Лекция 8

План лекции

1. **Алгоритмы порождения подмножеств**
2. **Генерирование всех подмножеств**
3. **Алгоритм генерации всех двоичных векторов длины n в лексикографическом порядке**
4. **Генерирование подмножеств с условием**
5. **Генерирование k-элементных подмножеств**
6. **Алгоритмы перестановок**
7. **Генерация сочетаний в лексикографическом порядке**
8. **Выбор с помощью сортировки**

8.1. Быстрая сортировка

8.2. Сортировка слиянием

8.3. Бинарный поиск элементов в массиве.

1. Алгоритмы порождения подмножеств.

Рассмотрим задачу генерирования подмножеств некоторого n-множества $A = \{a_1, a_2, ..., a_n\}$.

Существует несколько вариантов этой задачи, которые включают:

- генерирование всех возможных подмножеств данного множества,

- генерирование подмножеств с некоторыми условиями, накладываемыми на генерируемые подмножества.

Определено, что каждое n-множество A имеет точно 2^n подмножеств. Поэтому при создании компьютерных алгоритмов порождения подмножеств целесообразно каждое из полученных подмножеств представить в виде двоичной последовательности.

Некоторое подмножество $B \subset A$ может быть сопоставлено с двоичной последовательностью $b_1 b_2 ... b_j ... b_n$, определяемой следующим образом:

$$b_j = \begin{cases} 0, & a_j \notin B, \\ 1, & a_j \in B. \end{cases}$$

Такой подход позволяет установить взаимно однозначное соответствие между всеми булевыми векторами длины n и элементами множества B.

2. Генерирование всех подмножеств

Для генерации всех подмножеств n-множества A, очевидно, достаточно породить все двоичные наборы длины n.

Легко увидеть, что наиболее прямым способом их порождения является запись в системе счисления с основанием 2.

Пример. Сгенерировать все подмножества множества $M=\{a_0,a_1,a_2\}$

Решение. Обозначим $i-$е подмножество множества M через B_i, где $i=1,2,...,2^{|M|}$

Каждому подмножеству B_i поставим в соответствие двоичную последовательность $b_0b_1b_2$ с условием, что

$$b_j=\begin{cases}0,\ a_j\notin B,\\ 1,\ a_j\in B.\end{cases}$$

Результаты соответствия представим в таблице:

i	$b_0b_1b_2$	B_i
0	000	$\varnothing$
1	001	a_2
2	010	a_1
3	011	a_1,a_2
4	100	a_0
5	101	a_0,a_2
6	110	a_0,a_1
7	111	a_0,a_1,a_2

$$2^M=\{\varnothing,a_0,a_1,a_2,\{a_0,a_1\},\{a_0,a_2\},\{a_1,a_2\},\{a_0,a_1,a_2\}\}$$

3. Алгоритм генерации всех двоичных векторов длины n в лексикографическом порядке

Алгоритм порождает все двоичные векторы $b=(b_{n-1},b_{n-1},...,b_1,b_0)$ длины n в лексикографическом порядке, начиная с наименьшего элемента.

1. Будем использовать массив b[n],b[n-1], ….,b[1],b[0], установив b[n]:=0.
2. Просматривая справа налево, находим первую позицию b[i] такую, что b[i]=0.
3. Записываем b[i]:=1, а все элементы b[j], j<i, стоящие справа от b[i], полагаем равными 0.
4. Для всех порождаемых последовательностей элемент b[n] не изменяется, за исключением генерации последнего вектора

(1,1,…,1), i=n. Равенство b[n]=1 является условием остановки алгоритма.

Псевдокод программы генерации двоичных векторов длины n

```
For i:=0 to n do b[i]:=0; [начальный вектор]
While b[n]≠1 do   [проверка окончания алгоритма ]
begin
 Write(b[n-1], b[n-2],…, b[0]);
 i=0;
 While b[i]=1 do
 begin
  b[i]:=0;
  i:=i+1;
 end;
 b[i]:=1;
end;
```

Рассмотрим генерацию подмножеств множества $A=\{a_0,a_1,...,a_{n-1}\}$. Введем фиктивный элемент $a_n \notin A$.

Генерация всех двоичных векторов b длины $n=3$ и подмножеств B множества $A=\{a_0,a_1,a_2\}$.

```
B:=∅;
While a_n ∉ B do
begin
 Write(B);
 i:=0;
 While a_i ∈ B do
 begin
   B:=B\{a_i};
   i:=i+1;
 end;
 B:=B∪{a_i};
end;
```

$b^1=(0,0,0),\ B^1=\varnothing,\ i=1;$

$b^2=(0,0,1),\ B^2=\{a_2\},\ i=2;$

$b^3=(0,1,0),\ B^3=\{a_1\},\ i=0;$

$b^4=(0,1,1),\ B^4=\{a_1,a_2\},\ i=2;$

$b^5=(1,0,0),\ B^5=\{a_0\},\ i=0;$

$b^6=(1,0,1),\ B^6=\{a_0,a_2\},\ i=1;$

$b^7=(1,1,0),\ B^7=\{a_0,a_1\},\ i=0;$

$b^8=(1,1,1),\ B^8=\{a_0,a_1,a_2\},\ i=3.$

4. Генерирование подмножеств с условием

Генерация подмножеств с условием минимального отличия соседних порождаемых элементов.

Для такой генерации воспользуемся записью чисел в двоичном *коде Грея.*

Алгоритм получения кода Грея.

Пусть $b_1b_2...b_n$ – некоторое двоичное число.

Первый способ генерации кода Грея. Код Грея этого числа получают, сдвигая это число на один разряд вправо, и, отбросив самый правый (n-й разряд), складывают поразрядно по модулю два с этим же, но несдвинутым числом:

$$\begin{array}{l} b_1b_2b_3...b_{n-1}b_n \\ \underline{\oplus\ b_1b_2b_3.....b_{n-1}\cancel{b_n}} \\ c_1c_2c_3....c_{n-1}c_n \end{array}$$

Таким образом, каждый результирующий разряд c_i,. получают по формуле $c_i = b_i \oplus b_{i-1}$, полагая, что $b_0 = 0$.

Пример. Рассмотрим порядок генерации кода Грея для трехразрядных двоичных чисел:

i	Двоичн. число	Операция	Код Грея
0	000	$000 \oplus 00 = 000$	000
1	001	$001 \oplus 00 = 001$	001
2	010	$010 \oplus 01 = 011$	011
3	011	$011 \oplus 01 = 010$	010
4	100	$100 \oplus 10 = 110$	110
5	101	$101 \oplus 10 = 111$	111
6	110	$110 \oplus 11 = 101$	101
7	111	$111 \oplus 11 = 100$	100

Второй способ генерации кода Грея

Второй способ генерации кода Грея содержит следующие шаги:

1. В качестве базовой используем двухразрядную последовательность: 00,01,11,10.

2. Строим трехразрядную последовательность:

2а. Припишем к 00,01,11,10 справа 0:

000,010,110,100.

2б. Переставим элементы 00,01,11,10 в обратном порядке:

10,11,01,00.

2в. К элементам 10,11,01,00 припишем справа 1:

101,111,011,001.

2г. Объединим последовательности из п.2а и п.2в:

000, 010,110,100,101,111,011,001.

3. Для получения четырехразрядной последовательности перейдем к п.1 алгоритма, заменив двухразрядную последовательность последовательностью, полученной в п. 2г.

4. Повторяя действия $n-2$ раза, получим $n-$разрядный код Грея.

Правило.

Если последовательность $c_1, c_2, c_3, ..., c_k$ содержит все двоичные последовательности длины k и каждый член последовательности отличается от соседнего точно одним элементом, то, приписывая справа нуль к каждому члену этой последовательности и единицу также к каждому члену этой последовательности, записанной в обратном порядке, получим в результате новую последовательность, которая содержит все последовательности длины $k+1$, и каждые ее соседние члены отличаются друг от друга точно в одной координате.

Пример. Сгенерировать все подмножества множества $A=\{a_1, a_2, a_3\}$ с условием минимального отличия соседних порождаемых элементов.

Решение. Результаты представим в виде таблицы:

i	$b_1 b_2 b_3$	B_i
0	000	$\varnothing$
1	001	a_3
2	011	a_2, a_3
3	010	a_2
4	110	a_1, a_2
5	111	a_1, a_2, a_3
6	101	a_1, a_3
7	100	a_1

Псевдокод программы, генерирующей все подмножества с помощью кода Грея

```
 Program Gray;
Var
 i,M,N:byte;
 {N-разряднось, M=2^N-количество комбинаций}
  G:array[1..M] of byte;

 function BinToGray(b:byte):byte;
 begin
  BinToGray:=b xor (b shr 1)
 end;
```

```
begin (* главная программа *)
  For i:=1 to M do G[i]:=BinToGray(i);
end; (*конец программы*)
```

5. Генерирование k-элементных подмножеств

Сгенерируем все $k-$элементные подмножества $n-$элементного множества X. Пусть $X=\{1,2,...,n\}$. Тогда каждому $k-$элементному подмножеству соответствует возрастающая последовательность длины k, составленная из элементов множества X. Будем генерировать такие последовательности в лексикографическом порядке.

1. Рассмотрим последовательность $(a_1,a_2,...,a_k)$.

2. Тогда следующая за ней последовательность:

$(b_1,b_2,...,b_k)=(a_1,...,a_{p-1},a_p+1,a_p+2,...,a_p+k-p+1)$, где

$p=\max\{i\,|\,a_i<n-k+1\}$

3.Последовательность, следующая за $(b_1,b_2,...,b_k)$:

$(c_1,...,c_k)=(b_1,...,b_{p'-1},b_{p'}+1,b_{p'}+2,...,b_{p'}+k-p'+1)$, где

$$p'=\begin{cases}p-1, & если\, b_k=n,\\ k, & если\, b_k<n\end{cases}$$

Псевдокод алгоритма, генерирующего все $k-$элементные подмножества $n-$множества

```
begin
   For i:=0 to k do A[i]:=i;
   p:=k;
   while p≥1 do
   begin
    write (A[1],…,A[k]);
    if A[k]=n then p:=p-1
    else p:=k;
    If p≥1 then
    For i:=k downto p do
    A[i]:=A[p]+i-p+1;
   end;
end;
```

Справа приведен пример 4-элементных подмножеств множества {1,…,6}, построенных по данному алгоритму.

```
1234
1235
1236
1245
1246
1256
1345
1346
1356
1456
2345
2546
2356
2456
3456
```

6. Алгоритмы перестановок

Рассмотрим методы генерирования последовательностей $n!$ перестановок множества, составленного из n элементов. Для этого заданное множество представим в виде элементов массива $P[1], P[2], ..., P[n]$.

Методы, которые будут нами рассматриваться, базируются на применении к массиву $P[i]$, $i = 1, 2, ..., n$ элементарной операции, которая носит название **поэлементной транспозиции**. Суть операции состоит в обмене данными между элементами массива $P[i]$ и $P[j]$, $1 \le i, j \le n$ по такой схеме:

$$vrem := P[i],\ P[i] := P[j],\ P[j] := vrem,$$

где $vrem -$ некоторая вспомогательная переменная, используемая для временного хранения значения элемента массива $P[i]$.

Лексикографический порядок

Определение лексикографического порядка. Пусть существуют перестановки в виде последовательностей $\{x_1, x_2, x_3, ..., x_n\}, \{y_1, y_2, y_3, ..., y_n\}, ...$ одного и того же множества X. Перестановки из элементов множества X упорядочены в лексикографическом порядке, если $\{x_1, x_2, x_3, ..., x_n\} < \{y_1, y_2, y_3, ..., y_n\}$ тогда и только тогда, когда для некоторого k: $x_k \le y_k$ и $x_i = y_i$ для всех $i < k$.

Определение антилексикографического порядка. Пусть существуют перестановки в виде последовательностей $\{x_1, x_2, x_3, ..., x_n\}, \{y_1, y_2, y_3, ..., y_n\}, ...$ одного и того же множества X. Перестановки из элементов множества X упорядочены в антилексикографическом порядке, если $\{x_1, x_2, x_3, ..., x_n\} <$ ’$\{y_1, y_2, y_3, ..., y_n\}$ тогда и только тогда, когда для некоторого k: $x_k > y_k$ и $x_i = y_i$ для всех $i < k$.

Алгоритм построения перестановок в лексикографическом порядке

Начальная перестановка $(1, 2, ..., n)$.

Завершающая перестановка $(n, n-1, ..., 1)$.

Переходим от $(x_1, x_2, ..., x_n)$ до $(y_1, y_2, ..., y_n)$

Как строим перестановку $(y_1, y_2, ..., y_n)$?

1. Рассматриваем справа налево перестановку

$$x = \overleftarrow{(x_1, x_2, ..., x_i, x_{i+1} ..., x_n)}$$

и найдем такую позицию i, что $x_i < x_{i+1}$.

2. Если такой позиции нет, то тогда $x_1 > x_2 > ... > x_n$, то есть $x = (n, n-1, ..., 1)$. Данная перестановка является завершающей перестановкой нашего алгоритма.
3. Если позиция i найдена, то $x_i < x_{i+1} > x_{i+2} > ... > x_n$.

4. Ищем первую позицию j в диапазоне от позиции n к позиции i такую, что $x_i < x_j$. Тогда $i < j$.

$$x = \overleftarrow{(x_1, x_2, ..., x_i, x_{i+1}, ..., x_j, ..., x_n)}$$

5. Дальше выполняем операцию транспозиции над элементами x_i и x_j

$$x = \overleftarrow{(x_1, x_2, ..., x_i, x_{i+1}, ..., x_j, ..., x_n)}$$

6. В полученной перестановке часть элементов $x_{i+1}, ..., x_{n-1}, x_n$ переворачиваем, то есть меняем порядок их следования на противоположный.
7. Полученная перестановка является перестановкой $y = (y_1, y_2, ..., y_n)$.
Именно эта перестановка является следующей в лексикографическом порядке следования перестановок.

Пример. Пусть $x = (2, 6, 5, 8, 7, 4, 3, 1)$.
1. В соответствии с данным алгоритмом, $x_i = 5$, а $x_j = 7$.

2. Выполним транспозицию для элементов $i = 3$ и $j = 5$:

$$\tilde{x} = (2, 6, 7, 8, 5, 4, 3, 2, 1)$$

3. Перевернем часть элементов $x_3, ..., x_8 \to x_8, ..., x_3$:

$$(8, 5, 4, 3, 1) \to (1, 3, 2, 5, 8).$$

В результате получим следующую в лексикографическом порядке перестановку $y = (2, 6, 7, 1, 3, 4, 5, 8)$

Псевдокод программы генерирования перестановок в лексикографическом порядке

Элемент массива a[0]=0 используется только как признак окончания алгоритма.

For j:=0 **to** n **do** a[j]:=j;{генерация нач. посл.}

```
i:=1;
while i≠0 do
begin
 write(a[1],a[2],…,a[n]);
 i:=n-1;                    {поиск a[i]}
 while a[i]>a[i+1] do i:=i-1;
 j:=n;                      {поиск a[j]}
 while a[j]<a[i] do j:=j-1;
 Swap(a[i],a[j]);

 {переворачивание части последовательности}
 k:=i+1;
 m:=i+tranc((n-1)/2);
 while k≤m do
 begin
  Swap(a[k],a[n-k+i+1]);
  k:=k+1;
 end;
end;
```

Пример. При $n=3$ процесс работы данного алгоритма представлен следующей последовательностью перестроений перестановок a^k.

a^1={123}, $a^1[i]=2$, $a^1[j]=3$;
a^2={132}, $a^2[i]=1$, $a^2[j]=2$;
a^3={213}, $a^3[i]=1$, $a^3[j]=3$;
a^4={231}, $a^4[i]=1$, $a^4[j]=3$;
a^5={312}, $a^5[i]=1$, $a^5[j]=2$;
a^6={321}, i=0;

Перестановки множеств $X=\{1,2,3\}$ в лесикографическом (а) и антилексикографическом (б) порядке

	(а)	(б)
1	1 2 3	1 2 3
2	1 3 2	2 1 3
3	2 1 3	1 3 2
4	2 3 1	3 1 2
5	3 1 2	2 3 1
6	3 2 1	3 2 1

7. Генерация сочетаний в лексикографическом порядке

Сочетанием из n элементов по k называется неупорядоченная выборка k элементов из заданных n элементов.

Будем выполнять $k-$выборки из $n-$множества $A=\{1,2,...,n\}$.

Первая выборка равна: $\{1,2,...,k\}$.

Последняя выборка равна: $(n-k+1,n-k+2,...,n-1,n)$.

Определим по сочетанию $a=(a_1,a_2,...,a_k)$ вид следующего за ним сочетания: $b=(a_1,...,a_{m-1},a_m+1,a_m+2,...,a_m+k-m+1)$,

где $m=\max\{i|a_i<n-k+i,1\le i\le k\}$. Если b следует за a, то

$$b_i=\begin{cases}a_i, \text{ если } 1\le i<m,\\ a_m+i-m+1, \text{ если } m\le i\le k,\end{cases} \text{ где } m=\begin{cases}m-1, \text{ если } b_k=n,\\ k, \text{ если } b_k<n.\end{cases}$$

Псевдокод алгоритма сочетаний из n по k в лексикографическом порядке

```
Var p,i,k,n,m:integer;
    a: Array [0..19] of Integer;
begin
 For i:=1 to k do a[i]:=i;
 If k=n then p:=1 else p:=k;
 while p>=1  do
 begin
  For m:=1 to k do write(a[m]);
  writeln('');
  if a[k]=n then p:=p-1 else p:=k;
  if p>=1 then
  for i:=k downto p do
  a[i]:=a[p]+i-p+1;
 end;
end.
```

```
123
124
125
134
135
145
234
235
245
345
```

Справа приведен пример сочетаний из 5 по 3, построенных по данному алгоритму.

Словарный порядок

Пример. *Разбиение числа 7 в словарном порядке.*

(7·1) = (1, 1, 1, 1, 1, 1, 1),
(1·2, 5·1) = (2, 1, 1, 1, 1, 1),
(2·2, 3·1) = (2, 2, 1, 1, 1),
(3·2, 1·1) = (2, 2, 2, 1),
(1·3, 4·1) = (3, 1, 1, 1, 1),
(1·3, 1·2, 2·1) = (3, 2, 1, 1),
(1·3, 2·2) = (3, 2, 2),
(2·3, 1·1) = (3, 3, 1),
(1·4, 3·1) = (4, 1, 1, 1),
(1·4, 1·2, 1·1) = (4, 2, 1),
(1·4, 1·3) = (4, 3),

(1·5, 2·1) = (5, 1, 1),
(1·5, 1·2) = (5, 2),
(1·6, 1·1) = (6, 1),
(1·7) =(7).

8. Выбор с помощью сортировки

Задачу выбора можно свести к СОРТИРОВКЕ.

1. Упорядочить массив. Вычислительная сложность O(n log n)

2. Выполнить поиск нужного элемента.

Это эффективно в том случае, когда выбор нужно делать многократно

Рассмотрим наиболее распространенные алгоритмы сортировки.

Быстрая сортировка Quick Sort

Быстрая сортировка (англ. Quick Sort) - алгоритм сортировки , хорошо известный, как алгоритм разработанный Чарльзом Хоаром

Общая идея алгоритма состоит в следующем:

- Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива.
- Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на два непрерывных отрезка, следующие друг за другом — «меньшие опорного», «равные и больше опорного».
- Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

Быстрая сортировка использует стратегию «разделяй и властвуй». Шаги алгоритма таковы:

1. Выбираем в массиве некоторый элемент, который будем называть *опорным элементом*. С точки зрения корректности алгоритма выбор опорного элемента безразличен. С точки зрения повышения эффективности алгоритма выбираться должна медиана, но без дополнительных сведений о сортируемых данных её обычно невозможно получить. Известные стратегии: выбирать постоянно один и тот же элемент, например, средний или последний по положению; выбирать элемент со случайно выбранным индексом.
2. Операция *разделения* массива: реорганизуем массив таким образом, чтобы все элементы со значением меньшим или равным опорному

элементу, оказались слева от него, а все элементы, превышающие по значению опорный — справа от него. Обычный алгоритм операции:

1. Два индекса — L и R, приравниваются к минимальному и максимальному индексу разделяемого массива, соответственно.
2. Вычисляется индекс опорного элемента M.
3. Индекс L последовательно увеличивается до тех пор, пока L-й элемент не окажется больше либо равен опорному.
4. Индекс R последовательно уменьшается до тех пор, пока R-й элемент не окажется меньше либо равен опорному.
5. Если R=L — найдена середина массива — операция разделения закончена, оба индекса указывают на опорный элемент.
6. Если L < R — найденную пару элементов нужно обменять местами и продолжить операцию разделения с тех значений L и R, которые были достигнуты. Следует учесть, что если какая-либо граница (L или R) дошла до опорного элемента, то при обмене значение M изменяется на R-й или L-й элемент соответственно, изменяется именно индекс опорного элемента и алгоритм продолжает свое выполнение.

3. Рекурсивно упорядочиваем подмассивы, лежащие слева и справа от опорного элемента.
4. Базой рекурсии являются наборы, состоящие из одного или двух элементов. Первый возвращается в исходном виде, во втором, при необходимости, сортировка сводится к перестановке двух элементов. Все такие отрезки уже упорядочены в процессе разделения.

Поскольку в каждой итерации (на каждом следующем уровне рекурсии) длина обрабатываемого отрезка массива уменьшается, по меньшей мере, на единицу, терминальная ветвь рекурсии будет достигнута обязательно и обработка гарантированно завершится.

Пример реализации алгоритма Quick Sort на языке Pascal.

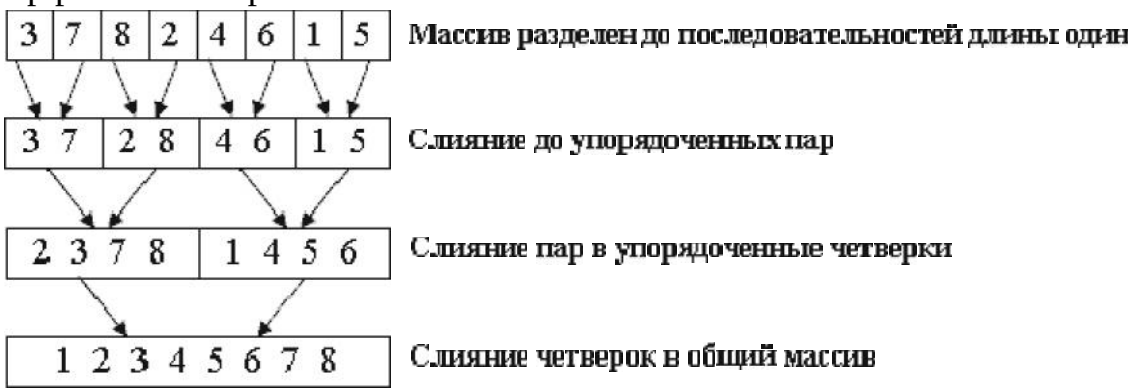
```
program Quick_Sort;
   var A:array[1..100] of integer;
   N,i : integer;
{В процедуру передаются левая и правая границы сортируемого
фрагмента}
  procedure QSort(L,R:integer);
  var M,X,y,i,j:integer;
  begin
   M=(L+R) div 2;
   X:=A[M];
   i:=L; j:=R;
   while i<=j do
    begin
     while A[i]<X do i:=i+1;
```

```
    while A[j]>X do j:=j-1;
      if i<=j then
      begin
          y:=A[i]; A[i]:=A[j]; A[j]:=y;
          i:=i+1; j:=j-1;
        end;
   end;
   if L<j then QSort(L,j);
   if i<R then QSort(i,R);
end;
begin
 write('Количество элементов массива ');
 read(N);
 for i:=1 to n do read(A[i]);
 QSort(1,n); {Упорядочить элементы с первого по n-ый}
{Упорядоченный массив}
 for i:=1 to n do write(A[i],' ');
end.
```

8.2. Сортировка слиянием

Сортировка слиянием также построена на принципе "разделяй-и-властвуй", однако реализует его несколько по-другому, нежели quickSort. А именно, вместо разделения по опорному элементу массив просто делится пополам.

Функция Merge на месте двух упорядоченных массивов A[left]...A[mid] и A[mid+1]...A[right] создает единый упорядоченный массив A[left]...A[right]. Пример работы алгоритма на массиве 3 7 8 2 4 6 1 5..



Рекурсивный алгоритм обходит получившееся дерево слияния в прямом порядке. Каждый уровень представляет собой *проход* сортировки слияния - операцию, полностью переписывающую массив.

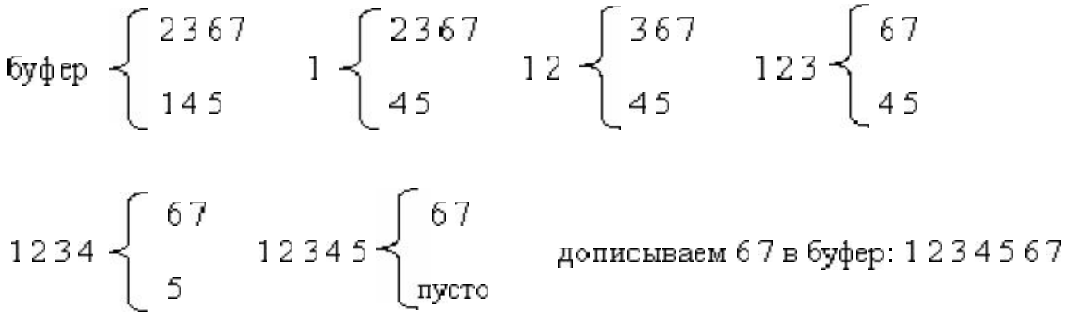
Обратим внимание, что деление происходит до массива из единственного элемента. Такой массив можно считать упорядоченным, а значит, задача сводится к написанию функции слияния merge.

```
Program MrgeSort;
Var A,B : array[1..1000] of integer;
    N  : integer;
{процедура сливающая массивы}
Procedure Merge(left,right : integer);
Var mid,i,j,k : integer;
Begin
 mid:=(left+right) div 2;
 i:=left;
 j:=mid+1;
 for k:=left to right do
 if (i<=mid) and ((j>right) or (A[i]<A[j])) then
 begin
   B[k]:=A[i];
   i:=i+1;
 end  else
 begin
   B[k]:=A[j];
   j:=j+1;
 end ;
 for k:=left to right do A[k]:=B[k];
End;
{left,right - индексы начала и конца сортируемой части
массива}
Procedure Sort(left,right : integer);
Begin
 {массив из одного элемента тривиально упорядочен}
 if left<right then
 begin
  mid:=(left+right) div 2
  Sort(left,mid);
  Sort((mid + 1,right);
  Merge(left,right);
 end;
End;
Begin
 {Определение размера массива A - N) и его заполнение}
 …
 {запуск сортирующей процедуры}
 Sort(1,N);
 {Вывод отсортированного массива A}
  …
End.
```

Один из способов состоит в слиянии двух упорядоченных последовательностей при помощи вспомогательного буфера, равного по размеру общему количеству имеющихся в них элементов. Элементы последовательностей будут перемещаться в этот буфер по одному за шаг.

```
merge ( упорядоченные последовательности A, B , буфер C )
{
пока A и B непусты {сравнить первые элементы A и B
          переместить наименьший в буфер
    }
    если в одной из последовательностей еще есть элементы
    дописать их в конец буфера, сохраняя имеющийся порядок
    }
```

Пример работы на последовательностях 2 3 6 7 и 1 4 5




```
}
```

Оценим быстродействие алгоритма: время работы определяется рекурсивной формулой T(n) = 2T(n/2) + Theta(n).

Ее решение: T(n) = n log n - результат весьма неплох, учитывая отсутствие "худшего случая". Однако, несмотря на хорошее общее быстродействие, у сортировки слиянием есть и серьезный минус: она требует Theta(n) памяти.

Хорошо запрограммированная внутренняя сортировка слиянием работает немного быстрее пирамидальной, но медленнее быстрой, при этом требуя много памяти под буфер. Поэтому MergeSort используют для упорядочения массивов, лишь если требуется устойчивость метода(которой нет ни у быстрой, ни у пирамидальной сортировок).

Сортировка слиянием является одним из наиболее эффективных методов для односвязных списков и файлов, когда есть лишь последовательный доступ к элементам.

8.3. Двоичный (бинарный) поиск элемента в массиве

Если у нас есть массив, содержащий упорядоченную последовательность данных, то очень эффективен двоичный поиск.

Идея поиска заключается в том, чтобы брать элемент посередине, между границами, и сравнивать его с искомым. В случае равенства возвращать его, а если искомое больше(в случае правостороннего - не меньше), чем элемент

сравнения, то сужаем область поиска так, чтобы новая левая граница была равна индексу середины предыдущей области. В противном случае присваиваем это значение правой границе. Проделываем эту процедуру до тех пор, пока правая граница больше левой более чем на 1, или же пока мы не найдём искомый индекс.

Двоичный поиск - очень мощный метод. Если, например, длина массива равна 1023, после первого сравнения область сужается до 511 элементов, а после второй - до 255. Легко посчитать, что для поиска в массиве из 1023 элементов достаточно 10 сравнений.

.