МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Тольяттинский филиал

Кафедра прикладной математики и информатики

Б. Ф. Мельников, Е. А. Мельникова

АЛГОРИТМЫ СОРТИРОВКИ МАССИВОВ.

СЛОЖНОСТЬ АЛГОРИТМОВ

Утверждено редакционно-издательским советом университета
в качестве учебного пособия

Самара
Издательство «Самарский университет»
2014

УДК 681.142.2
ББК 22.18
М48

Рецензенты : д-р педагог. наук, проф. Ульяновского Государственного университета Г.А. Жаркова,
канд. физ.-мат. наук, доц. Тольяттинского государственного университета Г.А. Тырыгина

Мельников, Б. Ф.

М48 Алгоритмы сортировки массивов. Сложность алгоритмов : учебное пособие / Б. Ф. Мельников, Е. А. Мельникова. – Самара: Изд-во «Самарский университет», 2014. – 40 с.

Пособие содержит учебный материал по двум темам курса: «Способы оценки сложности алгоритмов» и «Массивы данных – задачи поиска и сортировки». Именно на примере задач поиска и сортировки массивов лучше всего видны различные способы оценки сложности (эффективности) алгоритмов. Примеры программ и их фрагментов, приведённые в настоящем пособии, выполнены на языке Pascal и тщательно проверены.

Предназначено для студентов специальности «Бизнес-информатика» для освоения курса «Программирование».

УДК 681.142.2
ББК 22.18

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Данное пособие содержит первые две темы курса «Программирование» – «Способы оценки эффективности алгоритмов» и «Массивы данных – задачи поиска и сортировки». Объединение этих двух *тем* связано с *тем,* что различные способы оценки эффективности алгоритмов лучше всего видны именно на примере задач поиска и сортировки массивов.

Читающие данное методическое пособие должны быть знакомы с основами языком Паскаль. Желательно также, чтобы читатели были знакомы со следующими понятиями теории графов: *граф* (в том числе *ориентированный), дерево* (в том числе *двоичное),* а также с необходимыми для работы с графами и деревьями определениями: *инцидентности, смежности, пути, листьев* и т. п.

Алгоритмы разработаны и записаны в соответствии с принципами структурного программирования – в алгоритмах выделены фрагменты, которые оформлялись в виде отдельных подпрограмм. Элементы объектно-ориентированного программирования в данной первой части пособия не применяются – *однако она написана таким образом, чтобы в последующем сделать переход на ООП «безболезненным».*

Очень небольшое количество комментариев в примерах программ объясняется тем, что фактически все программы прокомментированы в основном тексте.

1. О ЗАДАЧАХ ПОИСКА И СОРТИРОВКИ ДЛЯ МАССИВОВ И ФАЙЛОВ

Задачи поиска и сортировки для массивов и файлов – очень часто встречающиеся задачи при работе с *данными*. Примерами множеств данных могут служить телефонные справочники, оглавления в книгах, списки книг в библиотеках, списки сотрудников организаций и т. д. Вот пример описания типа для информации о некоторой книге в библиотеке:

```
type Книга = record
     Автор: string;
     Название: string;
     ГодИздания: word;
     Тема: (Математика, Биология, ОБЖ);
     Стеллаж: word; {номер стеллажа, где эта книга
     находится в библиотеке}
     Количество: word; {штук в библиотеке}
end;
```

Здесь можно было бы “более детально” организовать почти все из имеющихся полей данных, например, тип поля Автор сделать таким:

```
record
     Фамилия, Имя, Отчество: string [20];
end;
```

однако ни здесь, ни ниже мы не будем заниматься подобной детализацией.

Из таких записей на Паскале может быть создана база данных, например, для описанного выше типа Книга:

```
array [1..N] of Книга;
```

для некоторой (достаточно большой) константы N, или

```
file of Книга;
```

Все поля в приведённом примере можно заменить на другие, удалить, записать новые и т. п. – для дальнейшего конкретный тип используемых данных несущественен.

Когда имеется много данных описанного типа, практически всегда удобно работать с *отсортированными* данными. Отметим, что для разных задач может потребоваться сортировка *по разным полям*. Например, для массива (или файла) данных – элементов типа Книга могут потребоваться такие виды сортировки:

- по отношению «меньше» – по одному из полей *ГодИздания*, *Стеллаж* или *Количество*,
- алфавитное – по одному из полей *Автор* или *Название*

В любом случае можно считать, что имеется некоторое отношение порядка, и именно по этому отношению производятся сортировка и поиск в отсортированном массиве (файле) данных.

Всюду ниже будем для упрощения записи сортировать массивы (файлы) по отношению “меньше” для типа `word`, и поиск в уже отсортированном массиве проводить также для этого отношения. Таким образом, «не ограничивая общности рассуждений», в дальнейшем можно будет работать только с данными следующего типа:

```
type Элемент = record
     Ключ: word;
     Информация: ТипИнформации;
end;
```

Здесь `Ключ` – поле, по которому ведётся сортировка (поиск), а тип `ТипИнформации` – абстракция для *всех остальных* полей типа. Например, для типа `Книга` вместо поля `Ключ` может быть применено поле `ГодИздания`, а поле `Информация` в этом случае обозначает совокупность полей `Автор`, `Название`, `Тема`, `Стеллаж` и `Количество`. Подробнее `ТипИнформации` конкретизировать не будем – приведённые далее программы работают с различными типами данных. Иногда будем называть элемент массива по его ключу (т. е. если `Ключ=51`, то будем говорить “элемент 51”).

Выше были описаны четыре возможные задачи – задачи сортировки и поиска для массивов и файлов данных. При этом имеются в виду *последовательные* файлы (что подчёркивается употреблением записи `file of`): действительно, задачи сортировки (поиска) для файлов *прямого доступа* аналогичны соответствующим задачам для массивов. Задача поиска для последовательных файлов тривиальна: никаких других алгоритмов, кроме последовательного просмотра всех элементов файла, придумать невозможно. Таким образом, из четырёх задач остаются три. Ниже (т.е. в данной части 1) будут рассмотрены две из них – обе задачи для массивов, они значительно проще задачи сортировки файлов, которая будет рассмотрена в одной из следующих частей.

Далее в обеих задачах про массивы мы не будем использовать никаких дополнительных структур данных (например, вспомогательных массивов), кроме, возможно, нескольких переменных простых типов – `word`, `boolean` и т.п., а также *одной* переменной описанного выше типа `Элемент` (эта переменная будет использоваться для обмена двух элементов массива). При этом, однако, алгоритмы сортировки вовсе не усложняются – т.е. они *не сложнее* тех алгоритмов, которые можно было бы создать, применяя вспомогательные массивы. А *сложность* алгоритмов – т.е. различные способы оценки их эффективности – будет рассмотрена в следующем разделе.

2. О СПОСОБАХ ОЦЕНКИ СЛОЖНОСТИ (ЭФФЕКТИВНОСТИ) АЛГОРИТМОВ

Определения *сложности* или *эффективности* алгоритмов, приводимые в разных монографиях ([5, 9, 15] и др.), весьма отличаются друг от друга, поэтому имеет смысл говорить не о строгом определении эффективности, а о различных *подходах* к её определению. А при необходимости в каждом конкретном случае один из подходов можно будет "развить" до строгого математического определения. Ниже приведены альтернативные варианты таких подходов. Альтернативы обозначены буквами (*A,B,C,D*) с нижними индексами, т.е. например, можно создать строгое определение на основе приведённых ниже альтернатив A_2, $B_{1,2}$, C_2 и D_1. Но в любом случае эффективность – некоторая функция, зависящая от размерности задачи (эта функция обычно является возрастающей)[1] Пока будем считать размерность зафиксированной.

Во-первых, исследование эффективности можно проводить:

(A_1) аналитическое;

(A_2) статистическое, на основе специальных тестов. Большинство дальнейших примеров этого раздела основаны на статистической оценке (при этом примеры более наглядны, кроме того, аналитическое исследование не всегда возможно). В следующих разделах, наоборот, будем оценивать эффективность аналитически, однако несколько различных программ для статистических оценок читателям всё же полезно написать самостоятельно.

Во-вторых, измерять эффективность можно:

(B_1) по времени;

(B_2) по объёму занимаемой памяти.

По-видимому, временная оценка эффективности алгоритмов на практике требуется значительно чаще, поэтому ниже в настоящем пособии будем рассматривать именно её.

В [5] можно найти таблицу зависимости максимальной размерности задачи, выполняемой за единицу времени (секунду, минуту, час), от вре-

[1] Таким образом, для строгого определения эффективности необходимо иметь определение размерности. В каждом конкретном случае определение размерности своё. Применительно к рассматриваемым нами задачам сортировки и поиска, размерность задачи – это просто размерность массива, т.е. Граница. Ниже это обозначение будет употребляться в текстах программ и их фрагментов, а в формулах вместо Граница будем писать n .

менной сложности алгоритма (n, n^2, n^2, 2^n или $nlogn$), а также таблицу возможного увеличения максимальной размерности при использовании более быстрого вычислительного устройства. При желании такие таблицы легко построить самостоятельно.

Имеются различные подходы к «единицам измерения» временной эффективности. В первом подходе для этого выбираются «обычные» временные единицы (например, миллисекунды), и для программы, соответствующей рассматриваемому алгоритму, измеряется время работы. Недостатки такого подхода следующие. Во-первых, имеется очень большая зависимость от качественного программирования, т.е. от эффективности *реализации* алгоритма. Во-вторых, даже при оптимальных (или близких к ней) реализациях эффективность зависит от применяемых компьютеров: *сравнительные* результаты работы двух разных алгоритмов могут сильно отличаться на компьютерах различных типов.

($B_{1,2}$) Попробуем избавиться от всех отмеченных недостатков предыдущего подхода, выбирая характерные для нескольких однотипных алгоритмов операции и подсчитывается число применений каждой из них (ср. измерение длины удава «в попугаях»)[3]. Различные способы подсчёта числа применений некоторой операции будут описаны ниже.

И у этого подхода легко заметить недостаток: где гарантия, что нами действительно были выбраны «самые важные» операции? Ведь во время работы алгоритма выполняются и другие операции, и, может быть, время выполнения остальных операций сравнимо со временем выполнения выбранных.

Итак, вряд ли может существовать оптимальный во всех отношениях способ измерения времени выполнения алгоритмов, – конкретный критерий приходится выбирать в каждой задаче свой.

Для любого из отмеченных способов оценки времени можно измерять:

(C_1) усреднённую;

(C_2) худшую

---

[2] Отметим ещё раз, что мы не будем использовать дополнительную память для сортировки массива.

[3] В рассматриваемых в данном пособии примерах в качестве таких операций удобно выбрать сравнение и пересылку элементов, но иногда имеет смысл выбирать “более сложные” операции - подалгоримы. Например, в большинстве алгоритмов сортировки массива можно было бы подсчитывать число обменов (вместо числа пересылок): ведь обмен двух элементов – простейший алгоритм.

эффективность для нескольких измерений. (Далее будем говорить «эффективность в среднем» и «в худшем».) При этом могут быть получены разные сравнительные результаты, один из подобных примеров имеется и в настоящем пособии – см. раздел 5.

Какие же случаи нужно рассматривать для выбора худшего из них или усреднения получаемой эффективности? Здесь тоже имеются по крайней мере два подхода:

(D_1) случайным образом генерируются несколько различных вариантов входных данных;

(D_2) Рассматриваются (или с помощью некоторой комбинаторной процедуры генерируются) все принципиально различные возможные входы[4]. Однако очевидно, что из-за «комбинаторного взрыва» такой способ можно применять только для очень малых размерностей задачи.

Сравнительная эффективность каких-либо двух алгоритмов часто зависит и от размерности задачи, которую мы пока считали зафиксированной. Так, при достаточно малых размерностях обычно лучше работают простые алгоритмы, а при больших размерностях – более сложные, не сразу очевидные алгоритмы. Иллюстрирующие примеры можно легко получить на основе алгоритмов из настоящего пособия[5]. Вследствие этого факта часто говорят о *предельной* эффективности алгоритма как некоторой функции $f(n)$ – в случае, если предел

$$\lim_{n\to\infty} \frac{Q(n)}{f(n)}$$

существует и отличен от 0 и ∞ (здесь n – размерность, а $Q(n)$ – эффективность рассматриваемого алгоритма). Или в других обозначениях: $Q(n)=O(f(n))$.

В заключение отметим, что далее в настоящем пособии будет применяться то определение эффективности алгоритмов, которое может быть записано на основе следующих альтернатив: A_1, $B_{1,2}$, C_2 и D_2.

---

[4] В задачах сортировки массивов в качестве такого множества входов обычно рассматривается множество перестановок чисел от 1 до n.

[5] Например, минимальная размерность, при которой среднее время работы «медленного» алгоритма `ОбменыМ` (раздел 4.3) более чем на 10% превышает время работы “быстрого” алгоритма `QuickSort` (раздел 5.2), равна 8 . А для размерностей, не превышающих 5 , алгоритм `ОбменыМ` работает даже быстрее. Числа 5 и 8 - могут, конечно, немного изменяться в зависимости от конкретного способа проверки, но это непринципиально.

3. АЛГОРИТМЫ ПОИСКА ЭЛЕМЕНТА В ОТСОРТИРОВАННОМ МАССИВЕ

Нами будет написано два варианта поиска в отсортированном массиве Массив (он отсортирован по неубыванию значений полей Ключ). Оба варианта будут оформлены в виде *функций*, возвращающих *номер найденного элемента* (т.е. элемента, ключ которого равен искомому или больше его; нами будет использована переменная с таким именем – ИскомыйКлюч).

Но что делать, если требуемого элемента в массиве нет? если подходящих элементов несколько? На эти вопросы можно отвечать по-разному – получатся различные алгоритмы. Например, в разделе 3.1 будем искать первый по порядку элемент, имеющий ключ, значение которого *больше или равно* значению искомого, а в разделе 3.2 – первый по порядку элемент, имеющий ключ, *больший* искомого (возможны и другие варианты). И в обоих случаях написанные нами функции будут возвращать *0*, если необходимого элемента в массиве нет (например, в разделе 3.2 – если ключи всех элементов массива меньше *или равны* искомому).

3.1 Линейный поиск

Алгоритм очень простой: перебираем все элементы, начиная с первого, и ищем первый подходящий.

```
function ЛинейныйПоиск (Граница,ИскомыйКлюч:word):
word
    var I: word;
    begin
      for I:=1 to Граница do
        if Массив[I].Ключ>=ИскомыйКлюч
           then begin ЛинейныйПоиск:=I; exit; end;
      ЛинейныйПоиск:=0;
end;
```

Ещё раз отметим, что выходом (т.е. возвращаемым значением) написанной нами функции является наименьший из номеров элементов, для которых выполнено условие Ключ>=ИскомыйКлюч (или 0, если таких эле-

ментов не найдено). Аналогично пишутся и другие, похожие варианты линейного поиска: найти в точности «равный» элемент, найти первый «больший» элемент и т.п.

3.2 Дихотомия (поиск делением пополам)

Рассмотрим такую игру: один игрок загадывает натуральное число (например, от 1 до 1024), а второй пытается его угадать с помощью минимального количества вопросов. При этом вопросы могут быть любыми, но есть ограничение: отвечать можно только «да» или «нет».

Алгоритм из предыдущего подраздела аналогичен такому методу угадывания: «это число равно 1 ? это число равно 2 ? это число равно 3 ? ... ” – и так до тех пор, пока удастся угадать.

Понятно, что такой метод угадывания далеко не самый быстрый. Если спросить «больше ли заданное число, чем 512 ?» (512 = 1024/2), то независимо от ответа мы сужаем границы поиска в два раза (т. е. должны искать либо от 1 до 512, либо от 513 до 1024). Так же и дальше; например, если известно, что искомое число лежит в границах от 513 до 1024, то вопрос должен быть таким: «больше ли заданное число, чем 768 ?» (768 = 512 + (1024 – 512)/2).

Поиск в упорядоченном массиве аналогичен описанному методу угадывания. Есть небольшая разница: мы выбираем для сравнения не «средний по значению» элемент (мы же не знаем его индекс), а «средний по номеру». Такой алгоритм поиска называется *дихотомией* (делением пополам). Важно отметить, что здесь мы впервые сталкиваемся с одним из алгоритмов, объединяемых общим названием «разделяй и властвуй» (см. также метод сортировки массива `QuickSort` в разделе 5.2. [6])

Комментарии к функции `Дихотомия`. В процессе работы искомое значение всегда находится на отрезке между элементами с индексами `Левый` и `Правый`. Собственно деление пополам – вычисление индекса `Средний`, равного их полусумме, и формирование нового отрезка для поиска, имеющего в 2 раза меньшую длину (границы нового отрезка – либо `Левый` и `Средний`, либо `Средний+1` и `Правый`).

[6] Ещё один интересный (и при этом несложный) пример удачного применения алгоритма “разделяй и властвуй” (не связанный, однако, с рассматриваемыми здесь темами) - *одновременный* поиск минимума и максимума в массиве. См., например, [5].

```
function Дихотомия (Граница,ИскомыйКлюч: word): word;
    var Левый,Правый,Средний: word;
    begin
      if Массив[Граница].Ключ<=ИскомыйКлюч
        then begin Дихотомия:=0; exit; end;
      Левый:=1; Правый:=Граница;
      while Левый<Правый do begin
        Средний:=(Левый+Правый) div 2; {вот дихотомия!}
        if Массив[Средний].Ключ<=ИскомыйКлюч
         then Левый:=Средний+1
         else Правый:=Средний;
        end;
      Дихотомия:=Правый;
    end;
```

Функция специально написана таким образом, чтобы она возвращала наименьший из номеров элементов, для которых выполнено условие `Ключ>ИскомыйКлюч` (т.е. здесь алгоритм находит первый «больший» элемент, ср. с функцией из предыдущего раздела). Это сделано не столько для демонстрации разных вариантов возврата (также см. предыдущий раздел), сколько для того, чтобы эту функцию можно было применить в качестве вспомогательного алгоритма в разделе 4.1. Отметим также, что оператор `Левый:=Средний+1` нельзя заменить на `Левый:=Средний`, иначе выполнение алгоритма может зациклиться.

Оценка эффективности обоих рассмотренных методов поиска крайне проста. Для линейного поиска получаем эффективность $\sim n$ – как «в среднем», так и «в худшем». А для дихотомии эффективность «в худшем» пропорциональна $log_2 n$ – поскольку можно построить двоичное дерево, содержащее n вершин, глубина которого будет равной $[log_2 n]$ [7]. С помощью несложных вычислений можно убедиться, что эффективность «в среднем» для дихотомии также $\sim log n$. Эти вычисления вряд ли представляют интерес для тем «Поиск» и «Сортировка», поэтому здесь они не приведены.

[7] Отметим, что обычно в литературе по информатике все логарифмы рассматриваются по основанию 2, поэтому основание логарифмов опускается. Есть ещё одно объяснение: для перехода к логарифму по другому основанию (a) достаточно применить умножение на константу $(log_0 2)$.

4. ПРОСТЫЕ АЛГОРИТМЫ СОРТИРОВКИ МАССИВА

Как было отмечено в разделе 1, в алгоритмах сортировки массива мы будем пользоваться только одной “вспомогательной” переменной типа Элемент. Эта переменная используется для обмена двух элементов массива. Поскольку обмен некоторых двух элементов массива многократно используется во всех алгоритмах сортировки, оформим этот небольшой подалгоритм в виде процедуры:

```
procedure Обмен (I,J: word);
    var Эл: Элемент;
    begin
      Эл:=Массив[J];
      Массив^] :=Массив[!] ;
      Массив[I]:=Эл;
end;
```

Иногда будем пользоваться модификацией этого алгоритма: менять элементы только тогда, когда ключ (Ключ) первого параметра больше ключа второго. В программах будет использоваться результат работы этого алгоритма – т.е. действительно ли мы поменяли элементы местами, – поэтому оформим этот подалгоритм в виде такой логической функции:

```
function Упорядочили (I, J: word): boolean;
    begin
      if Массив[I].Ключ>Массив[J].Ключ
        then begin Обмен(I,J); Упорядочили:=true; end
        else Упорядочили:=false;
    end;
```

Удобно выделить в качестве подалгоритма и циклический сдвиг нескольких элементов массива (ср. с процедурой Обмен):

```
procedure Сдвиг (I,J: word);
    {Циклический сдвиг элементов:
    либо I -> I+1 -> I+2 ->...-> J-1 -> J (при I<J),
    либо I -> I-1 -> I-2 ->...-> J+1 -> J (при I>J).
    }
    var Эл: Элемент;
        K: word;
    begin
```

```
  Эл:=Массив[J];
  if J>I
   then for K:=J downto I+1 do Массив[K]:=МассивК-1]
   else for K:=J to I-1 do Массив[K]:=МассивК+1];
  Массив[I]:=Эл;
end;
```

Рассмотренную ранее процедуру обмена двух элементов массива можно было бы также написать как частный случай алгоритма Сдвиг, - однако это, по-видимому, нерационально. Отметим также, что мы только один раз – а именно, при написании процедуры Сдвиг – выясняем, какой из наших двух элементов (I-й или J-й) имеет меньший номер. В дальнейшем же, т.е. *при применении* процедуры Сдвиг, мы будем заботиться только о том, чтобы направление от первого из номеров-параметров ко второму было именно направлением сдвига.

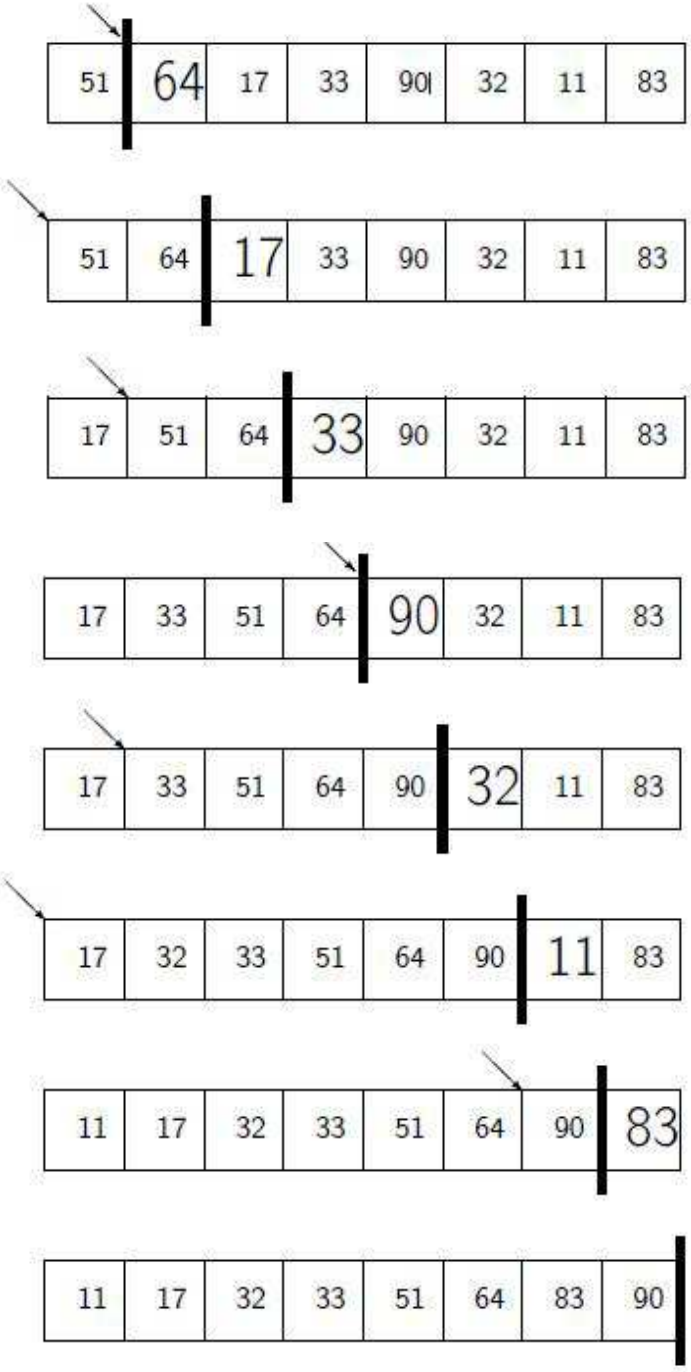
Ниже, при печати текстов программ сортировки, мы не будем специально отмечать то, что описания приведённых трёх алгоритмов обмена и сдвига - Обмен, Упорядочили и Сдвиг – всегда опущены. Отметим, что если несколько различных процедур сортировки собраны в единый модуль, то три упомянутых подалгоритма должны оформляться “наравне” с остальными – т.е. быть доступными для всех других процедур[8].

4.1 Метод прямого включения

Неформально алгоритм этого метода (итерационный процесс) описывается следующим образом: в процессе работы алгоритма все элементы делятся на *уже отсортированную* часть, находящуюся в начале массива и *ещё не отсортированную*, находящуюся в конце. База итерационного процесса: можно считать, что в начале работы эта граница находится между первым и вторым элементами (т. к. массив из одного элемента всегда является отсортированным). Шаг процесса: выбирается первый элемент второй части и переносится на необходимое место в первой части; при этом граница между частями массива сдвигается на один элемент.

[8] Однако в раздел interface модуля их включать, по-видимому, незачем.

Продемонстрируем алгоритм на примере. Будем здесь и всюду ниже на рисунках, поясняющих метод, изображать только ключи элементов (т.е. поля `Ключ` переменных типа `Элемент`). Кроме этого, предполагаем, что в нарисованных горизонтально массивах первый элемент находится слева, последний – справа, а в нарисованных вертикально – первый элемент является верхним. Изменения массива показаны здесь сверху вниз, граница между частями массива – толстая линия, рассматриваемый элемент выделен шрифтом большего размера, место в первой части, куда происходит включение – стрелкой.



Как мы видим, граница между частями массива “уехала” за границу массива – на этом сортировка кончается.

Программа для метода прямого включения несложна, отметим лишь использование в ней двух ранее рассмотренных подалгоритмов – дихотомии (при использовании вместо неё линейного поиска[9] эффективность была бы ещё ниже), а также сдвига.

```
procedure ПрямоеВключение (Граница: word); var I,J:
word;
  Куда: word;
begin
    for I:=2 to Граница do begin
      Куда:=Дихотомия(I-1,Массив[I].Ключ);
      {дихотомией нашли,
       куда переписать рассматриваемый  элемент
      }
      if Куда>0 then Сдвиг(Куда,I);
     { "else" рассматриваемый элемент не надо
        переписывать
     }
    end;
end;
```

Оценим эффективность «в худшем» метода прямого включения. Количество сравнений дихотомии мы оценивали ранее – эта величина для массива из *n* элементов имеет порядок *logn*. Мы применяем дихотомию n–*1* раз, причём для *i*-го применения массив состоит из *i* элементов, поэтому общее число сравнений есть

$$\sum_{i=1}^{n-1} \log i \sim \int_1^n logx\,dx \sim n \log n\,.$$

Для подсчёта наибольшего возможного числа пересылок заметим, что на *i*-м шаге в худшем случае (а именно – когда рассматриваемый элемент пересылается в *1*-ю позицию) будет *i+1* пересылка, т.е. их общее число не превышает

[9] Написанную нами в разделе 3.1 функцию ЛинейныйПоиск при этом пришлось бы немного изменить.

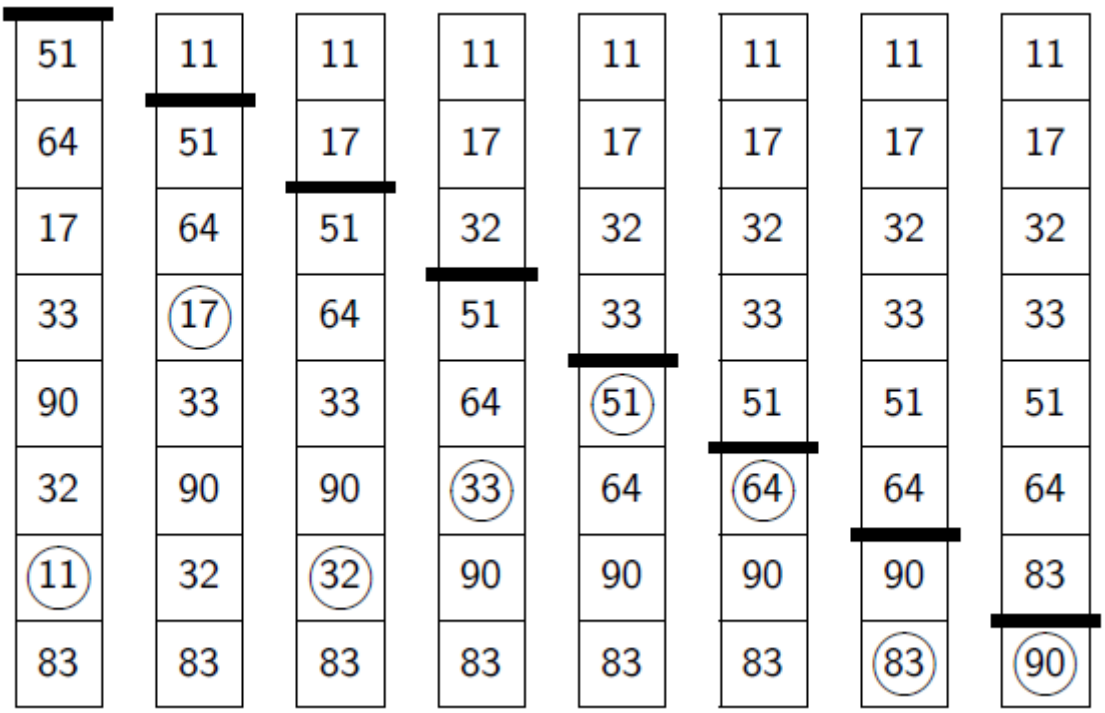
$$\sum_{i=1}^{n-1}(i+1) \sim n^2 .$$

Из последнего заключаем, что для метода прямого включения эффективность “в худшем” есть n^2 .

4.2 Методы прямого выбора

Общее во всех методах прямого выбора – то, что на каждом шаге алгоритмов среди непросмотренной части массива выбирается минимальный (или максимальный) элемент и переносится на первое (последнее) возможное место.

Первым рассмотрим метод *пузырька*, названный так из-за аналогии со всплывающим пузырьком газа в воде.[10] Алгоритм: на *i*-м шаге среди непросмотренных элементов (их *n–i+1*) выбирается минимальный, записывается на *i*-е место массива, при этом промежуточные элементы сдвигаются (всплыл пузырёк):



[10] Из-за этой аналогии будем на рисунках, иллюстрирующих методы прямого выбора, располагать массивы сверху вниз.

При внимательном рассмотрении рисунка (или процедуры сортировки, составленной на основе описанного алгоритма [11])

```
procedure Пузырек (Граница: word);
{==========}
function Минимум (Левый: word): word;
    var I,Num,Key: word;
begin
    Num:=Левый; Key:=Массив[Левый].Ключ;
    {предполагаемый минимум}
    for !:=Левый+1 to Граница do
     if Массив[I].Ключ then begin
       Num:=I; Key:=Массив[I].Ключ;
      {новый предполагаемый минимум}
     end;
    Min:=M;
end;
{==========}
var I: word;
begin
    for I:=1 to Граница-1 do Сдвиг^МинимумЦ));
end;
```

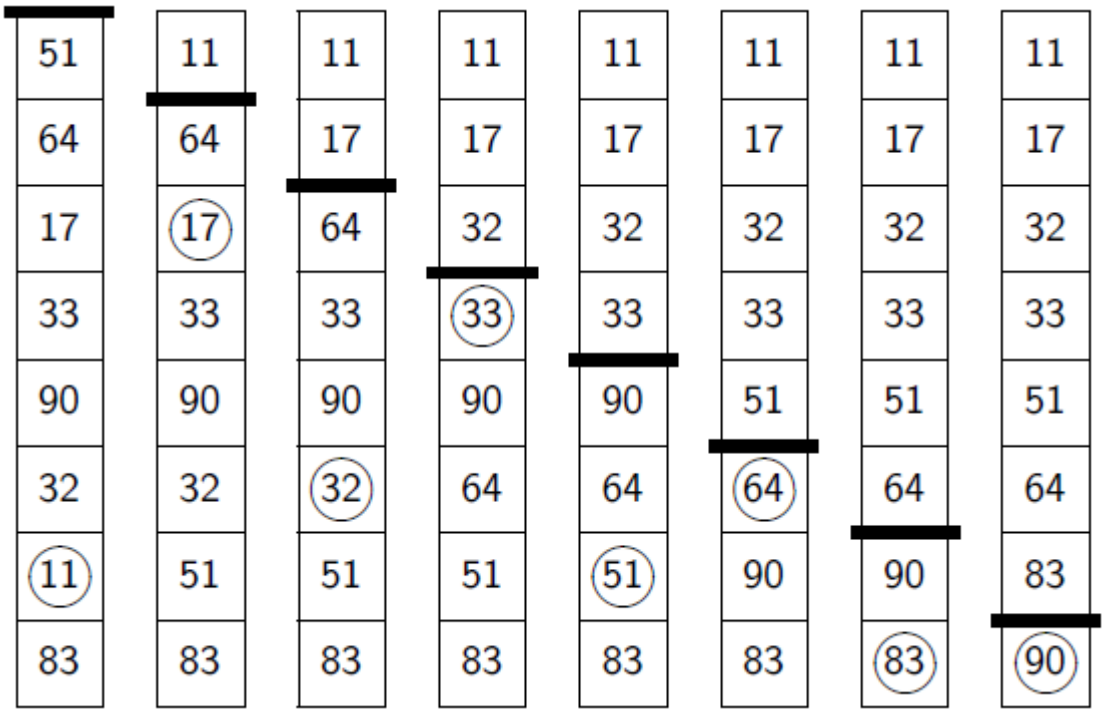
становится непонятным применение подалгоритма `Сдвиг`: ведь у нас ещё нет никакой информации об *относительных* значениях элементов с номерами от `I+1` до `Минимум(I)`. Поэтому подалгоритм `Сдвиг` можно заменить на более быстрый `Обмен`, при этом процедура сортировки становится такой[12]:

```
procedure ПузырекМ (Граница: word);
    var I: word;
    begin
      for I:=1 to Граница-1 do Обмен(I,Минимум(I));
    end;
```

Вот соответствующий этой процедуре рисунок:

[11] Дочерняя функция Минимум процедуры Пузырек возвращает номер того элемента между параметрами Левый и Граница, который имеет минимальный ключ.

[12] Описание дочерней функции Минимум здесь опущено.



Эффективность метода пузырька посчитаем для последнего варианта. Немного переформулируем ранее отмеченный факт: перед `i`-м шагом алгоритма у нас ещё нет никакой информации об *относительных* значениях элементов с номерами от `i` до `n`. Из этого можно сделать такое заключение: на `i`-м шаге, вообще говоря, нельзя найти минимальный элемент, не сделав `n–i` сравнений. Таким образом, в худшем случае общее число сравнений пропорционально

$$\sum_{i=1}^{n-1}(n-i) \sim n^2 .$$

При подсчёте общего числа пересылок естественно учитывать не только те из них, которые вызываются подпрограммой `Обмен`, но и необходимые для работы функции `Минимум`[13]. На любом шаге возможны 3 пересылки «первого рода», а максимально возможное число пересылок «второго рода» зависит от номера шага алгоритма: на `i`-м шаге их число равно `2*(n–i+1)`. Таким образом, для всего алгоритма максимально возможное число пересылок есть

[13] Отметим однако, что *при выбранном нами оформлении* методов сортировки, пересылки, вызываемые подпрограммой `Обмен` и имеющие дело с данными типа `Элемент`, выполняются, по-видимому, дольше вызываемых функцией `Минимум` и работающих с данными типа `word`.

$$\sum_{i=1}^{n-1}(3+2*(n-i+1)) \sim n^2,$$

т.е. и для метода пузырька мы получаем такую же оценку эффективности: n^2.

Далее рассмотрим один из вариантов т. н. *шейкерной* сортировки, также являющейся развитием метода пузырька[14]. Сначала отметим следующий факт: во всех рассмотренных ранее алгоритмах возможны ситуации, когда наш массив оказался отсортированным *до* окончания работы алгоритма. Интуитивно ясно, что многократные проверки такой ситуации (для досрочного окончания работы алгоритмов в случае её обнаружения) приведёт к существенному снижению эффективности. Однако возможны ситуации, когда заранее известно, что входной массив «почти отсортирован» [15]. Рассмотрим два примера.

Во-первых, применяя метод пузырька к такому массиву:

```
33 11 51 64 17 83 90 32
```

увидим, что три «пузырька» – `11, 17` и `32` – полностью отсортируют массив. Однако массив

```
90 11 17 32 33 51 64 83,
```

который тем более хочется назвать «почти отсортированным» – требует для сортировки максимально возможное (для размерности `8`) число «пузырьков» 7 .

Последний пример подсказывает такой алгоритм: чередовать перенос наверх минимального (из оставшихся) элемента и перенос вниз максимального («топорик»?). При таком алгоритме сортировка обоих «почти отсортированных» массивов завершается менее чем за 7 шагов – это, по-видимому, легко понять и без поясняющих рисунков. Процедура получается такая:

```
procedure Шейкер (Граница: word);
 var Вверх: boolean;
     Левый,Правый: word;
 begin
```

[14] По английски shake – трясти, a shaker – прибор для перемешивания коктейлей. “Развивать” мы будем первый вариант метода пузырька, т. е. алгоритм подпрограммы `Пузырек` (а не `ПузырекМ`).

[15] Мы не будем определять критерий “почти отсортированности”, ограничимся лишь интуитивными размышлениями.

```
    Вверх:=true;
    Левый:=1; Правый:=Граница;
    while not ПоПорядку(Левый,Правый) do begin
     if Вверх then begin
          Сдвиг(Левый,Минимум(Левый,Правый));
          inc(Левый);
        end
     else begin
          Сдвиг(Правый,Максимум(Левый,Правый));
          dec(Правый);
         end;
    Вверх:=not Вверх;
    end;
end;
```

Функция ПоПорядку в принципе может быть использована и в других алгоритмах, поэтому опишем её отдельно:

```
function ПоПорядку (Левый,Правый: word): boolean;
    var I: word;
    begin
      ПоПорядку:=false; {пока}
      for I:=Левый to Правый-1 do
        if Массив[I].Ключ>Массив[I+1].Ключ then exit;
      ПоПорядку:=true;
end;
```

К написанному необходимы следующие пояснения. Опущенные подал- горитмы Минимум и Максимум – функции, возвращающие номера минимального и максимального элементов среди находящихся между их параметрами (аналогичная функция была написана нами ранее). Условие

Левый<Правый в цикле while основной процедуры можно не проверять потому, что это делается при работе функции ПоПорядку. Переменная Вверх используется в качестве переключателя «пузырёк/топорик», а подалгоритм ПоПорядку применяется только для «внутренних» элементов массива (т.е. тех, которые расположены *между* двумя отсортированными частями – этого достаточно). Ещё раз отметим применение подалгоритма Сдвиг (при применении вместо него подалгоритма Обмен терялся бы смысл всего метода).

Оценку эффективности метода шейкера проводить не будем по следующим причинам. Во-первых, очевидно, что эффективность «в худшем» совпадает с соответствующей величиной метода пузырька, а нам уже известно, что даже улучшенная процедура `ПузырекМ` имеет эффективность $\sim n^2$. Во-вторых, посчитать эффективность «в среднем» для удачных («почти отсортированных») вариантов расположения элементов (где и можно было бы ожидать меньшую величину) вряд ли вообще возможно – мы применяли лишь интуитивные размышления об удачных вариантах расположения элементов. В-третьих, в следующем разделе будет написана несколько более эффективная «процедура-гибрид» методов шейкера и последовательных обменов.

4.3 Методы последовательных обменов

Рассмотренный нами в предыдущем подразделе алгоритм `ПоПорядку` наводит на мысль *не только* сравнивать порядок расположения соседних элементов массива, *но и* переупорядочивать их в случае неверного расположения. Таким образом, возникает следующий цикл [16]:

```
for I:=1 to Граница-1 do Упорядочили(I,I+1);
```

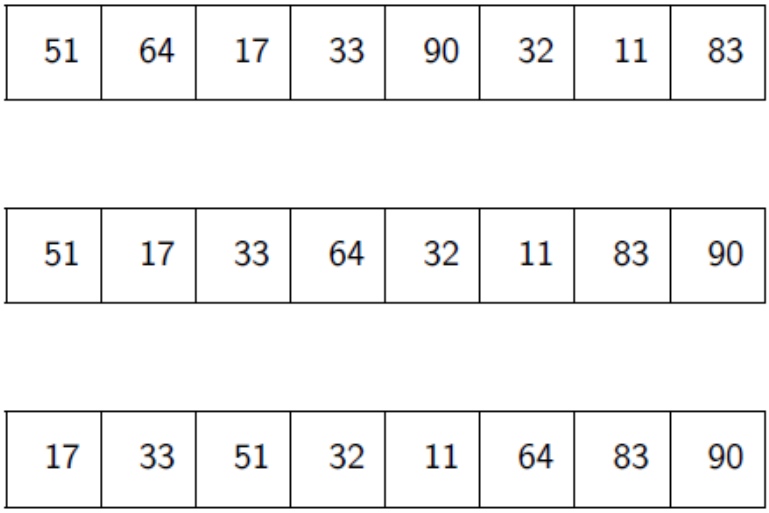
(этот цикл можно назвать подалгоритмом). Подалгоритм «уменьшает энтропию» массива, поэтому за некоторое число его применений массив станет отсортированным[17]. Но – чему равно необходимое число применений последнего подалгоритма? Рассматривая начало его работы, мы убеждаемся что после каждого очередного прохода по крайней мере один элемент – максимальный из оставшихся – попадает на своё место, причём на следующих проходах этот элемент сдвигаться уже не будет.

Отсюда получаем следующие два вывода: во-первых, для массива размерности n подалгоритм достаточно применить $n-1$ раз; во-вторых, его можно изменить, каждый раз заканчивая цикл на элементе с номером на 1 меньше, чем в предыдущем случае.

---

[16] В нём функция Упорядочили применяется как процедура - в смысле расширенного синтаксиса языка Borland-Pascal.

[17] Последнее предложение можно легко переформулировать, чтобы оно стало, во-первых, строгим определением (например, энтропия упорядоченного массива полагается равной 0), во-вторых, формулировкой алгоритма, и, в-третьих, доказательством его сходимости.



Будем называть описанный алгоритм методом *последовательных обменов*, для него получается следующая процедура сортировки:

```
procedure Обмены (Граница: word);
    var I,K: word;
begin
      for K:=1 to Граница-1 do
      for I:=1 to Граница-K do Упорядочили(I,I+1);
end;
```

Заметим, что после очередного прохода на своём месте может оказаться не только максимальный из оставшихся элементов, но и ещё несколько (сейчас нас интересуют только следующие по «старшинству»). Пример такой ситуации нами уже был рассмотрен: на последнем рисунке после первого прохода на своём месте оказался не только элемент (с ключом) 90, но и 83. Это обстоятельство подсказывает такую модификацию алгоритма последовательностных обменов:

```
procedure ОбменыM (Граница: word);
    var I,M,maxОбмен: word;
begin
      maxОбмен:=Граница-1;
      while max0бмен>0 do begin
        M:=1;
        for I:=1 to max0бмен do
         if Упорядочили(I,I+1) then M:=I;
        maxОбмен:=M-1;
      end;
end;
```

Здесь во время каждого прохода мы запоминаем максимальный из номеров элементов, обмененных со следующими за ними, получая границу, отделяющую заведомо упорядоченную часть массива (перед очередным проходом эта граница имеет значение max0обмен+1).

Последняя процедура напоминает такой вариант сортировки, часто встречающийся в учебниках и задачниках по информатике:

```
procedure ОбменыП (Граница: word);
    var 1,Проход: word;
        Поменяли: boolean;
begin
  repeat
    Поменяли:=false;
    Проход:=1;
    for I:=1 to Граница-Проход do begin
        Поменяли:=Упорядочили(I,I+1) or Поменяли;
        inc(Проход);
    end;
  until not Поменяли;
end;
```

Вместо Граница-Проход может “для простоты” быть Граница-1, и переменная Проход вообще не использоваться. Отметим порядок логических выражений в дизъюнкции – при применении иного порядка процедура перестаёт работать, т.к. при этом не гарантируется просмотр всей оставшейся части массива, и, следовательно, постановка на место максимального элемента.

Несмотря на очевидные преимущества процедуры ОбменыМ по сравнению с Обмены и ОбменыП, понятно, что эти преимущества влияют только на эффективность «в среднем», нами не считаемую. Эффективность же «в худшем» у всех трёх модификаций метода последовательных обменов одинакова; это лучше всего видно на примере «инверсного» массива:

```
90 83 64 51 33 32 17 11
```

Попытки закончить сортировку раньше времени здесь успеха не приносят, причём и число сравнений, и число пересылок пропорциональны

$$\sum_{i=1}^{n-1} i \sim n^2$$

(лучше всего последнюю формулу объясняет вариант Обмены).

А теперь вспомним ещё раз про метод шейкера. Оказывается, он имеет много общего с только что написанной нами процедурой – ведь в функции ПоПорядку мы тоже сравнивали соседние элементы массива. Чтобы результаты этих сравнений не пропадали, попробуем менять элементы в случае неудачного сравнения. А чтобы было совсем похоже на метод шейкера, будем каждый нечётный раз двигаться слева направо (ставя при этом на место максимальный элемент, заодно “уменьшая энтропию” оставшейся части массива), а каждый чётный раз – справа налево (на место ставится минимальный элемент). Получается такая процедура:

```
procedure ШейкерМ (Граница:word);
    var I,M,minОбмен,maxОбмен: word;
        Вверх: boolean;
begin
    Вверх:=false;
    minОбмен:=1; maxОбмен:=Граница-1;
    while minОбмен<=maxОбмен do
      if Вверх then begin
        M:=Граница;
        for I:=maxОбмен downto minОбмен do
          if Упорядочили(I,I+1) then M:=I;
        minОбмен:=M+I;
        Вверх:=false;
      end
      else begin M:=1;
        for I:=minОбмен to maxОбмен do
          if Упорядочили(I,I+1) then M:=I;
        maxОбмен:=M-1;
        Вверх:=true;
      end;
end;
```

являющаяся “гибридом” методов шейкера и последовательных обменов. Очевидно, что эффективность процедуры ШейкерМ совпадает с эффективностью каждого из методов последовательных обменов.

Таким образом, можно сделать следующий вывод: эффективность каждого из простых методов сортировки массива ~ n^2 .

5. СЛОЖНЫЕ АЛГОРИТМЫ СОРТИРОВКИ МАССИВА

5.1 Сортировка с помощью двоичного дерева

В отличие от разобранных выше методов сортировки, описание рассматриваемой в данном разделе сортировки *с помощью двоичного дерева* (поскольку русское название этого метода сортировки не устоялось, будем далее называть её английским сокращением – *HeapSort*) состоит из двух частей – собственно алгоритма и его реализации[18]. Это связано с тем, что уже сейчас может возникнуть такой вопрос: какое отношение к сортировке массива имеет двоичное дерево? А если читающие данное методическое пособие уже знакомы с динамическими структурами данных, создаваемыми с помощью ссылочных переменных, то вопрос может быть таким: как связаны с массивами двоичные деревья (являющиеся одним из примеров динамических структур данных для представления множеств)? В данном разделе есть ответы на эти вопросы.

Сначала рассмотрим *алгоритм* сортировки. Будем называть *двоичными* такие деревья, у которых каждая из вершин имеет *не более* 2 *потомков*[19]. Ниже будем рассматривать только т.н. *правильные* двоичные деревья – такие, у которых максимально возможное число уровней заполнено полностью, а у не до конца заполненного уровня все имеющиеся вершины (листья) расположены слева. Очевидно, что для заданного n существует единственное (непомеченное) правильное двоичное дерево, содержащее ровно n вершин.

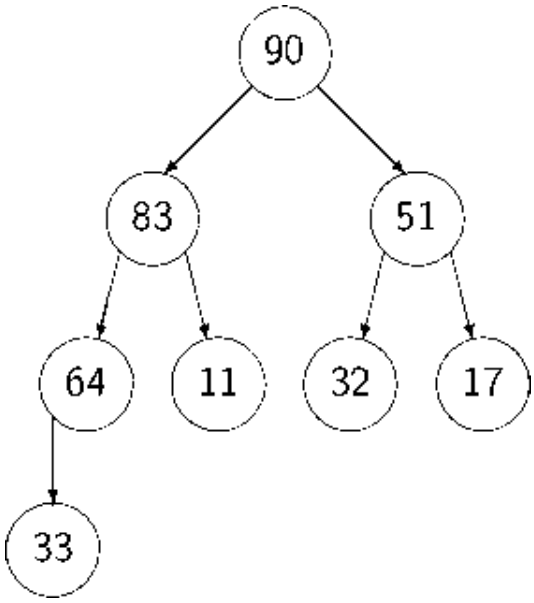
Представим себе сортируемые элементы массива расположенными в вершинах (как листьях, так и нелистьях) правильного двоичного дерева[20], причём так, что *число, расположенное в любой вершине, не меньше, чем расположенное в любой из дочерних*. Будем называть такие правильные деревья *правильно заполненными*. Существует много способов «правильно

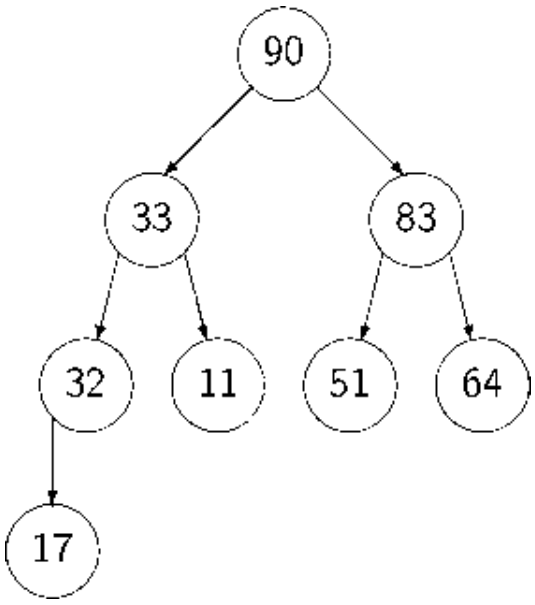
[18] В рассмотренных ранее методах сортировки реализация алгоритмов не отличалась сложностью, поэтому обычно были приведены только тексты программ.

[19] Иногда двоичными называют деревья, у которых каждая из вершин имеет либо 0 потомков (такие вершины называются *листьями*), либо 2 (но не 1).

[20] Или с точки зрения теории графов: вершины дерева *помечены* числами – элементами сортируемого массива.

заполнить» вершины дерева “нашими” 8 элементами [21]; два из этих способов см. на приведённых рисунках.





Отметим, что для любого из таких расположений каждая вершина помечена максимумом определяемого ею поддерева, и, в частности, корень обязательно помечен максимальным из элементов массива.

Если бы мы умели строить подобное двоичное дерево, то алгоритм сортировки можно было бы записать таким образом:

1. построить дерево;
2. «забрать» корень – максимальный элемент, и перенести его в начало отсортированной части массива;
3. если ещё остались элементы, то перестроить дерево (так, чтобы полученное дерево подчинялось ранее сформулированным правилам);
4. перейти на шаг 2.

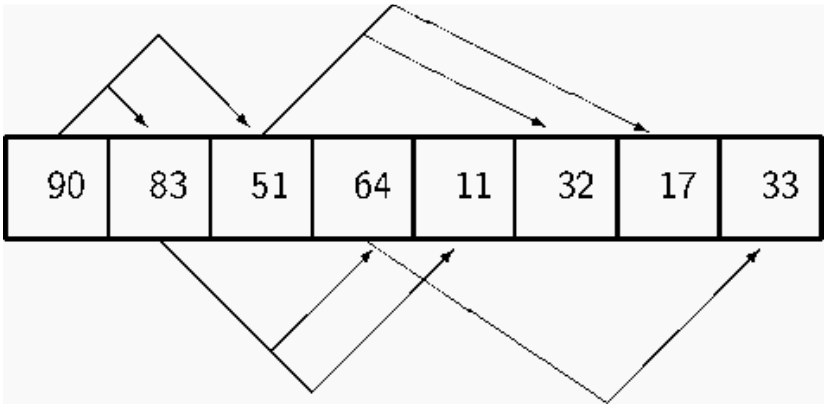
Мы написали только “заготовку” для алгоритма сортировки HeapSort поскольку ещё остались невыясненными многие вопросы: «как связано двоичное дерево с массивом?»[22], «как построить дерево?», «как его изменять при переносе элемента в отсортированную часть?» и др.

Ответим на главный из этих вопросов – как представить наше двоичное дерево в виде массива. Будем предполагать, что вершина дерева соответствует элементу массива, причём корнем является элемент с номером

[21] Задача по комбинаторике: сколько существует таких способов для N различных чисел?

[22] Согласно материалу раздела 1, мы не должны использовать никаких дополнительных структур данных, кроме переменных типа `word`.

1, а у вершины-элемента с номером `i` потомками являются элементы массива с номерами `2*i` и `2*i+1` (если такие существуют). Нарисуем одно из рассмотренных нами правильных деревьев таким образом:

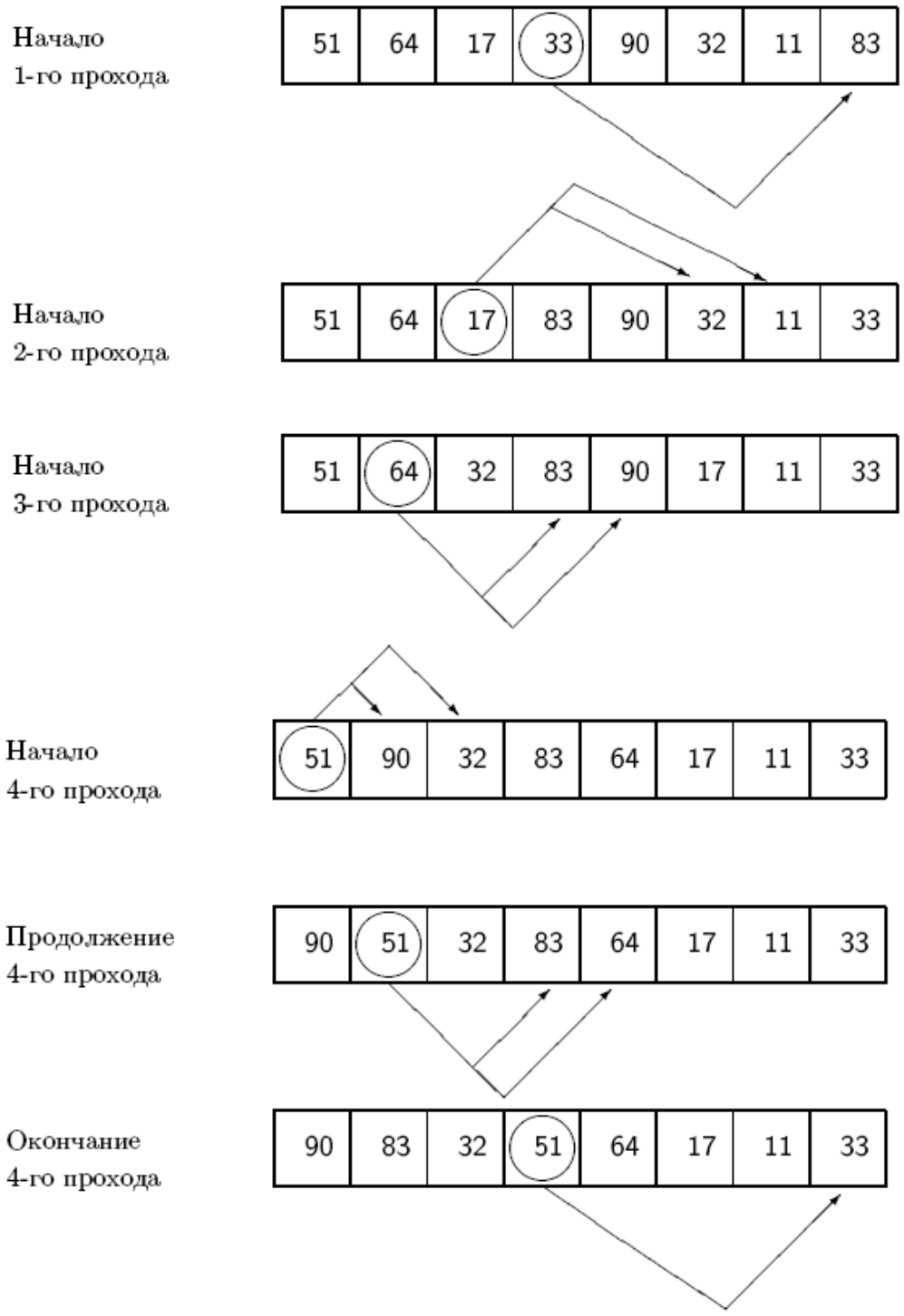


Вопросы всё же остаются: «как построить правильно заполненное дерево?», «как его изменять?».

Сначала – построим. Будем рассматривать вершины дерева (т.е. элементы массива) с конца, *перестраивая* при этом поддерево, определяемое рассматриваемой вершиной (корнем), *если* это поддерево *не является правильно заполненным*. Отметим, что поддеревья, определяемые листьями общего дерева, заведомо правильно заполненные (т.к. состоят из единственной вершины-корня, не имеющей потомков), поэтому будем перебирать с конца вершины-нелистья. Несложно убедиться, что независимо от чётности `n` – числа вершин (правильного) дерева, *не*листьями являются вершины с номерами от `1` до `[n/2]`. Итак, опишем подалгоритм *перестройки* поддерева. Предполагаем («предположение индукции»), что все *собственные* поддеревья рассматриваемого поддерева уже правильно заполнены («базой индукции» при этом являются листья).

Выберем первую вершину поддерева (корень) в качестве текущей. Рассмотрим прямые потомки текущей вершины (дочерние вершины; их не более `2`). Если значение текущей вершины не меньше, чем значение максимальной из дочерних, то процесс обработки поддерева закончен (т.к. по предположению индукции поддеревья дочерних вершин – правильно заполненные). В противном случае поменяем местами значения текущей вершины и максимальной из дочерних, после чего эту вершину (куда переместилось значение из текущей) объявим новой текущей. Корректность подалгоритма перестройки доказывается тем, что после описанного обмена *только* поддерево с корнем в новой текущей вершине может не быть пра-

вильно заполненным[23]. Рассмотрим работу подалгоритма на старом примере. “Просеиваемое” число на рисунках обведено кружком, а стрелками показаны прямые потомки вершины, в которой оно расположено. Каждое применение подалгоритма названо проходом.



[23] Отсюда же следует, что подалгоритм перестройки поддерева легко реализуется *не* рекурсивной процедурой, несмотря на то, что двоичное дерево - рекурсивная структура данных. Этот подалгоритм часто называют процедурой *просеивания* элемента через дерево (по-английски – Sift).

После приведённого обсуждения процедура Sift сложности не представляет:

```
procedure Sift (КореньПоддерева,ГраницаПоддерева: word);
    var I,J,J1,J2: word;
begin
    I:=КореньПоддерева;
    repeat
      J:=I;
      {вычисляем номера дочерних вершин для J-й}
      J1:=2*J; if J1>ГраницаПоддерева then exit;
      J2:=J1+1;
      if (J2>Lim) or (Массив[J1].Ключ>Массив[J2].Ключ)
        then I:=J1
        else I:=J2; until not Упорядочили(I,J);
end;
```

Отметим, что функция Упорядочили используется здесь не совсем обычным способом – для “обратного” обмена, т.к. при этом, в отличие от предыдущих использований, значение первого параметра *больше* значения второго: I>J. Пока не понятен смысл введения второго параметра (т.е. ГраницаПоддерева – ведь Sift, по-видимому, будет использована как «внутренняя» процедура алгоритма сортировки), – но этот момент прояснится в дальнейшем.

Таким образом, для (первоначального) построения правильно заполненного дерева необходимо выполнить следующее:

```
for I:=(Граница div 2) downto 1 do Sift(I,Граница);
```

(для примера на рисунке этот цикл выполняет все 4 прохода).

Итак, нами выполнена первая часть алгоритма – построение дерева[24]. Перенесём максимальный элемент в конец массива (на своё место), и далее будем рассматривать дерево (массив), содержащие на 1 вершину (элемент) меньше. В начало массива (т.е. в корень дерева) при этом обмене записы-

[24] Отметим, что мы специально строили дерево с корнем «слева», а в принципе можно было и наоборот. А ещё можно было пользоваться другим определением правильно заполненного дерева: заменить в данном выше определении слова «не меньше» на «не больше». По аналогии с методом прямого выбора может показаться, что так было бы удобнее: после построения правильно заполненного дерева минимальный элемент оказывался бы на своём месте – в начале массива. Однако скоро будет понятно, что из 4 возможных вариантов нами выбран наилучший.

вается число из конца массива. Это – не обязательно минимальный элемент массива, но, поскольку последний элемент массива есть лист дерева, полученное после такой перестановки дерево – скорее всего не будет правильно заполненным.

Но: где нарушается определение правильной заполненности? *Только в корне!* А с похожей ситуацией мы уже встречались – *на последнем проходе при первоначальном построении дерева,* и умеем её обрабатывать. (Теперь становится понятным смысл введения второго параметра процедуры `Sift` – `ГраницаПоддерева`.) В итоге мы получаем следующую процедуру сортировки методом `HeapSort` (дочерняя процедура `Sift` в тексте опущена):

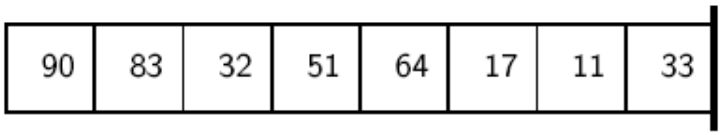
```
procedure HeapSort (Граница: word);
    var I: word;
begin
    for !:=(Граница div 2) downto 1 do
    Sift(I,Граница); for !:=Граница downto 2 do begin
      Обмен(1,I);
      Sift (1,I-1);
    end;
end;
```

Отметим, что при наличии в массиве нескольких одинаковых элементов приведённый алгоритм может содержать лишние действия, поэтому, может быть, второй цикл лучше записать таким образом:

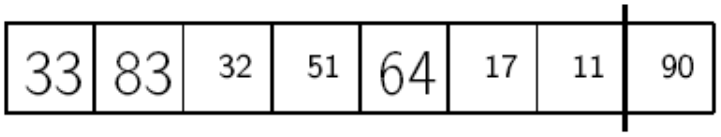
```
for I:=Граница downto 2 do
    if Упорядочили(1,I) then Sift(1,I-1);
```

Рассмотрим второй цикл на старом примере (начнём с того момента, когда мы правильно заполнили дерево). На рисунке показаны границы между отсортированной и неотсортированной частями массивов (отсортированная часть- справа), а работа процедуры Sift показана «конспективно» (участвующие в «просеивании» элементы выделены):

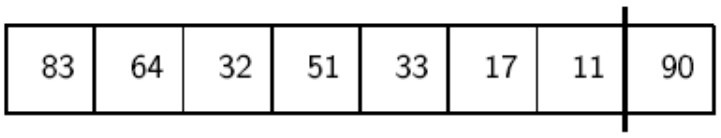
Стартовое правильно заполненное дерево



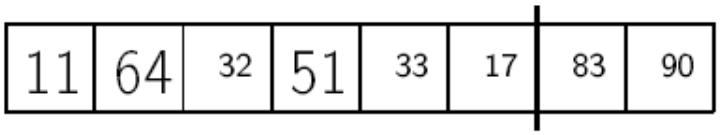
Просеиваем элемент 33



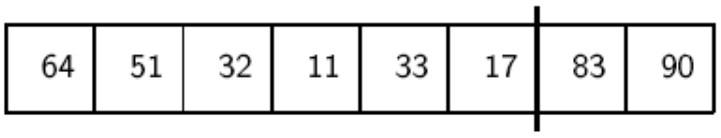
Второе правильно заполненное дерево



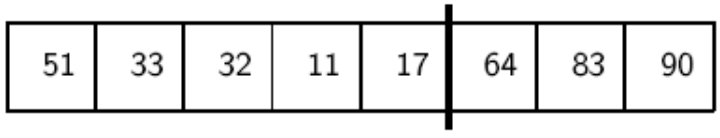
Просеиваем элемент 11



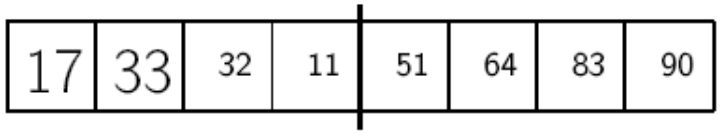
Третье правильно заполненное дерево



Четвёртое правильно заполненное дерево



Опять просеиваем элемент 17



И так далее...

Оценим эффективность метода `HeapSort`. Отметим, что вовсе не очевидно, что такой сложный алгоритм сортировки даёт хорошие результаты. Однако из очень простых рассуждений следует, что эффективность метода (как число сравнений, так и число обменов – в этом алгоритме будем оценивать их одновременно) пропорциональна *n*logn*. Действительно, число операций, необходимых для «просеивания элемента через правильное дерево, содержащее *к* элементов, пропорционально глубине этого дерева, т.е. *logк*. Таким образом, число операций и для первой, и для второй частей алгоритма пропорционально

$$\sum_{i=1}^{n/2} \log i \sim n \log n \,.$$

В итоге получаем эффективность метода *~n*logn*, что лучше, чем у любого из методов, рассмотренных ранее.

5.2 «Быстрая» сортировка

Как и для предыдущего метода, будем иногда употреблять английское сокращение названия – `QuickSort`.

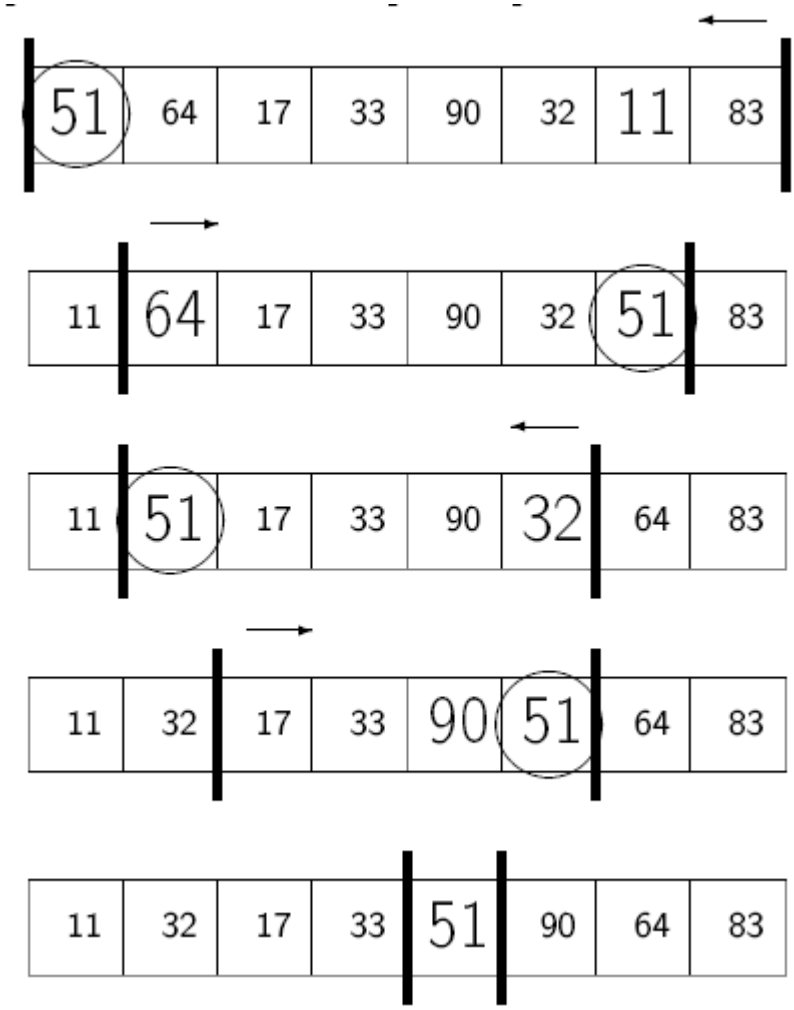
Аналогично дихотомии, `QuickSort` тоже является одним из алгоритмов группы «разделяй и властвуй». При поиске методом дихотомии мы делили рассматриваемый массив (примерно) пополам, чтобы искать нужный элемент только в одной половине. Нельзя ли здесь осуществить аналогичную идею –*поставить* элемент с номером $\left[\frac{n+1}{2}\right]$ *на место*, слева от него записать меньшие элементы, справа – большие, после чего сортировать две половины массива независимо друг от друга? В принципе это возможно, но алгоритмы поиска элемента с номером (т.н. медианы) сложны. Однако существуют достаточно простые алгоритмы постановки на своё место произвольно выбранного элемента массива (например, первого по счёту), причём такой постановки, что в полученном массиве все элементы с меньшими номерами не больше рассматриваемого, и наоборот, все элементы с большими номерами не меньше рассматриваемого. Один из таких алгоритмов будет описан ниже.

Поставив *таким образом* выбранный элемент на место, будем сортировать две части массива независимо друг от друга[25]. И уже сейчас мы можем сказать, что, к сожалению, эффективность «в худшем» рассматриваемого метода сортировки пропорциональна n^2 (как у простых методов): действительно, если массив в самом начале работы *уже* упорядочен, то работа рассматриваемого алгоритма практически не отличается от работы алгоритма прямого выбора. А название `QuickSort` было дано методу за хорошую эффективность «в среднем» (подробнее см. ниже).

Итак, опишем алгоритм такой постановки на место выбранного элемента, что в полученном массиве все элементы с меньшими номерами не больше рассматриваемого, и наоборот. Выберем первый элемент массива; предположим, что он *уже* находится на своём месте. Чтобы это подтвердить (или опровергнуть), необходимо, вообще говоря, сравнить его со всеми оставшимися элементами массива; будем перебирать их «справа налево» (т.е. от больших номеров к меньшим). Если при этом найдётся “неправильный” элемент (т.е. не удовлетворяющий сделанному предположению о том, что все элементы справа от первого больше него), то придётся поменять местами этот элемент и первый. *Основная идея алгоритма:* будем предполагать, что рассматриваемый нами элемент (бывший первый, ниже – «главный») *теперь* находится на своём месте (т.е. больше всех элементов, стоящих слева от него). Заметим, что для проверки этого условия нужно сравнить его только с элементами, стоящими *между* его новым и старым положениями в массиве. Продолжая этот процесс (т.е. после каждого обмена предполагая, что главный элемент уже находится на своём месте), мы каждый раз уменьшаем число элементов, необходимых для проверки сделанного предположения (это число уменьшается по крайней мере на 1), т.о. описанный алгоритм постановки элемента на место конечен.

Продемонстрируем работу алгоритма на примере. Главный элемент (`51`) всюду выделен крупным шрифтом и обведён в кружок. Направление сравнения показано стрелкой. Элемент, с которым обменивается главный, также выделен крупным шрифтом. Как обычно, жирными линиями показаны границы между частями массива: между двумя просмотренными и одной непросмотренной.

[п] Заранее отметим, что нами впервые будет использована *рекурсивная* процедура: ведь сортировать часть массива мы будем по тому же самому алгоритму, что и весь массив. Подробнее рекурсивные алгоритмы будут рассмотрены в части 2 настоящего пособия.



Итак, элемент 51 находится на своём месте, причём слева от него стоят меньшие элементы, а справа – большие.

Теперь несложно написать процедуру для алгоритма QuickSort. Ещё раз отметим, что она содержит рекурсивные (само)вызовы.

```
procedure QuickSort (Левый,Правый: word);
    label 10,20,30;
begin
    if Левый>=Правый then exit;
    Л:=Левый; П:=Правый;
    10: while Л<П do
         if Упорядочили(Л,П)
            then begin inc(Л); goto 20;  end
            else dec(П);
    goto 30;
    20: while Л<П do
         if Упорядочили(Л,П)
```

```
            then begin dec(П); goto 10; end
            else тс(Л) ;
    30: QuickSort(Левый,Л-1); QuickSort(П+1, Правый);
end;
```

Несмотря на неоднократное использование оператора `goto`, написанная нами процедура, по-видимому, удовлетворяет основным требованиям структурного программирования. Действительно, в тексте явно выделены 3 подзадачи – движение налево, движение направо и рекурсивный вызов. Если же обилие `goto` покажется нежелательным, то приведённый далее альтернативный вариант[26] должен удовлетворить самых взыскательных приверженцев канонов структурного программирования:

```
procedure QuickSort (Левый,Правый: word);
    var Л,П: word;
        Направо,У: boolean;
begin
    if Левый>=Правый then exit;
    Л:=Левый; П:=Правый;
    Направо:=true;
    repeat
      У:=Упорядочили(Л,П);
      if У=Направо then inc^) else dec(n);
      if У then Направо:=not Направо;
    until Л>=П;
    QuickSort (Левый,Л-1); QuickSort(П+1,Правый);
end;
```

Для удобства желательно написать следующую процедуру `Sort` глобальную по отношению к `QuickSort`:

```
procedure Sort (Граница: word);
begin
    QuickSort(1,Граница);
end;
```

Выше было отмечено, что эффективность «в худшем» у метода `QuickSort` не лучше, чем у простых методов сортировки, и преимущест-

[26] Он построен на основе предыдущего с помощью двух вспомогательных логических переменных: Направо, показывающей направление движения по массиву, и У, показывающей, был ли переставлен рассматриваемый элемент.

во рассмотренного метода можно увидеть только «в среднем». Однако точно посчитать эффективность «в среднем» довольно сложно (это тоже было замечено ранее, в разделе 2), но на помощь приходит следующее соображение. Выбранный элемент массива может с равными вероятностями оказаться *1-м*, *2-м*, …, *n-м*, поэтому если знать количество операций (сравнений и/или пересылок), необходимых для обработки массивов размерностей $1, 2, \ldots, n-1$ (пусть для размерности i это число есть C_i), то несложно найти усреднённое число операций для массива размерности n:

$$C_n = \frac{(\sum_{i=1}^{n}(C_{i-1} + C_{n-i}))}{n} + n - 1\,.$$

Отметим, что в последней формуле учитывается и число операций (также – сравнений и/или пересылок) необходимых для реализации описанного выше подалгоритма постановки выбранного элемента на своё место: это – слагаемое *n-1* .

Приведённая формула хорошо оценивает эффективность метода (читателю полезно проверить это самостоятельно). Однако программа подсчёта эффективности по такой формуле (точнее – программа сравнения количества операций, необходимых «в среднем» для сортировки массива размерности *n* , с числом *n*logn*) не относится к рассматриваемой нами теме «сортировка массива». Эта программа будет приведена в части 2 настоящего пособия, при описании способов преобразования рекурсивных алгоритмов в нерекурсивные. Отметим ещё, что при сравнении эффективности «в среднем» алгоритмов `HeapSort` и `QuickSort` (например, методом случайного выбора начального массива) преимущество оказывается у последнего метода: коэффициент при *n*logn* для него оказывается меньше. (Провести такое сравнительное исследование двух методов сортировки читателю также полезно самостоятельно.)

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Абрамов С. Лекции о сложности алгоритмов. М.: МЦНМО, 2009.
2. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы. М.: Вильямс, 2001
3. Алексеев В. Введение в теорию сложности алгоритмов. М.: Изд-во ВМиК МГУ, 2002
4. Алексеев В., Таланов В. Графы и алгоритмы. Структуры данных. Модели вычислений. М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2006
5. Вирт Н. Алгоритмы и структуры данных. М.: Мир, 1989.
6. Гагарина Л.Г., Колдаев В.Д. Алгоритмы и структуры данных. М: Финансы и статистика; ИНФРА-М, 2009.
7. Головешкин В, Ульянов М. Теория рекурсии для программистов. М.: ФИЗМАТЛИТ, 2006
8. Громкович Ю. Теоретическая информатика. Введение в теорию автоматов, теорию вычислимости,теорию сложности, теорию алгоритмов, теорию связи, и криптографию. СПб.: БХВ, 2010.
9. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М: Вильямс, 2006
10. Кнут Д. Искусство программирования, том 1. Основные алгоритмы. М.: Вильямс, 2006.
11. Крупский В. Введение в сложность вычислений. М.: Факториал Пресс, 2006
12. Левитин А. Алгоритмы: введение в разработку и анализ алгоритмы. М.: Вильямс, 2006.
13. Липский В. Комбинаторика для программистов. М.: Мир,1989
14. Макконнелл Дж. Основы современных алгоритмов М.: Техносфера, 2004.
15. Немнюгин С. Turbo Pascal. Программирование на языке высокого уровня. СПб.: Питер, 2007
16. Окулов С. Программирование в алгоритмах. М.: Бином, 2004
17. Сапоженко А. Некоторые вопросы сложности алгоритмов. М.: Изд-во ВМиК МГУ, 2001
18. Харари Ф. Теория графов. М.: КомКнига, 2006
19. Шень А. Программирование: теоремы и задачи, М., Изд-во МЦНМО, 1995.
20. Яблонский С. Введение в дискретную математику. М.: Высшая школа, 2003.

Учебное издание

Мельников Борис Феликсович,
Мельникова Елена Анатольевна

АЛГОРИТМЫ СОРТИРОВКИ МАССИВОВ. СЛОЖНОСТЬ АЛГОРИТМОВ

Учебное пособие

Публикуется в авторской редакции
Титульное редактирование *Т. И. Кузнецовой*
Компьютерная верстка, макет *Т. В. Кондратьевой*

Подписано в печать 16.12.14. Формат 60x84/16. Бумага офсетная. Печать оперативная.
Усл.-печ. л. 2,3; уч.-изд. л. 2,5. Гарнитура Times. Тираж 100 экз. Заказ № 2579.
Издательство «Самарский университет», 443011, г. Самара, ул. Акад. Павлова, 1.
Тел. 8 (846) 334-54-23.
Отпечатано на УОП СамГУ