Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
«Вятский государственный гуманитарный университет»

Дополнительная подготовка школьников по дисциплине «Информатика и информационные технологии»

Учебный модуль
Сортировки

С. Ю. Иванов

Киров
2011

СОДЕРЖАНИЕ

1. Введение

Сортировка данных является одной из наиболее частых операций, которые реализуются в приложениях самого разного вида и уровня сложности. В зависимости от объемов данных и требований к времени работы алгоритма могут применяться различные виды сортировок. В данном модуле мы рассмотрим сортировки с трудоемкостью порядка $O(N^2)$ в худшем случае и порядка $O(N \cdot \log_2 N)$ в лучшем. Также рассмотрим алгоритм сортировки подсчётом, работающий за $O(N)$, и алгоритмы поиска данных за $O(\log_2 N)$.

Пусть дано множество объектов M. Между объектами этого множества должно быть определено отношение порядка, чтобы мы могли сказать, какой объект из двух меньше, а какой больше. Например, если у нас есть множество учеников, для которых указан рост в сантиметрах, то мы можем определить отношение порядка между учениками по их росту, следовательно, мы сможем их отсортировать. Аналогично, если для ученика задана фамилия, то мы можем отсортировать учеников в лексикографическом порядке, например, «петров» меньше чем «пупков», так как полагаем вторую букву в первом слове «е» меньше, чем «у».

В простейшем случае будем считать, что множество, состоящее из N элементов, представлено в виде массива длины N. В каждом элементе массива закодирован элемент множества. В наших примерах будем полагать, что элементами массива являются целые числа. Сами алгоритмы сортировки и поиска будут справедливы и для других типов данных, главное чтобы для типа было задано отношение порядка.

Большинство алгоритмов, осуществляющих сортировку массивов, основано на целенаправленном сравнении пар элементов и в случае необходимости их перемещении в массиве так, чтобы в итоге массив оказался упорядоченным в соответствии с критерием сортировки.

Существует большое количество различных стратегий, в соответствии с которыми может осуществляться перестановка элементов массива. При описании таких стратегий будем рассматривать задачу сортировки массива в следующей упрощенной форме. Задан числовой массив $a[1..N]$, требуется переставить его элементы, то есть обменять содержимое элементов массива, так чтобы в результате перестановок выполнилось условие: $a[1] \le a[2] \le ... \le a[N]$.

Для всех алгоритмов будем использовать массив целых чисел, заданный следующим образом:

```
type mas=array[1..NMax] of integer;
var a: mas;
```

Мы объявили собственный тип `mas`, чтобы массив можно было передавать в качестве параметра в функции сортировки.

Во многих представленных ниже алгоритмах нам потребуется операция перестановки местами двух элементов массива. Её можно выполнить с помощью трёх операторов присваивания. Чтобы поменять местами содержимое переменных $a[i]$ и $a[j]$ будем использовать процедуру `swap`, которая меняет местами значения в двух переменных:

```
procedure swap(var x,y:integer);
var temp:integer;
begin
  temp:=x;
  x:=y;
  y:=temp;
end;
```

Оценивая качество различных алгоритмов, обычно интересуются тем, как зависит время его работы от длины N сортируемой последовательности и требуется ли для этого дополнительная память, размер которой зависит от параметра N.

Поскольку время работы таких алгоритмов существенно зависит от количества сравнений и перемещений элементов в массиве, следует обращать внимание на то, как эти количества зависят от длины сортируемого массива.

При представлении алгоритмов в качестве кода будем использовать язык Паскаль. При этом код будет организован так, чтобы он компилировался в следующих средах: PascalABC.Net, Borland Delphi, Free Pascal.

2. Трудоемкость, устойчивость

Для оценки алгоритмов сортировки часто выделяют следующие характеристики:

1. *Время*. Это один из основных параметров, характеризующих алгоритмы сортировки. Он определяет быстродействие алгоритма, его трудоёмкость или временную сложность. Обозначается как O(N), где N – размерность задачи, в случае сортировки данных N – это количество элементов массива. Простейшие алгоритмы сортировки работают за время O(N^2), в то время как более сложные реализации позволяют значительно сократить эту оценку до O($N \cdot \log_2 N$). Существуют также некоторые специализированные алгоритмы, работающие за O(N), например, сортировка подсчётом.

2. *Память*. Некоторые алгоритмы для своей работы требуют выделения дополнительной памяти под хранение промежуточных данных, что позволяет увеличить скорость работы алгоритмы. Например, алгоритм сортировки слиянием предполагает использование дополнительной памяти.

3. *Устойчивость*. В исходных данных может оказаться несколько элементов с одинаковым значением, по которому осуществляется сортировка, однако в некоторых случаях важно, чтобы такие элементы сохранили исходный порядок относительно друг друга в отсортированном массиве. Сортировки, которые удовлетворяют этому требованию, и называются *устойчивыми*. Не все сортировки являются устойчивыми изначально, но практически любую из них можно достаточно легко модифицировать для решения этой задачи: достаточно при сравнении элементов анализировать не только их значение, но и место в исходном массиве, если их значения совпали. Такого рода модификация может потребовать дополнительного объёма памяти и процессорного времени на обработку.

4. *Использование операции сравнения*. Большинство сортировок используют операции сравнения элементов массива и осуществляют их перестановку. Однако существуют и такие алгоритмы, которые не подразумевают сравнения элементов. Примером такой сортировки является сортировка подсчётом. Как правило, сортировки с использованием операции сравнения являются универсальными и могут применяться для сортировки любых данных, а сортировки без использования операции сравнения – в специальных случаях, когда о данных известны какие-то дополнительные сведения.

5. *Естественность поведения*. Данная характеристика говорит об эффективности алгоритма при обработке отсортированных или частично отсортированных данных. Если алгоритм учитывает данный факт и оптимизирует свою работу, то говорят, что алгоритм ведёт себя естественно.

3. «Простые» сортировки

Алгоритмы сортировок, рассматриваемые в данной части, уже обсуждались ранее в теме «Массивы и строки», поэтому мы не будем детально останавливаться на них, а отметим различные варианты их реализации и оптимизации, а также оценим временную сложность алгоритмов в лучшем и худшем случаях. Условно мы назовём эти сортировки *простыми*.

3.1. Сортировка простым выбором

Идея сортировки состоит в том, чтобы на i-м шаге найти в массиве элемент, который должен стоять на i-м месте, и поставить его туда.

Рассмотрим следующую процедуру:

```
procedure sort(var a: mas; n: integer);
var i,j: integer;
begin
  for i:=1 to n-1 do
    for j:=i+1 to n do
      if a[i] > a[j] then swap(a[i], a[j]);
end;
```

Вложенный цикл `for` как раз занимается тем, что ищет минимальный элемент на интервале $[i..N]$. Таким образом, по окончании этого цикла $a[i]$ будет содержать искомый элемент.

Несложно заметить, что данную процедуру можно оптимизировать. При поиске элемента, который должен стоять на i-м месте, мы меняем местами $a[i]$ и $a[j]$ каждый раз, когда выполняется условие $a[i] > a[j]$, вместо этого можно найти минимальный элемент на данном участке массива и поставить его на нужную позицию. Рассмотрим модифицированный алгоритм:

```
procedure sort(var a: mas; n: integer);
var i,j, min: integer;
begin
  for i:=1 to N-1 do begin
    min:=i; {ищем минимальный элемент на интервале [i..N]}
    for j:=i+1 to N do
      if a[j] < a[min] then min:=j;
```

```
    if i < min then swap(a[i], a[min]); {если минимальный
элемент стоит не на своем месте, то ставим его на i-ую
позицию}
  end;
end;
```

Переменная `min` будет содержать индекс минимального элемента на интервале [*i*..*N*]. После завершения вложенного цикла `for` нам остаётся лишь, проверить не находится ли уже минимальный элемент на *i*-м месте. Если находится, то нам ничего не надо делать, иначе необходимо поменять элементы местами.

Оценим количество операций сравнения. На первом шаге при $i = 1$ нам потребуется $N–1$ сравнение, при $i = 2$ потребуется $N–2$ сравнение, а на последнем шаге – лишь одно сравнение. Получается

$$C = (N – 1) + (N – 2) + (N – 3) + \ldots + 1 = N\cdot(N–1) / 2.$$

Следует заметить, что этими подсчетами не ограничивается реальное число операций, так как организация циклов и обращений к процедуре `swap` требует скрытых от нас операций с памятью, зависящих от используемой системы программирования. Однако мы обоснованно можем считать, что эти накладные расходы не более чем в константное число раз превышают результаты наших подсчетов.

Не имея возможности оценивать упомянутые константы, при анализе сложности алгоритмов ограничиваются асимптотическими оценками. В нашем случае приведенные оценки числа сравнений и перемещений позволяют оценить время работы сортировки в худшем случае величиной $O(N^2)$.

3.2. Сортировка простым обменом или методом «пузырька»

Данный метод в качестве основной операции использует перестановку двух соседних элементов $a[i]$ и $a[i+1]$, если $a[i] > a[i+1]$. Если мы выполним такую операцию для i от 1 до N, то на последнем месте будет стоять максимальный элемент. После этого нам остаётся повторить эти же действия $N–1$ раз, но при этом окончание массива можно будет уже не рассматривать, т. к. там элементы будут упорядочены. Итак, рассмотрим процедуру с реализацией данного алгоритма:

```
procedure sort(var a: mas; n: integer);
var i,k: integer;
begin
  for k:=1 to n-1 do
    for i:=1 to n-k do
      if a[i]<a[i+1] then swap(a[i],a[i+1]);
end;
```

Заметим, что после первого выполнения внутреннего цикла (при $k = 1$) на последнем месте в массиве $a[1..n]$ окажется максимальный элемент этого массива. После второго его выполнении (при $k = 2$) на предпоследнем месте в массиве $a[1..n]$ окажется элемент меньше или равен максимальному, но не меньше остальных и т. д. После последнего выполнения внутреннего цикла (при $k = n–1$) массив окажется полностью отсортированным.

При сортировке методом «пузырька» выполняется $N–1$ просмотров, на каждом i-м просмотре производится $N–i$ сравнений. Таким образом, общее количество сравнений $C = N·(N–1)/2$, или $O(N^2)$.

Данный алгоритм можно оптимизировать, так как при некотором значении k может оказаться, что массив уже отсортирован, то дальнейшие проверки можно исключить. Определить отсортирован ли массив достаточно просто – можно проверить была ли вызвана процедура `swap` хотя бы один раз при текущем значении k. Итак, модифицированная версия процедуры:

```
procedure sort(var a: mas; n: integer);
  var i,k: integer; moved: boolean;
begin
  for k:=1 to n-1 do begin
    moved:=false;
    for i:=1 to n-k do
      if a[i]>a[i+1] then begin {проверяем два соседних
элемента}
        swap(a[i],a[i+1]); {меняем соседние элементы
местами}
        moved:=true; {помечаем, что на текущей итерации
хоты бы одна пара элементов поменяла свое место}
      end;
    if not moved then break; {если ни один элемент не был
переставлен, то это значит, что массив отсортирован, и
действия цикла можно прервать}
  end;
end;
```

В лучшем случае, если массив уже был упорядочен до вызова `sort`, нам потребуется всего лишь O(N). Однако общая временн*а*я сложность алгоритма от этого не изменится и в худшем случае будет по-прежнему O(N^2).

3.3. Сортировка вставками

В основе сортировки вставками лежит предположение, что первые k элементов массива уже упорядочены. Так как массив из одного элемента уже отсортирован, то мы начнем рассматривать массив со второго элемента, в этом случае нам требуется определить либо оставить элемент на месте, либо поставить его перед первым элементом. Далее берём третий элемент и ищем позицию от 1 до 3, на которую его надо поставить. Общий алгоритм будет следующим: берётся k-й элемент, и для него подбирается место в отсортированной части массива таким образом, чтобы сохранить упорядоченность первых k элементов. Описанные действия можно реализовать следующим образом:

```
procedure sort(var a: mas; n: integer);
var i,k,x: integer;
begin
  for k:=2 to N do begin
    x:=a[k]; {запоминаем k-й элемент, т.к. его значение
будет потеряно при сдвиге элементов в массиве}
    i:=k-1;
    while (i>0) and (x<a[i]) do begin {сдвигаем вправо на
один все элементы больше X}
      a[i+1]:=a[i];
      Dec(i);
    end;
    a[i+1]:=x; {ставим X на свое место}
  end;
end;
```

Для упорядоченного массива количество сравнений будет N–1, но в худшем случае, когда массив отсортирован в обратном порядке, нам потребуется $N\cdot(N–1)/2$ сравнений, или O(N^2).

4. Стратегии «разделяй и властвуй»

4.1. Общая идея

Принцип «*разделяй и властвуй*» (англ. *divide and conquer*) является одной из парадигм разработки алгоритмов, которая состоит в рекурсивном разбиении задачи на две или более подзадачи того же типа, но меньшего размера, после чего решения меньших задач комбинируются для получения ответа к исходной задаче.

Разбиение на подзадачи происходит до тех пор, пока подзадачи не окажутся элементарными. В нашем случае для сортировки подзадачи будут элементарными, когда каждая часть будет содержать лишь по одному элементу.

Благодаря тому, что на каждом шаге мы разбиваем задачу на две, фактически уменьшая ее размерность в два раза, то, очевидно, что мы выполним $\log_2 N$ таких разбиений. Для сортировки каждой части нам потребуется не более N операций, следовательно, общая сложность алгоритма будет $O(N \cdot \log_2 N)$, что существенно быстрее рассмотренных ранее алгоритмов, работающих за $O(N^2)$, и позволяет сортировать большие объёмы данных.

4.2. Сортировка слиянием

Сортировка слиянием (англ. *merge sort*) является разновидностью метода «разделяй и властвуй» и была предложена Джоном фон Нейманом в 1945 году. Основная идея заключается в том, что если у нас есть два отсортированных массива, тогда мы можем объединить их в новый упорядоченный массив за время $O(N)$, где N – длина каждого исходного массива. Согласно принципу «разделяй и властвуй», будем разбивать исходный массив на подмассивы до тех пор, пока они не станут одноэлементными. Следовательно, мы получим два упорядоченных одноэлементных массива и сможем их объединить в новый двухэлементный упорядоченный массив. Затем мы продолжим процесс и будем объединять двухэлементные массивы в четырехэлементные и так далее, пока не вернёмся к исходной размерности N.

Реализуем процедуру `merge`, которая объединяет два упорядоченных «массива», а точнее два упорядоченных участка одного и того же массива:

```
procedure merge(var a: mas; l, r, m: integer);
var b: mas;
    i, k, left: integer;
begin
  k:=m;
  left:=l;
  for i:=left to r do begin
    if (k > r) or ((l < m) and (a[l] < a[k])) then begin
{выбираем из какой части - левой или правой должны взять
очередной элемент}
      b[i]:= a[l]; {выбираем очередной элемент из левой
части и сдвигаем в ней указатель вправо}
      Inc(l);
    end
    else begin
      b[i]:= a[k]; {выбираем очередной элемент из правой
части и сдвигаем в ней указатель вправо}
      Inc(k);
    end;
  end;
  for i:=left to r do a[i]:=b[i]; {копируем элементы из
временного массива B обратно в массив A}
end;
```

Поясним параметры процедуры: `a` – сортируемый массив, `l` и `r` – индексы левого и правого элементов, образующих интервал для объединения, `m` – средний элемент на данном участке. Таким образом, у нас есть две отсортированные части: [`l`, `m`–1] и [`m`, `r`]. Для объединения нам потребуется временный массив `b`, из которого мы в конце процедуры скопируем элементы обратно в `a`. На `i`-м шаге выбираем, из какой части [`l`, `m`–1] или [`m`, `r`] мы должны взять элемент и поместить на `i`-ую позицию. Из первой части [`l`, `m`–1] мы берем элемент в двух случаях:

1) если элементы во второй части [`m`, `r`] ещё есть и текущий элемент первой части меньше текущего элемента второй части;

2) если элементы во второй части [`m`, `r`] уже закончились, то есть они уже были помещены в массив `b` ранее.

Если ни одно из этих двух условий не выполняется, то тогда на `i`-ую позицию помещаем очередной элемент из второй части [`m`, `r`].

Функция сортировки будет выглядеть следующим образом:

```
procedure sort(var a:mas; i,j: integer);
var m:integer;
begin
  if i < j then begin
    m:=(i+j) div 2;          {находим середину текущего
                              интервала в массиве}
    sort(a, i, m);           {сортируем левую часть}
    sort(a, m+1, j);         {сортируем правую часть}
    merge(a, i, j, m+1);     {объединяем две отсортированные
                              части в одну}
  end
end;
```

Первый раз мы вызовем функцию `sort` со следующими параметрами:

```
sort(a, 1, N);
```

Как уже отмечалось, сложность данного алгоритма $O(N \cdot \log_2 N)$, в этом легко убедиться: функция `merge` производит слияние за M итераций цикла, где M – длина объединяемых массивов, при этом максимальное значение $M = N / 2$, так как влияние констант для оценки временной сложности алгоритма несущественно, то можно сказать, что функция `merge` работает за время $O(N)$. Если посмотреть на реализацию функции `sort`, то можно посчитать, что функция `merge` будет вызвана $\log_2 N$ раз, так как при каждом рекурсивном вызове `sort` длина массива сокращается в два раза. Таким образом, общая временная сложность алгоритма получается $O(N \cdot \log_2 N)$.

Однако следует отметить, что функция `merge` требует дополнительного расхода памяти. В нашей реализации массив `b` создается такой же размерности что и исходный массив `a`**.** Причем это происходит при каждом вызове функции `merge`, для оптимизации можно предусмотреть создание такого массива лишь один раз перед началом сортировки, а затем его использование в процедуре слияния массивов.

4.3. Быстрая сортировка

Быстрая сортировка (англ. *quick sort*) – метод был предложен Чарльзом Хоаром в 1962 году, а так как алгоритм получился достаточно эффективным в большинстве случаев, то автор назвал его быстрой сортировкой.

Суть алгоритма: выберем в массиве барьерный элемент и запомним его значение в *X*, а все элементы в массиве переставим так, чтобы слева от *X* были элементы меньше, а справа больше, чем *X*. В качестве барьерного элемента можно брать средний элемент, хотя его выбор может быть различным и его даже можно выбирать случайным образом. Для перестановки элементов массива относительно *X* мы будем использовать две переменные: *i* будет двигаться слева направо и *j* справа налево, если $a[i] > x$ и $a[j] < x$, то мы будем менять местами $a[i]$ и $a[j]$. Следует отметить, что после таких перестановок *X* будет стоять уже на своем месте, поэтому будет достаточно повторить алгоритм для части массива слева от *X* и для части массива справа от *X*, что в полной мере соответствует принципу «разделяй и властвуй».

Рассмотрим следующую реализацию быстрой сортировки:

```
procedure sort(var a:mas; l,r: integer);
var i,j,x: integer;
begin
  If l>=r Then Exit;
  i:=l;
  j:=r;
  x:=a[(l+r) div 2]; {запоминаем барьерный элемент}
  repeat
    while a[i]<x do Inc(i); {пропускаем элементы слева от
X, которые стоят на своем месте}
    while a[j]>x do Dec(j); {пропускаем элементы справа от
X, которые стоят на своем месте}
    if i<=j then begin
      swap(a[i], a[j]); {меняем местами два элемента,
стоящие не на своем месте относительно X}
      Inc(i);
      Dec(j);
    end
  until i>j;
  sort(a,l,j); {сортируем левую часть}
  sort(a,i,r); {сортируем правую часть}
end;
```

Поскольку алгоритм рекурсивный, то первым делом необходимо задать «заглушку» – `l` (левая граница, сортируемого участка массива) должна быть меньше `r` (правая граница). Переменную `i` устанавливаем в начало участка, `r` в конец, а в `x` запоминаем средний элемент. Пока `i` и `j` не достигнут друг

друга, мы будем искать такую пару, что `a[i]>x` и `a[j]<x`, если такая пара найдена, то вызовем `swap(a[i],a[j])`, чтобы поменять их значения местами и упорядочить относительно `x`. После цикла вызовем алгоритм для левой и правой частей соответственно.

Первый вызов функции `sort` выглядит так:

```
sort(a, 1, N);
```

В среднем временная сложность алгоритма составляет $O(N \cdot \log_2 N)$, однако в худшем случае получается $O(N^2)$. Худшим случаем будет та ситуация, когда барьерный элемент равен минимальному или максимальному значению на текущем интервале, так как это приведет к тому, что при рекурсивном вызове `sort` мы сократим длину массиву не в два раза, а всего лишь на один элемент. В связи с этим существуют различные оптимизации, в том числе и по выбору барьерного элемента, который можно выбирать случайным образом или, например, выбирать средний по значению элемент из первого, последнего и среднего элементов текущего участка массива.

4.4. Поиск *k*-й порядковой статистики

K-й порядковой статистикой набора из *n* чисел называется элемент, который после сортировки набора в возрастающем порядке оказывается на *k*-м месте. В частности первая порядковая статистика равна минимальному элементу массива и может быть найдена сканированием массива за время $O(N)$, это же справедливо для максимального элемента, если мы хотим найти *n*-ую порядковую статистику. Однако для произвольного *k* решить задачу за $O(N)$ не так легко. Для решения задачи можно отсортировать массив, но, как правило, это потребует $O(N \cdot \log_2 N)$ операций, если не рассматривать сортировку подсчётом, которая не всегда применима. В отсортированном массиве нам достаточно вернуть *k*-й элемент. Существует несколько решений, позволяющих найти *k*-ую порядковую статистику за $O(N)$, мы рассмотрим вариант, основанный на алгоритме быстрой сортировки.

Итак, как мы уже отмечали, после перестановки элементов массива относительно *X*, сам барьерный элемент встает на свое место в уже отсортированном массиве. Следовательно, нам нужно проверить, не совпадает ли место *X* со значением *k*, если совпадает, то мы нашли решение и алгоритм можно остановить, иначе нам нужно сравнить место *X* и значение *k*, чтобы определить в какой части массива следует продолжить поиск. Однако,

как и быстрая сортировка, данный алгоритм может потребовать $O(N^2)$ операций в худшем случае.

Рассмотрим один из возможных вариантов реализации алгоритма:

```
function Part(var a:mas; l,r:integer):integer;
var x,i,j: integer;
begin
  x:=a[r];
  i:=l-1;
  for j:=l to r-1 do
    if a[j]<=x then begin
      Inc(i);
      swap(a[i],a[j])
    end;
  swap(a[i+1],a[r]);
  Part:=i+1
end;

function GetK(var a:mas; l,r,k: integer): integer;
  var m: integer;
begin
  m:=Part(a,l,r);
  if m = k then GetK:=a[m]
  else if m > k then GetK:=GetK(a,l,m-1,k)
  else GetK:=GetK(a,m+1,r,k);
end;
```

Первый вызов функции и получение ответа выглядит так:

```
X:=GetK(a, 1, N, k);
```

где a – массив, N – количество элементов массива, k – искомая порядковая статистика.

Как уже было сказано, существуют и другие алгоритмы нахождения k-й статистики, например, алгоритм BFPRT, созданный Мануэлем Блюмом, Робертом Флойдом, Воганом Рональдом Праттом, Роном Ривестом и Робертом Тарьяном в 1973 году и работающий за время $O(N)$.

4.5. Двоичный поиск элемента в отсортированном массиве

Для произвольного массива найти произвольный элемент можно за время O(*N*), перебрав последовательно все элементы. Однако если на массив наложить некоторые ограничения, то алгоритм поиска можно ускорить. В частности если мы знаем, что массив отсортирован, то можно воспользоваться *двоичным* или *бинарным поиском*.

Идея, лежащая в основе этого алгоритма, уже нам знакома: на каждом шаге мы будем делить массив пополам и выбирать ту половину, в которой необходимо продолжить поиск.

```
function find(var a:mas; n,x: integer): integer;
var ind,l,r,m: integer;
begin
  l:=1;
  r:=n;
  ind:=-1;    {если элемент X не найден, то вернем -1}
  while (l<=r) and (ind=-1) do begin {продолжаем поиски,
пока не найден элемент и пока левая и правая граница не
сошлись}
    m:=(l+r) div 2; {находим средний элемент}
    if x<a[m] then r:=m-1 {продолжаем поиск в левой
половине}
    else if x>a[m] then l:=m+1 {продолжаем поиск в правой
половине}
    else ind:=m; {элемент найден, запоминаем его индекс}
  end;
  find:=ind;
end;
```

В цикле на каждом шаге мы сравниваем средний элемент `a[m]` с искомым значением `x`, если `a[m]` больше `x`, то следует искать в левой половине массива, поэтому мы сдвигаем правую границу `r` вправо, если `a[m]` меньше `x`, то сдвигаем левую границу `l` вправо, а если `a[m]` равно `x`, то мы нашли искомое число и запоминаем ответ в переменной `ind`. Если число в массиве не было найдено, то функция `find` вернёт –1.

Оценив максимальное количество операций, получаем временную сложность алгоритма $O(\log_2 N)$, так как на каждом шаге мы уменьшаем длину массива в два раза.

4.6. Сортировка подсчетом

В *сортировке подсчетом* (*counting sort*) предполагается, что исходный массив $a[1..n]$ содержит целые числа из интервала $L..H$, где L и H – целые числа, таким образом, массив может содержать K различных чисел, где $K = H - L + 1$. Суть алгоритма заключается в том, что значение массива $a[i]$ используется в качестве индекса другого массива, в котором мы будем считать, сколько раз нам встретилось значение $a[i]$. Реализация алгоритма может выглядеть следующим образом:

```
procedure sort(var a:mas; n:integer);
var i,j,x: integer;
    c: array[L..H] of integer;
begin
  For i:=L to H do c[i]:=0;
  For i:=1 to n do Inc(c[a[i]]);
  x:=1;
  for i:=L to H do
    for j:=1 to c[i] do begin
      a[x]:=i;
      Inc(x);
    end;
end;
```

Первый цикл `for` просто заполняет массив с нулями, то есть мы считаем, что любая цифра от `L` до `H` встретилась ноль раз. Вторым циклом `for`, мы перебираем все элементы массива `a` и увеличиваем счётчик в массиве `c`, по завершению цикла `b[i]` будет содержать сколько раз встретилось число `i` в массиве `a`. На последней стадии мы перебираем все числа от `L` до `H`, и если для какого-то из них в массиве `c` содержится число больше 0, то это значит что оно встречалось в массиве и мы должны его поместить обратно в `a`. Поскольку числа в диапазоне `L..H` мы перебираем по порядку, то они уже являются упорядоченными и нам остается их лишь поместить обратно в массив `a`.

Преимуществом данной сортировки является то, что она работает фактически за линейное время, которое не зависит от входных данных, т.е. в данном случае неважно в каком порядке располагаются числа в массиве `a`, алгоритм будет всегда работать одно и то же время. Оценим его временную сложность: нам требуется `K` операций для инициализации массива `c`, затем `N` операций для подсчёта количества каждого числа в массиве `a`, и в конце у

нас есть вложенный цикл, который потребует `Max(K,N)` операций. Недостатком сортировки подсчётом может являться большой объём памяти, например, нам требуется отсортировать 10 чисел в диапазоне от 1 до 10^9, это потребует от нас массива на один миллиард элементов, если тип элемента будет `integer` (4 байта), то общий объем памяти составит порядка 4 Гбайт. Кроме того, это потребует от нас и цикла от 1 до 10^9, что очень отрицательно скажется на времени работы алгоритма. Поэтому сортировка подсчётом больше подходит для случаев, когда диапазон чисел не велик, но при этом каждое число может встречаться очень часто. Таким образом, общую сложность алгоритма можно оценить как O(N+K), где N – количество элементов в массиве, а K – количество чисел в диапазоне значений элементов массива.

4.7. Внешняя сортировка

Внешней сортировкой называют сортировку данных, располагающихся во внешней памяти и слишком больших, чтобы можно было целиком переместить их в оперативную память и применить один из методов внутренней сортировки.

Внешняя память характеризуется последовательным доступом к элементам данных в отличие от прямого (адресуемого) доступа к данным во внутренней (оперативной) памяти. Методы внешней сортировки появились, когда устройствами внешней памяти были магнитные ленты с чисто последовательным доступом. Современные дисковые запоминающие устройства реализуют так называемый индексно-последовательный доступ, при котором время обращения к данным состоит из двух этапов: практически прямого доступа к блоку памяти и затем последовательного считывания данных из выбранного блока во внутреннюю память. Эти особенности работы с внешней памятью естественно следует учитывать при разработке алгоритмов.

Следует заметить, что скорость выполнения алгоритмов зависит от размера блоков, которые способна принять внутренняя память. Мы рассмотрим метод внешней сортировки, работающие при минимальных расходах оперативной памяти.

Посмотрим, как можно использовать метод слияния при внешней сортировке. Предположим, что в файле A, записаны последовательно элементы сортируемого числового массива $a[1..n]$. Для простоты предположим, что n представляет собой степень числа 2. Будем считать, что каждая запись состоит ровно из одного элемента, представляющего собой

ключ сортировки. Для сортировки используются два вспомогательных файла *B* и *C*. Размер каждого из них будет *n*/2.

Сортировка состоит из последовательности шагов, в каждом из которых выполняется распределение содержимого файла *A* в файлы *B* и *C*, а затем слияние файлов *B* и *C* в файл *A*.

На первом шаге последовательно читается файл *A*, из которого записи с нечетными номерами *a*[1], *a*[3], ..., *a*[*n*–1] переписываются в файл *B*, а записи *a*[2], *a*[4], ..., *a*[*n*] с четными номерами – в файл *C*. Первое слияние производится над парами (*a*[1], *a*[2]), (*a*[3], *a*[4]), ..., (*a*[*n*–1], *a*[*n*]), в результате чего каждая пара оказывается упорядоченной по неубыванию и эти пары последовательно записываются в файл *A*.

На втором шаге из файла *A* последовательно переписываются в файл *B* уже пары с нечетными номерами, а пары с четными номерами, а в файл *C*. При очередном слиянии образуются и пишутся в файл *A* упорядоченные четверки записей. И так далее. Перед выполнением последнего шага файл *A* будет содержать две упорядоченные подпоследовательности размером *n*/2 каждая. При распределении первая из них попадет в файл *B*, а вторая – в файл *C*. После слияния файл *A* будет содержать полностью упорядоченную последовательность записей.

Заметим, что для выполнения внешней сортировки методом слияния в основной памяти требуется расположить всего лишь две переменные – для размещения очередных записей из файлов *B* и *C*. Файлы *A*, *B* и *C* будут $O(\log_2 n)$ раз прочитаны и столько же раз записаны.

5. Задания для самостоятельного решения

1. Измените сортировку слиянием так, чтобы временный массив создавался только один раз, а не при каждом вызове функции `merge`.

2. Считается, что выбор элемента X в быстрой сортировке случайным образом (вместо выбора среднего элемента) оставляет среднюю эффективность неизменной при всех типах массивов (упорядоченный, распределенный по какому-то закону и т.д.). Реализуйте эту идею.

3. Замените рекурсивную схему реализации быстрой сортировки на нерекурсивную.

4. Реализуйте рекурсивный вариант бинарного поиска элемента в массиве. Модифицируйте алгоритм для работы со строками, а не с числами.

5. Ниже по тексту приведены три версии процедуры бинарного поиска. Какая из них верная? Исправьте ошибки. Какая из них более эффективная?

```
Procedure Search;
Var i,j,k: Integer;
Begin
  i:=l;j:=N;
  Repeat
    k:=(i+j) Div 2;
    If X<=A[k] Then
      j:=k-l;
    If A[k]<=X Then
      i:=k+l;
  Until i>j
End;
```

```
Procedure Search;
Var i,j,k: Integer;
Begin
  i:=l;j:=N;
  Repeat
    k:=(i+j) Div 2;
    If X<A[k] then
      j:=k
    Else i:=k+l;
  Until i>=j
End;
```

```
Procedure Search;
Var i,j,k: Integer;
Begin
  i:=l;j:=N;
  Repeat
    k:=(i+j) Div 2;
    If A[k]<X Then
      i:=k
    Else j:=k;
  Until  (A[k]=X)  Or
(i>=j)
End;
```

6. Решить задачу поиска числа X в двухмерном массиве $A[1..N, 1..M]$ (элементы в строках и столбцах упорядочены) за время, пропорциональное $O(N + M)$.

7. Реализуйте устойчивый вариант сортировки подсчетом, чтобы сохранялся исходный порядок равных элементов.

8. Для нахождения *k*-й порядковой статистики мы использовали функцию `Part`, измените алгоритм быстрой сортировки так, чтобы он тоже использовал эту функцию.

9. Можно ли для поиска *k*-й порядковой статистики применить оптимизацию для быстрой сортировки, и выбирать не средний элемент, а произвольный? Если можно, то реализуйте этот подход и проверьте на больших наборах данных, чтобы время работы не было $O(N^2)$.

10. Можно ли сортировку подсчетом использовать для упорядочивания слов, а не чисел? Ответ обоснуйте, предложите варианты решения.

6. Заключение

Мы рассмотрели далеко не все способы сортировки массивов. В частности среди рассмотренных нами алгоритмов внутренней сортировки нет алгоритма, который бы имел оценку трудоемкости $O(N \cdot \log_2 N)$ на худший случай и не требовал бы дополнительной памяти, зависящего от N. Одним из таких алгоритмов является так называемая *пирамидальная сортировка*, для ознакомления с которой мы отсылаем читателя к списку литературы.

Литература

1. Алексеев В. Е., Таланов В. А. Графы и алгоритмы. Структуры данных. Модели вычислений. – М: Интернет-Университет Информационных Технологий. БИНОМ. Лаборатория знаний, 2006.

2. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. – М: Мир, 1979.

3. Кнут Д. Искусство программирования. Том 3. Сортировка и поиск – 2-е изд. – М.: Издательский дом «Вильямс», 2007.

4. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. – 2-е изд.: Пер. с англ. – М.: Издательский дом «Вильямс», 2005.

5. Окулов С. М. Основы программирования. – 5-е изд., испр. – М.: БИНОМ. Лаборатория знаний, 2010.

6. Окулов С. М., Пестов А. А. 100 задач по информатике. – Киров: изд-во ВГПУ, 2000.