Рекурсия

ЛЕКЦИЯ 7

Опеределение

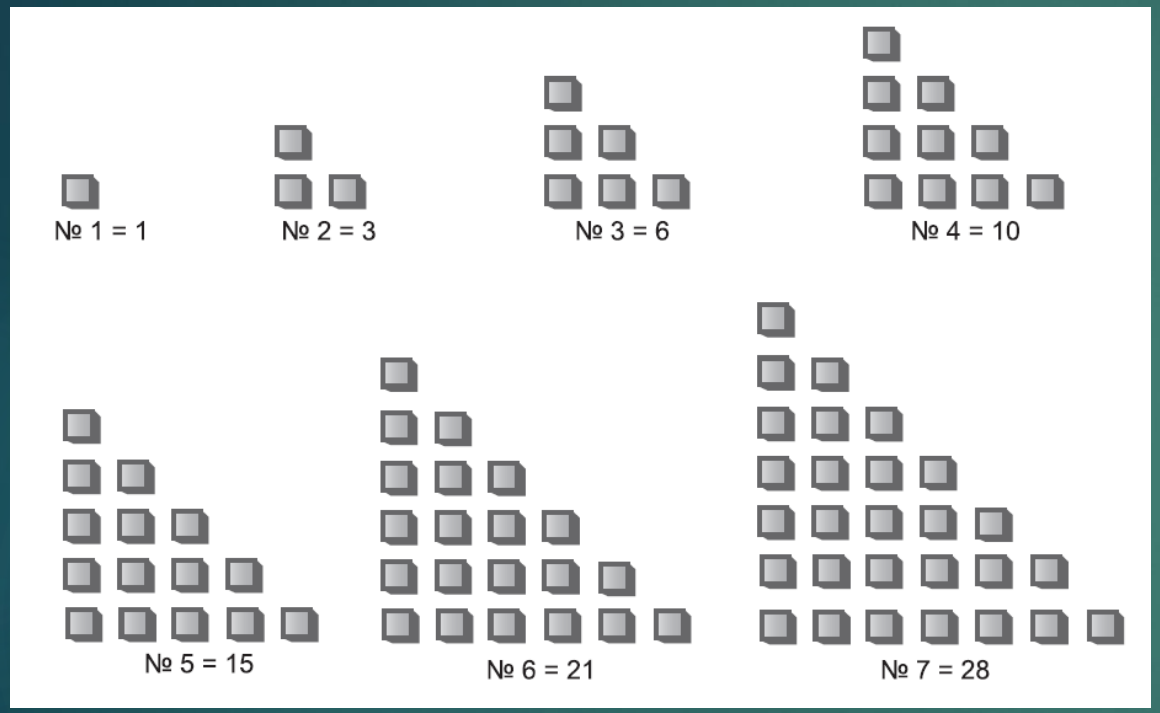
Рекурсией называется методология программирования, при которой метод (функция) вызывает сам себя.

Треугольные числа

Говорят, пифагорейцы — группа древнегреческих математиков, работавших под началом Пифагора (автора знаменитой теоремы) — ощущали мистическую связь с числовыми рядами вида 1, 3, 6, 10, 15, 21, … . Сможете ли вы угадать следующее число в этом ряду?

N-е число ряда получается прибавлением n к предыдущему числу. Таким образом, чтобы получить второе число, мы увеличиваем первое (1) на 2; 1 + 2 = 3. Третье число получается увеличением второго (3) на 3; 3 + 3 = 6 и т. д.

Треугольные числа

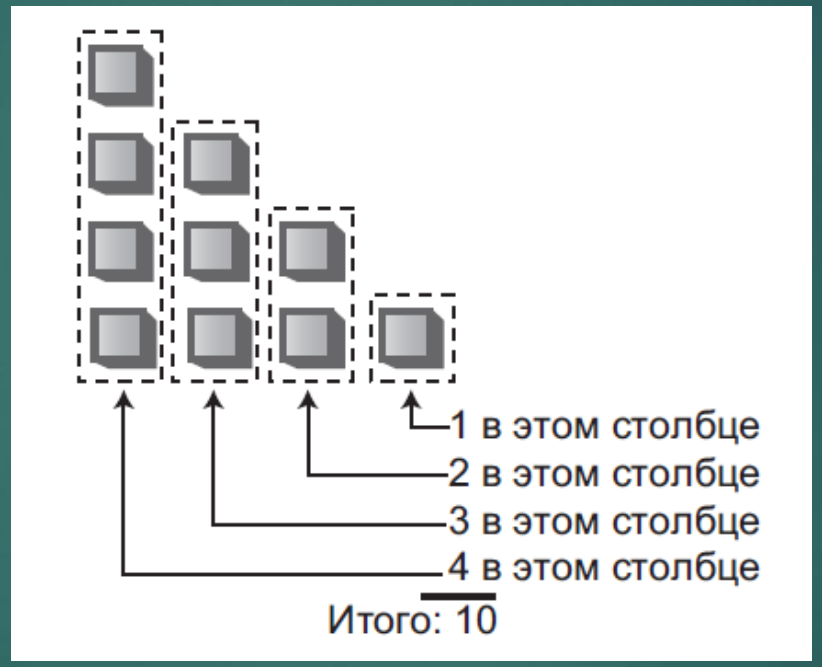


Такие ряды называются *треугольными числами*, потому что их можно наглядно представить как число объектов, которые могут быть расставлены в форме треугольника.

Вычисление n-го треугольного числа в цикле

Допустим, вы хотите определить значение произвольного *n*-го числа в серии — допустим, 4-го (10). Как его вычислить?

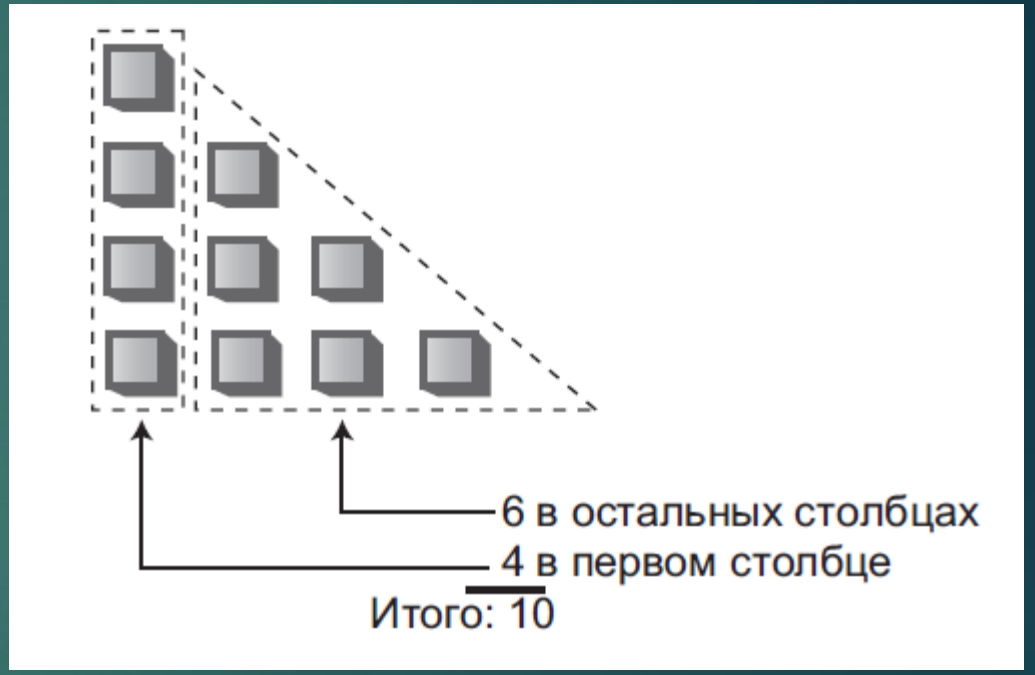
Вычисление n-го треугольного числа в цикле



Для четвертого числа первый столбец состоит из четырех квадратиков, второй — из трех и т. д. Суммирование 4 + 3 + 2 + 1 дает 10.

Вычисление n-го треугольного числа с применением рекурсии

```
public static int triangle(int n)
{
    if(n==1)
        return 1;
    else
        return( n + triangle(n-1) );
}
```



Характеристики рекурсивных методов

- Он вызывает сам себя.
- Рекурсивный вызов предназначен для решения упрощенной задачи.
- Существует версия задачи, достаточно простая для того, чтобы метод мог решить ее и вернуть управление без рекурсии.

Насколько эффективна рекурсия?

Вызов метода сопряжен с определенными непроизводительными затратами ресурсов. Управление должно передаваться из текущей точки вызова в начало метода. Кроме того, аргументы метода и адрес возврата заносятся во внутренний стек, чтобы метод мог обратиться к значениям аргументов и знать, по какому адресу следует вернуть управление.

Другой фактор снижения эффективности — затраты памяти на хранение всех промежуточных аргументов и возвращаемых значений во внутреннем стеке. Большой объем данных может создать проблемы и привести к переполнению стека.

Математическая индукция

Рекурсия является программным аналогом математической индукции — способа определения чего-либо в контексте определяемой сущности. (Этим термином также обозначается способ доказательства теорем.)

Анаграммы

Перестановкой называется расположение объектов в определенном порядке. Предположим, требуется составить полный список анаграмм заданного слова, то есть всех возможных перестановок букв (независимо от того, являются они допустимыми словами или нет). Например, для слова cat программа должна вывести следующий список анаграмм:

cat

cta

atc

act

tca

tac

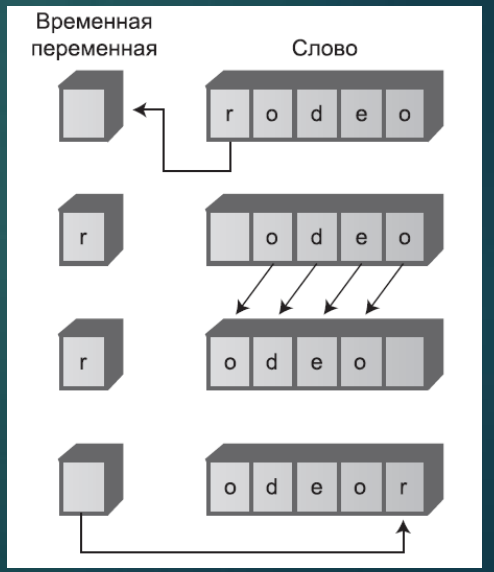
Анаграммы

Как написать программу для построения списка анаграмм слова?

1.Построить анаграммы для правых n–1 букв.

2. Выполнить циклический сдвиг всех n букв.

3. Повторить эти действия n раз.



Как строится список анаграмм правых *n* – 1 букв?

Рекурсивный метод doAnagram() получает единственный параметр с размером слова, для которого строятся анаграммы. Предполагается, что это слово содержит правые *n* букв исходного слова. Каждый раз, когда метод doAnagram() рекурсивно вызывает себя, он делает это для уменьшенного на единицу количества букв

Анаграммы слова cat

Слово	Выводить слово?	Первая буква	Остальные буквы	Действие
cat	Да	c	at	Циклический сдвиг at
cta	Да	c	ta	Циклический сдвиг ta
cat	Нет	c	at	Циклический сдвиг cat
atc	Да	a	tc	Циклический сдвиг tc
act	Да	a	ct	Циклический сдвиг ct
atc	Нет	a	tc	Циклический сдвиг atc
tca	Да	t	ca	Циклический сдвиг ca
tac	Да	t	ac	Циклический сдвиг ac
tca	Нет	t	ca	Циклический сдвиг tca
cat	Нет	c	at	Завершение

```
public static void doAnagram(int newSize)
{
    if(newSize == 1) // Если слово слишком маленькое,
        return; // не продолжать.

    for(int j=0; j<newSize; j++) // Для каждой позиции
    {
        doAnagram(newSize-1); // Построить анаграммы остальных букв
        if(newSize==2) // Если внутреннее состояние
            displayWord(); // Вывести слово

        rotate(newSize); // Циклический сдвиг всего слова
    }
}
```

```
public static void rotate(int newSize)
{
    int j;
    int position = size - newSize;
    char temp = arrChar[position]; // Сохранение первой буквы

    for(j=position+1; j<size; j++) // Сдвиг остальных букв влево
        arrChar[j-1] = arrChar[j];

    arrChar[j-1] = temp; // Перемещение первой буквы
}
```

```
public int find(long searchKey)
{
    int lowerBound = 0;
    int upperBound = nElems-1;
    int curIn;

    while(true){
        curIn = (lowerBound + upperBound ) / 2;
        if(a[curIn]==searchKey)
            return curIn; // Элемент найден
        else if(lowerBound > upperBound)
            return nElems; // Элемент не найден
        else // Деление диапазона
        {
            if(a[curIn] < searchKey)
                lowerBound = curIn + 1; // В верхней половине
            else
                upperBound = curIn - 1; // В нижней половине
        }
    }
}
```

Рекурсивный двоичный поиск

```
private int recFind(long searchKey, int lowerBound, int upperBound){
    int curIn;
    curIn = (lowerBound + upperBound ) / 2;

    if(a[curIn]==searchKey)
        return curIn; // Элемент найден
    else if(lowerBound > upperBound)
        return nElems; // Элемент не найден
    else // Деление диапазона
    {
        if(a[curIn] < searchKey) // В верхней половине
            return recFind(searchKey, curIn+1, upperBound);
        else // В нижней половине
            return recFind(searchKey, lowerBound, curIn-1);
    }
}
```

Рекурсивный двоичный поиск

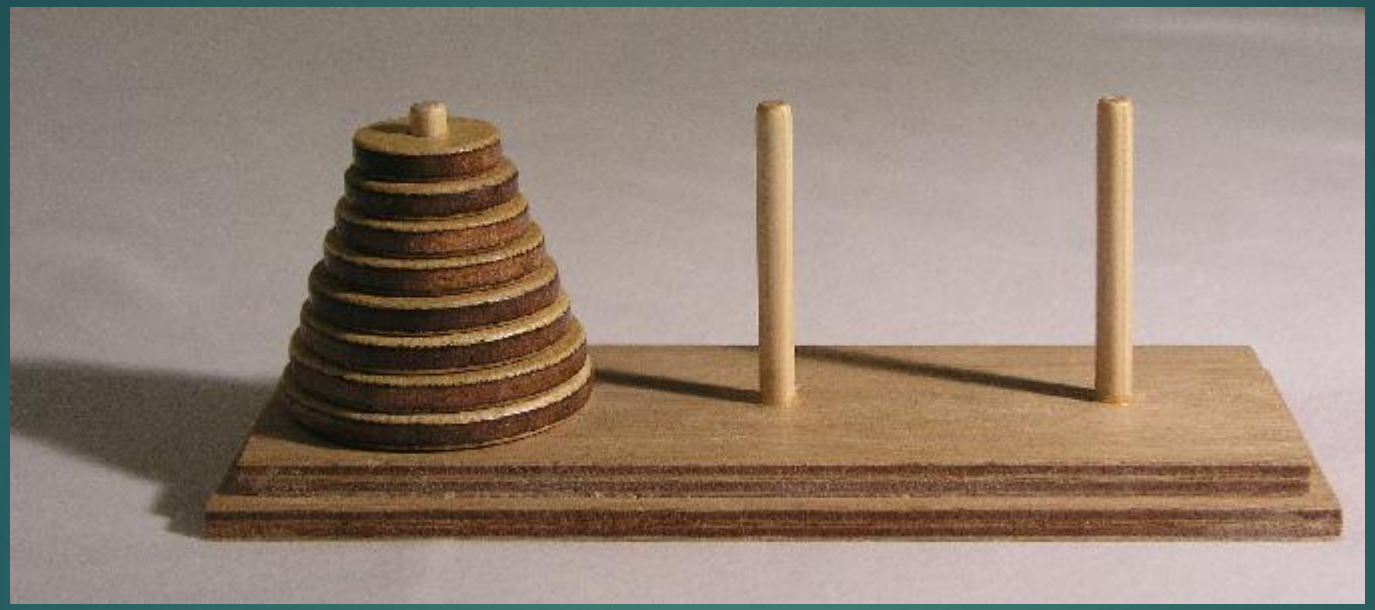
Пользователь класса (представленный методом main()) при вызове find() может не знать, сколько элементов содержит массив; в любом случае его не следует обременять поиском начальных значений upperBound и lowerBound. По этой причине в программе определяется вспомогательный открытый метод find(), который вызывается методом main() с единственным аргументом — искомым значением ключа.

```
public int find(long searchKey)
{
    return recFind(searchKey, 0, nElems-1);
}
```

Алгоритмы последовательного разделения

Рекурсивный двоичный поиск является примером *алгоритма последовательного разделения*. Большая задача делится на две меньшие задачи, решаемые по отдельности. Меньшие задачи решаются одинаково: каждая из них делится на две еще меньшие задачи. Процесс продолжается до тех пор, пока не доберется до базового ограничения, которое тривиально решается без дальнейшего деления.

Ханойская башня



Все диски имеют разный диаметр, а в середине у них просверлено отверстие, что позволяет складывать их в стопки. Изначально все диски находятся на стержне A.

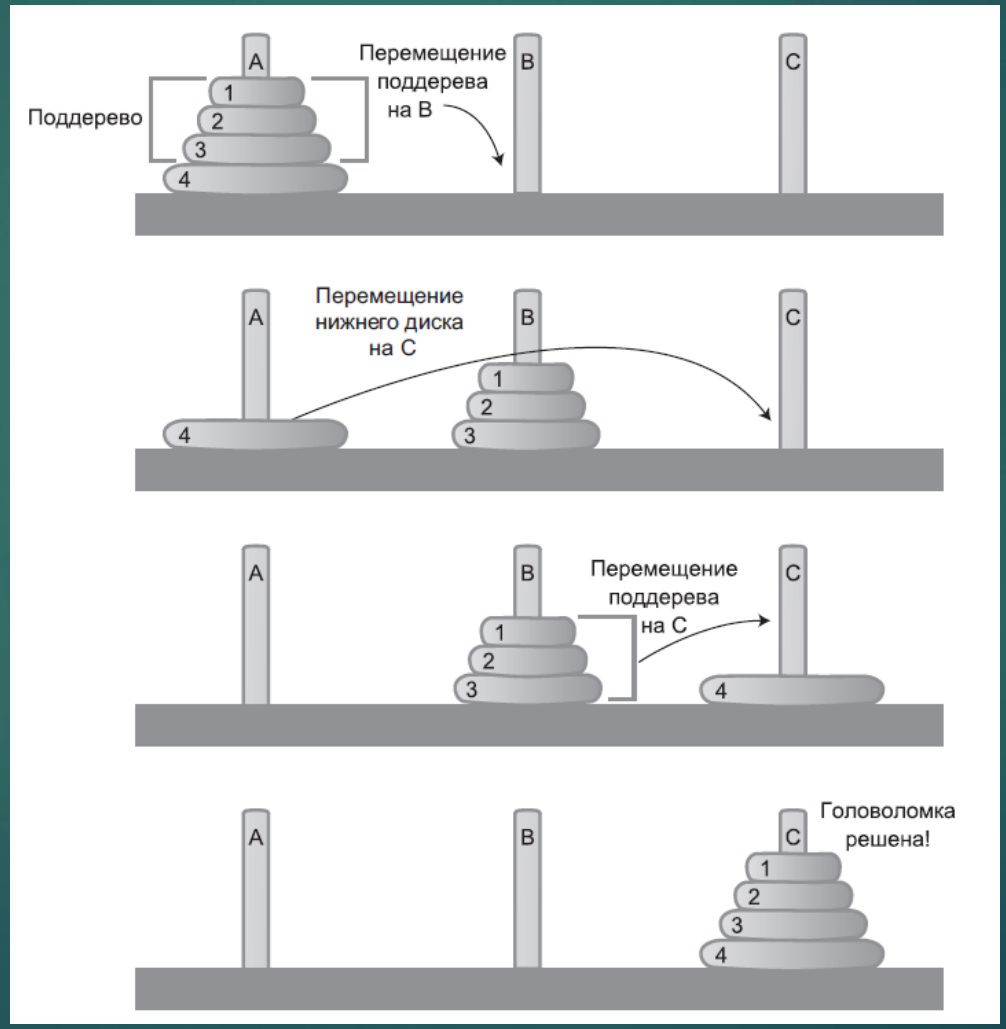
Цель головоломки — переложить все диски со стержня A на стержень C. Диски перекладываются по одному, причем запрещается класть диск на другой диск меньшего размера.

Рекурсивный алгоритм

Предположим, требуется переместить все диски с исходного стержня (назовем его S) на приемный стержень (D). Также имеется промежуточный стержень (I). На стержне S находятся *n* дисков. Алгоритм перемещения выглядит так:

1. Переместить поддерево, состоящее из верхних *n* – 1 дисков, с S на I.
2. Переместить оставшийся (самый большой) диск с S на D.
3. Переместить поддерево с I на D.

Рекурсивное решение головоломки



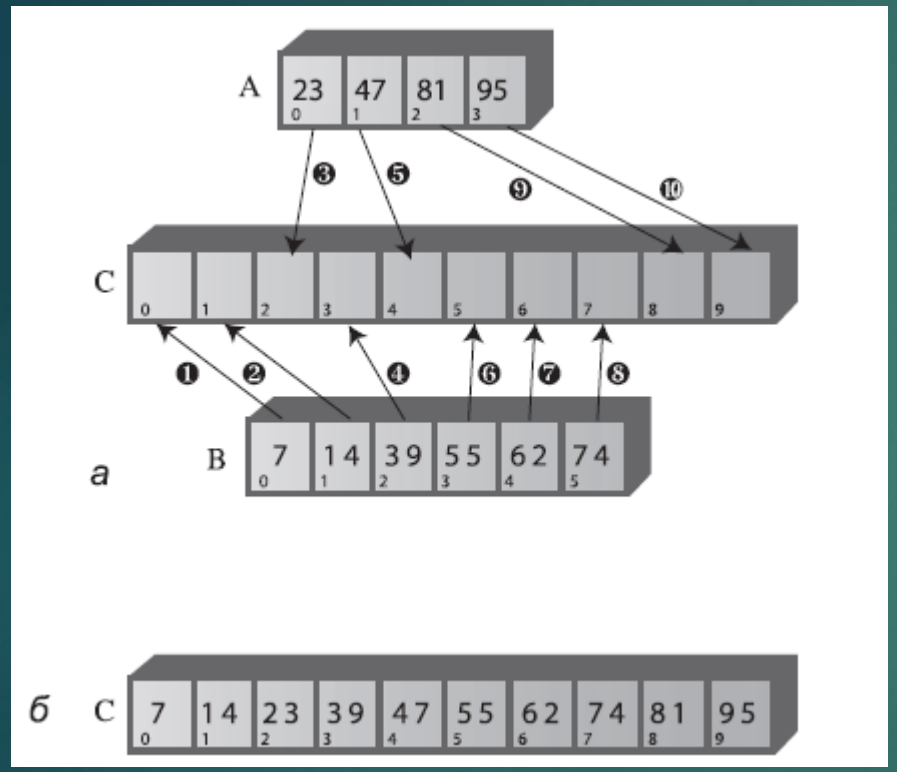
Рекурсивное решение головоломки

```
public static void doTowers(int topN, char from, char inter, char to){


    if(topN==1)
        System.out.println("Disk 1 from " + from + " to "+ to);
    else{
        doTowers(topN-1, from, to, inter); // from-->inter
        System.out.println("Disk " + topN + " from " + from + " to "+ to);
        doTowers(topN-1, inter, from, to); // inter-->to
    }
}
```

Enter (3 disks): s=A, i=B, d=C
    Enter (2 disks): s=A, i=C, d=B
        Enter (1 disk): s=A, i=B, d=C
            Base case: move disk 1 from A to C
        Return (1 disk)
        Move bottom disk 2 from A to B
        Enter (1 disk): s=C, i=A, d=B
            Base case: move disk 1 from C to B
        Return (1 disk)
    Return (2 disks)
    Move bottom disk 3 from A to C
    Enter (2 disks): s=B, i=A, d=C
        Enter (1 disk): s=B, i=C, d=A
            Base case: move disk 1 from B to A
        Return (1 disk)
        Move bottom disk 2 from B to C
        Enter (1 disk): s=A, i=B, d=C
            Base case: move disk 1 from A to C
        Return (1 disk)
    Return (2 disks)
Return (3 disks)

Сортировка слиянием



Центральное место в алгоритме сортировки слиянием занимает слияние двух предварительно отсортированных массивов. В результате слияния двух сортированных массивов A и B создается третий массив C, который содержит все элементы A и B, также расположенные в порядке сортировки.

Эффективность

Если пузырьковая сортировка, сортировка методом вставки и сортировка методом выбора выполняются за время O(N 2), то сортировка слиянием выполняется за время O(N × logN).

Например, если N (количество сортируемых элементов) равно 10 000, то значение N^2 будет равно 100 000 000, а N × logN — всего 40 000. Если сортировка слиянием для такого количества элементов займет 40 секунд, то сортировка методом вставок потребует около 28 часов. Кроме того, сортировка слиянием относительно легко реализуется.

Недостаток сортировки слиянием заключается в том, что она требует выделения в памяти дополнительного массива с размером, равным размеру сортируемого массива.

```
// Слияние массивов A и B в массив C
public static void merge( int[] arrayA, int sizeA,
int[] arrayB, int sizeB,
int[] arrayC ){

    int aDex=0, bDex=0, cDex=0;

    while(aDex < sizeA && bDex < sizeB) // Ни один из массивов не пуст
        if( arrayA[aDex] < arrayB[bDex] )
            arrayC[cDex++] = arrayA[aDex++];
        else
            arrayC[cDex++] = arrayB[bDex++];

    while(aDex < sizeA) // Массив arrayB пуст,
        arrayC[cDex++] = arrayA[aDex++]; // в arrayA остались элементы

    while(bDex < sizeB) // Массив arrayA пуст,
        arrayC[cDex++] = arrayB[bDex++]; // в arrayB остались элементы
}
```

Сортировка слиянием

Метод merge() содержит три цикла while.

- Первый цикл перебирает массивы arrayA и arrayB, сравнивает элементы и копирует меньший из них в arrayC.
- Второй цикл используется в ситуации, в которой из массива arrayB уже были извлечены все элементы, а в массиве arrayA элементы все еще остаются. Цикл просто копирует оставшиеся элементы из arrayA в arrayC.
- Третий цикл решает аналогичную задачу, когда все элементы были извлечены из массива arrayA, а массив arrayB еще не пуст; оставшиеся элементы копируются в arrayC.

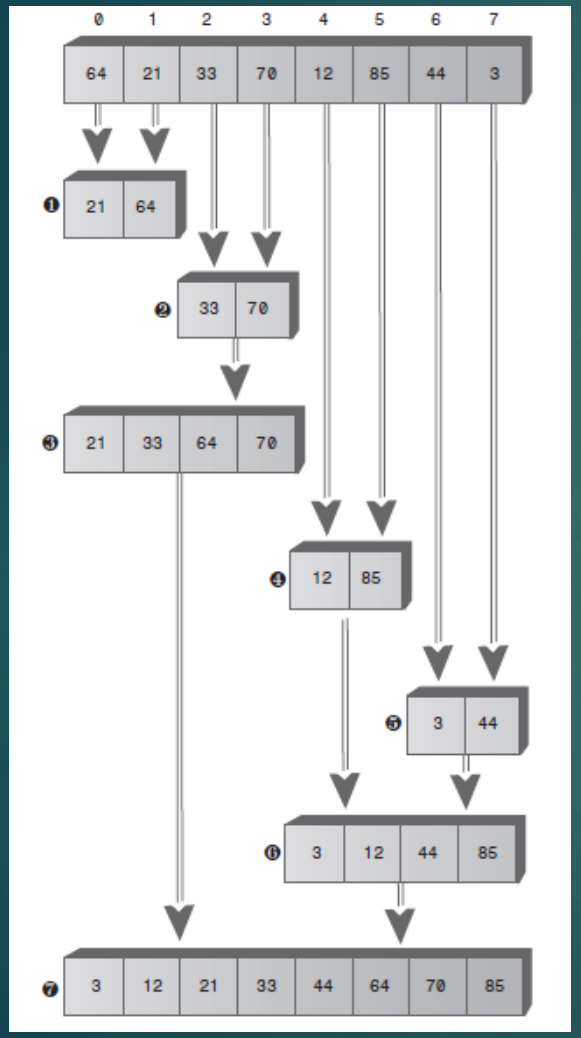
Сортировка слиянием

Идея сортировки слиянием заключается в том, чтобы разделить массив пополам, отсортировать каждую половину, а затем воспользоваться методом merge() для слияния двух половин в один отсортированный массив. Как отсортировать каждую половину?

Ответ уже известен:

половина делится на две четверти, каждая четверть сортируется по отдельности, после чего две четверти объединяются в отсортированную половину.

Сортировка слиянием



Когда метод mergeSort() возвращает управление после обнаружения двух массивов, содержащих по одному элементу, он объединяет их в отсортированный массив из двух элементов. Далее каждая пара полученных 2-элементных массивов объединяется в массив из 4-х элементов. Процесс продолжается с массивами постоянно увеличивающихся размеров до тех пор, пока весь массив не будет отсортирован.

```
private void recMergeSort(long[] workSpace, int lowerBound,
int upperBound){

    if(lowerBound == upperBound) // Если только один элемент,
        return; // сортировка не требуется.
    else { // Поиск середины
        int mid = (lowerBound+upperBound) / 2;
        // Сортировка нижней половины
        recMergeSort(workSpace, lowerBound, mid);
        // Сортировка верхней половины
        recMergeSort(workSpace, mid+1, upperBound);
        // Слияние
        merge(workSpace, lowerBound, mid+1, upperBound);
    }
}
```

```
private void merge(long[] workSpace, int lowPtr, int highPtr, int upperBound) {

     int j = 0; // Индекс в рабочей области
     int lowerBound = lowPtr;
     int mid = highPtr-1;
     int n = upperBound-lowerBound+1; // Количество элементов

     while(lowPtr <= mid && highPtr <= upperBound)
          if( theArray[lowPtr] < theArray[highPtr] )
               workSpace[j++] = theArray[lowPtr++];
          else
               workSpace[j++] = theArray[highPtr++];

     while(lowPtr <= mid)
          workSpace[j++] = theArray[lowPtr++];

     while(highPtr <= upperBound)
          workSpace[j++] = theArray[highPtr++];

     for(j=0; j<n; j++)
          theArray[lowerBound+j] = workSpace[j];
}
```

Эффективность рекурсии

Между рекурсией и стеками существует тесная связь. Компиляторы чаще всего применяют стеки для реализации рекурсии. Как уже говорилось ранее, при вызове метода компилятор заносит аргументы метода и адрес возврата (по которому должно быть передано управление при выходе из метода) в стек, после чего передает управление методу. При выходе из метода значения извлекаются из стека. Аргументы игнорируются, а управление передается по адресу возврата.

Эффективность рекурсии

```
int triangle(int n)
{ if(n==1) return 1;
 else
return( n + triangle(n-1) ); }
```

Мы разобьем этот алгоритм на отдельные операции, каждая из которых станет отдельной секцией case в конструкции switch. (В C++ и некоторых других языках аналогичная декомпозиция может быть реализована командами goto, но в Java goto не поддерживается.) Команда switch заключается в метод с именем step. При каждом вызове step() выполняется одна секция case. Многократный вызов step() в конечном итоге приведет к выполнению всего кода алгоритма.

Эффективность рекурсии

Приведенный выше метод triangle() выполняет операции двух видов. Вопервых, он выполняет арифметические действия, необходимые для вычисления треугольных чисел: сравнение n с 1 и суммирование n с результатом предыдущих рекурсивных вызовов. Однако triangle() также выполняет операции, необходимые для управления самим методом, включая передачу управления, обращение к аргументам и адресу возврата. Эти операции не видны в программном коде, они встраиваются во все методы. В общих чертах при вызове метода происходит следующее:

- Аргументы и адрес возврата заносятся в стек.
- Метод обращается к своим аргументам, читая значения с вершины стека.
- Непосредственно перед возвращением управления метод читает из стека адрес возврата, после чего извлекает этот адрес и свои аргументы из стека и уничтожает их.

Эффективность рекурсии

```
// stackTriangle.java
// Вычисление треугольных чисел с заменой рекурсии стеком
// Запуск программы: C>java StackTriangleApp
import java.io.*; // Для ввода/вывода
////////////////////////////////////////////////////////////////
class Params // Параметры, сохраняемые в стеке
 {
 public int n;
 public int returnAddress;
 public Params(int nn, int ra)
 {
 n=nn;
 returnAddress=ra;
 }
 } // Конец класса Params
```

Эффективность рекурсии

```
class StackX
 {
 private int maxSize; // Размер массива StackX
 private Params[] stackArray;
 private int top; // Вершина стека
//--------------------------------------------------------------
 public StackX(int s) // Конструктор
 {
 maxSize = s; // Определение размера массива
 stackArray = new Params[maxSize]; // Создание массива
 top = -1; // Массив пока не содержит элементов
 }
//--------------------------------------------------------------
 public void push(Params p) // Размещение элемента на вершине стека
 {
 stackArray[++top] = p; // Увеличение top, вставка элемента
 }
```

Эффективность рекурсии

```
 public Params pop() // Извлечение элемента с вершины стека
 {
 return stackArray[top--]; // Обращение к элементу, уменьшение top
 }
//--------------------------------------------------------------
 public Params peek() // Чтение элемента на вершине стека
 {
 return stackArray[top];
 }
//--------------------------------------------------------------
 } // Конец класса StackX
class StackTriangleApp
 {
 static int theNumber;
 static int theAnswer;
 static StackX theStack;
 static int codePart;
 static Params theseParams;
```

Эффективность рекурсии

```
public static void main(String[] args) throws IOException
{
System.out.print("Enter a number: ");
theNumber = getInt();
recTriangle();
System.out.println("Triangle="+theAnswer);
}
//--------------------------------------------------------
public static void recTriangle()
{
theStack = new StackX(10000);
codePart = 1;
while( step() == false) // Вызывать, пока step() не вернет true
; // Пустое тело цикла
}
```

Эффективность рекурсии

```
 public static boolean step()
{
switch(codePart)
{
case 1: // Исходный вызов
theseParams = new Params(theNumber, 6);
theStack.push(theseParams);
codePart = 2;
break;
case 2: // Вход в метод
theseParams = theStack.peek();
if(theseParams.n == 1) // Проверка
{
theAnswer = 1;
codePart = 5; // Выход
}
```

Эффективность рекурсии

```
else
 codePart = 3; // Рекурсивный вызов
 break;
 case 3: // Вызов метода
 Params newParams = new Params(theseParams.n - 1, 4);
 theStack.push(newParams);
 codePart = 2; // Вход в метод
 break;
 case 4: // Вычисление
 theseParams = theStack.peek();
 theAnswer = theAnswer + theseParams.n;
 codePart = 5;
 break;
```

Эффективность рекурсии

```
case 5: // Выход из метода
theseParams = theStack.peek();
codePart = theseParams.returnAddress; // (4 или 6)
theStack.pop();
break;
case 6: // Точка возврата
return true;
}
return false;
}
```

Эффективность рекурсии

```
 public static String getString() throws IOException
 {
 InputStreamReader isr = new InputStreamReader(System.in);
 BufferedReader br = new BufferedReader(isr);
String s = br.readLine();
 return s;
 }
//-------------------------------------------------------
 public static int getInt() throws IOException
 {
 String s = getString();
 return Integer.parseInt(s);
 }
//-------------------------------------------------------
 }
```

Итоги

- Рекурсивный метод повторно вызывает сам себя с разными аргументами.
- При некоторых значениях аргументов рекурсивный метод возвращает управление без дальнейших рекурсивных вызовов. Это называется *базовым ограничением*.
- При возврате управления рекурсивным вызовом наибольшей глубины происходит процесс «раскрутки» — незавершенные вызовы завершаются по убыванию вложенности, от внутренних вызовов к исходному.
- Рекурсивное решение может оказаться неэффективным. Иногда его удается заменить простым циклом или решением на базе стека.