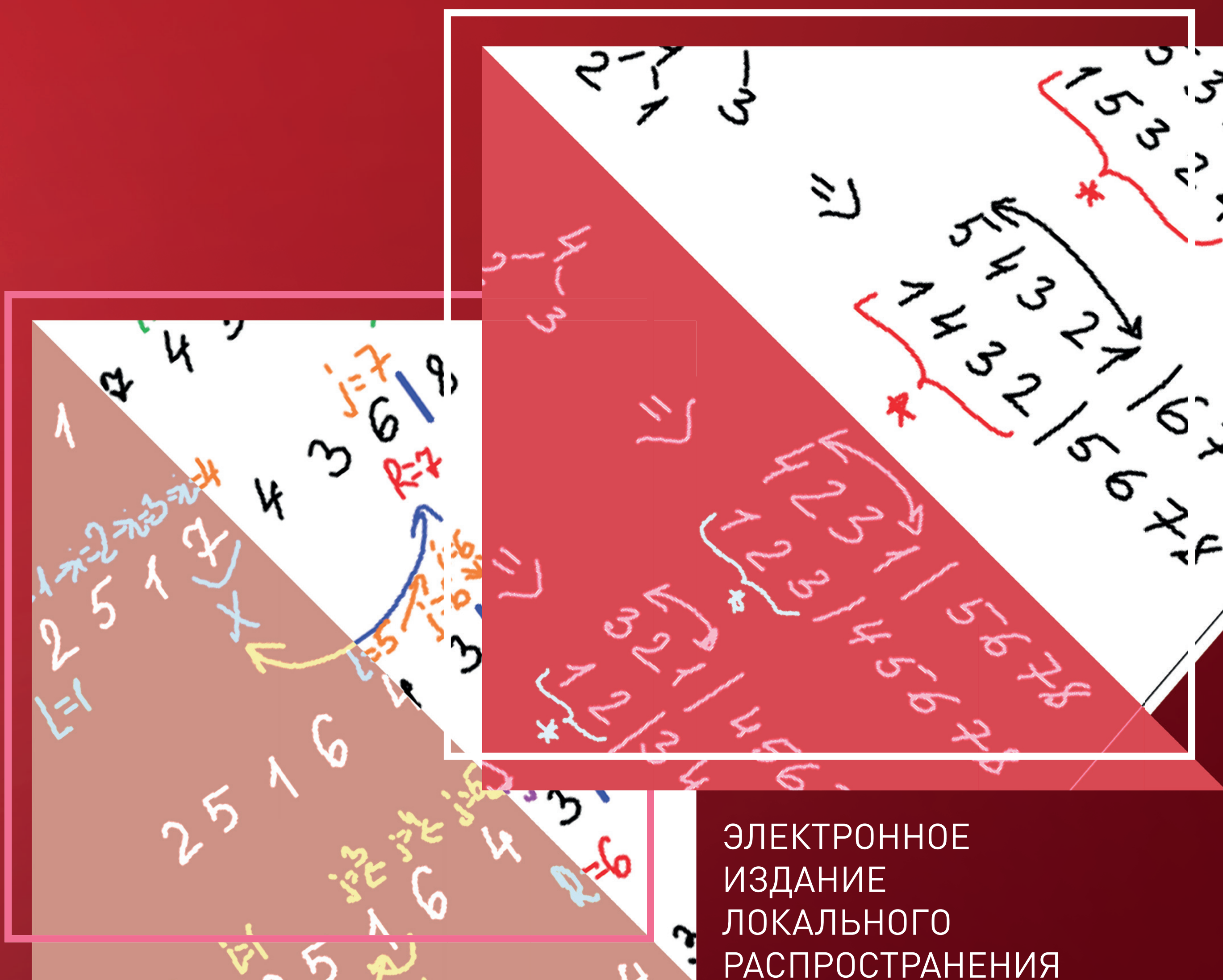
Н.Ф. Долганова
В.М. Долганов
А.Н. Стась

ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ: НЕКОТОРЫЕ МЕТОДЫ СОРТИРОВКИ МАССИВОВ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ



ЭЛЕКТРОННОЕ
ИЗДАНИЕ
ЛОКАЛЬНОГО
РАСПРОСТРАНЕНИЯ

МИНИСТЕРСТВО ПРОСВЕЩЕНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Томский государственный педагогический университет»
(ТГПУ)

Н.Ф. Долганова, В.М. Долганов, А.Н. Стась

ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ: НЕКОТОРЫЕ МЕТОДЫ СОРТИРОВКИ МАССИВОВ

Учебно-методическое пособие

Электронное издание
локального распространения

Томск 2024

© Томский государственный педагогический университет, 2024
ISBN 978-5-907791-00-8

УДК 510.5
ББК 22.127я73+32.973я73+16.0я73
Д64

Рекомендовано к изданию
редакционно-издательским советом
Томского государственного
педагогического университета

Рецензент:
кандидат технических наук, доцент кафедры государственного и муниципального
управления Томского государственного университета
Н.Л. Ерёмина

Долганова Н.Ф.

С30 Теоретические основы прикладной математики и информатики : некоторые методы сортировки массивов : учебно-методическое пособие [Электронный ресурс] / Н.Ф. Долганова, В.М. Долганов, А.Н. Стась. – Томск : Издательство Томского государственного педагогического университета, 2024. – 66 с. – 1 электрон. опт. диск (CD-ROM). – Загл. с титул. экрана.

ISBN 978-5-907791-00-8

Учебно-методическое пособие содержит основные сведения из теории сортировок массива: определения, основные методы, алгоритмы и оценку их временной трудоемкости, а также примеры, позволяющие наглядно продемонстрировать материал как с теоретической, так и с практической точек зрения. Представлен раздаточный материал для студентов и тексты лабораторных работ по конкретным темам.

Предназначено для изучения основ теории алгоритмов и реализации базовых алгоритмов внутренних сортировок обучающимися различных направлений подготовок, связанных с математикой, информатикой и вычислительной техникой.

УДК 510.5
ББК 22.127я73+32.973я73+16.0я73

Системные требования:
ПК не ниже класса Pentium II; RAM 512 Mb; Windows XP/7–10 (32-разрядная
или 64-разрядная версии); разрешение экрана 1 024 × 768 (768 × 1 024); CD-ROM-дисковод, мышь;
Adobe Acrobat Reader DC (либо другое, открывающее PDF-файлы).

© Долганова Н.Ф., 2024
© Долганов В.М., 2024
© Стась А.Н., 2024
ISBN 978-5-907791-00-8
© Томский государственный педагогический университет, 2024

СОДЕРЖАНИЕ

Предисловие 4

1. Постановка задачи 5

2. Простые методы сортировки массивов 7

2.1. Сортировка с помощью прямого включения 7

2.2. Сортировка с помощью прямого выбора 8

2.3. Сортировка с помощью прямого обмена 9

3. Улучшенные методы сортировки массивов 12

3.1. Метод Шелла 12

3.2. Пирамидальная сортировка 14

3.3. Сортировка с помощью разделения 22

Список рекомендуемой литературы 27

Приложение 1. Временная трудоемкость и ее оценка 28

Приложение 2. Постановка задачи сортировки. Нижняя оценка трудоемкости сортировок, основанных на сравнениях 39

Приложение 3. Сортировка с помощью прямого включения 41

Приложение 4. Сортировка с помощью прямого выбора 43

Приложение 5. Сортировка с помощью прямого обмена 45

Приложение 6. Сортировка Шелла 48

Приложение 7. Пирамидальная сортировка (Heapsort) 50

Приложение 8. Быстрая сортировка (Quicksort) 50

Приложение 9. Лабораторная работа 1 «Простые методы сортировки массивов» 55

Приложение 10. Лабораторная работа 2 «Улучшенные методы сортировки массивов» 59

ПРЕДИСЛОВИЕ

Понятие алгоритма является одним из центральных понятий в области прикладной математики и информатики. Непосредственно же разработка алгоритма представляет собой один из основных этапов решения любой прикладной задачи с использованием электронных вычислительных машин. Важным навыком является решение задачи наиболее эффективным способом из возможных. В качестве примера одной из базовых задач является сортировка данных. Под сортировкой понимается упорядочение некоторой последовательности элементов. Эффективность алгоритма сортировки зависит от формы хранения сортируемой последовательности. Важным аспектом является наличие прямого доступа к элементу. Как правило, такой доступ возможен для массивов, хранимых во внутренней памяти, поэтому сортировку массивов или других структур с прямым доступом называют внутренней сортировкой.

В пособии рассматриваются основные понятия теории внутренних сортировок, приведены примеры простых и улучшенных методов сортировок массивов с доказательством их временной трудоемкости. Трудоемкость простых методов в наихудшем и среднем имеет порядок $O(n^2)$, улучшенных – $O(n \cdot \log n)$. Отмечается, что пирамидальная сортировка является наиболее эффективной с точки зрения сортировки в наихудшем, а быстрая сортировка Хоара – с точки зрения трудоемкости в среднем.

Приведены коды алгоритмов на обобщенном алгоритмическом языке, на скриптовом языке программирования Python и на структурном языке программирования Pascal. В приложениях представлены раздаточные материалы с выжимками теоретических сведений, демонстрационных примеров и кодом программ (приложения 1–8), а также разработанные лабораторные работы «Простые методы сортировки массивов» (приложение 9), «Улучшенные методы сортировки массивов» (приложение 10).

1. ПОСТАНОВКА ЗАДАЧИ

Сортировка – класс задач, связанных с упорядочиванием последовательностей по возрастанию. Сортировка по убыванию практически та же задача, но используется противоположная операция сравнения. Речь идет не просто о решении этой задачи, а об эффективном решении, причем эффективность может пониматься как в наихудшем, так и в среднем. Кроме того, многое зависит от структуры данных, сортировка массивов и сортировка последовательных файлов – сильно различающиеся задачи.

Выбор алгоритма зависит от структуры обрабатываемых данных, причем в случае сортировки такая зависимость столь глубока, что соответствующие методы в информатике делятся на два класса – сортировку массивов и сортировку файлов (последовательностей) (рис. 1). Иногда их называют внутренней и внешней сортировкой, поскольку массивы хранятся в быстрой, оперативной, внутренней памяти машины со случайным доступом, а файлы обычно размещаются в более медленной, но и более емкой внешней памяти, на устройствах, основанных на механических перемещениях (дисках или лентах). Отличительной особенностью внутренних методов сортировок от внешних – доступ к элементам: в массиве – прямой, в файлах – последовательный.

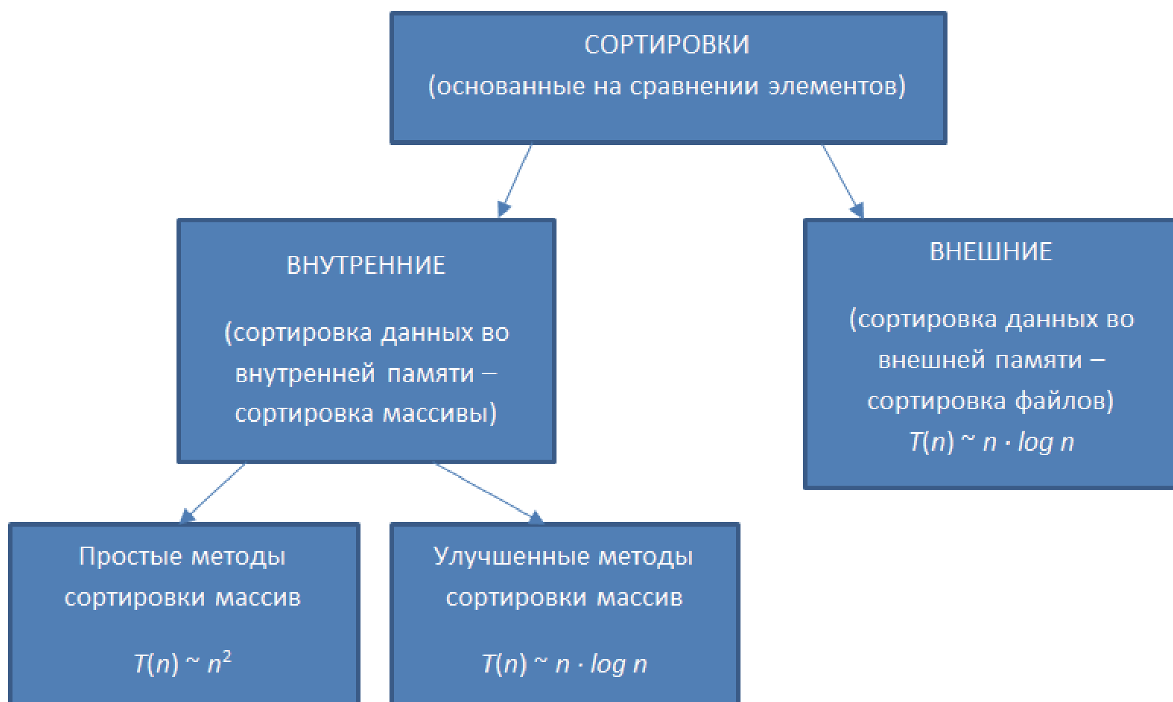


Рис. 1. Классификация сортировок

Проведем нижнюю оценку трудоемкости алгоритмов, основанных на сравнении элементов. Для этого выясним, сколько сравнений минимум надо сделать, чтобы отсортировать произвольный массив. Фактически мы получим нижнюю оценку трудоемкости в наихудшем, при достижении которой алгоритм улучшить будет уже невозможно.

Теорема. Нижняя оценка $T(n)$ для алгоритмов сортировки, основанных на сравнении, имеет трудоемкость порядка $O(n \cdot \log n)$.

Доказательство. В комбинаторике доказывается, что массив длины n можно записать $n!$ различными способами (называемыми перестановками), если допускается менять только порядок элементов и не касаться содержимого. Нам надо выбрать подходящую перестановку.

Очевидно, что при проведении одного сравнения число возможно подходящих перестановок сужается вдвое. Еще одно сравнение сужает круг поиска еще в 2 раза и т.д. Таким образом мы найдем одну-единственную нужную перестановку за $\log n!$ сравнений.

$$\frac{n!}{2^{\mathrm{k}}} = 1,$$
$$n! = 2^k,$$
$$k = \log_2 n!,$$
$$\log_2 n! = \log_2(1 \cdot 2 \cdot \ldots \cdot n) = \log 1 + \log 2 + \cdots + \log n \leq$$
$$\leq \log n + \log n + \cdots + \log n \sim n \cdot \log n.$$

При доказательстве мы использовали известные свойства алгоритмов:

$$\log(ab) = \log a + \log b,$$
$$a \leq b \ \longrightarrow \log a \leq \log b$$

Можно доказать и обратное утверждение:

$$n \cdot \log n = \log n + \log n + \log n + \ldots + \log n \leq$$
$$\leq \log(1 \cdot n) + \log(2 \cdot (n-1)) + \log(3 \cdot (n-2)) + \ldots + \log(n \cdot 1) =$$
$$= [\log 1 + \log n] + [\log 2 + \log(n-1)] + [\log 3 + \log(n-2)] + \ldots + [\log n + \log 1] =$$
$$= 2 \cdot \log 1 + 2 \cdot \log 2 + 2 \cdot \log 3 + \ldots + 2 \cdot \log n =$$
$$= 2 \cdot (\log 1 + \log 2 + \log 3 + \ldots + \log n) =$$
$$= 2 \cdot \log(1 \cdot 2 \cdot 3 \cdot \ldots \cdot n) =$$
$$= 2 \cdot \log n!$$

Таким образом, $T(n) \sim \mathrm{O}(n \cdot \log n)$.

2. ПРОСТЫЕ МЕТОДЫ СОРТИРОВКИ МАССИВОВ

2.1. Сортировка с помощью прямого включения

Метод прямого (простого) включения основан на следующем приеме: на каждом шаге элементы мысленно делятся на уже отсортированную последовательность a_1, …, a_{i-1} и неотсортированную последовательность, при этом на первом шаге отсортированной последовательностью считается первый элемент исходной последовательности. При каждом шаге, начиная со второго ($i = 2$) и увеличивая i каждый раз на единицу, из неотсортированной последовательности извлекается i-й элемент и путем последовательных сравнений включается в отсортированную последовательность на нужное место.

В табл. 1 показан в качестве примера процесс сортировки с помощью включения восьми случайно выбранных чисел.

Таблица 1

Пример сортировки с помощью прямого включения

Начальные ключи	2	5	1	8	4	3	6	7
$i = 2$	2	5	1	8	4	3	6	7
$i = 3$	1	2	5	8	4	3	6	7
$i = 4$	1	2	5	8	4	3	6	7
$i = 5$	1	2	4	5	8	3	6	7
$i = 6$	1	2	3	4	5	8	6	7
$i = 7$	1	2	3	4	6	5	8	7
$i = 8$	1	2	3	4	6	5	7	8

Алгоритм этой сортировки таков:

```
for i: = 2 to n do begin
x: = a[i];
   включение x на соответствующее место среди a[1], ... ,a[i];
end;
```

В табл. 2 приведены примеры алгоритмов реализации прямого включения на структурном языке программирования Pascal и на скриптовом языке программирования Python.

Таблица 2

Алгоритмы реализации прямого включения

<table>
<tr><th>Pascal</th><th>Python</th></tr>
<tr><td><pre>
for i:= 2 to n do begin
  x:= A[i];
  A[0]:= x;
  j:= i;
  while x < A[j – 1] do begin
   A[j]:=A[j – 1];
   dec (j)
  end;
  A[j]:= x
 end;
</pre></td><td><pre>
for i in range(1,n):
    x = A[i]
    j = i – 1
    while j >= 0 and x < A[j]:
        A[j + 1] = A[j]
        j-=1
    A[j + 1] = x
</pre></td></tr>
</table>

Очевидно, $T(n) \sim O(n^2)$, $T_{ср.}(n) \sim O(n^2)$.

2.2. Сортировка с помощью прямого выбора

Метод прямого (простого) выбора (метод минимальных элементов) работает следующим образом. Выбирается наименьший элемент и меняется с первым элементом исходной последовательности. Затем этот процесс повторяется с оставшимися $(n - 1)$ элементами, $(n - 2)$ элементами и так далее до тех пор, пока не останется один, самый большой элемент.

В табл. 3 показан в качестве примера процесс сортировки с помощью прямого выбора восьми случайно выбранных чисел.

Таблица 3

Пример сортировки с помощью прямого выбора

Начальные ключи	2	5	1	8	4	3	6	7
$i = 2$	1	5	2	8	4	3	6	7
$i = 3$	1	2	5	8	4	3	6	7
$i = 4$	1	2	3	8	4	5	6	7
$i = 5$	1	2	3	4	8	5	6	7
$i = 6$	1	2	3	4	5	8	6	7
$i = 7$	1	2	3	4	5	6	8	7
$i = 8$	1	2	3	4	5	6	7	8

Алгоритм формулируется так:

for *i*: = 1 to *n* – 1 do begin
присвоить k индекс наименьшего из a[*i*] ... *a*[*n*];
поменять местами a[*i*] ... *a*[*k*];
end;

В табл. 4 приведены примеры алгоритмов реализации прямого выбора на структурном языке программирования Pascal и на скриптовом языке программирования Python.

Таблица 4

Алгоритмы реализации прямого выбора

Pascal	Python

for *i*:=1 to *n*-1 do begin
  min:=*i*;
  x:=A[*i*];
  for *j*:=*i*+1 to *n* do
  if A[*j*]<*x* then begin
   min:=*j*;
   x:=A[*min*]
  end;
  A[*min*]:=A [*i*];
  A[*i*]:=*x*
 end;
	for *i* in range(*n*):
	  min = *i*
	
	  for *j* in range(*i*+1,*n*):
	   if A[*j*]<A[*min*]:
	    min=*j*
	  x = A[*min*]
	
	  A[*min*] = A[*i*]
	  A[*i*]=*x*

Доказано, что $T(n) \sim T_{ср.}(n) \sim O(n^2)$.

2.3. Сортировка с помощью прямого обмена

Метод прямого обмена («метод пузырька») основывается на сравнении и смене мест пары соседних элементов и продолжении этого процесса до тех пор, пока не будут упорядочены все элементы. Мы повторяем проходы по массиву в обратном порядке, сдвигая каждый раз минимальный элемент из двух соседних к началу. Очевидно, что каждый раз минимальный элемент будет подобно легкому пузырьку воздуха всплывать на первую позицию. Поэтому повторные проходы уже отсортированную часть не затрагивают. Таким образом, всего необходим $(n - 1)$ проход. Для бóльшей наглядности «эффекта пузырька» массивы данных удобнее представлять вертикально.

В табл. 5 показан в качестве примера процесс сортировки с помощью прямого обмена восьми случайно выбранных чисел на первом шаге (при $i = 1$).

Таблица 5

Пример работы сортировки с помощью прямого обмена на первом шаге

$i = 1$							
2	2	2	2	2	2	2	1
5	5	5	5	5	5	1	2
1	1	1	1	1	1	5	5
8	8	8	8	3	3	3	3
4	4	4	3	8	8	8	8
3	3	3	4	4	4	4	4
6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7

Таким образом, после первого прохода отсортированной частью массива будет первый элемент: 1| 2 5 3 8 4 6 7. Далее осуществляем аналогичные проходы по массиву в обратном порядке, не затрагивая уже отсортированную часть (табл. 6).

Таблица 6

Пример пузырьковой сортировки

$i = 1$	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
5	3	3	3	3	3	3	3
3	5	4	4	4	4	4	4
8	4	5	5	5	5	5	5
4	8	6	6	6	6	6	6
6	6	8	7	7	7	7	7
7	7	7	8	8	8	8	8

Алгоритм сортировки:

i = 2 to *n* do *j* := *n* downto *i* do

если a[*j*] *больше a*[*j* – 1], *поменяй их местами*;

end;

В табл. 7 приведены примеры алгоритмов реализации прямого обмена на структурном языке программирования Pascal и на скриптовом языке программирования Python.

Таблица 7

Алгоритмы реализации прямого обмена

Pascal	**Python**

for *i*:=2 to *n* do
for *j*:=*n* downto *i* do
if A[*j*-1]>A[*j*] then begin
  x:=A[*j*-1];
  A[*j*-1]:=A[*j*];
  A[*j*]:=*x*
end;
	for i in range(*n*-1):
	    for j in range(*n*-1, *i*, -1):
	        if A[*j*-1]>A[*j*]:
	            x = A[*j*]
	            A[*j*] = A[*j*-1]
	            A[*j*-1] = *x*

Легко видно, что $T(n) \sim T_{ср.}(n) \sim O(n^2)$.

3. УЛУЧШЕННЫЕ МЕТОДЫ СОРТИРОВКИ МАССИВОВ

3.1. Метод Шелла

Метод основан на усовершенствовании сортировки с помощью прямого включения. Метод предложен Д. Шеллом в 1959 г. Метод основан на так называемых Н-цепочках. Полагаем, что существует конечная последовательность расстояний между сортируемыми элементами простым включением $h_1, \dots, h_t$, причем $h_t = 1$, $h_{i+1} < h_i$.

Описание массива при этом выглядит так a: array $[-h_1 .. n]$ of integer.

Алгоритм работает следующим образом: сначала методом простого включения сортируются элементы, отстающие друг от друга на расстоянии h_1, затем сортируются элементы на расстоянии h_2 и т.д. На заключительной стадии происходит сортировка с шагом $h_t = 1$, т.е. выполняется обычный метод простого включения.

На первый взгляд, первые h_{t-1} шагов кажутся бессмысленными и даже увеличивающими сложность алгоритма, однако это впечатление обманчиво. В конечном итоге $(t-1)$ сортировки позволяют в разы снизить трудоемкость последнего шага.

В табл. 8 приведены примеры сортировок с помощью включений с уменьшающимися расстояниями ($h_1 = 4$, $h_2 = 2$, $h_{t=3} = 1$).

Таблица 8

Сортировка с помощью включений с уменьшающимися расстояниями

2	5	1	8	4	3	6	7
Четверная сортировка дает							
2	3	1	7	4	5	6	8
Двойная сортировка дает							
1	3	2	5	4	7	6	8
Одинарная сортировка дает							
1	2	3	4	5	6	7	8

Запишем алгоритм метода Шелла на обобщенном Паскале:

```
// определяем h-цепочку и заносим ее в массив h;
// цикл по m для перебора всех расстояний
begin
  k := h[m]; s := – k; // k – шаг сортировки
  for i := k + 1 to n do begin
    x := a[i]; j := i – k;
```

```
      if s = 0 then s := –k; //отсечение случая
      inc(s); a[s] := x; // выхода за пределы массива
      while x < a[j] do begin  //простые вставки с шагом k
        a[j + k] := a[j];
        j := j – k
      end;
      a[j + k] := x
    end
  end;
```

В табл. 9 приведены примеры алгоритмов реализации сортировки Шелла на структурном языке программирования Pascal и на скриптовом языке программирования Python.

Таблица 9

Алгоритмы реализации сортировки Шелла

<table>
<tr><th>Pascal</th><th>Python</th></tr>
<tr><td><pre>program shells;
const t=4;
 h: array [1..t] of integer = (15,7,3,1);
var
 i,j,k,s,x,n,m: integer;
 A: array [-15..50] of integer;
begin
 // ввод массива
 for m:=1 to t do begin
  k:=h[m];
  s:=-k;
  for i:=k+1 to n do begin
   x:=A[i];
   j:=i-k;
   if s=0 then s:=-k;
   inc(s);
   A[s]:=x;
   while x<A[j] do begin
    A[j+k]:=A[j];
    j:=j-k
   end;
   A[j+k]:=x
  end;
 end;
 // вывод массива
end.</pre></td><td><pre>// ввод массива
// определить длину массива
// ввод массива расстояний HL
for h in HL:
    for i in range(h,n):
        x = A[i]
        j = i-h
        while j>=0 and x<A[j]:
            A[j+h]=A[j]
            j-=h
        A[j+h]=x
// вывод массива</pre></td></tr>
</table>

$T(n) \sim O(n \cdot \log n)$, $T_{ср.}(n) \sim O(n^{1.2})$, $T_{наих.}(n) \sim O(n^2)$.

Замечание. Неизвестно, какие расстояния дают наилучший результат. Но они не должны быть множителями один другого. Д.Э. Кнут показывает, что имеет смысл использовать такую последовательность, в которой $h_{k-1} = 3h_k + 1$, $h_t = 1$ и $t = [\log_2 n] - 1$.

Итак, в наихудшем случае метод остается квадратичным, а вот $T_{ср}(n) = n^{1.2}$. Таким образом, достигается прогресс в средних показателях. Также доказано, что $T(n) \sim O(n \cdot \log n)$.

3.2. Пирамидальная сортировка

Попробуем улучшить метод прямого выбора. При поиске минимального элемента будем производить множество сравнений и запоминать информацию о них. Совершаем попарное сравнение элементов: первый со вторым, третий с четвертым, пятый с шестым и так далее, при этом минимумы запоминаем в пары. Затем эти минимумы опять группируем в пары и сравниваем уже их. Таким образом, получим дерево, на вершине которого будет минимальный элемент (рис. 2).

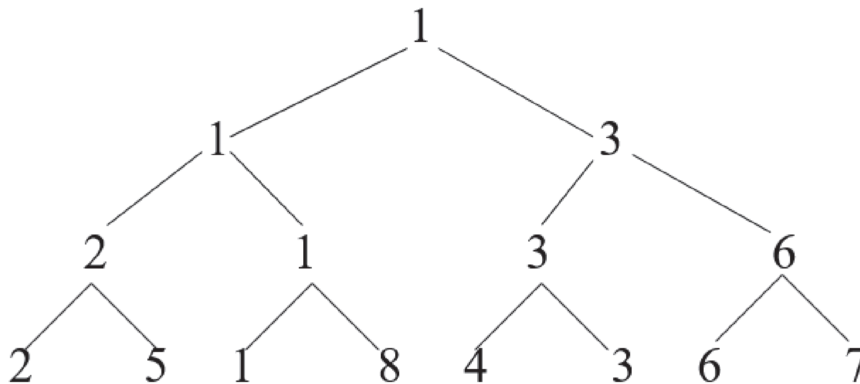


Рис. 2. Дерево, на вершине которого минимальный элемент

Для построения такого дерева требует порядка n операций, т.е. столько же, сколько на простой поиск минимума.

После того как минимальный элемент будет найден, его легко удалить из дерева (рис. 3).

После чего получим аналогичное дерево для $(n - 1)$ элемента (рис. 4) и т.д. (рис. 5–11).

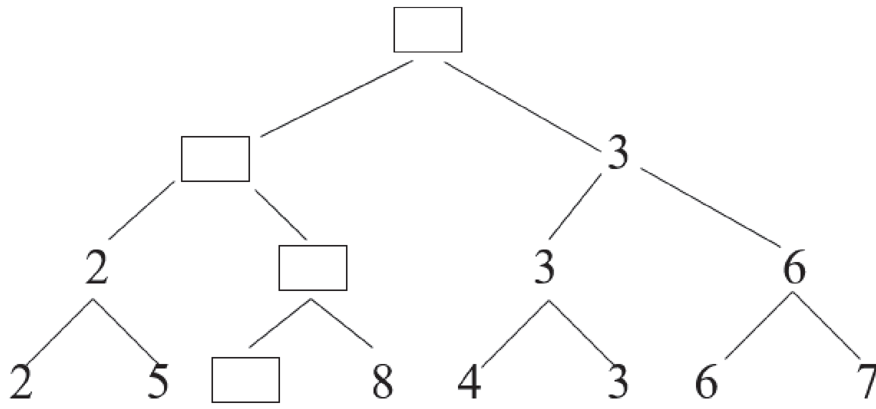


Рис. 3. Дерево после удаления наименьшего элемента

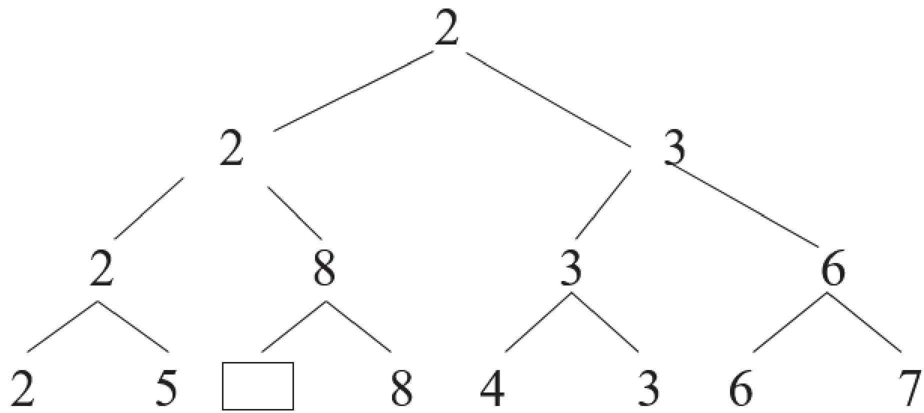


Рис. 4. Дерево для (n – 1) элемента

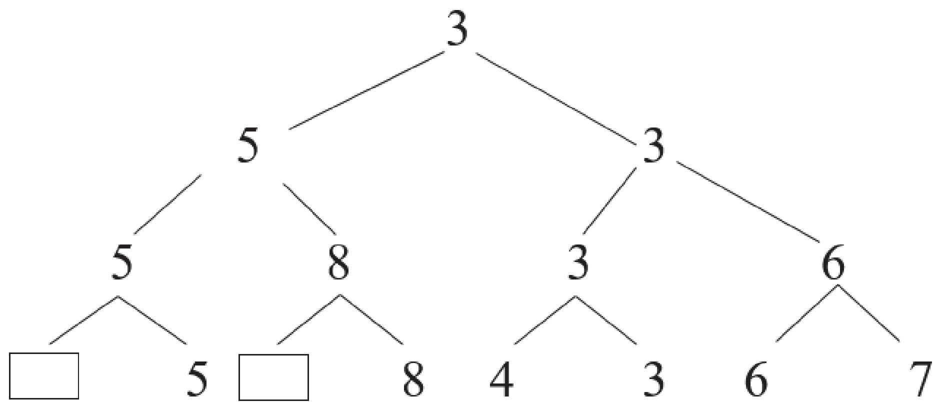


Рис. 5. Дерево для (n – 2) элементов

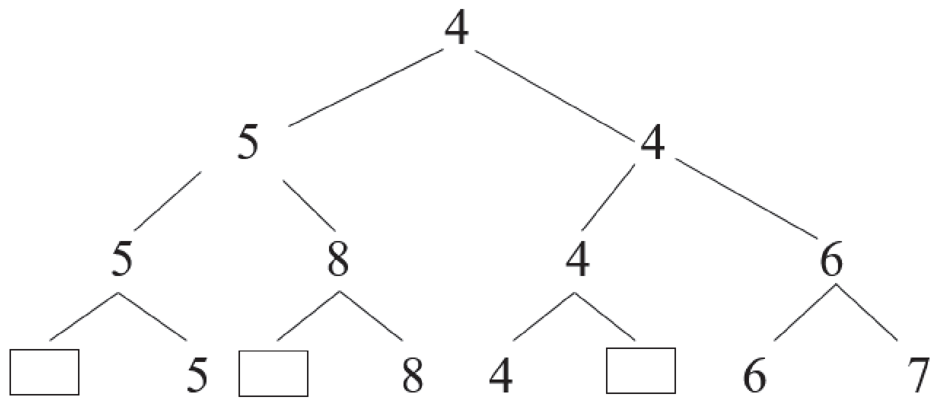


Рис. 6. Дерево для (n – 3) элементов

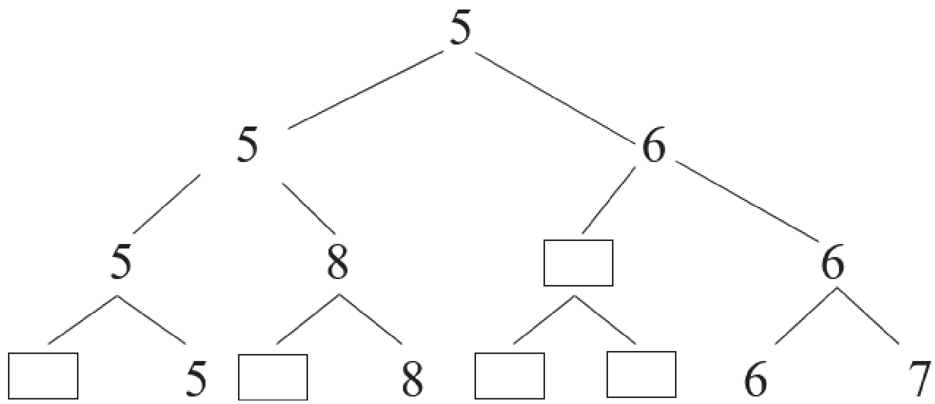


Рис. 7. Дерево для (n – 4) элементов

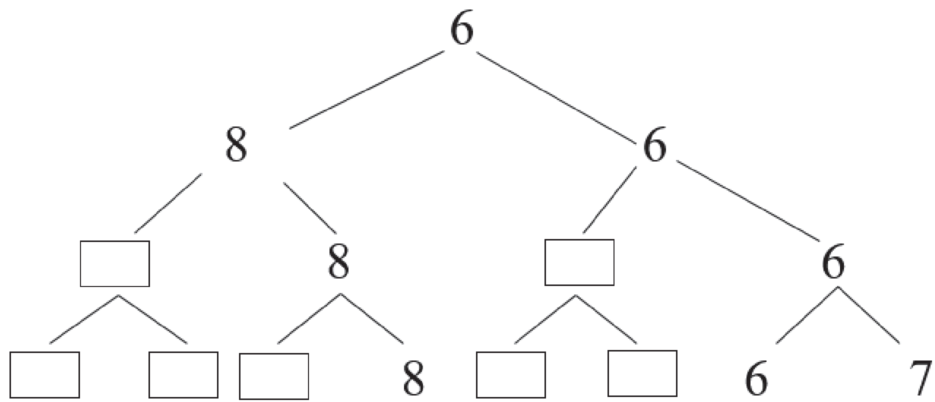


Рис. 8. Дерево для (n – 5) элементов

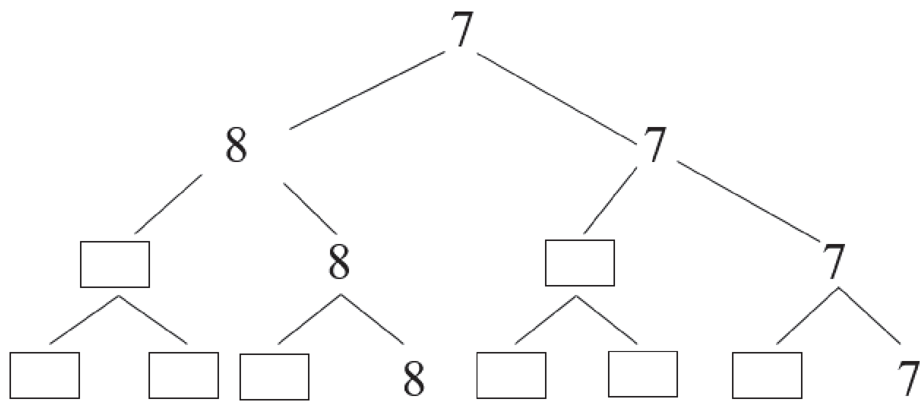


Рис. 9. Дерево для (n – 6) элементов

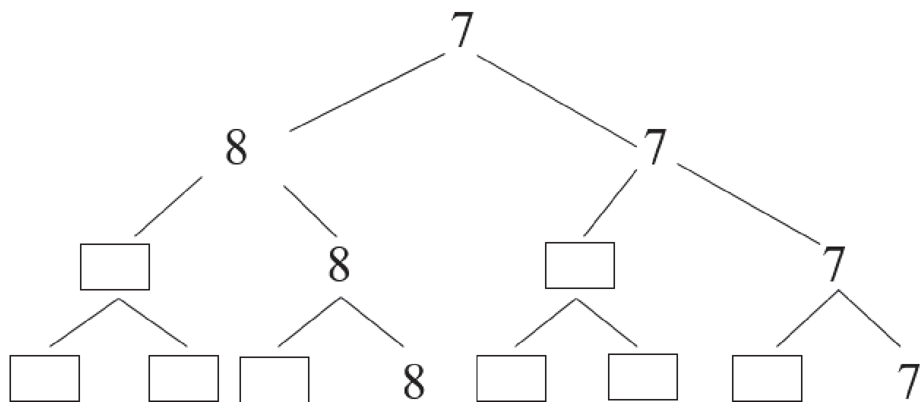


Рис. 10. Дерево для (n – (n – 2)) элементов

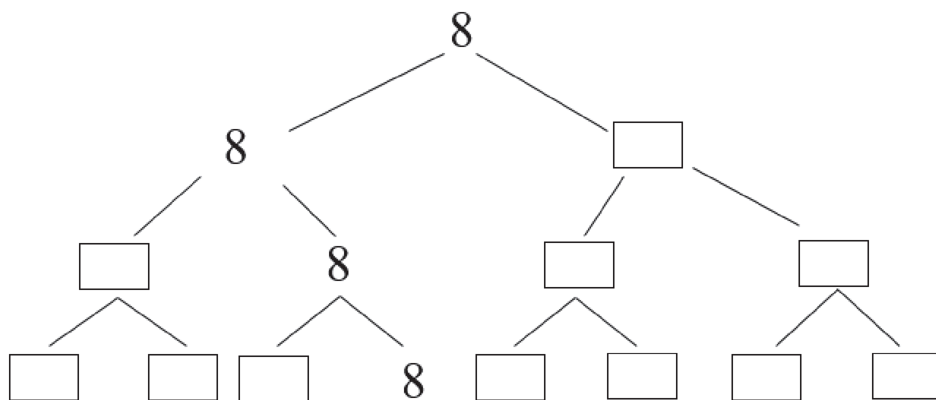


Рис. 11. Дерево для 1 элемента (n – (n – 1) = 1)

После n шагов дерево становится пустым (состоит только из «дырок»), следовательно, процесс сортировки заканчивается. На каждом шаге требуется

log$n$ сравнений, поэтому на весь процесс понадобится порядка $n \cdot \log n$ операций, плюс еще n шагов на построение дерева, таким образом, $T(n) = n + n \cdot \log n \sim O(n \cdot \log n)$. Однако требуются дополнительные затраты памяти для хранения отсортированных элементов.

Д. Уилльямс изобрел метод *Heapsort*, в котором было получено существенное улучшение традиционных сортировок с помощью деревьев. Пирамида определяется как последовательность ключей a[L], a[L+1], …, a[R], такая, что $a[i] \le a[2 \cdot i]$ и $a[i] \le a[2 \cdot i+1]$ для i = L…R/2.

Р. Флойдом был предложен некий «лаконичный» способ построения пирамиды на «*том же месте*». Здесь a[1] … a[n] – некий массив, причем a[m]…a[n] (m = [n div 2] + 1) уже образуют пирамиду, поскольку индексов i и j, удовлетворяющих соотношению $j = 2 \cdot i$ (или $j = 2 \cdot i + 1$), просто не существует.

Заметим, что элементы массива можно представлять в виде двоичного дерева. Считается, что a[i] порождает элементы a[$2 \cdot i$] и a[$2 \cdot i$+1] (рис. 12).

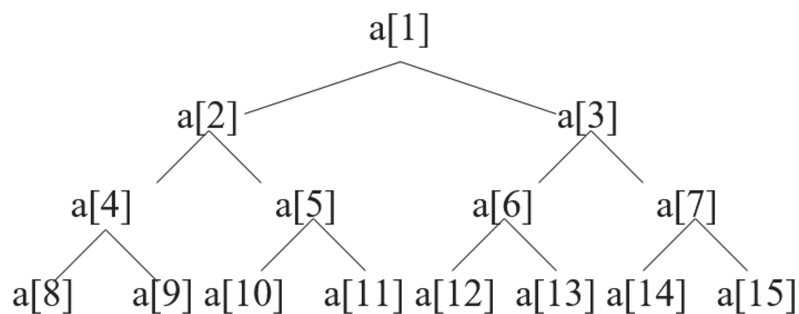


Рис. 12. Массив, представленный в виде двоичного дерева

Эти элементы (a[m]…a[n], где m = [n div 2+1]) образуют как бы нижний уровень соответствующего двоичного дерева, для них никакой упорядоченности не требуется. Далее пирамида расширяется влево, каждый раз добавляется и сдвигами ставится в надлежащую позицию новый элемент (a[L] = a[n div 2], a[L–1], …, a[1]), причем в процессе его просеивания будем проверять выполнение условий-неравенств $a[i] \ge a[2 \cdot i]$ и $a[i] \ge a[2 \cdot i+1]$ для i = 1,…, R/2 (рис. 13).

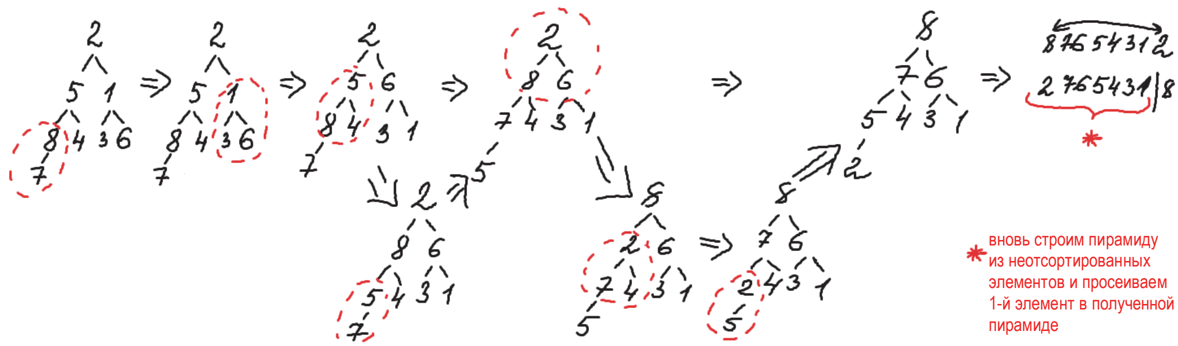


Рис. 13. Первое просеивание

В табл. 10 приведено первоначальное построение пирамиды с просеиванием половины элементов, имеющих потомков (просеиваемый элемент в строке таблицы отмечен знаком «]»).

Таблица 10

Первоначальное построение пирамиды

2	5	1	8]	4	3	6	7
2	5	1]	8	4	3	6	7
2	5]	6	8	4	3	1	7
2]	8	6	7	4	3	1	5
8	7	6	5	4	3	1	2

Далее менять первый и последний элемент местами, декрементировать правую границу массива R и просеивать первый элемент в пирамиде, образованной неотсортированными элементами массива от L до R (рис. 14).

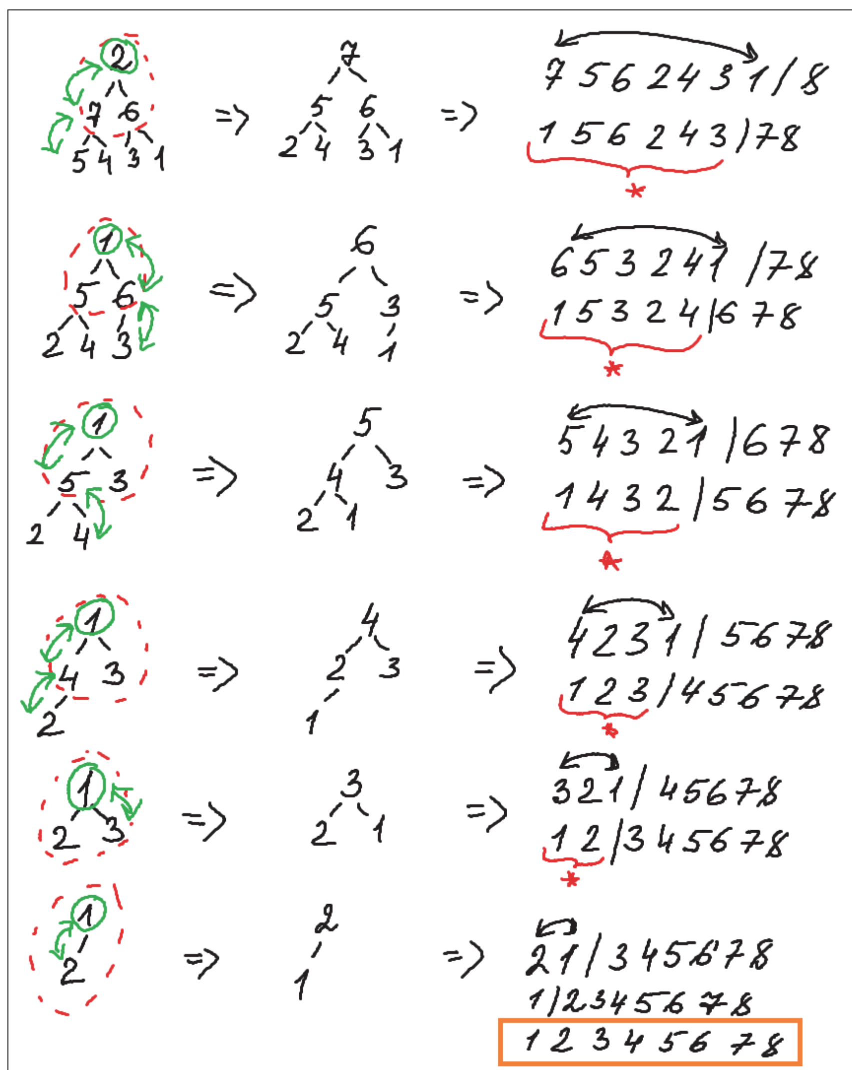


Рис. 14. Дальнейшее просеивание

На рис. 14 продемонстрирован процесс просеивания первого элемента при построении пирамиды из неотсортированной части элементов массива.

В табл. 11 приведены необходимые в этом случае $n-1$ шагов.

Таблица 11

Примеры процесса сортировки с помощью *Heapsort*

2	7	6	5	4	3	1|	8
1	5	6	2	4	3|	7	8
1	5	3	2	4|	6	7	8
1	4	3	2|	5	6	7	8
1	2	3|	4	5	6	7	8
1	2|	3	4	5	6	7	8
1|	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

Запишем алгоритм пирамидальной сортировки на обобщенном Паскале:

```
PROGRAM HS;
VAR I,X,L,N,R:INTEGER;
   A:ARRAY[0..50] OF INTEGER;

PROCEDURE SIFT(L,R: INTEGER);
VAR
 I,J,X: INTEGER;
BEGIN
 I:=L;
 J:=2*L;
 X:=A[L];
 IF (J<R)AND(A[J]<A[J+1]) THEN INC(J);
 WHILE (J<=R)AND(X<A[J]) DO BEGIN
  A[I]:=A[J];
  A[J]:=X;
  I:=J;
  J:=2*J;
  IF (J<R)AND(A[J]<A[J+1]) THEN INC(J);
 END
END;


BEGIN
 //ввод массива длины n
 // определить левую границу L как позицию среднего элемента плюс один в массиве A;
 // определить правую границу R, равную количеству элементов массива A;
```

Пока элементы слева есть делай
уменьши L на единицу;
вызови процедуру просеивания SIFT для фактических параметров L,N
END;
Пока правая граница больше единицы делай
поменяй местами A[1] и A[R] элементы;
уменьши правую границу на единицу;
вызови процедуру просеивания SIFT для фактических параметров 1,R
END;
//вывод отсортированного массива на экран
END.

В табл. 12 приведены примеры алгоритмов реализации пирамидальной сортировки на структурном языке программирования Pascal и на скриптовом языке программирования Python.

Таблица 12

Алгоритмы реализации пирамидальной сортировки

Pascal	**Python**

program hs;
var i,x,l,n,r: integer;
A: array[0..50] of integer;
 	
procedure sift(l,r: integer);
var
i,j,x: integer;
begin
i:=l;
j:=2*l;
x:=A[l];
if (j<r)and(A[j]<A[j+1]) then inc(j);
while (j<=r)and(x<A[j]) do begin
A[i]:=A[j];
A[j]:=x;
i:=j;
j:=2*j;
if (j<r)and(A[j]<A[j+1]) then inc(j);
end
end;
 	
begin
//ввод массива
	def sift(l,r):
	i = l
	j = 2*i+1
	if j+1<=r and A[j+1]>A[j]:
	j = j+1
	while j<=r and A[j]>A[i]:
	x = A[i]
	A[i] = A[j]
	A[j] = x
	i = j
	if j+1<=r and A[j+1]>A[j]:
	j = j+1
	
	// ввод массива
	// определение длины массива
	for i in range(n//2-1, -1, -1):
	sift(i, n-1)
	//поменяй местами 1-ый и последний элементы
	//уменьши правую границу на единицу
	
	for i in range(n-2):

Pascal	Python

l:=(n div 2)+1;
r:=n;
while l>1 do begin
dec(l);
sift(l,n)
end;
while r>1 do begin
//поменяй местами 1-ый и последний элементы
//уменьши правую границу на единицу
sift(1,r)
end;
//вывод массива
end.
	sift(0, r)
	x = A[0]
	A[0] = A[r]
	A[r] = x
	r-=1
	// вывод массива

$T(n) = T_{ср.}(n) = T_{наих.}(n) \sim O(n \cdot log\ n)$.

Heapsort очень эффективна для большого числа элементов *n*; чем больше *n*, тем лучше она работает.

В худшем случае нужно *n*/2 сдвигающих шагов, они сдвигают элементы на *log* (*n*/2), *log* (*n*/2–1), …, *log* (*n*–1) позиций (логарифм по [основанию 2] «обрубается» до следующего меньшего целого). Следовательно, фаза сортировки будет *n*–1 сдвигов с самое большое *log* (*n*–1), *log* (*n*–2), …, 1 перемещениями.

Можно сделать вывод, что даже в самом плохом из возможных случаев *Heapsort* потребуется $O(n \cdot log\ n)$ шагов. Великолепная производительность в таких случаях – одно из привлекательных свойств *Heapsort.*

3.3. Сортировка с помощью разделения

Данный метод сортировки основан на обмене. Это самый лучший из всех известных на данный момент методов сортировки массивов. Его производительность столь впечатляюща, что изобретатель Ч. Хоар назвал этот метод *быстрой сортировкой* (*Quicksort*). В *Quicksort* исходят из того соображения, что для достижения наилучшей эффективности сначала лучше производить перестановки на большие расстояния. Предположим, что у нас есть *n* элементов, расположенных по ключам в обратном порядке. Их можно отсортировать за *n*/2 обменов, сначала поменять местами самый левый с самым правым, а затем последовательно сдвигаться с двух сторон. Это возможно в том

случае, когда мы знаем, что порядок действительно обратный. Однако полученный при этом алгоритм может оказаться и неудачным, что, например, происходит в случае n идентичных ключей: для разделения нужно $n/2$ обменов. Этих необязательных обменов можно избежать, если операторы просмотра заменить на такие:

$$\text{while } A[i] \le x \text{ do } i := i + 1;$$

$$\text{while } x \le A[j] \text{ do } j := j - 1$$

В этом случае x не работает как барьер для двух просмотров. В результате просмотры массива со всеми идентичными ключами приведут к переходу через границы массива.

Цель – не только провести разделение на части исходного массива элементов, но и отсортировать его. Будем применять процесс разделения к получившимся двум частям до тех пор, пока каждая из частей не будет состоять из одного-единственного элемента. Эти действия описываются в программе на обобщенном Pascal ниже.

```
PROGRAM QS;
VAR N,I:INTEGER;
   A:ARRAY[0..50] OF INTEGER;

PROCEDURE SORT(L,R: INTEGER);
VAR
 I,J,X,W: INTEGER;
BEGIN
//определить значение переменной I как соответствующее значению левой границе просмотра L;
//определить значение переменной J как соответствующее значению левой границе просмотра R;
//определить значение переменной X, равной значению серединного элемента;
 REPEAT
   Пока A[I] расположено верно относительно X инкрементируй I;
   Пока A[J] расположено верно относительно X декрементируй J;
   Если I «не забежало» за J поменяй местами A[I] и A[J] и сдвинь индексы I и J навстречу друг
другу
UNTIL I>J;
 IF L<J THEN SORT(L,J);
 IF I<R THEN SORT(I,R);
END;

BEGIN
 //ввод массива длины n
 //вызови процедуру просеивания SORT для фактических параметров 1,N
 //вывод отсортированного массива
END.
```

В табл. 13 приведены примеры алгоритмов реализации сортировки с помощью разделения на структурном языке программирования Pascal и на скриптовом языке программирования Python.

Таблица 13

Алгоритмы реализации быстрой сортировки

Pascal	Python

program qs;
var n,i: integer;
A: array[0..50] of integer;
 	
procedure sort(l,r: integer);
var
i,j,x,w: integer;
begin
i:=l;
j:=r;
x:=A[(l+r) div 2];
repeat
while A[i]<x do inc(i);
while x<A[j] do dec(j);
if i<=j then begin
w:=A[i];
A[i]:=A[j];
A[j]:=w;
inc(i);
dec(j)
end
until i>j;
if l<j then sort(l,j);
if i<r then sort(i,r);
end;
 	
begin
//ввод массива
sort(1, n);
//вывод массива
end.
	def sort(l,r):
	if r>l:
	x = A[(l+r)//2]
	i = l
	j = r
	while i<=j:
	while A[i]<x:
	i+=1
	while A[j]>x:
	j-=1
	if i<=j:
	x = A[i]
	A[i] = A[j]
	A[j] = x
	i+=1
	j-=1
	sort(l,j)
	sort(i,r)
	
	//ввод массива
	//определение длины массива
	sort(0, n-1)
	//вывод массива

На рис. 15 продемонстрирован процесс работы быстрой сортировки Хоара.

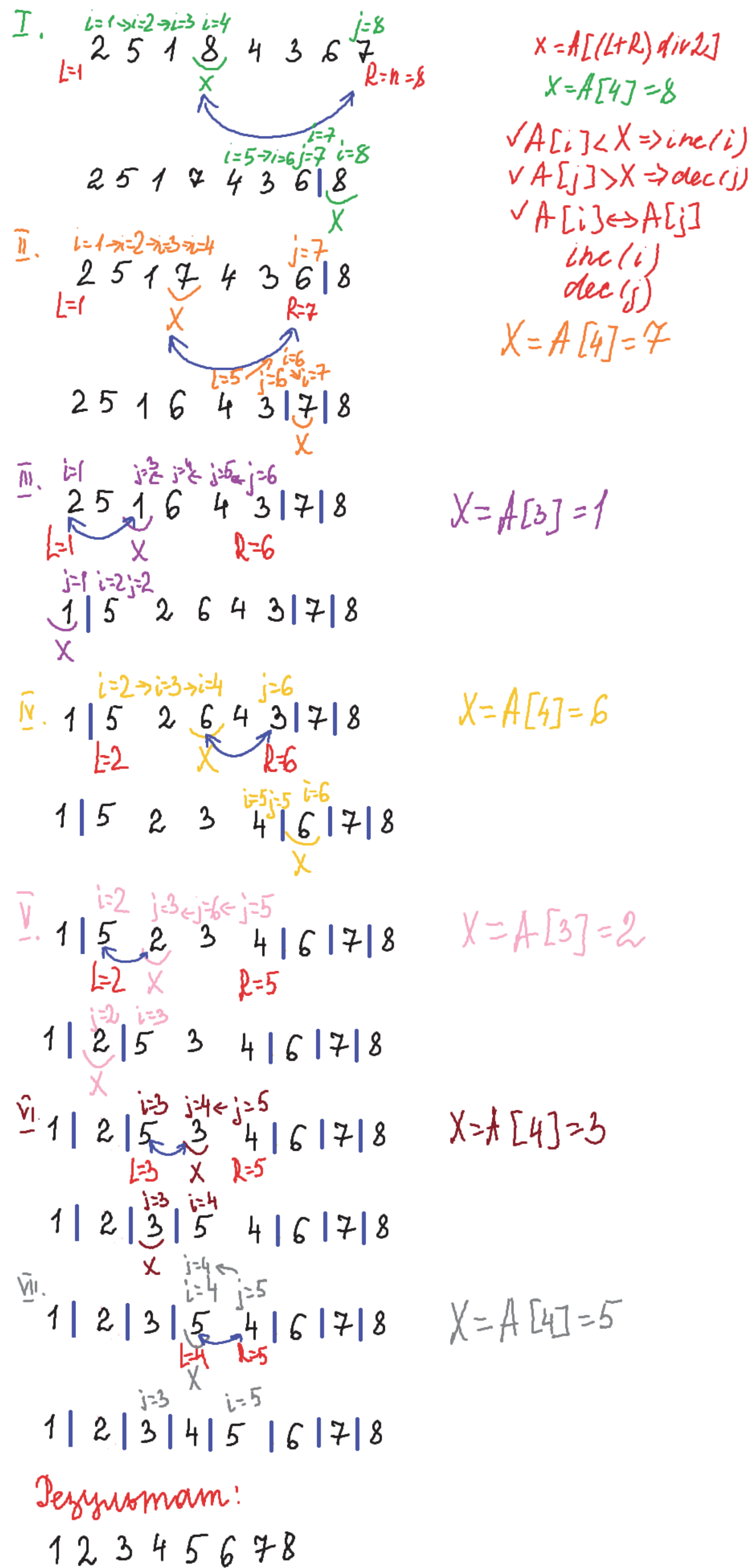


Рис. 15. Пример работы быстрой сортировки

$T(n) = T_{ср.}(n) \sim O(n \cdot \log n)$, $T_{наих.}(n) \sim O(n^2)$.

Анализ Quicksort. Процесс разделения идет следующим образом: выбрав некоторое граничное значение x, мы затем проходим целиком по всему массиву. При этом выполняется точно n сравнений. Ожидаемое число обменов есть среднее этих ожидаемых значений для всех возможных границ x.

$$M = [\mathrm{Sx}:1 \le x \le n:(x-1)\cdot(n-(x-1))/n]/n = [\mathrm{Su}:0 \le u \le n-1:\ u\cdot(n-u)]/n^2 =$$

$$= n\cdot(n-1)/2n - (2n^2-3n+1)/6n = (n-1/n)/6$$

В том случае, если бы нам всегда удавалось выбирать в качестве границы медиану, то каждый процесс разделения расщеплял бы массив на две половинки, и для сортировки требовалось бы всего $n \cdot \log n$ подходов. В результате общее число сравнений было бы равно $n \cdot \log n$, а общее число обменов – $n \cdot \log(n)/6$. Но вероятность этого составляет только $1/n$.

Главный из недостатков *Quicksort* – недостаточно высокая производительность при небольших n, впрочем, этим грешат все усовершенствованные методы. Одним из достоинств является то, что для обработки небольших частей в него можно легко включить какой-либо из прямых методов сортировки.

Список рекомендуемой литературы

1. Ахо, А. Структуры данных и алгоритмы / А. Ахо, Дж. Хопкрофт, Дж. Ульман. – Москва : Вильямс, 2000. – 384 с.

2. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт. – Москва : Мир, 1989. – 360 с.

3. Долганова, Н.Ф. Теоретические основы прикладной математики и информатики : элементы теории разработки эффективных алгоритмов / Н.Ф. Долганова, В.М. Долганов, А.Н. Стась. – Томск : Изд-во ТГПУ, 2019. – 34 с.

4. Кнут, Д. Искусство программирования для ЭВМ / Д. Кнут. – Т. 3. – Сортировка и поиск. – Москва : Мир, 1978. – 848 с.

Приложение 1

Временная трудоемкость и ее оценка

Пример. Рассмотрим алгоритм вычисления значения многочлена степени n в заданной точке x:

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_i x^i + \ldots + a_1 x^1 + a_0.$$

Алгоритм 1. Для каждого слагаемого, кроме a_0 возвести x в заданную степень последовательным умножением и затем домножить на коэффициент. Затем слагаемые сложить.
Вычисление i-го слагаемого ($i = 1..n$) требует i умножений. Значит, всего $1 + 2 + 3 + \ldots + n = n \cdot (n + 1)/2$ умножений. Кроме того, требуется n сложений. Всего $n \cdot (n + 1)/2 + n = n^2/2 + 3n/2$ операций.

Алгоритм 2. Вынесем x-ы за скобки и перепишем многочлен в виде:

$$P_n(x) = a_0 + x(a_1 + x(a_2 + \ldots (a_i + \ldots x(a_{n-1} + a_n x))).$$

Например, $P_3(x) = a_3x^3 + a_2x^2 + a_1x^1 + a_0 = a_0 + x(a_1 + x(a_2 + a_3x))$.

Будем вычислять выражение изнутри. Самая внутренняя скобка требует 1 умножение и 1 сложение. Ее значение используется для следующей скобки… и так, 1 умножение и 1 сложение на каждую скобку, которых $n - 1$ штука. И еще после вычисления самой внешней скобки умножить на x и прибавить a_0. Всего n умножений плюс n сложений, т.е. $2n$ операций.

Зачастую такая подробная оценка не требуется. Вместо нее приводят лишь асимптотическую скорость возрастания количества операций при увеличении n.

Трудоемкостью (временной сложностью, временной трудоемкостью) алгоритма называется количество элементарных шагов, выполняемых алгоритмом для данного входа.

Элементарным шагом называется любое действие алгоритма, время выполнения которого не зависит от длины входа.

Длина входа – это любая характеристика, влияющая на трудоемкость.

Функцию $T(n)$, где n – длина входа, называют функцией трудоемкости.

$$\lim_{n\to\infty} T(n) = O(f(n))$$

Замечание. Действия, которые можно считать элементарным шагом, определяются алгоритмическим языком. Мы будем записывать алгоритмы на языке Паскаль и считать элементарными шагами следующие операции:

- выполнение оператора присваивания;
- проверку условия в условных операторах;
- однократное выполнение заголовка цикла (проверка условия, инкремент/декремент переменной цикла);
- вызов и выход из функции (только передача параметров и управления).

Указанные операции являются элементарными шагами, если они не включают действий, которые нельзя считать элементарными (например, исполнение некоторой функции или обработку всех элементов некоторого массива). В противном случае необходимо учитывать, сколько элементарных шагов требуется для выполнения неэлементарных действий.

Таким образом, временная сложность алгоритма определяется количеством входных данных. Для простоты входные данные представляются параметром n. Этот параметр пропорционален величине обрабатываемого набора данных и может обозначать:

- размер массива или файла при сортировке или поиске;
- степень полинома;
- количество символов в строке;
- другую абстрактную меру объема рассматриваемой задачи.

Скорость роста часто используемых функций оценки временной сложности алгоритмов

n	$\log n$	$n \cdot \log n$	n^2
1	0	0	1
16	4	64	256
256	8	2 048	65 536
4 096	12	49 152	16 777 216
65 536	16	1 048 565	4 294 967 296
1 048 476	20	20 969 520	1 099 301 922 576
16 775 616	24	402 614 784	281 421 292 179 456

Если считать, что числа соответствуют микросекундам, то для задачи с 1048476 элементами алгоритму со временем работы $O(\log n)$ потребуется 20 микросекунд, а алгоритму со временем работы $O(n^2)$ – более 12 дней.

Классы сложности алгоритмов в зависимости от функции трудоемкости

Случаи оценок трудоемкости	Характеристика класса алгоритмов	Примеры алгоритмов
$T(n) = O(1)$	Константная сложность. Большинство инструкций большинства функций запускается один или несколько раз. Если все инструкции программы обладают таким свойством, то время выполнения программы постоянно.	Выполнение оператора присваивания.
$T(n) = O(\log n)$	Логарифмическая сложность. Такое время выполнения обычно присуще программам, которые сводят большую задачу к набору меньших подзадач, уменьшая на каждом шаге размер задачи на некоторый постоянный фактор. Изменение основания не оказывает заметного влияния на изменение значения логарифма: при $n = 1\ 000$, $\log_{10} n = 3$, $\log_2 n \approx 10$	Поиск элемента в упорядоченном массиве длины n.
$T(n) = O(n)$	Линейная сложность. Когда время выполнения программы является линейным, это обычно значит, что каждый входной элемент подвергается небольшой обработке.	Поиск элемента в неупорядоченном массиве длины n.

Классы сложности алгоритмов в зависимости от функции трудоемкости

Случаи оценок трудоемкости	**Характеристика класса алгоритмов**	**Примеры алгоритмов**
$T(n) = O(n \cdot \log n)$	Линейно-логарифмическая сложность. Время выполнения, пропорциональное $n \cdot \log n$, возникает тогда, когда алгоритм решает задачу, разбивая ее на меньшие подзадачи, решая их независимо и затем объединяя решения.	Сортировка слиянием массива длины n.
$T(n) = O(n^p)$, где $p > 1$, $p = const$.	Полиномиальная сложность. Например, когда время выполнения алгоритма является квадратичным, он полезен для практического использования при решении относительно небольших задач. Квадратичное время выполнения обычно появляется в алгоритмах, которые обрабатывают все пары элементов данных (возможно, в цикле двойного уровня вложенности). Или похожий алгоритм, который обрабатывает тройки элементов данных (наиболее часто в цикле тройного уровня вложенности), имеет кубическое время выполнения и практически применим лишь для малых задач.	$p = 2$ – сортировка массива длины n пузырьком, $p = 3$ – умножение квадратных матриц размерности $n \times n$.
$T(n) = O(2^{kn})$, где $k = const$.	Лишь несколько алгоритмов с *экспоненциальным* временем выполнения имеют практическое применение, хотя такие алгоритмы возникают естественным образом при попытках прямого решения задачи, например полного перебора.	$k = 1$ – Ханойские башни с n дисками

Классы сложности алгоритмов в зависимости от функции трудоемкости

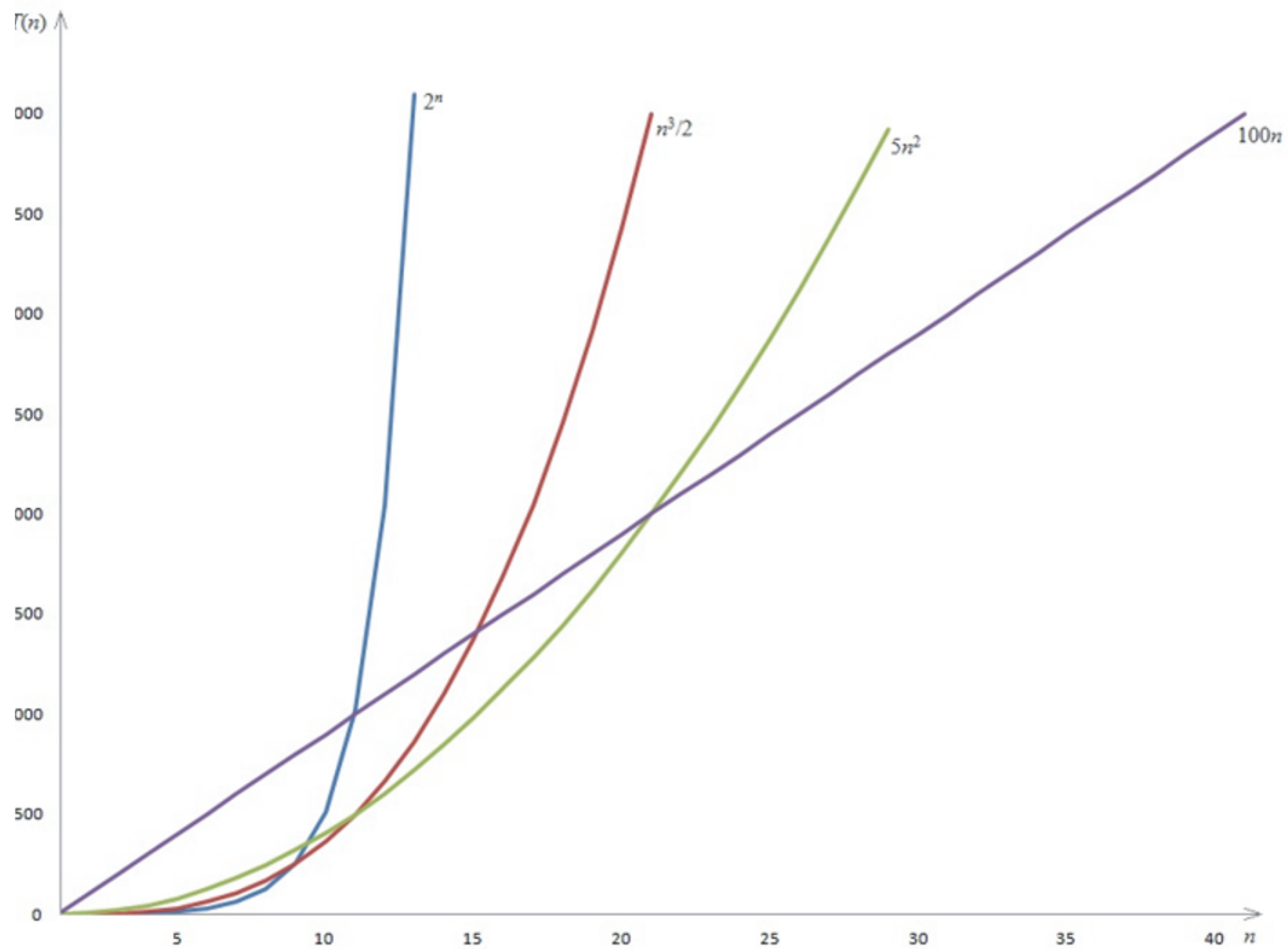


Рис. 1. Функции времени выполнения четырех программ

Оценка временной трудоемкости алгоритма

- Правило суммы.
- Правило произведения.
- Оценка рекурсивных алгоритмов.

Примеры оценки некоторых алгоритмических конструкций

Вид управляющей структуры	**Сложность**
Присваивание	$O(1)$
Простое выражение	$O(1)$
Подряд идущие фрагменты программы: P_1 P_2	доминанта из $\{O(\text{выч.}P_1), O(\text{выч.}P_2)\}$

if <условие> then
S_1
else
S_2
	доминанта из $\{O(\text{выч.}S_1), O(\text{выч.}S_2)\}$ и $O(\text{выч. условия})$

for i:= 1 to n do P	$O(n \times \text{выч.}P)$

Примеры оценки временной трудоемкости алгоритма

Алгоритм обработки элементов массива:

```
for i:= 1 to n do begin
    writeln (A[i]);
    inc (k);
end;
```

Сложность этого алгоритма $O(n)$, так как тело цикла выполняется n раз, и сложность тела цикла равна $O(1)$.

```
for i:= 1 to n do
for j:= 1 to n do begin
    …
end;
```

Сложность этого алгоритма $O(n^2)$.

```
function fact (n: integer): integer;
begin
    if n <= 1 then
    fact:= 1
    else
    fact:= n · fact(n – 1)
end;
```

$$T(n) = \begin{cases} c + T(n-1), \text{ если } n > 1; \\ d, \text{ если } n \le 1. \end{cases}$$

Полагая, что $n > 2$, и раскрывая в соответствии с вышеприведенным соотношением выражение $T(n-1)$ (т.е. подставляя в него $n-1$ вместо n), получим $T(n) = 2c + T(n-2)$. Аналогично, если $n > 3$, раскрывая $T(n-2)$, получим $T(n) = 3c + T(n-3)$. Продолжая этот процесс, в общем случае для некоторого $n > i$, имеем $T(n) = ic + T(n-i)$. Положив в последнем выражении $i = n-1$, окончательно получаем

$$T(n) = c(n-1) + T(1) = c(n-1) + d.$$

Примеры оценки временной трудоемкости алгоритма

```
  A{ Writeln('Введите ограничение');
     Readln(N);
     small:=1;

     While small < N do
       begin
  B{   next:=small;
       While next <= N do
         begin
  C{       last:=next;
           While last<=N do
             begin
               if (last<=small*2) and (next<=small*2) and
               (last*last=small*small+next*next) then
                 begin
  H{               writeln(small);
                   writeln(next);
                   writeln(last);
                 end;
  J{           inc(last);
             end;
  K{       inc(next);
         end;
  L{   inc(small);
       end;
```

$O(H) = O(1) + O(1) + O(1) = O(1)$;
$O(I) = O(n)*(O(F) + O(J)) = O(n)*O$(доминанты условия) $= O(n)$;
$O(G) = O(n)*(O(C) + O(I) + O(K)) = O(n)*(O(1) + O(n) + O(1)) = O(n^2)$;
$O(E) = O(n)*(O(B) + O(G) + O(L)) = O(n)*O(n^2) = O(n^3)$;
$O(D) = O(A) + O(E) = O(1) + O(n^3) = O(n^3)$.

Сложность данного алгоритма $O(n^3)$.

```
function search (low, high, key: integer): integer;
   var
   mid, data: integer;
begin
   while low <= high do
   begin
mid:= (low + high) div 2;
data:= A[mid];
if key = data then
search:= mid
else
if key < data then
high:= mid – 1
else
low:= mid + 1;
   end;
   search:= –1;
end;
```

Первая итерация цикла имеет дело со всем списком. Каждая последующая итерация делит пополам размер подсписка. Так, размерами списка для алгоритма являются: n, $n/2^1$, $n/2^2$, $n/2^3$, $n/2^4$, …, $n/2^m$.

В конце концов будет такое целое m, что $n/2^m < 2$ или $n < 2^{m+1}$.

Так как m – это первое целое, для которого $n/2^m < 2$, то должно быть верно $n/2^{m-1} >= 2$ или $2^m =< n$.

Из этого следует, что $2^m =< n < 2^{m+1}$.

Возьмем логарифм каждой части неравенства и получим $m =< \log_2 n = x < m+1$.

Значение m – это наибольшее целое, которое $=< x$.
Итак, $O(\log_2 n)$.

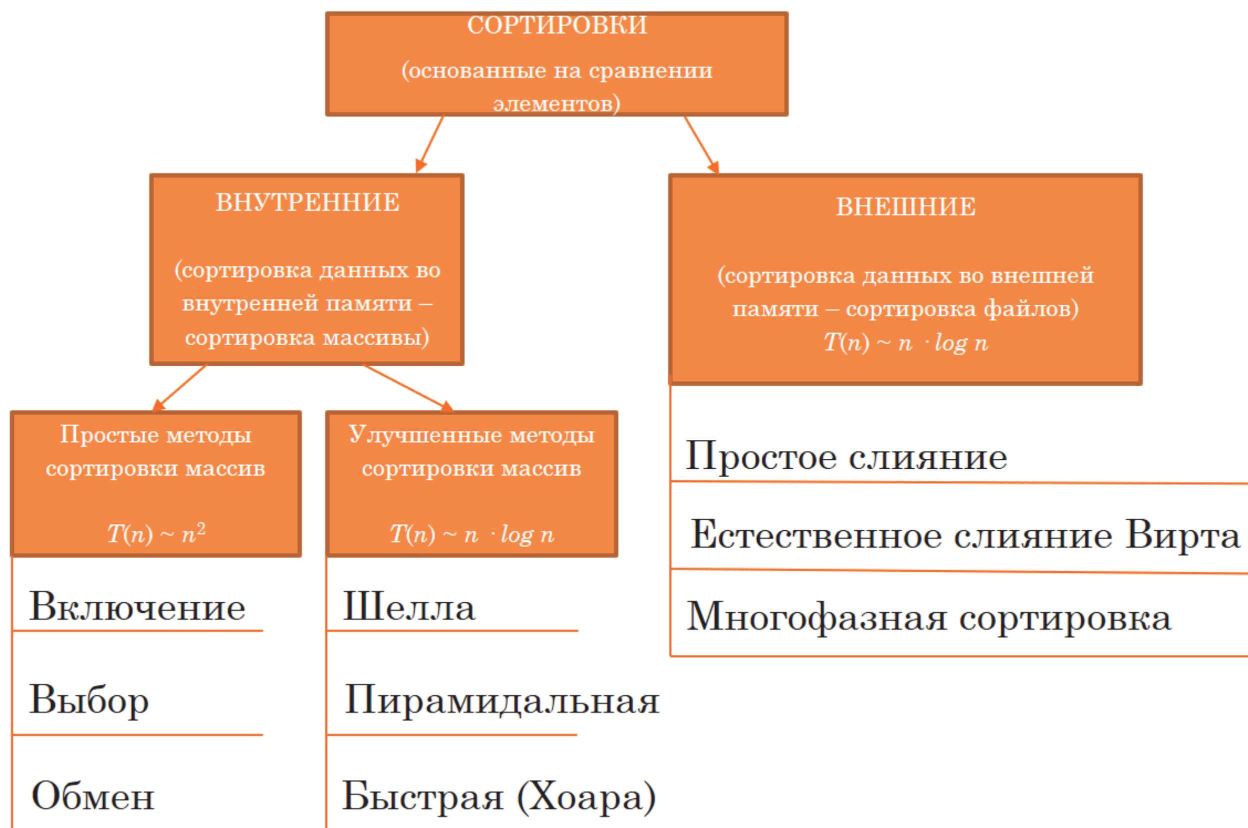
Временной сложностью задачи называется временная трудоемкость наиболее эффективного алгоритма ее решения.

Однако для оценки сложности задачи помимо оценки конкретного алгоритма необходимо доказать, что он является самым эффективным.

Приложение 2

Постановка задачи сортировки. Нижняя оценка трудоемкости сортировок, основанных на сравнениях

Сортировка – КЛАСС ЗАДАЧ, СВЯЗАННЫХ С УПОРЯДОЧИВА ПОСЛЕДОВАТЕЛЬНОСТЕЙ ПО ВОЗРАСТАНИЮ (УБЫВАНИЮ).



Теорема. Нижняя оценка $T(n)$ для алгоритмов сортировки, основанных на сравнении, имеет трудоемкость порядка $O(n \cdot log\ n)$.

$$\frac{n!}{2^{k}} = 1,$$

$$n! = 2^{k},$$

$$k = \log_2 n!,$$

$$\log_2 n! = \log_2(1 \cdot 2 \cdot \dots \cdot n) = \log 1 + \log 2 + \dots + \log n \le$$

$$\le \log n + \log n + \dots + \log n \sim n \cdot \log n.$$

При доказательстве мы использовали известные свойства алгоритмов:

$$\log(ab) = \log a + \log b,$$

$$a \le b \ \longrightarrow \log a \le \log b$$

Можно доказать и обратное утверждение:

$$n \cdot \log n = \log n + \log n + \log n + \ldots + \log n \le$$

$$\le \log(1 \cdot n) + \log(2 \cdot (n-1)) + \log(3 \cdot (n-2)) + \ldots + \log(n \cdot 1) =$$

$$= [\log 1 + \log n] + [\log 2 + \log(n-1)] + [\log 3 + \log(n-2)] + \ldots + [\log n + \log 1] =$$

$$= 2 \cdot \log 1 + 2 \cdot \log 2 + 2 \cdot \log 3 + \ldots + 2 \cdot \log n = 2 \cdot (\log 1 + \log 2 + \log 3 + \ldots + \log n) =$$

$$= 2 \cdot \log(1 \cdot 2 \cdot 3 \cdot \ldots \cdot n) = 2 \cdot \log n!$$

Таким образом, $T(n) \sim O(n \cdot \log n)$.

Приложение 3

Сортировка с помощью прямого включения

Сортировка с помощью прямого включени

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
Начальные ключи		4	5	1	3	6	8	2	7
i = 2	5	4	5	1	3	6	8	2	7
i = 2		4	5	1	3	6	8	2	7
i = 3	1	4	5	1	3	6	8	2	7
i = 3		1	4	5	3	6	8	2	7
i = 4	3	1	4	5	3	6	8	2	7
i = 4		1	3	4	5	6	8	2	7
i = 5	6	1	3	4	5	6	8	2	7
i = 5		1	3	4	5	6	8	2	7
i = 6	8	1	3	4	5	6	8	2	7
i = 6		1	3	4	5	6	8	2	7
i = 7	2	1	3	4	5	6	8	2	7
i = 7		1	2	3	4	5	6	8	7
i = 8	7	1	2	3	4	5	6	8	7
Результат		1	2	3	4	5	6	7	8

```
for  i:= 2 to n do begin
 x:= A[i];
 A[0]:= x;
 j:= i;
 while  x  <  A[j  –  1]  do
begin
  A[j]:=A[j – 1];
  dec (j)
 end;
 A[j]:= x
end;
```

Алгоритм сортировки с помощью прямого включения

на каждом шаге, начиная со второго (***i* = 2**) *и увеличивая i каждый раз на единицу*

извлекается ***i****-й элемент* (***x* = *A*[*i*]**);

включение ***x*** *на соответствующее место среди* ***a*[1]**, … , ***a*[*i*]**

Pascal	Python

for i:= 2 to n do begin
x:= A[i];
A[0]:= x;
j:= i;
while x < A[j – 1] do begin
A[j]:=A[j – 1];
dec (j)
end;
A[j]:= x
end;
	for i in range(1,n):
	x = A[i]
	j = i – 1
	while j >= 0 and x < A[j]:
	A[j + 1] = A[j]
	j-=1
	A[j + 1] = x

Очевидно, $T(n) \sim O(n^2)$, $T_{ср.}(n) \sim O(n^2)$.

Приложение 4

Сортировка с помощью прямого выбора

Сортировка с помощью прямого выбора

Начальные ключи	4	5	1	3	6	8	2	7
i = 1	1	5	4	3	6	8	2	7
i = 2	1	2	3	4	6	8	5	7
i = 3	1	2	3	4	6	8	5	7
i = 4	1	2	3	4	6	8	5	7
i = 5	1	2	3	4	5	6	8	7
i = 6	1	2	3	4	5	6	8	7
i = 7	1	2	3	4	5	6	7	8
Результат	1	2	3	4	5	6	7	8

```
for i:= 1 to n-1 do
    Присвоить min индекс наименьшего из a[i] … a[n];
    Поменять местами a[i] и a[min];
end
```

```
for i:=1 to n-1 do begin
  min:=i;
  x:=A[i];
  for j:=i+1 to n do
  if A[j]<x then begin
   min:=j;
   x:=A[min]
  end;
  A[min]:=A [i];
  A[i]:=x
 end;
```

Алгоритм сортировки с помощью прямого выбора

```
for i:= 1 to n-1 do
    Присвоить min индекс наименьшего из a[i] … a[n];
    Поменять местами a[i] и a[min];
end
```

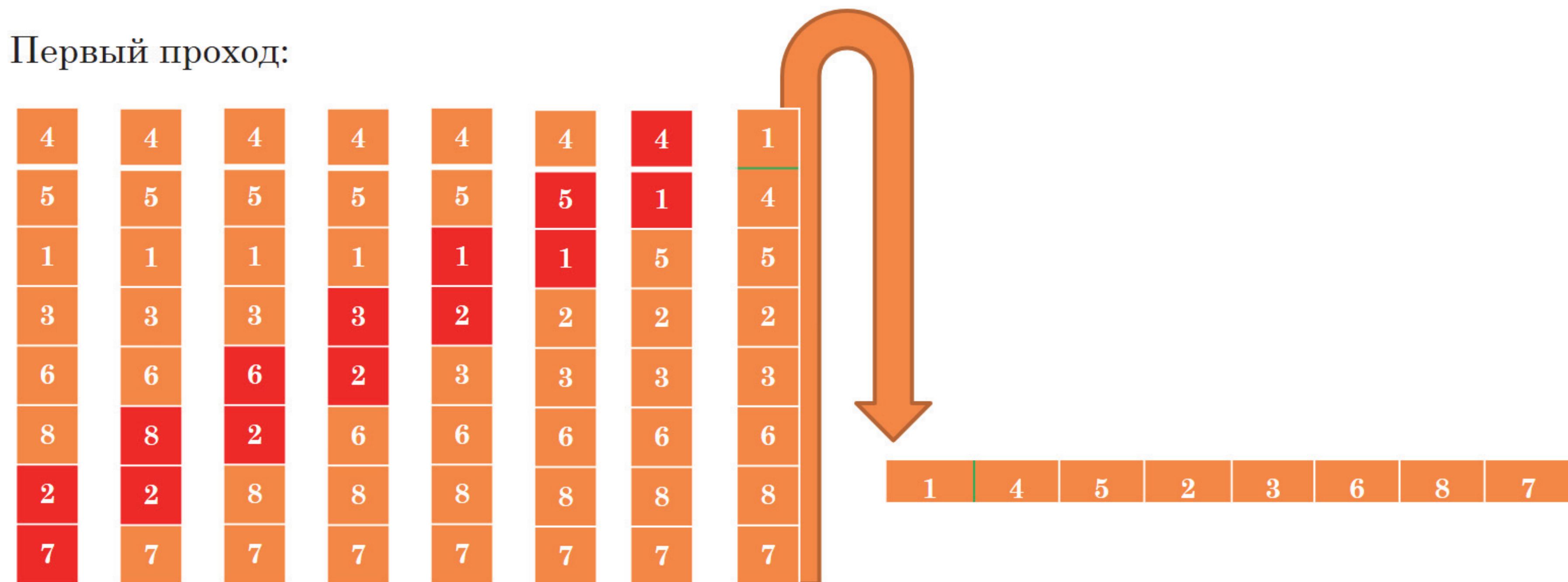
Pascal	Python

for i:=1 to n-1 do begin
  min:=i;
  x:=A[i];
  for j:=i+1 to n do
  if A[j]<x then begin
   min:=j;
   x:=A[min]
  end;
  A[min]:=A [i];
  A[i]:=x
 end;
	for i in range(n):
	    min = i
	
	    for j in range(i+1,n):
	        if A[j]<A[min]:
	            min=j
	    x = A[min]
	
	    A[min] = A[i]
	    A[i]=x

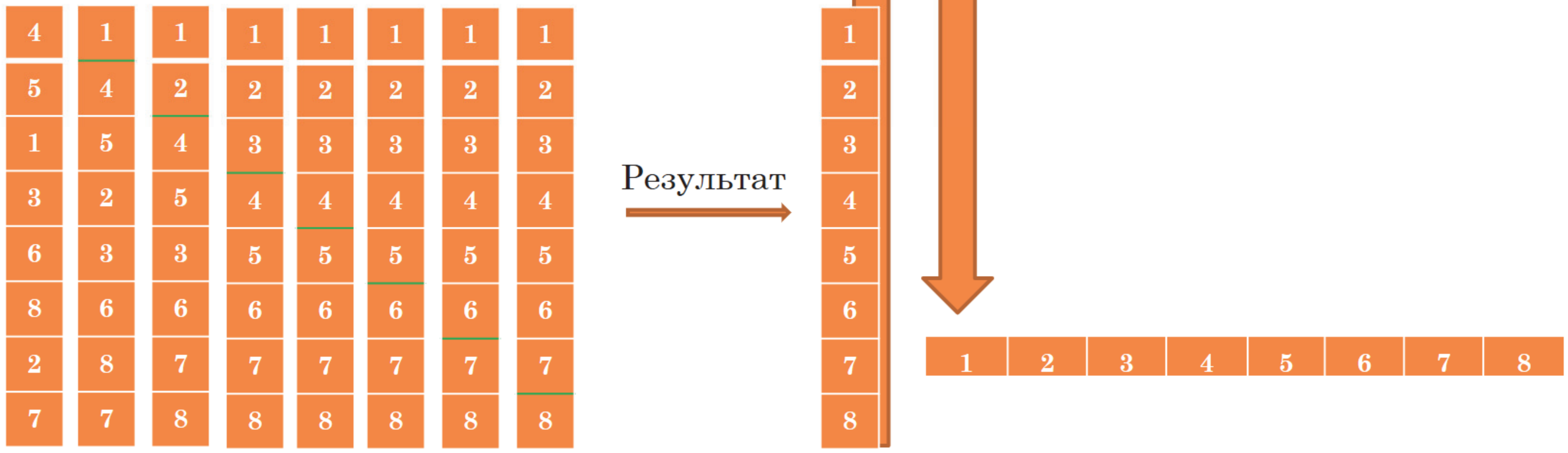
Очевидно, $T(n) \sim O(n^2)$, $T_{ср.}(n) \sim O(n^2)$.

Приложение 5

Сортировка с помощью прямого обмена



Сортировка с помощью прямого обмена



Алгоритм сортировки с помощью прямого обмена

i = 2 to n do j := n downto i do
если A[j] *больше* A[j – 1], *поменяй их местами*;
end;

Pascal	Python

for i:=2 to n do
for j:=n downto i do
if A[j-1]>A[j] then begin
  x:=A[j-1];
  A[j-1]:=A[j];
  A[j]:=x
end;
	for i in range(n-1):
	  for j in range(n-1, i, -1):
	    if A[j-1]>A[j]:
	      x = A[j]
	      A[j] = A[j-1]
	      A[j-1] = x

Очевидно, $T(n) \sim O(n^2)$, $T_{\text{ср.}}(n) \sim O(n^2)$.

Приложение 6

Сортировка Шелла

Сортировка Шелла

4	5	8	3	6	1	2	7

Четверная сортировка (h_3=4)							
4	5	8	3	6	1	2	7
Результат четвертной сортировки							
4	1	2	3	6	5	8	7

Двойная сортировка (h_2=2)							
4	1	2	3	6	5	8	7
Результат двойной сортировки							
2	1	4	3	6	5	8	7

Одинарная сортировка (включение: h_1=1)							
2	1	4	3	6	5	8	7
Результат одинарной сортировки							
1	2	3	4	5	6	7	8

Запишем алгоритм метода Шелла на обобщенном паскале:

```
// определяем h-цепочку и заносим ее в массив h;
// цикл по m для перебора всех расстояний
begin
  k := h[m]; s := – k; // k – шаг сортировки
  for i := k + 1 to n do begin
     x := a[i]; j := i – k;
     if s = 0 then s := –k; //отсечение случая
     inc(s); a[s] := x; // выхода за пределы массива
     while x < a[j] do begin  //простые вставки с шагом k
       a[j + k] := a[j];
       j := j – k
    end;
    a[j + k] := x
  end
end;
```

Алгоритм сортировки Шелла

$T(n) \sim O(n \cdot \log n)$, $T_{ср.}(n) \sim O(n^{1.2})$, $T_{наих.}(n) \sim O(n^2)$.

Pascal

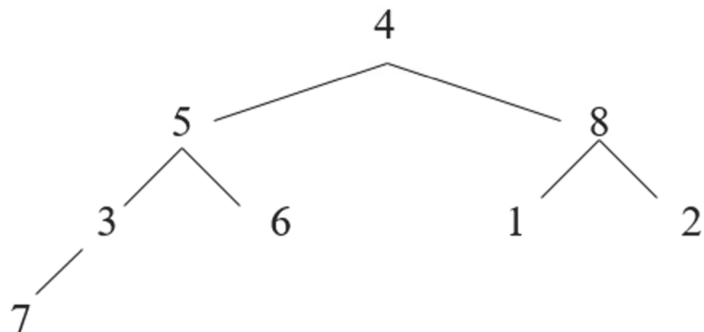
```pascal
program shells;
const  t=4;
 h: array [1..t] of integer = (15,7,3,1);
var
 i,j,k,s,x,n,m: integer;
 a:array [-15..50] of integer;
begin
 // ввод массива
 for m:=1 to t do begin
  k:=h[m];
  s:=-k;
  for i:=k+1 to n do begin
   x:=a[i];
   j:=i-k;
   if s=0 then s:=-k;
   inc(s);
   a[s]:=x;
   while x<a[j] do begin
    a[j+k]:=a[j];
    j:=j-k
   end;
   a[j+k]:=x
  end;
 end;
 // вывод массива
end.
```

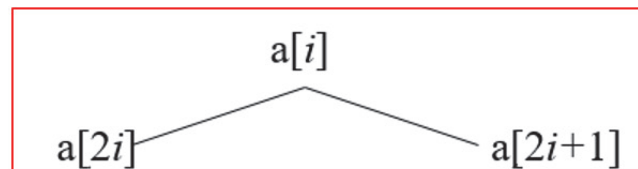
Python

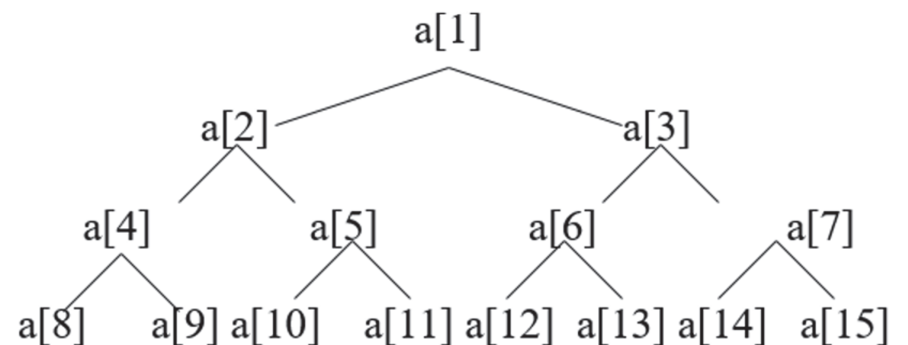
```python
A = list(map(int, input().split()))
n = len(A)
HL = [15, 7, 3, 1]
for h in HL:
    for i in range(h,n):
        x = A[i]
        j = i-h
        while j>=0 and x<A[j]:
            A[j+h]=A[j]
            j-=h
        A[j+h]=x
for x in A:
    print(x, end=" ")
```

Пирамидальная сортировка (Heapsort)

Пирамидальная сортировка

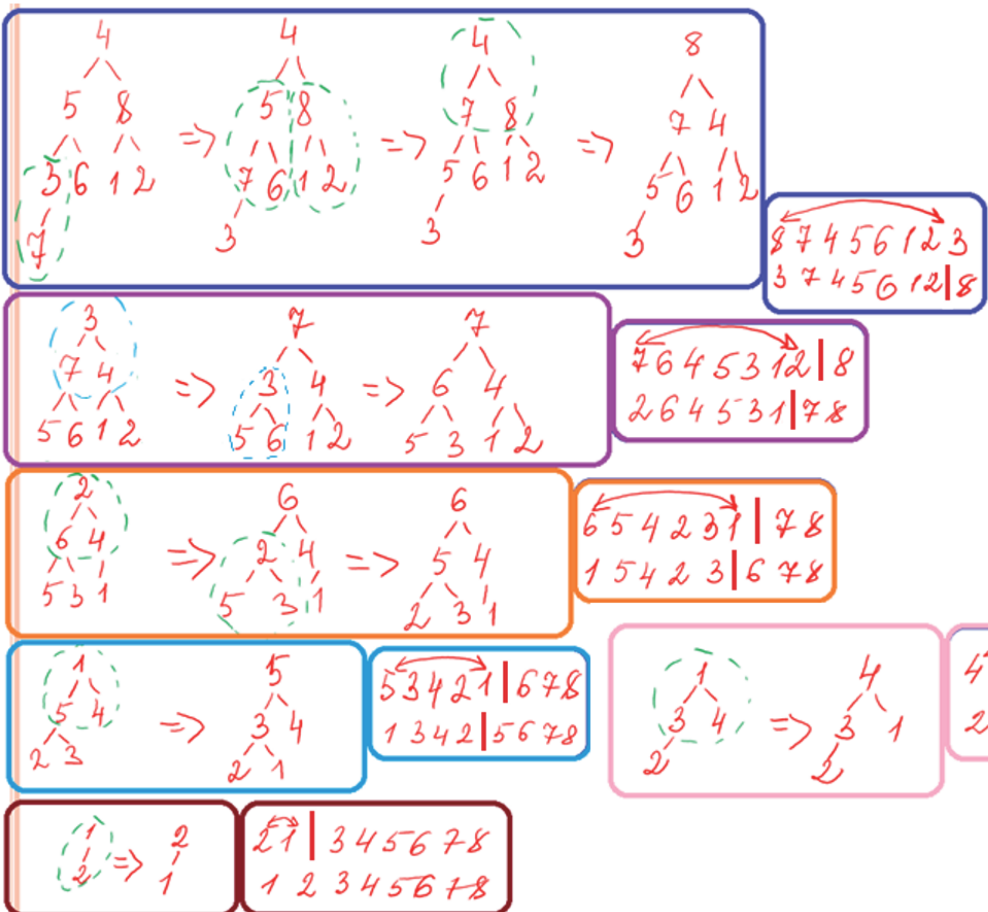


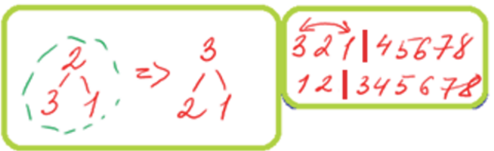


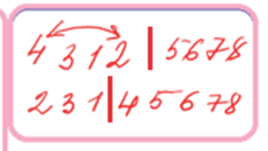


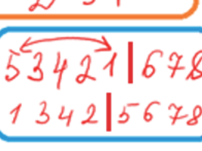
Пирамидальная сортировка

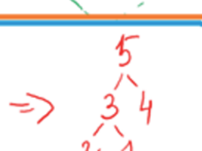
4	5	8	3	6	1	2	7











$$a[i] \geq a[2i] \text{ и } a[i] \geq a[2i+1]$$

BEGIN

//ввод массива длины n

// определить левую границу L как позицию среднего элемента плюс один в массиве A;

// определить правую границу R, равную количеству элементов массива A;

Пока элементы слева есть делай

уменьши L на единицу;

вызови процедуру просеивания SIFT для фактических параметров L,N

END;

Пока правая граница больше единицы делай

поменяй местами A[1] и A[R] элементы;

уменьши правую границу на единицу;

вызови процедуру просеивания SIFT для фактических параметров 1,R

END;

//вывод отсортированного массива на экран

END.

Алгоритм пирамидальной сортировки

$$T(n) = T_{\text{ср.}}(n) = T_{\text{наих.}}(n) \sim O(n \cdot \log n).$$

Pascal

```
PROGRAM HS;
VAR I,X,L,N,R:INTEGER;
   A:ARRAY[0..50] OF INTEGER;

PROCEDURE SIFT(L,R: INTEGER);
VAR
 I,J,X: INTEGER;
BEGIN
 I:=L;
 J:=2*L;
 X:=A[L];
 IF (J<R)AND(A[J]<A[J+1]) THEN INC(J);
 WHILE (J<=R)AND(X<A[J]) DO BEGIN
  A[I]:=A[J];
  A[J]:=X;
  I:=J;
  J:=2*J;
  IF (J<R)AND(A[J]<A[J+1]) THEN INC(J);
 END
END;

BEGIN
 //ввод массива
 L:=(N DIV 2)+1;
 R:=N;
 WHILE L>1 DO BEGIN
  DEC(L);
  SIFT(L,N)
 END;
 WHILE R>1 DO BEGIN
  X:=A[1];
  A[1]:=A[R];
  A[R]:=X;
  DEC(R);
  SIFT(1,R)
 END;
 //вывод массива
END.
```

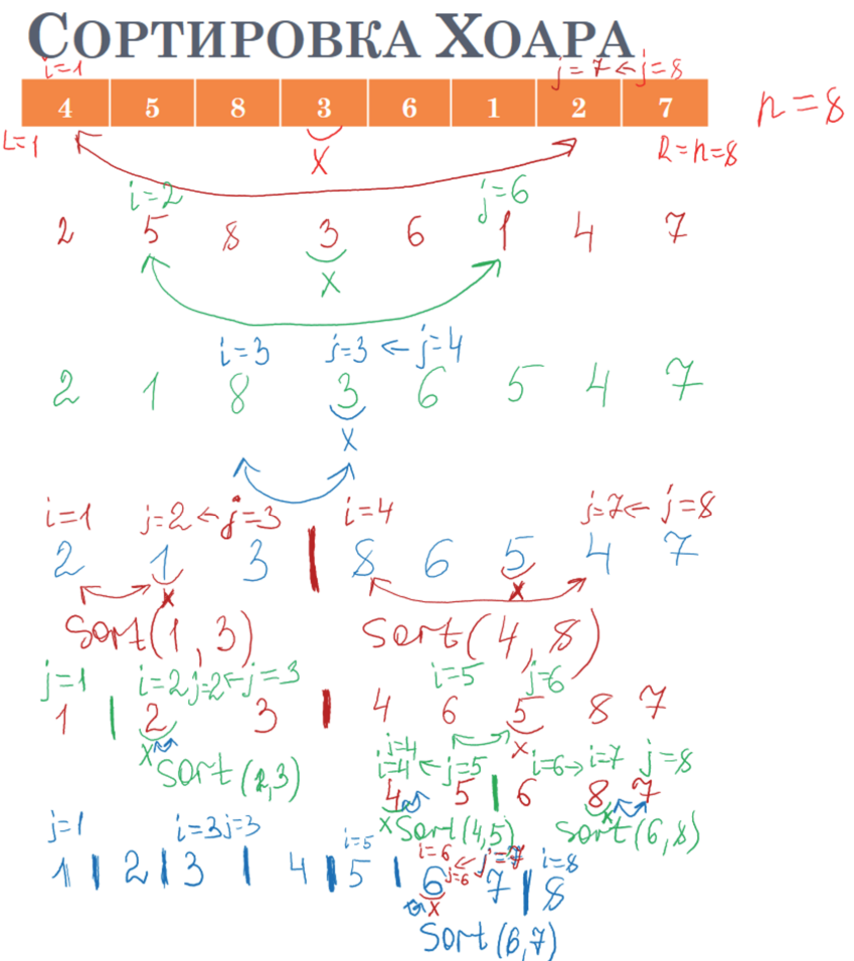
Python

```
def sift(l,r):
    i = l
    j = 2*i+1
    if j+1<=r and A[j+1]>A[j]:
        j = j+1
    while j<=r and A[j]>A[i]:
        x = A[i]
        A[i] = A[j]
        A[j] = x
        i = j
        if j+1<=r and A[j+1]>A[j]:
            j = j+1

A = list(map(int, input().split()))
n = len(A)
for i in range(n//2-1, -1, -1):
    sift(i, n-1)
x = A[0]
A[0] = A[n-1]
A[n-1] = x
r = n-2
for i in range(n-2):
    sift(0, r)
    x = A[0]
    A[0] = A[r]
    A[r] = x
    r-=1
for x in A:
    print(x, end=" ")
```

Приложение 8

Быстрая сортировка (Quicksort)



SORT (1,n)

```
PROCEDURE SORT(L,R: INTEGER);
VAR
 I,J,X,W: INTEGER;
BEGIN
//определить значение переменной I как соответствующее значению левой границе просмотра L;
//определить значение переменной J как соответствующее значению левой границе просмотра R;
//определить значение переменной X, равной значению серединного элемента;
 REPEAT
   Пока A[I] расположено верно относительно X инкрементируй I;
   Пока A[J] расположено верно относительно X декрементируй J;
   Если I «не забежало» за J поменяй местами A[I] и A[J] и сдвинь индексы I и J навстречу друг другу
UNTIL I>J;
 IF L<J THEN SORT(L,J);
 IF I<R THEN SORT(I,R);
END;
```

Алгоритм быстрой сортировки

$T(n) = T_{ср.}(n) \sim O(n \cdot \log n)$, $T_{наих.}(n) \sim O(n^2)$.

Pascal

```
PROGRAM QS;
VAR N,I:INTEGER;
   A:ARRAY[0..50] OF INTEGER;

PROCEDURE SORT(L,R: INTEGER);
VAR
 I,J,X,W: INTEGER;
BEGIN
 I:=L;
 J:=R;
 X:=A[(L+R) DIV 2];
 REPEAT
  WHILE A[I]<X DO INC(I);
  WHILE X<A[J] DO DEC(J);
  IF I<=J THEN BEGIN
   W:=A[I];
   A[I]:=A[J];
   A[J]:=W;
   INC(I);
   DEC(J)
  END
 UNTIL I>J;
 IF L<J THEN SORT(L,J);
 IF I<R THEN SORT(I,R);
END;


BEGIN
 //ввод массива
 SORT(1, N);
 //вывод массива
END.
```

Python

```
def sort(l,r):
    if r>l:
        x = A[(l+r)//2]
        i = l
        j = r
        while i<=j:
            while A[i]<x:
                i+=1
            while A[j]>x:
                j-=1
            if i<=j:
                x = A[i]
                A[i] = A[j]
                A[j] = x
                i+=1
                j-=1
        sort(l,j)
        sort(i,r)

A = list(map(int, input().split()))
n = len(A)
sort(0, n-1)
for x in A:
    print(x, end=" ")
```

Приложение 9

Лабораторная работа 1 «Простые методы сортировки массивов»

План изучения:

1) основные понятия;
2) сортировка с помощью прямого включения;
3) сортировка с помощью прямого выбора;
4) сортировка с помощью прямого обмена.

Цель работы:

освоить алгоритмы простых методов сортировки массивов.

Основные понятия

Сортировка – задача, заключающаяся в расположении элементов последовательности в определенном порядке.

Рассмотрим и проанализируем простые методы сортировки, которые являются основными. Именно на них основаны все известные методы упорядочивания данных в зависимости от условия задачи (по убыванию, по возрастанию, по алфавиту).

Простые методы сортировки данных во внутренней памяти являются неэффективными с точки зрения временной трудоемкости и имеют трудоемкость квадратичного порядка от длины входных данных.

В данной работе остановимся на следующих простых методах сортировки массивов: сортировка с помощью прямого включения; сортировка с помощью прямого выбора; сортировка с помощью прямого обмена (см. п. 1–3).

1. *Сортировка с помощью прямого включения*

Такой метод широко используется при игре в карты. Элементы мысленно делятся на уже «готовую» последовательность a_1, …, a_{i-1} и исходную последовательность. При каждом шаге, начиная с I = 2 и увеличивая I каждый раз на единицу, из исходной последовательности извлекается *i*-й элементы и перекладывается в готовую последовательность, при этом он вставляется на нужное место. В табл. 1 в качестве примера показан процесс сортировки с помощью включения восьми случайно выбранных чисел.

Таблица 1

Пример сортировки с помощью прямого включения

Начальные ключи	44	55	12	42	94	18	06	67
I = 2	44	55	12	42	94	18	06	67
I = 3	12	44	55	42	94	18	06	67
I = 4	12	42	44	55	94	18	06	67
I = 5	12	42	44	55	94	18	06	67
I = 6	12	18	42	44	55	94	06	67
I = 7	06	12	18	42	44	55	94	67
I = 8	06	12	18	42	44	55	67	94

Алгоритм сортировки с помощью прямого включения:

```
FOR  I: = 2  TO  n  DO
X: = A[I];
   включение  X  на  соответствующее  место  среди А [1], … , A[I]
END
```

В реальном процессе поиска подходящего места удобно, чередуя сравнения и движения по последовательности, как бы просеивать X, т.е. X сравнивается с очередным элементом A[I], а затем либо X вставляется на свободное место, либо A[J] сдвигается вправо и процесс «уходит» влево. Обратите внимание, что процесс просеивания может закончиться при выполнении одного из двух следующих различных условий:

1) найден элемент A[J] с ключом, меньшим чем ключ у X;

2) достигнут левый конец готовой последовательности.

Очевидно, $T(n) \sim O(n^2)$, $T_{\text{ср.}}(n) \sim O(n^2)$.

2*. Сортировка с помощью прямого выбора*

Этот прием основан на следующих принципах:

1) выбирается элемент с наименьшим ключом;

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, самый большой элемент.

Процесс работы этим методом с теми же восемью ключами, что и в табл. 1, приведен в табл. 2.

Алгоритм сортировки с помощью прямого выбора:

```
FOR  I:= 1 TO n-1 DO
        Присвоить  K  индекс  наименьшего  из  A[I] … A[n];
        Поменять  местами  A[I]  и  A[K];
END
```

Таблица 2

Пример сортировки с помощью прямого выбора

Начальные ключи	44	55	12	42	94	18	06	67
I = 2	06	55	12	42	94	18	44	67
I = 3	06	12	55	42	94	18	44	67
I = 4	06	12	18	42	94	55	44	67
I = 5	06	12	18	42	94	55	44	67
I = 6	06	12	18	42	44	55	94	67
I = 7	06	12	18	42	44	55	67	94
I = 8	06	12	18	42	44	55	67	94

Доказано, что $T(n) \sim T_{\text{ср.}}(n) \sim O(n^2)$.

3. *Сортировка с помощью прямого обмена*

Как и в методе прямого выбора, мы повторяем проходы по массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к левому концу массива. Если мы будем рассматривать как вертикальные, а не горизонтальные построения, то элементы можно интерпретировать как пузырьки в чане с водой, причем вес каждого соответствует его ключу (табл. 3). Такой метод сортировки известен под именем «пузырьковая сортировка».

Таблица 3

Пример «пузырьковой сортировки»

I = 1	2	3	4	5	6	7	8
44	06	06	06	06	06	06	06
55	44	12	12	12	12	12	12
12	55	44	18	18	18	18	18
42	12	55	44	42	42	42	42
94	42	18	55	44	44	44	44
18	94	42	42	55	55	55	55
06	18	94	67	67	67	67	67
67	67	67	94	94	94	94	94

Алгоритм сортировки с помощью прямого обмена:

```
I = 2 TO N DO J := N DOWNTO I DO
        ЕСЛИ A[J] БОЛЬШЕ A[J – 1], ПОМЕНЯЙ ИХ МЕСТАМИ;
END;
```

Легко видно, что $\mathrm{T}(n) \sim \mathrm{T}_{\text{ср.}}(n) \sim O(n^2)$.

Задания лабораторной работы 1:

Задание 1. Реализовать алгоритм сортировки массива с помощью прямого включения.

Задание 2. Реализовать алгоритм сортировки массива с помощью прямого выбора.

Задание 3. Реализовать алгоритм сортировки массива с помощью прямого обмена.

Уточнения к реализации алгоритмов:

1. Реализовать алгоритмы можно на любом языке программирования.
2. В программе необходимо отсортировать по возрастанию элементы заданного неотсортированного массива, состоящего минимум из восьми элементов. После завершения работы программы выходными данными является отсортированный массив. Пример входных и выходных данных приведен в табл. 4.

Таблица 4

Пример входных и выходных данных программ сортировок

INPUT	OUTPUT
44 55 12 42 94 18 06 67	06 12 18 42 44 55 67 94

Результаты лабораторной работы – файлы с кодом программ, наименовать соответственно:

ФамилияИО_номер группы_лаб.1_номер задания (1–3).

Например, «ДолгановВМ_412_лаб.1_1».

Приложение 10

Лабораторная работа 2 «Улучшенные методы сортировки массивов»

План изучения:

1) основные понятия;

2) сортировка Шелла;

3) сортировка с помощью пирамиды;

4) сортировка с помощью разделения (быстрая сортировка Хоара).

Цель работы:

освоить алгоритмы улучшенных методов сортировки массивов.

Основные понятия

Сортировка – задача, заключающаяся в расположении элементов последовательности в определенном порядке.

Рассмотрим и проанализируем улучшенные методы сортировки, наиболее часто используемые при возникновении необходимости упорядочивания большого объема входных данных.

Улучшенные методы сортировки данных во внутренней памяти являются эффективными с точки зрения временной трудоемкости и имеют линейно-логарифмический порядок временной трудоемкости.

В данной работе остановимся на следующих улучшенных методах сортировки массивов: сортировка Шелла; сортировка с помощью пирамиды; сортировка с помощью разделения (быстрая сортировка Хоара) (см. п. 1–3).

1. *Сортировка Шелла*

В 1959 г. Д. Шеллом было предложено усовершенствование сортировки с помощью прямого включения. Сам метод представлен на нашем стандартном примере в табл. 1. Сначала отдельно сортируются и группируются элементы, отстоящие друг от друга на расстояние 4. Такой процесс называется четверной сортировкой. В нашем примере восемь элементов, и каждая группа состоит из двух элементов. После первого прохода элементы перегруппировываются – теперь каждый элемент группы отстоит от другого на две позиции – и вновь сортируются. Это называется двойной сортировкой. На третьем подходе идет обычная (или одинарная) сортировка. На каждом этапе либо сортируется относительно мало элементов, либо элементы уже довольно хорошо упорядочены и требуют сравнительно немного перестановок.

Таблица 1

Сортировка с помощью включений с уменьшающимися расстояниями

44	55	12	42	94	18	06	67
Четверная сортировка дает							
44	18	06	42	94	55	12	67
Двойная сортировка дает							
06	18	12	42	44	55	94	67
Одинарная сортировка дает							
06	12	18	42	44	55	67	94

Сортировка не ориентирована на некоторую определенную последовательность расстояний. Все t расстояний обозначаются, соответственно, $h_1, h_2, \ldots, h_t$, для них выполняются условия: $h_t = 1$, $h_{i+1} < h_i$.

Описание массива на Pascal *при* этом выглядит так

A: ARRAY [-h_1..n] OF INTEGER

Алгоритм сортировки Шелла на обобщенном паскале:

```
// определяем h-цепочку и заносим ее в массив h;
// цикл по m для перебора всех расстояний
begin
   k := h[m]; s := – k; // k – шаг сортировки
   for i := k + 1 to n do begin
      x := a[i]; j := i – k;
      if s = 0 then s := –k; //отсечение случая
      inc(s); a[s] := x; // выхода за пределы массива
      while x < a[j] do begin  //простые вставки с шагом k
        a[j + k] := a[j];
        j := j – k
      end;
      a[j + k] := x
   end
end;
```

Замечание. Неизвестно, какие расстояния дают наилучший результат. Но они не должны быть множителями один другого. Кнут показал, что имеет смысл использовать такую последовательность, в которой $h_{k-1} = 3h_k + 1$, $h_t = 1$ и $t = [\log_2 n] - 1$.

Итак, в наихудшем случае метод остается квадратичным, а вот $T_{ср}(n) = n^{1,2}$. Таким образом, достигается прогресс в средних показателях. Также доказано, что $T(n) \sim O(n \cdot \log n)$.

2. *Сортировка с помощью пирамиды*

Элементы массива можно представлять в виде двоичного дерева. Считается, что a[i] порождает элементы a[2i] и a[2i+1] (рис. 1).

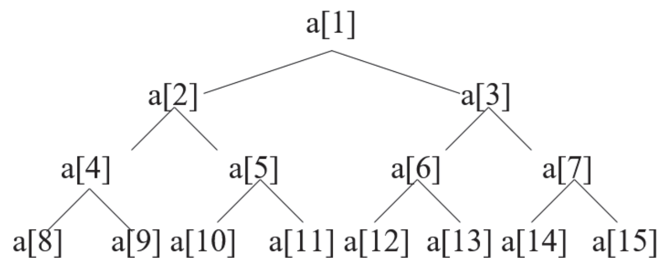


Рис. 1. Массив, представленный в виде двоичного дерева

Д. Уилльямс изобрел метод *Heapsort*, в котором было получено существенное улучшение традиционных сортировок с помощью деревьев. Пирамида определяется как последовательность ключей a[L], a[L+1], …, a[R], такая, что $a[i] \leq a[2i]$ и $a[i] <= a[2i+1]$ для $i = L \dots R/2$.

Р. Флойдом был предложен некий «лаконичный» способ построения пирамиды на «*том же месте*». Здесь $a[1] \dots a[n]$ – некий массив, причем $a[m] \dots a[n]$ ($m = [n \text{ DIV } 2] + 1$) уже образуют пирамиду, поскольку индексов i и j, удовлетворяющих соотношению $j = 2i$ (или $j = 2i + 1$), просто не существует.

Эти элементы образуют как бы нижний уровень соответствующего двоичного дерева, для них никакой упорядоченности не требуется. Теперь пирамида расширяется влево; каждый раз добавляется и сдвигами ставится в надлежащую позицию новый элемент.

Таблица 2

Построение пирамиды

44	55	12	42)	94	18	06	67
44	55	12)	42	94	18	06	67
44	55)	06	42	94	18	12	67
44)	42	06	55	94	18	12	67
06	42	12	55	94	18	44	67

Каждый раз будем брать последнюю компоненту пирамиды (скажем *x*), прятать верхний элемент пирамиды в освободившемся теперь месте, а *x* сдвигать в нужное место. В табл. 3 приведены необходимые в этом случае *n*–1 шагов.

Таблица 3

Примеры процесса сортировки с помощью *Heapsort*

06	42	12	55	94	18	44	67
12	42	18	55	94	67	44	06
18	42	44	55	94	67	12	06
42	55	44	67	94	18	12	06
44	55	94	67	42	18	12	06
55	67	94	44	42	18	12	06
67	94	55	44	42	18	12	06
94	67	55	44	42	18	12	06

Пример из табл. 3 показывает, что получающийся порядок фактически является обратным. Однако это можно легко исправить.

Алгоритм пирамидальной сортировки на обобщенном Pascal:

```
PROGRAM HS;
VAR I,X,L,N,R:INTEGER;
   A:ARRAY[0..50] OF INTEGER;

PROCEDURE SIFT(L,R: INTEGER);
VAR
 I,J,X: INTEGER;
BEGIN
 I:=L;
 J:=2*L;
 X:=A[L];
 IF (J<R)AND(A[J]<A[J+1]) THEN INC(J);
 WHILE (J<=R)AND(X<A[J]) DO BEGIN
  A[I]:=A[J];
  A[J]:=X;
  I:=J;
  J:=2*J;
  IF (J<R)AND(A[J]<A[J+1]) THEN INC(J);
 END
END;

BEGIN
```

```
 //ввод массива длины n
 // определить левую границу L как позицию среднего элемента плюс один в массиве A;
 // определить правую границу R, равную количеству элементов массива A;
Пока элементы слева есть делай
   уменьши L на единицу;
   вызови процедуру просеивания SIFT для фактических параметров L,N
END;
Пока правая граница больше единицы делай
   поменяй местами A[1] и A[R] элементы;
   уменьши правую границу на единицу;
   вызови процедуру просеивания SIFT для фактических параметров 1,R
END;
//вывод отсортированного массива на экран
END.
```

Доказано, что $T(n) = T_{\text{ср.}}(n) = T_{\text{наих.}}(n) \sim O(n \cdot \log n)$.

3. *Сортировка с помощью разделения (быстрая сортировка Хоара)*

Этот улучшенный метод сортировки основан на обмене. Это самый лучший из всех известных на данный момент методов сортировки массивов. Его производительность столь впечатляюща, что изобретатель Ч. Хоар назвал этот метод *быстрой сортировкой* (*Quicksort*). В *Quicksort* исходят из того соображения, что для достижения наилучшей эффективности сначала лучше производить перестановки на большие расстояния. Предположим, что у нас есть n элементов, расположенных по ключам в обратном порядке. Их можно отсортировать за $n/2$ обменов, сначала поменять местами самый левый с самым правым, а затем последовательно сдвигаться с двух сторон. Это возможно в том случае, когда мы знаем, что порядок действительно обратный. Однако полученный при этом алгоритм может оказаться и неудачным, что, например, происходит в случае n идентичных ключей: для разделения нужно $n/2$ обменов. Этих необязательных обменов можно избежать, если операторы просмотра заменить на такие:

$$\text{while } A[i] \le x \text{ do } i := i + 1;$$

$$\text{while } x \le A[j] \text{ do } j := j - 1$$

В этом случае x не работает как барьер для двух просмотров. В результате просмотры массива со всеми идентичными ключами приведут к переходу через границы массива.

Наша цель – не только провести разделение на части исходного массива элементов, но и отсортировать его. Будем применять процесс разделения к

получившимся двум частям до тех пор, пока каждая из частей не будет состоять из одного-единственного элемента. Эти действия представлены в программе на обобщенном Pascal.

Алгоритм пирамидальной сортировки на обобщенном паскале:

```
PROGRAM QS;
VAR N,I:INTEGER;
   A:ARRAY[0..50] OF INTEGER;

PROCEDURE SORT(L,R: INTEGER);
VAR
 I,J,X,W: INTEGER;
BEGIN
//определить значение переменной I как соответствующее значению левой границе просмотра L;
//определить значение переменной J как соответствующее значению левой границе просмотра R;
//определить значение переменной X, равной значению серединного элемента;
 REPEAT
   Пока A[I] расположено верно относительно X инкрементируй I;
   Пока A[J] расположено верно относительно X декрементируй J;
   Если I «не забежало» за J поменяй местами A[I] и A[J] и сдвинь индексы I и J навстречу друг другу
UNTIL I>J;
 IF L<J THEN SORT(L,J);
 IF I<R THEN SORT(I,R);
END;

BEGIN
 //ввод массива длины n
 //вызови процедуру просеивания SORT для фактических параметров 1,N
 //вывод отсортированного массива
END.
```

Доказано, что $T(n) = T_{\text{ср.}}(n) \sim O(n \cdot \log n)$, $T_{\text{наих.}}(n) \sim O(n^2)$.

Задания лабораторной работы 2:

Задание 1. Реализовать алгоритм сортировки массива с помощью метода Шелла.

Задание 2. Реализовать алгоритм пирамидальной сортировки массива.

Задание 3. Реализовать алгоритм сортировки быстрой сортировки Хоара.

Уточнения к реализации алгоритмов:

1. Реализовать алгоритмы можно на любом языке программирования.

2. В программе необходимо отсортировать по возрастанию элементы заданного неотсортированного массива, состоящего минимум из восьми элементов. После завершения работы программы выходными данными является отсортированный массив. Пример входных и выходных данных приведен в табл. 4.

Таблица 4

Пример входных и выходных данных программ сортировок

INPUT	OUTPUT
44 55 12 42 94 18 06 67	06 12 18 42 44 55 67 94

Результаты лабораторной работы – файлы с кодом программ, наименовать соответственно:

ФамилияИО_номер группы_лаб.2_номер задания (1–3).

Например, «ДолгановВМ_412_лаб.2_1».

Учебное издание

ДОЛГАНОВА Надежда Филипповна
ДОЛГАНОВ Виталий Михайлович
СТАСЬ Андрей Николаевич

Теоретические основы
прикладной математики и информатики:
некоторые методы сортировки массивов

Учебно-методическое пособие

Текстовое электронное издание

Ответственный за выпуск: *Ю.Ю. Афанасьева*
Корректор: *Ю.П. Готфрид*
Технический редактор: *А.И. Лелоюр*

Подписано к использованию: 18.01.2024
Гарнитура Times. Объем издания: 16,74 Mb. Комплектация издания – 1 CD.
Тираж 100 CD. Заказ № 032/ЭУ.

Издательство Томского государственного педагогического университета
634061, г. Томск, ул. Киевская, 60
тел. 8(3822)311-484
E-mail: izdatel@tspu.edu.ru