Методические рекомендации к лабораторной работе

«Рекурсивные алгоритмы сортировки одномерных массивов»

Рекурсия – фундаментальное понятие в математике и компьютерных науках. В языках программирования рекурсивной программой называется программа, которая обращается *сама к себе* (подобно тому, как в математике рекурсивная функция определяется через понятия самой этой функции). Рекурсивная программа не может вызывать себя до бесконечности, следовательно, вторая важная особенность рекурсивной программы – наличие *условия завершения*, позволяющее программе прекратить вызывать себя.

Таким образом, рекурсия в программировании может быть определена как *сведение задачи к такой же задаче, но манипулирующей более простыми данными.*

Как следствие, рекурсивная программа должна иметь *как минимум два пути выполнения*, один из которых предполагает рекурсивный вызов (случай «сложных» данных), а второй – без рекурсивного вызова (случай «простых» данных).

Сущность рекурсии

Процедура или функция может содержать вызов других процедур или функций. В том числе **процедура или функция может вызвать саму себя**. Никакого парадокса здесь нет – компьютер лишь последовательно выполняет встретившиеся ему в программе команды и, если встречается вызов процедуры, просто начинает выполнять эту процедуру. Без разницы, какая процедура дала команду это делать.

Пример рекурсивной процедуры (Слева Delphi, справа C):

```
procedure Rec(a: in-
teger);
begin
  if a>0 then
     Rec(a-1);
  writeln(a);
end;
```

```
void Rec(int a);
void Rec(int a)
{
  if (a>0) Rec(--a);
  printf("%d",a);
}
```

Важно!
Поскольку по стандартам C функция должна быть объявлена до использования, при задании рекурсии необходимо объявлять прототип функции.

Рассмотрим, что произойдет, если в основной программе поставить вызов, например, вида Rec(3). Ниже представлена блок-схема, показывающая последовательность выполнения операторов.

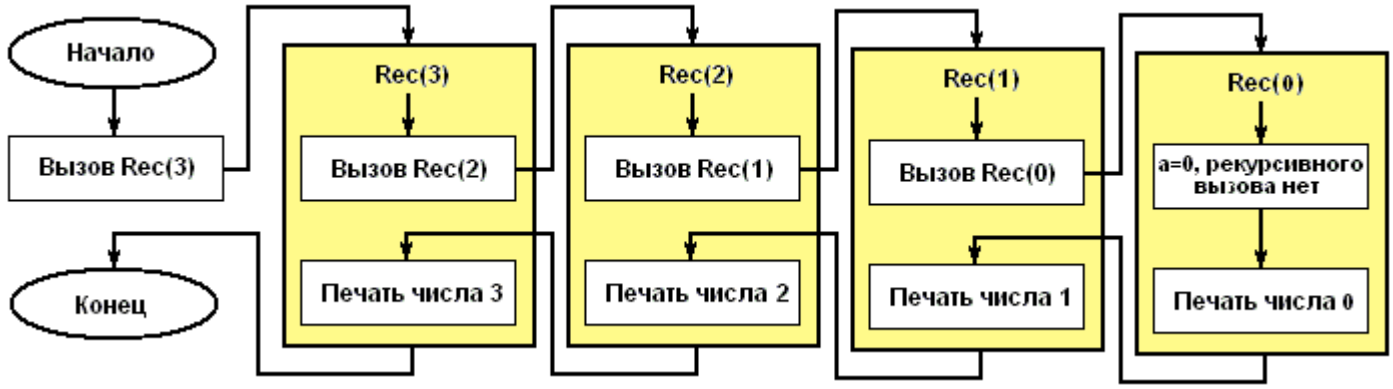


Рис. 1. Блок схема работы рекурсивной процедуры.

Процедура Rec вызывается с параметром a = 3. В ней содержится вызов процедуры Rec с параметром a = 2. Предыдущий вызов еще не завершился, поэтому можете представить себе, что создается еще одна процедура и до окончания ее работы первая свою работу не заканчивает. Процесс вызова заканчивается, когда параметр a = 0. В этот момент одновременно выполняются 4 экземпляра процедуры. Количество одновременно выполняемых процедур называют *глубиной рекурсии*.

Четвертая вызванная процедура (Rec(0)) напечатает число 0 и закончит свою работу. После этого управление возвращается к процедуре, которая ее вызвала (Rec(1)) и печатается число 1. И так далее пока не завершатся все процедуры. Результатом исходного вызова будет печать четырех чисел: 0, 1, 2, 3.

Еще один визуальный образ происходящего представлен на рис. 2.

Рис. 2. Выполнение процедуры Rec с параметром 3 состоит из выполнения процедуры Rec с параметром 2 и печати числа 3. В свою очередь выполнение процедуры Rec с параметром 2 состоит из выполнения процедуры Rec с параметром 1 и печати числа 2. и т. д.

Обратите внимание, что в приведенном примере рекурсивный вызов стоит внутри условного оператора. Это необходимое условие для того, чтобы рекурсия когда-нибудь закончилась. Также обратите внимание, что сама себя процедура вызывает с другим параметром, не таким, с каким была вызвана она сама. Если в процедуре не используются глобальные переменные, изменение параметра, передаваемого в процедуру необходимо, чтобы рекурсия не продолжалась до бесконечности.

Рекуррентные соотношения. Рекурсия и итерация

Говорят, что последовательность векторов $\{\vec{x}_n\}$ задана рекуррентным соотношением, если задан начальный вектор $\vec{x}_0 = (x_0^1, \ldots, x_0^D)$ и функциональная зависимость последующего вектора от предыдущего

$$\vec{x}_n = \vec{f}(\vec{x}_{n-1}) \qquad (1)$$

Простым примером величины, вычисляемой с помощью рекуррентных соотношений, является факториал

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$$

Очередной факториал $n!$ можно вычислить по предыдущему как:

$$n! = (n-1)! \cdot n \qquad (2)$$

Введя обозначение $x_n = n!$, получим соотношение:

$$x_n = x_{n-1} \cdot n,\ x_0 = 1 \qquad (3)$$

Вектора $\vec{x}_n$ из формулы (1) можно интерпретировать как наборы значений переменных. Тогда вычисление требуемого элемента последовательности будет состоять в повторяющемся обновлении их значений. В частности для факториала (Слева Delphi, по центру и справа два варианта написания на C) :

```
x := 1;
for i := 2 to n do
   x := x * i;
writeln(x);
```

```
x = 1;
for (i=2; i<n; i++)
       x = x *i;
printf("%d",s);
```

```
for (i=2, x=1; i<n; x*=i++);
printf("%d",s);
```

Каждое такое обновление (x = x * i) называется *итерацией*, а процесс повторения итераций – *итерированием*.

Обратим, однако, внимание, что соотношение (1) является чисто рекурсивным определением последовательности и вычисление n-го элемента есть на самом деле многократное взятие функции f от самой себя:

$$x_n = \underbrace{f(f(\ldots f(x_0)))}_{n} \qquad (4)$$

В частности для факториала можно написать:

```
function Factorial(n:integer):integer;
begin
  if n > 1 then
    Factorial:=n*Factorial(n-1)
  else
    Factorial := 1;
end;
```

```
int Factorial(int n);
int Factorial(int n)
{
  if (n>1)
    return n*Factorial(--n);
  else return 1;
}
```

Следует понимать, что вызов функций влечет за собой потерю производительности и выделение дополнительной памяти, поэтому первый вариант вычисления факториала будет несколько более быстрым. В целом итерационные решения работают быстрее рекурсивных.

Прежде чем переходить к ситуациям, когда рекурсия полезна, обратим внимание на один пример, где ее использовать не следует.

Рассмотрим частный случай рекуррентных соотношений, когда следующее значение в последовательности зависит не от одного, а сразу от нескольких предыдущих значений. Примером может служить известная последовательность Фибоначчи, в которой каждый следующий элемент есть сумма двух предыдущих:

$$x_n = x_{n-1} + x_{n-2},\ x_0 = 1,\ x_1 = 1 \qquad (5)$$

Через рекурсии это можно реализовать следующим образом:

```
function Fib(n: integer): integer;
begin
  if n > 1 then
    Fib := Fib(n-1) + Fib(n-2)
  else
    Fib := 1;
end;
```

```
int Fib (int n);
int Fib (int n)
{
  if (n>1)
     return Fib(n-1)+Fib(n-2);
  else return 1;
}
```

Каждый вызов Fib создает сразу две копии себя, каждая из копий – еще две и т.д. Количество операций растет с номером *n* экспоненциально, хотя при итерационном решении достаточно линейного по *n* количества операций.

Приведенный пример, скорее учит не **КОГДА** рекурсию не следует использовать, а тому, **КАК** ее не следует использовать. Если существует быстрое итерационное (на базе циклов) решение, то тот же цикл можно реализовать с помощью рекурсивной процедуры или функции:

```
// x1, x2 - начальные условия (1, 1)
// n - номер требуемого числа Фибоначчи

function Fib(x1,x2,n:integer):integer;
begin
  if n > 1 then
    Fib := Fib(x2,x1+x2, n-1);
  else
    Fib := x2;
end;
```

```
int Fib(int x1,int x2,int n);
int Fib(int x1,int x2,int n)
{
  if(n>1)
    return Fib(x2,x1+x2,--n);
    else return x2+x1;
}
```

Любые рекурсивные процедуры и функции, содержащие всего один рекурсивный вызов самих себя, легко заменяются итерационными циклами. Если процедура или функция вызывает себя два и более раз простого нерекурсивного аналога у нее не будет. В этом случае множество вызываемых процедур образует уже не цепочку, как на рис. 1, а целое дерево. Существуют широкие классы задач, когда вычислительный процесс должен быть организован именно таким об-

разом. Как раз для них рекурсия будет наиболее простым и естественным способом решения.

Метод «разделяй и властвуй».

Многие алгоритмы используют два рекурсивных вызова, каждый из которых работает приблизительно с половиной входных данных. Такая рекурсивная схема, по-видимому, представляет собой наиболее важный случай хорошо известного метода «разделяй и властвуй» (divide and conquer) разработки алгоритмов.

В качестве примера рассмотрим задачу отыскания максимального из N элементов, сохраненных в массиве a[0], . . . , a[N-1] с элементами типа Item. Эта задача легко может быть решена за один проход массива:

```
Max:=a[0];
for i:=1 to N do
   if a[i] > Max then  Max:=a[i];
```

```
Max=a[0];
for (i=1; i<N; i++)
   if (a[i] > Max) Max=a[i]
```

Рекурсивное решение типа «разделяй и властвуй» - еще один простой (хотя совершенно иной) способ решения той же задачи:

```
//l,r – начальный и конечный элемент обрабатываемого массива
//Начальные условия (a,0,N-1)
function Max(a:array of Item;
l,r:integer):Item;
var
  u, v : Item;
  m : integer;
begin
   m := (l+r) / 2;
   if l = r then Max := a[l]
   else
     begin
       u := Max (a, l, m);
       v := Max (a, m+1, r);
       if u > v then Max := u
       else Max := v
     end
end;
```

```
Item Max(Item a[], int l, int r);
Item Max(Item a[], int l, int r)
{
  Item u,v;
  int m=(l+r)/2;
  if (l==r) return a[l];
  else
  {
    u=Max(a, l, m);
    v=Max(a, m+1, r);
    if (u>v) return u;
    else return v;
  }
}
```

На Рис.3 показано разбиение массива на отдельные подмножества, в процессе рекурсивного поиска наибольшего числа.

Чаще всего подход “разделяй и властвуй» используют из-за того, что он обеспечивает более быстрые решения, чем итерационные алгоритмы.

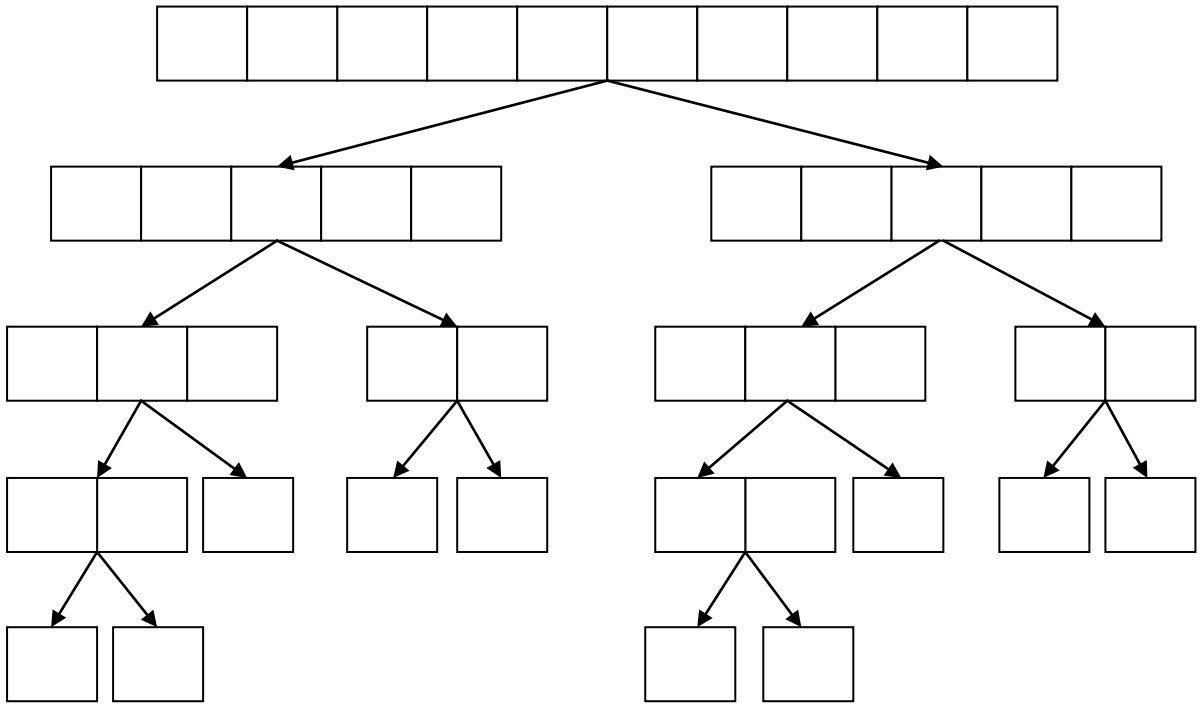


Рис. 3. Последовательность разбиения массива, в процессе рекурсивного обхода, при поиске наибольшего числа

Основным недостатком и одновременно преимуществом алгоритмов типа «разделяй и властвуй» является то, что они делят задачи на *независимые* подзадачи. Когда подзадачи независимы, это может приводит к увеличению ресурсов для общего решения, так как некоторые фрагменты подзадачи могут решаться по многу раз. В то же время, при грамотном планировании задач можно добиться существенного преимущества, за счет того, что независимые задачи могут выполняться параллельно на многопоточных системах, поскольку современные процессоры поддерживают от четырех параллельных вычислительных потоков и более рекурсивные методы могут обеспечить существенное преимущество.

Быстрые сортировки

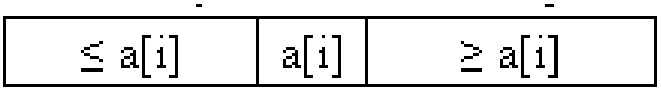
Простые методы сортировки вроде метода выбора или метода пузырька сортируют массив из n элементов за $O(n^2)$ операций. Однако с помощью принципа «разделяй и властвуй» удается построить более быстрые, работающие за $O(n \log_2 n)$ алгоритмы. Суть этого принципа в том, что решение получается путем рекурсивного разделения задачи на несколько простые подзадачи того же типа до тех

пор, пока они не станут элементарными. Приведем в качестве примеров несколько быстрых алгоритмов такого рода.

Алгоритм 1: «Быстрая» сортировка (quicksort).

Быстрая сортировка – усовершенствованный метод обменной сортировки, предложенный Чарльзом Хоаром. Метод основан на подходе «разделяй и властвуй».

Общая схема такова: из массива произвольным образом выбирается некоторый опорный элемент; запускается процедура разделения массива, которая перемещает все ключи, меньшие, либо равные a[i], влево от него, а все ключи, большие, либо равные a[i] - вправо, теперь массив состоит из двух подмножеств, причем левое меньше, либо равно правому, для обоих подмассивов:



если в подмассиве более двух элементов, рекурсивно запускаем для него ту же процедуру. В конце получится полностью отсортированная последовательность.

Рассмотрим алгоритм подробнее:

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, ока-

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

a) Два индекса — L и H, приравниваются к минимальному и максимальному индексу разделяемого массива, соответственно.
b) Вычисляется индекс опорного элемента m.
c) Индекс L последовательно увеличивается до тех пор, пока l-й элемент не окажется больше либо равен опорному.
d) Индекс H последовательно уменьшается до тех пор, пока r-й элемент не окажется меньше либо равен опорному.
e) Если H = L — найдена середина массива — операция разделения закончена, оба индекса указывают на опорный элемент.
f) 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.

Теперь массив разделен на две части: все элементы левой меньше либо равны x, все элементы правой - больше, либо равны x. Разделение завершено.

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.

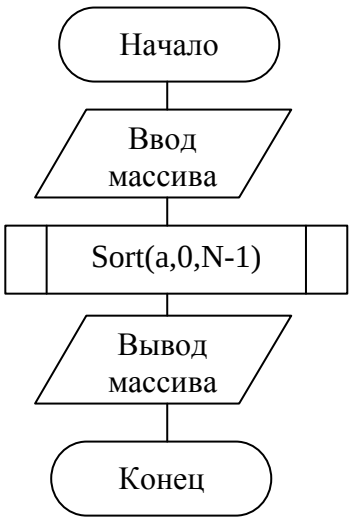
Поскольку в каждой итерации (на каждом следующем уровне рекурсии) длина обрабатываемого отрезка массива уменьшается, по меньшей мере, на единицу, терминальная ветвь рекурсии будет достигнута обязательно и обработка гарантированно завершится.

Необходимо отметить, что на скорость сортировки сильно влияет правильность выбора опорного числа X, наибольшее быстродействие достигается, когда число X равно медиане упорядочиваемого множества, однако не имея информации о структуре множества предположить какое значение имеет его медиана невозможно.

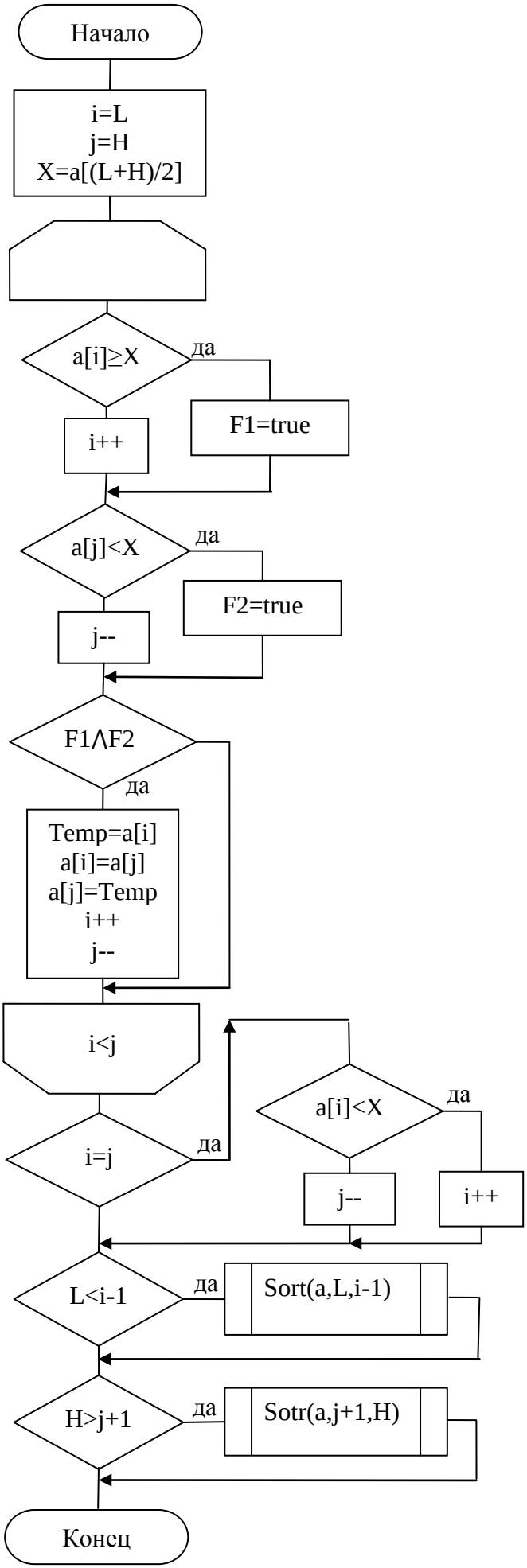
Для большинства задач в качестве опорного элемента X можно взять центральный элемент подмножества (как сделано в приведенном ниже примере), можно взять случайный элемент подмножества, либо среднее арифметическое по множеству. Последний вариант спорный, поскольку может существенно увеличить быстродействие если множество имеет нормальное распределение, либо наоборот его уменьшит, в противном случае, поскольку дополнительные ресурсы будут потрачены на вычисление среднего арифметического.

Важно! Поскольку при сортировке массив разбивается на независимые подмножества можно в каждом подмножестве работать с основным массивом, а не с его копией. Если каждый раз, при создании нового экземпляра рекурсивной функции копировать массив может произойти переполнение памяти.

Блок-схема алгоритма сортировки:



Блок-схема процедуры Sort (a,L,H):



Алгоритм 2: Сортировка слиянием (merge sort).

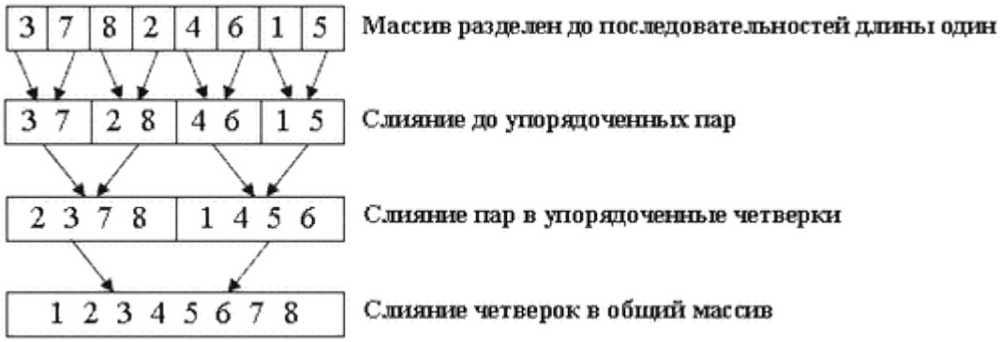
Алгоритм предложен фон Нейманом еще в 1945г. Суть алгоритма такова:

1. Сортируемый массив разбивается на две части примерно одинакового размера. (Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы, любой массив длины 1 можно считать упорядоченным.);
2. Каждая из получившихся частей сортируется отдельно, тем же самым алгоритмом;
3. Два упорядоченных массива половинного размера соединяются в один:

    a) На каждом шаге мы берём меньший из двух первых элементов упорядоченых подмассивов и записываем его в результирующий массив. Счетчики номеров элементов результирующего массива и подмассива из которого был взят элемент увеличиваем на 1.

    b) "Прицепление" остатка.

Пример работы алгоритма на массиве 3 7 8 2 4 6 1 5.



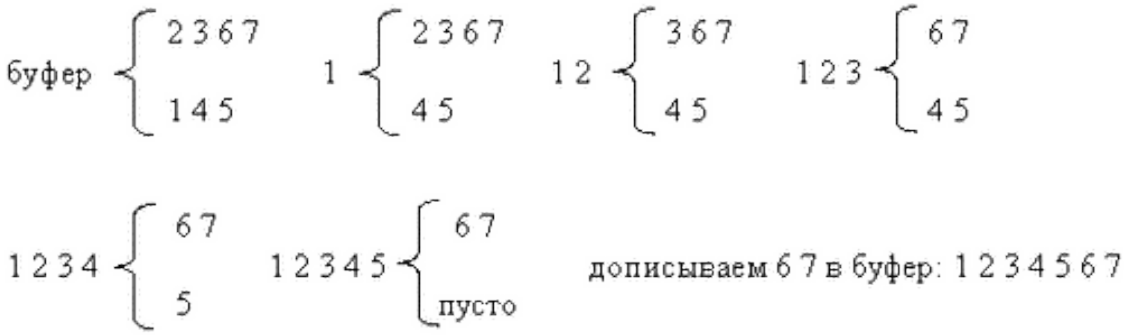
Алгоритм работает по принципу: разбивает массив на две части, отсортировывает каждую из них, а затем сливает обе части в одну отсортированную. Процесс объединения: подмассивы объединяются в рабочий массив, а затем копируются в исходный массив.

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

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.

1. Пока A и B не пусты сравнить элементы и переместить наименьший в буфер.
2. Если в одной из последовательностей остались элементы – дописать их в конец буфера, сохраняя имеющийся порядок.

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



Результатом является упорядоченная последовательность, находящаяся в буфере. В случае упорядочивания массивов нужно не забыть переписать данные из буфера обратно в массив.

Важно! Поскольку при сортировке массив разбивается на независимые подмножества можно в каждом подмножестве работать с основным массивом, а не с его копией. Если каждый раз, при создании нового экземпляра рекурсивной функции копировать массив может произойти переполнение памяти. По этой же причине можно использовать общий буфер, равный по величине размеру исходного массива. В показанной ниже блок-схеме буфер обозначен как b.

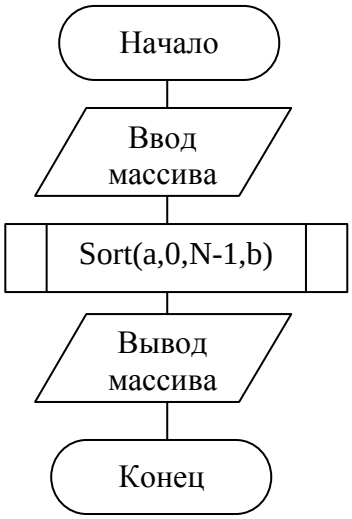
Если использовать динамическое задание массивов, можно задавать массивы переменной длинны, в этом случае нет необходимости использовать общий статический буфер. Вместо этого в каждой функции Merge можно создавать массив минимально необходимой длинны с помощью оператора new и удалять его после использования с помощью оператора delete[].

Пример для типа int:

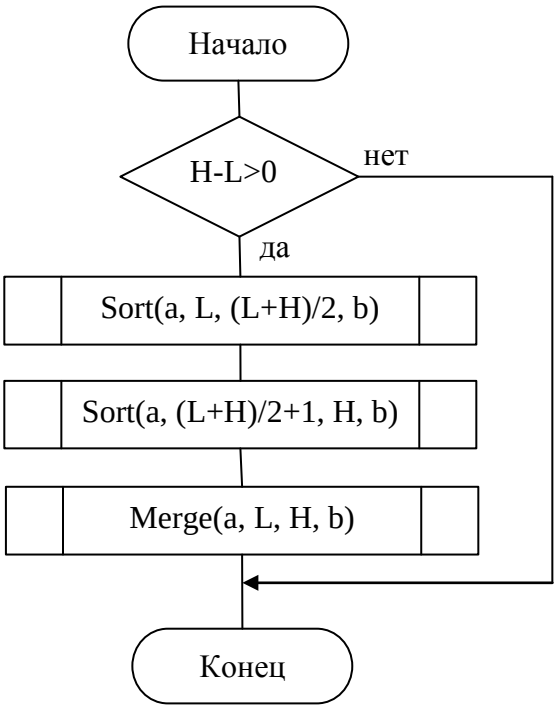
```
int *Buf = new int[H-L+1]; // Создание массива Buf, типа int размером H-L+1
delete[] Buf; // Удаление массива Buf
```

Важно! При завершении функции Merge необходимо удалить динамический буфер, поскольку автоматически он удаляться не будет и после многократного запуска функции Merge это может привести к переполнению памяти.

Блок-схема алгоритма сортировки:



Блок-схема сортирующей процедуры Sort (a,L,H,b) , реализована в рекурсивном виде, где L и H – индексы начала и конца сортируемой части массива, a – указатель на массив, b – указатель на общий буфер.



Блок-схема функции merge(p,q), реализующая непосредственное слияние двух упорядоченных массива.