

14 Рекурсия

14.1 Основные понятия

Рекурсия — определение, описание, изображение какого-либо объекта или процесса внутри самого этого объекта или процесса.

В математике рекурсия имеет отношение к методу определения функций и числовых рядов: рекурсивно заданная функция определяет своё значение через обращение к себе самой с другими аргументами.

Классический пример: рекурсивно-определённый факториал целого неотрицательного числа:

$$n! = \begin{cases} 1 & \text{для } n = 0, \\ n * (n - 1)! & \text{для } n > 0 \end{cases}$$

Поскольку n , по определению, целое неотрицательное число, через n рекурсивных обращений вычисление функции гарантированно придёт к частному случаю $0! = 1$, на котором рекурсия прекратится. Таким образом, несмотря на рекурсивность определения, вычисление функции для любого аргумента из области определения окажется конечным.

14.2 Нисходящая рекурсия

В программировании подпрограмма может вызывать сама себя, и тогда подпрограмма будет являться рекурсивной.

Рассмотрим пример вычисления факториала по описанной ранее формуле при помощи функции:

```
//Функция расчета факториала
//при помощи рекурсии
//Входные значения:
//  n - значение аргумента,
//      должно быть неотрицательным
//Входные значения:
//  f_down - значение факториала
1 function f_down(n: integer):integer;
2 begin
3   if n = 0 then //терминальная ситуация
4     f_down := 1
5   else
6     f_down := n * f_down(n - 1);
7 end;
```

В строке (3) описана так называемая **терминальная ситуация** — это условие в рекурсивном алгоритме вычислений при котором значение может быть вычислено без рекурсивного обращения. Для вычисления факториала такой терминальной ситуацией будет вычисление $0!$. В строке (6) в операторе присваивания в левой части стоит имя функции, а в правой — выражение, содержащее обращение к этой же функции с аргументом $(n - 1)$.

Добавим в текст операторы для промежуточных вычислений и вывода, чтобы посмотреть, как будет вычисляться значение факториала:

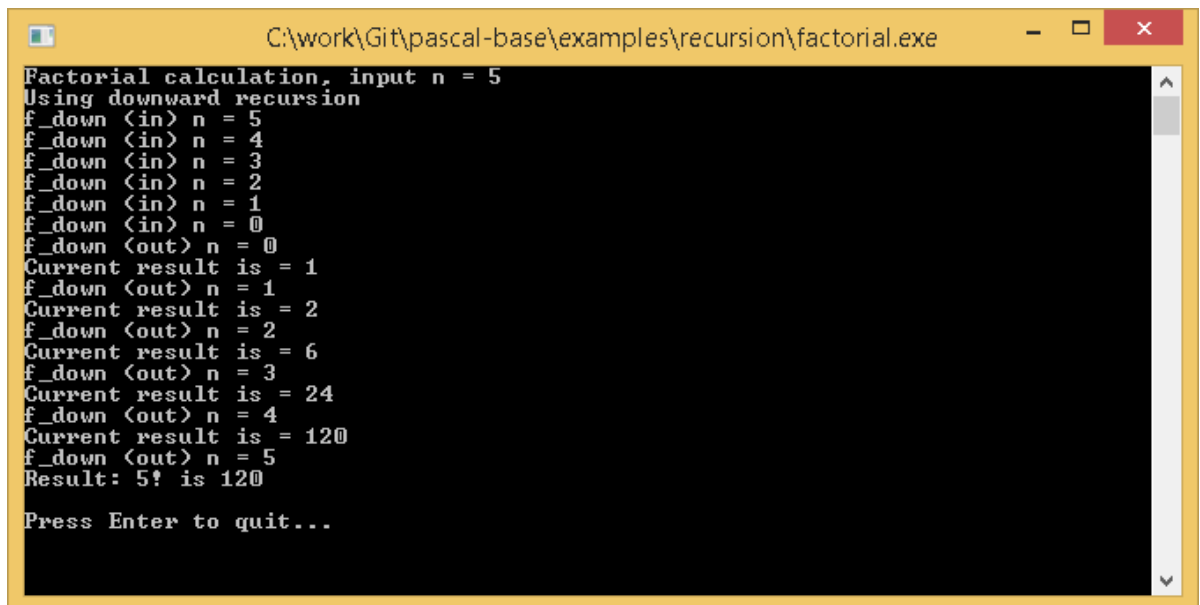
```
1 function f_down(n: integer):integer;
2 var
3   r: integer;
4 begin
5   WriteLn('f_down (in) n = ', n);
```

```

6   if n = 0 then //терминальная ситуация
7       f_down := 1
8   else
9       begin
10          //вычисление по формуле  $n! = n \cdot (n-1)!$ 
11          r := n * f_down(n - 1);
12          Writeln('Current result is = ', r);
13          f_down := r;
14      end;
15      Writeln('f_down (out) n = ', n);
16 end;
17 var
18     f, n: integer;
19 begin
20     Write('Factorial calculation, input n = ');
21     ReadLn(n);
22     Writeln('Using downward recursion');
23     f := f_down(n);
24     Writeln('Result: ', n, '! is ', f);
25     Writeln;
26     Write('Press Enter to quit...');
27     ReadLn;
28 end.

```

В функции мы сначала в строке (5) печатаем значение аргумента, потом - значение вычисленного результата (12), потом — значение аргумента при завершении функции. Результат показан на рисунке 1.



```

C:\work\Git\pascal-base\examples\recursion\factorial.exe
Factorial calculation, input n = 5
Using downward recursion
f_down (in) n = 5
f_down (in) n = 4
f_down (in) n = 3
f_down (in) n = 2
f_down (in) n = 1
f_down (in) n = 0
f_down (out) n = 0
Current result is = 1
f_down (out) n = 1
Current result is = 2
f_down (out) n = 2
Current result is = 6
f_down (out) n = 3
Current result is = 24
f_down (out) n = 4
Current result is = 120
f_down (out) n = 5
Result: 5! is 120
Press Enter to quit...

```

Рис. 1: Результат выполнения программы расчета факториала. Нисходящая рекурсия

На рисунке видно, что сначала идут последовательные вызовы функции с уменьшением параметра n . Потом вычисление доходит до терминальной ситуации при $n = 0$, и затем начинается последовательный возврат значений и завершение вызовов функций. После завершения вызова исходной функции в основную программу мы получаем результат - $5! = 120$.

Данный вид рекурсии называется **нисходящей рекурсией** — в ней процесс вычислений осуществляется на «обратном ходе» вычислительного процесса.

14.3 Восходящая рекурсия

Рассмотрим текст программы, в которой реализована **восходящая рекурсия** вычисления факториала.

```
//Процедура расчета факториала при помощи восходящей рекурсии
//Входные значения:
//  n - значение аргумента,
//      должно быть неотрицательным
//Входные значения:
//  res - накапливаемое значение факториала
1 procedure f_up(n: integer; var res: integer);
2 begin
3   WriteLn('f_up (in) n = ', n);
4   if n = 0 then //терминальная ситуация
5     exit
6   else
7     begin
8       Writeln('Current (in) result is = ', res);
9       res := res * n;
10      f_up(n-1, res);
11      Writeln('Current (out) result is = ', res);
12    end;
13   WriteLn('f_up (out) n = ', n);
14 end;
15 var
16   f, n: integer;
17 begin
18   Write('Factorial calculation, input n = ');
19   ReadLn(n);
20   Write('Using upward recursion');
21   //начальное значение f должно быть инициализированно
22   f := 1;
23   f_up(n, f);
24   WriteLn('Result: ', n, '! is ', f);
25   Writeln;
26   Write('Press Enter to quit...');
27   ReadLn;
28 end.
```

В данной программе имеются следующие отличия:

- вместо функции используется процедура, так как для передачи рассчитанного значения будет использоваться параметр подпрограммы *res*;
- непосредственно вычисление факториала будет происходить в строке (9), где текущее рассчитанное значение будет умножено на параметр *n*;
- далее в строке (10) происходит рекурсивный вызов, для расчета факториала со значением параметра $n - 1$;
- так как для расчета используется параметр-переменная, то необходима ее предварительная инициализация (в строке (22));

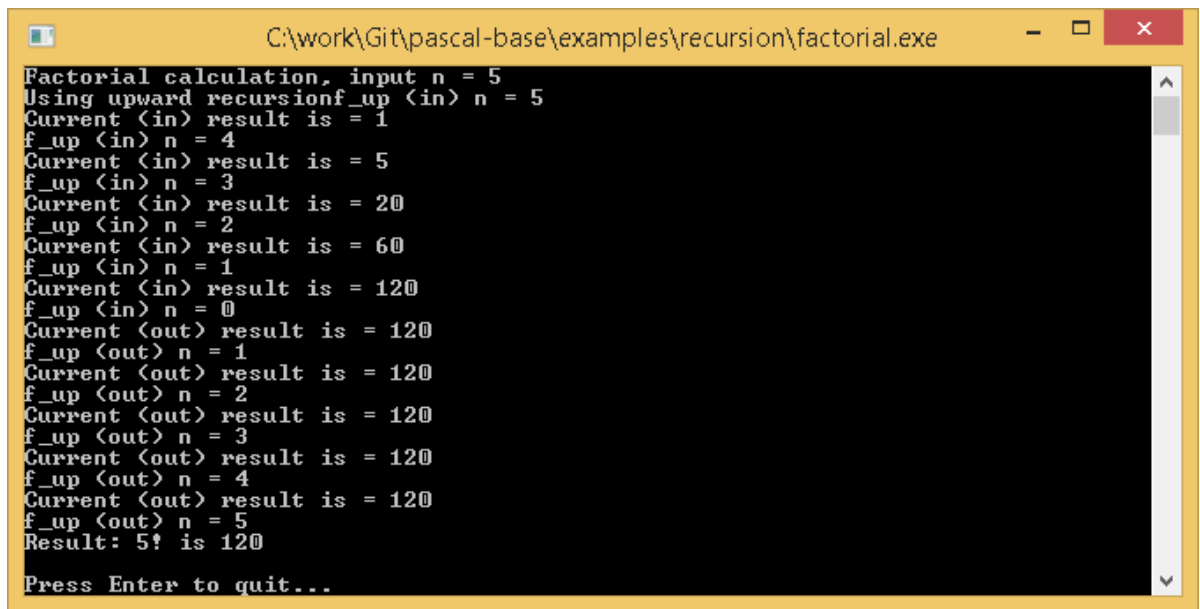


Рис. 2: Результат выполнения программы расчета факториала. Восходящая рекурсия

На рисунке 2 видно, что вычисление происходит на прямом ходе рекурсии, факториал вычисляется на шестом рекурсивном вызове, после чего все процедуры последовательно завершаются.

Правильно написанная рекурсивная функция должна гарантировать, что через конечное число рекурсивных вызовов будет достигнуто выполнение условия прекращения рекурсии, в результате чего цепочка последовательных рекурсивных вызовов прервётся и выполнится возврат.

Пример с факториалом — во многом искусственный, так как существует простой итерационный способ вычисления факториала как произведения последовательных натуральных чисел:

$$n! = \prod_{i=1}^n i$$

И алгоритм вычисления очень прост:

```
1 var
2   i, n, f: integer;
3 begin
4   Write('Factorial calculation, input n = ');
5   ReadLn(n);
6   Write('Using iterations:');
7   f := 1;
8   for i:=2 to n do
9     f := f * i;
10  Write('Press Enter to quit...');
11  ReadLn;
12 end.
```

Несмотря на то, что описание алгоритма в рекурсивном виде во многих задачах (которые будут рассмотрены далее) выглядит кратко и элегантно, на практике реализацию алгоритма делают при помощи итерационных процессов.

Реализация рекурсивных вызовов функций опирается на механизм стека вызовов — адрес возврата и локальные переменные функции записываются в стек, благодаря чему каждый

следующий рекурсивный вызов этой функции пользуется своим набором локальных переменных и за счёт этого работает корректно. Объем стека ограничен, и если глубина рекурсии будет большой, то в стеке может закончиться место, что приведет к ошибке **stack overflow**.

Теоретически, любую рекурсивную функцию можно заменить циклом и стеком.

14.4 Параллельная рекурсия

Рекурсия называется **параллельной**, если в рекурсивной ветви алгоритма происходит несколько рекурсивных вызовов.

Рассмотрим пример вычисления n -го числа Фибоначчи. Числа Фибоначчи задаются при помощи следующей рекурсивной формулы:

$$\begin{cases} f_0 = 1, \\ f_1 = 1, \\ f_n = f_{n-1} + f_{n-2} \text{ для } n > 1 \end{cases}$$

Напишем рекурсивный алгоритм вычисления n -го числа Фибоначчи:

```
//Рекурсивная функция вычисления n-го
//числа Фибоначчи
//Входные параметры:
//  n - номер числа (n>=0)
//Выходное значение:
//  f_req - число Фибоначчи
1 function f_req(n: integer): integer;
2 begin
3   if (n = 0) or (n = 1) then
4     f_req := 1
5   else
6     f_req := f_req(n-1) + f_req(n-2)
7 end;
```

В строке (6) в правой части оператора присваивания находятся два вызова функции : `f_req(n-1)` и `f_req(n-2)`. Несмотря на название — «параллельная рекурсия», эти вызовы не будут выполняться одновременно. Доработаем наш алгоритм, добавив вывод информации о вызове функций и расчет общего количества вызовов.

```
//Рекурсивная функция вычисления n-го
//числа Фибоначчи
//Входные параметры:
//  n - номер числа (n>=0)
//Выходное значение:
//  f_req - число Фибоначчи
//  n_count - глобальная переменная,
//             счетчик вызовов функции
1 var
2   n_count: integer; //счетчик итераций для рекурсивной функции
3 function f_req(n: integer): integer;
4 begin
5   inc(n_count);
6   WriteLn('Call f (', n, ')');
7   if (n = 0) or (n = 1) then
8     f_req := 1
```

```

9   else
10      f_req := f_req(n-1) + f_req(n-2)
11 end;
12 var
13   n, fn: integer;
14 begin
15   Write('Input n = ');
16   ReadLn(n);
17   n_count := 0;
18   fn := f_req(n);
19   WriteLn(n, '-th Fib number is ', fn);
20   WriteLn('We need ', n_count, ' function call in requsive case!');
21 end.

```

Результат запуска рекурсивной программы для расчета числа Фибоначчи представлен на рисунке 3. Для расчета f_7 потребовался 41 вызов функции `f_req(n)`.

Для вычисления *седьмого* числа Фибоначчи потребовался *двадцать один* вызов функции.

Если построить график зависимости количества вызовов функции в зависимости от порядкового номера числа, то мы получим график, представленный на рисунке 4. На графике видно, что зависимость количества вызовов от числа является экспоненциальной.

The screenshot shows a window titled "C:\work\Git\pascal-base\examples\recursion\fib.exe". The output text is as follows:

```

Input n = 7
Call f (7)
Call f (6)
Call f (5)
Call f (4)
Call f (3)
Call f (2)
Call f (1)
Call f (0)
Call f (1)
Call f (2)
Call f (1)
Call f (0)
Call f (3)
Call f (2)
Call f (1)
Call f (0)
Call f (1)
Call f (4)
Call f (3)
Call f (2)
Call f (1)
Call f (0)
Call f (1)
Call f (2)
Call f (1)
Call f (0)
Call f (5)
Call f (4)
Call f (3)
Call f (2)
Call f (1)
Call f (0)
Call f (1)
Call f (2)
Call f (1)
Call f (0)
Call f (3)
Call f (2)
Call f (1)
Call f (0)
Call f (1)
7-th Fib number is 21
We need 41 function call in requsive case!
Press Enter to quit...

```

Рис. 3: Результат выполнения программы расчета числа Фибоначчи. Параллельная рекурсия

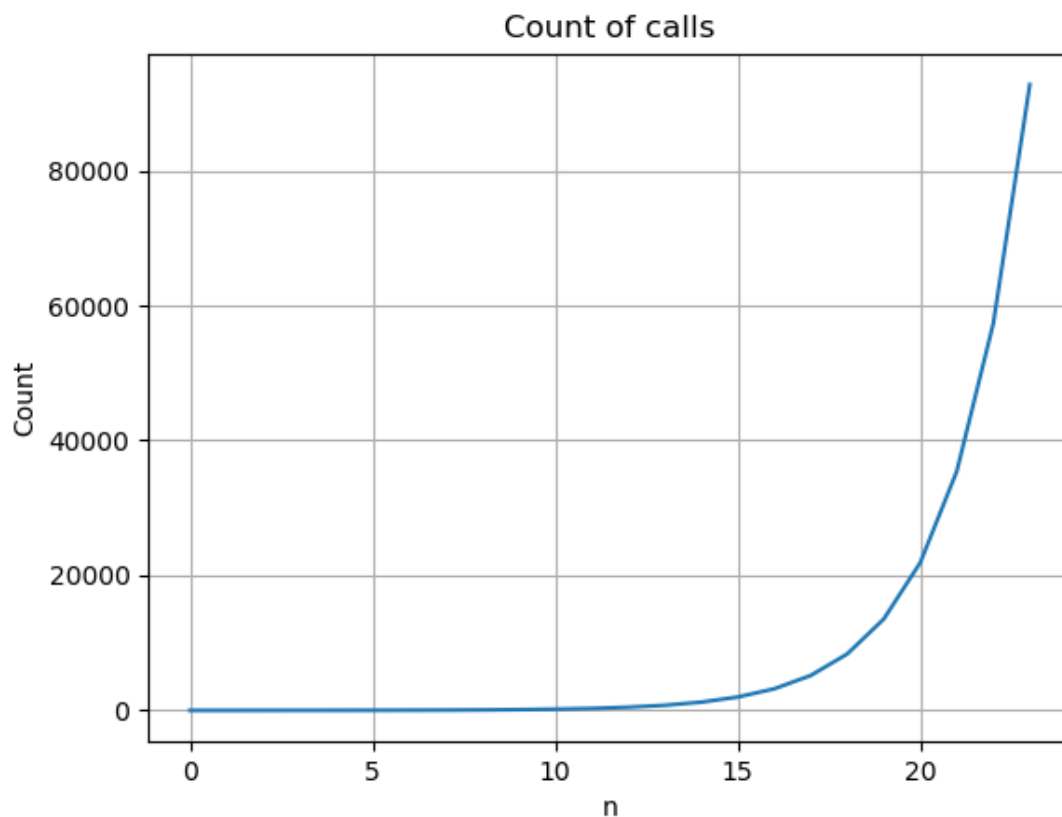


Рис. 4: Зависимость количества вызовов функции `f_req` от номера числа Фибоначчи

Для задачи вычисления числа Фибоначчи также существует простой итерационный алгоритм, который потребует всего две вспомогательные переменные для хранения значений чисел Фибоначчи, вычисленных на двух предыдущих шагах:

```
//Итерационная функция вычисления числа Фибоначчи
1 function f_iter(n: integer): integer;
2 var
3     i: integer;
4     a, b, c: integer;
5 begin
6     a := 1; b := 1;
7     for i := 2 to n do
8         begin
9             c := b + a;
10            a := b;
11            b := c;
12        end;
13    f_iter := b;
14 end;
```

Рассмотрим следующую задачу, в которой покажем, как хранение промежуточных результатов вычислений поможет сократить число рекурсивных вызовов и увеличить скорость расчетов.

Задача: необходимо определить, сколько существует различных способов разменять сумму в N рублей монетами достоинством 1, 5 и 10 рублей. Примечание: варианты, которые отличаются порядком монет (например, 1-10-5-1 и 5-10-1-1 считаются различными).

Данная задача имеет простое решение: для того, чтобы посчитать количество вариантов для размена суммы в N рублей, мы должны сложить варианты размена сумм $(N-1)$, $(N-5)$ и $(N-10)$.

```
1 function variants(sum: integer):integer;
2 begin
3   if sum < 0 then
4     variants := 0
5   else
6     if sum = 0 then
7       variants := 1
8     else
9       variants := variants(sum-1) + variants(sum-5) + variants(sum-10);
10 end;
```

В строке (9) в правой части оператора присваивания присутствует *три* рекурсивных вызова, и количество рекурсивных вызовов будет больше, чем при вычислении числа Фибоначчи. Рассмотрим пример реализации алгоритма вычислений, который будет использовать глобальный массив для хранения результатов вычислений, что приведет к соращению числа рекурсивных вызовов.

```
1 var
2   calls: integer;
3 function variants(sum: integer):integer;
4   const
5     NO_DATA = -1;
6   var
7     counts: array of integer;
8     i: integer;
9   function calc_variants(sum: integer):integer;
10  begin
11    inc(calls);
12    if sum < 0 then
13      calc_variants := 0
14    else
15      if sum = 0 then
16        calc_variants := 1
17      else
18        begin
19          if counts[sum] = NO_DATA then
20            counts[sum] := calc_variants(sum-1)
21              + calc_variants(sum-5)
22              + calc_variants(sum-10);
23          calc_variants := counts[sum];
24        end;
25    end;
26  begin
```



```

27   SetLength(counts, sum+1);
28   for i := 0 to sum do
29       counts[i] := NO_DATA;
30   variants := calc_variants(sum);
21 end;

```

Рассмотрим данную реализацию:

- в строке (2) описана глобальная переменная для подсчета числа обращений к функции;
- функция `variants` содержит блок описания констант, переменных и рекурсивную функцию `calc_variants`;
- в строке (5) описана константа `NO_DATA` для описания факта, что для заданной суммы количество вариантов еще не рассчитывалось;
- в строке (7) описан глобальный для функции `calc_variants` массив `counts`, который будет хранить результаты промежуточных вычислений;
- функция `calc_variants` выполняет вычисление по следующему алгоритму:
 1. в (11) увеличивается значение счетчика вызовов;
 2. в (12) проверяется условие $sum < 0$ (аналогичное рекурсивному вызову);
 3. в (15) проверяется условие $sum = 0$ (аналогичное рекурсивному вызову);
 4. в (19) осуществляется проверка $counts[sum] = NO_DATA$ — что для значения для sum не было рассчитано значение количества вариантов;
 5. если значение не было рассчитано, то осуществляется расчет с обращением к рекурсивным вызовам (20-22);
 6. в (20) в элемент массива будет записан результат расчета, который будет использоваться, если будет необходимость в получении ранее рассчитанного значения;
 7. в строке (23) возвращается значение (либо ранее рассчитанное, либо вычисленное);
 8. в (27) задается размер глобального массива `counts`, который используется для хранения промежуточных вычислений;
 9. в (28, 29) инициализируется массив значений (в каждый элемент записывается признак `NO_DATA`);
 10. в (30) в качестве значения функции возвращается результат расчета — через обращение к локальной функции `calc_variants`.

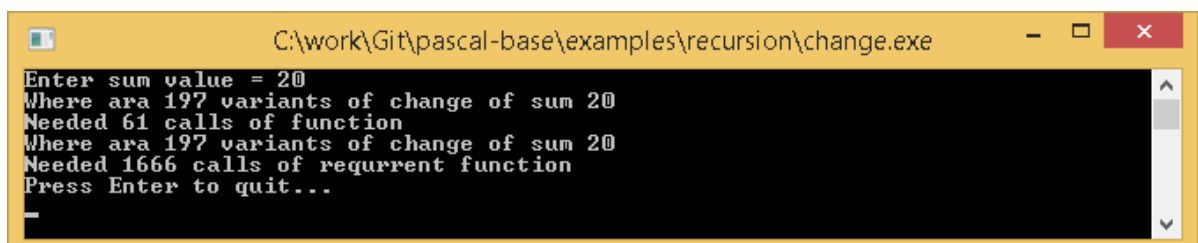


Рис. 5: Результат расчета количества вариантов размена при помощи рекурсивной функции и рекурсивной функции с использованием вспомогательного глобального массива

Использование дополнительной памяти позволило сократить количество рекурсивных вызовов за счет выделения дополнительной памяти. Как правило, во всех задачах приходится искать компромисс между использованием памяти и скоростью работы алгоритма.

```

1 function variants(sum: integer):integer;
2 begin
3   if sum < 0 then
4     variants := 0
5   else
6     if sum = 0 then
7       variants := 1
8     else
9       variants := variants(sum-1) + variants(sum-5) + variants(sum-10);
10 end;

```

В строке (9) в правой части оператора присваивания присутствует *три* рекурсивных вызова, и количество рекурсивных вызовов будет больше, чем при вычислении числа Фибоначчи. Рассмотрим пример реализации алгоритма вычислений, который будет использовать глобальный массив для хранения результатов вычислений, что приведет к сокращению числа рекурсивных вызовов.

14.5 Бинарный поиск

Рассмотрим алгоритм бинарного поиска в упорядоченном массиве значений.

Пусть задан упорядоченный (по возрастанию) набор значений a_1, a_2, \dots, a_n . Требуется определить, имеется ли в данном наборе значений значение *value*.

Алгоритм бинарного поиска можно описать следующим образом:

1. задаем начальные границы поиска `left=1`, `right=n`;
2. вычисляем середину последовательности `middle`;
3. если элемент с индексом `middle` равен искомому, то завершаем алгоритм, возвращаем признак, что элемент найден;
4. если элемент с индексом `middle` больше искомого, то осуществляем бинарный поиск в границах (`left`, `middle-1`)
5. если элемент с индексом `middle` меньше искомого, то осуществляем бинарный поиск в границах (`middle+1`, `right`)
6. поиск завершается неудачей, если левая граница оказывается больше правой.

Текст программы бинарного поиска представлен ниже.

Функция рекурсивного поиска описана в строках (3-21). В строке (7) выводится отладочная информация.

В строке (12) осуществляется расчет значения середины. Обратите внимание, что использовалась конструкция `middle := left + (right - left) div 2`, так как вычисление значения выражения `(right + left) div 2` может привести к переполнению значений - сумма `right + left` может выйти за пределы максимального значения типа.

В строках (17) и (19) осуществляются рекурсивные вызовы для поиска в левой и правой частях массива соответственно.

В строках (22-24) описана функция, которая вызывает рекурсивную функцию бинарного поиска с начальными параметрами.

В разделе операторов осуществляется проверка работы функции бинарного поиска - осуществляется поиск значений, как присутствующих в заданном наборе, так и отсутствующих в нем.

```

1 type
2   TArray = array of integer;
3 function search_req(const arr: TArray; left, right, key: integer): integer;
4 var
5   middle: integer;
6 begin
7   Writeln('find ', key, ' left = ', left, ' right = ', right);
8   if right < left then
9     search_req := -1
10  else
11    begin
12      middle := left + (right - left) div 2; //(right + left) div 2;
13      if arr[middle] = key then
14        search_req := middle
15      else
16        if arr[middle] > key then
17          search_req := search_req(arr, left, middle-1, key)
18        else
19          search_req := search_req(arr, middle+1, right, key);
20    end;
21  end;
22 function search(const arr: TArray; key: integer): integer;
23 begin
24   Result := search_req(arr, 0, length(arr), key);
25 end;
26 procedure print_array(const arr: TArray);
27 var
28   i: integer;
29 begin
30   for i := 0 to High(arr) do
31     Write(arr[i], ' ');
32   Writeln;
33 end;
34 procedure test_search(const arr: TArray; key: integer);
35 var
36   index: integer;
37 begin
38   Writeln('Trying to find ', key, ' in array: ');
39   index := search(arr, key);
40   Writeln('Result index is ', index);
41   Writeln;
42 end;
43 var a: TArray;
44 begin
45   a := TArray.Create(0, 1, 2, 2, 2, 4, 4, 5, 5, 5);
46   Write('Array is ');
47   print_array(a);
48   test_search(a, 1);
49   Write('Press Enter to quit...');
50   ReadLn;
51 end.

```

Результат выполнения программы для поиска значения 1 показан на рисунке 6.

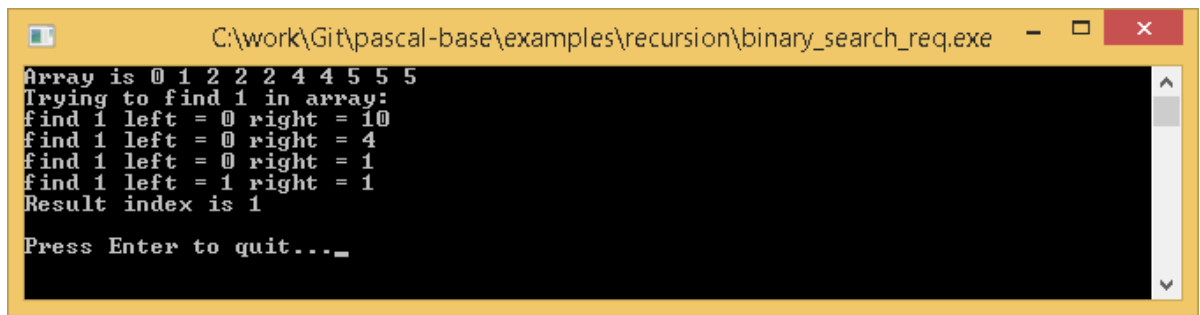


Рис. 6: Результат выполнения бинарного поиска

Несмотря на то, что данная реализация алгоритма также рекурсивная, даже такая реализация может быть использована в программах, так как с каждым последующим рекурсивным вызовом количество элементов уменьшается в два раза. Таким образом, для массива из N элементов потребуется максимум $\log_2(N)$ рекурсивных вызовов. Например, для массива из 1024 элементов потребуется максимум 5 вызовов функции, чтобы определить наличие или отсутствие искомого элемента.

Бинарный поиск также может быть реализован в виде итерационного алгоритма. В следующем примере приведены две функции, которые вычисляют границы (левую и правую), внутри которых лежит искомое(искомые) значения.

Если в наборе значений искомое значение (например, 2) встречается несколько раз (например, в наборе: 0, 1, 2, 2, 2, 3, 7, 12), то функция `left_bound` вернет значение индекса = 1, в функция `right_bound` — значение 5.

```
1 function left_bound(arr: array of integer; key: integer): integer;
2 var
3   left, right, middle: integer;
4 begin
5   left := -1;
6   right := length(arr);
7   while right - left > 1 do
8     begin
9       middle := left + (right - left) div 2;
10      if arr[middle] < key then
11        left := middle
12      else
13        right := middle;
14    end;
15   Result := left;
16 end;

17 function right_bound(arr: array of integer; key: integer): integer;
18 var
19   left, right, middle: integer;
20 begin
21   left := -1;
22   right := length(arr);
23   while right - left > 1 do
24     begin
25       middle := left + (right - left) div 2;
26       if arr[middle] <= key then
27         left := middle
```

```
28     else
29         right := middle;
30     end;
31     Result := right;
32 end;
```

На основе рекурсии построен алгоритм *быстрой сортировки*, но для его рассмотрения необходима целая лекция.

Контрольные вопросы

1. В чем отличие восходящей рекурсии от нисходящей?
2. Может ли в рекурсивной функции отсутствовать терминальная ситуация?
3. В чем недостаток рекурсивных алгоритмов по сравнению с итерационными? В чем преимущество?
4. Определите характер зависимости ($y = f(n)$) между количеством рекурсивных вызовов и входным параметром N для алгоритма вычисления числа N -ого числа Фибоначчи. Характер зависимости может быть: линейным, полиномиальным, экспоненциальным и т.д.
5. Сравните время выполнения алгоритмов бинарного поиска для итерационной и рекурсивной реализации.
6. В реализации алгоритма бинарного поиска с функциями `left_bound` и `right_bound`, определите:
 - количество искомых элементов в массиве;
 - отсутствие искомого элемента в массиве;
7. Реализуйте алгоритм поиска максимального значения в одномерном массиве при помощи рекурсии.
8. В клетку A8 на шахматной доске поставлена фишка. Фишку можно перемещать только влево или вниз. Сколько существует различных путей из клетки A8 в клетку H1?