В данном разделе будут рассмотрены некоторые алгоритмы для решения прикладных математических задач, в частности, рекуррентные соотношения, суммы функциональных рядов.

1. Рекуррентные соотношения

В математике очень часто приходится иметь дело с вычислениями значений функций в какой-то момент времени, с прогнозированием поведения какой-либо системы и т. д., определять, каким будет поведение системы в какой-либо момент времени, если известно ее состояние на текущий момент времени. Другим словами, есть последовательность чисел $a_0, a_1, \dots, a_{n-1}$ и необходимо определить a_n. Для этого используются рекуррентные соотношения.

Рекуррентные соотношения — это формулы вида

$$a_n = f(a_{n-1}, \dots, a_{n-p}),$$

позволяющие определить n-ый член последовательности $a_1, a_2, \dots, a_n$ по p ее предыдущим членам.

Также существуют операторы управления, позволяющие прерывать выполнение операторов в зависимости от некоторых условий. Самый простой пример рекуррентной последовательности — это арифметическая или геометрическая прогрессия. Например, если известна разность d арифметической прогрессии, то рекуррентное соотношение для нее записывается в виде

$$a_k = a_{k-1} + d. \tag{1}$$

Как видно, для того, чтобы найти n-ый член прогрессии, необходимо знать $n-1$-ый член прогрессии и ее разность. Следовательно, зная первый член прогрессии и разность, можно последовательно определить все последующие члены прогрессии.

Пример 0.1. Рассмотрим пример программы, вычисляющей первые `n` членов арифметической прогрессии:

Листинг 1.

```
#include <iostream>
using namespace std;
int main(){
  setlocale (LC_ALL,"Russian");                    // русская клавиатура
```

```
  int a, d, n;
  cout << "Введите первый член прогрессии\n";
  cout << "a1="; cin >> a;                    //первый член прогрессии
  cout << "Введите разность прогрессии\n";
  cout << "d="; cin >> d;                     //разность
  cout << "Введите количество членов прогрессии\n";
  cout << "n="; cin >> n;                     //количество членов прогрессии
  cout << "a" << 1 << "=" << a <<endl;        //вывод 1-ого члена прогрессии
  for (int i = 2; i <= n; i++){               //вычисление i-ого члена
    a += d;
    cout << "a" << i << "=" << a << endl;    //вывод на экран
  }
  return 0;
}
```

Результат работы программы при `a1 = 5, d = 4, n = 5`:

i	a_{i-1}	$a_i = a_{i-1} + d$
1		5
2	5	9
3	9	13
4	13	17
5	17	21

□

Рассмотрим теперь в качестве примера старинную задачу, описанную итальянским математиком Леонардо Пизанским, более известным под прозвищем Фибоначчи, в XIII веке.

> Пара новорожденных кроликов находится в некоем замкнутом пространстве (нет хищников, питания достаточно, за год ни один кролик не умирает). Сколько пар кроликов родится при этом в течении года, если природа кроликов такова, что каждый месяц пара кроликов производит на свет другую пару, а способность к производству потомства у них появляется по достижению двухмесячного возраста.

Как видно из рис. 1, первые два месяца будет одна пара кроликов, обозначенная на рис. 0 ($a_1 = a_2 = 1$). В третий месяц крольчиха произведет пару кроликов (01 на рисунке 1), т. е., будет уже две пары кроликов ($a_3 = a_1 + a_2$). На четвертый месяц родители произведут на свет еще одну пару крольчат (02), соответственно, будет три пары кроликов (родители и две пары приплода). На следующий месяц приплод принесут

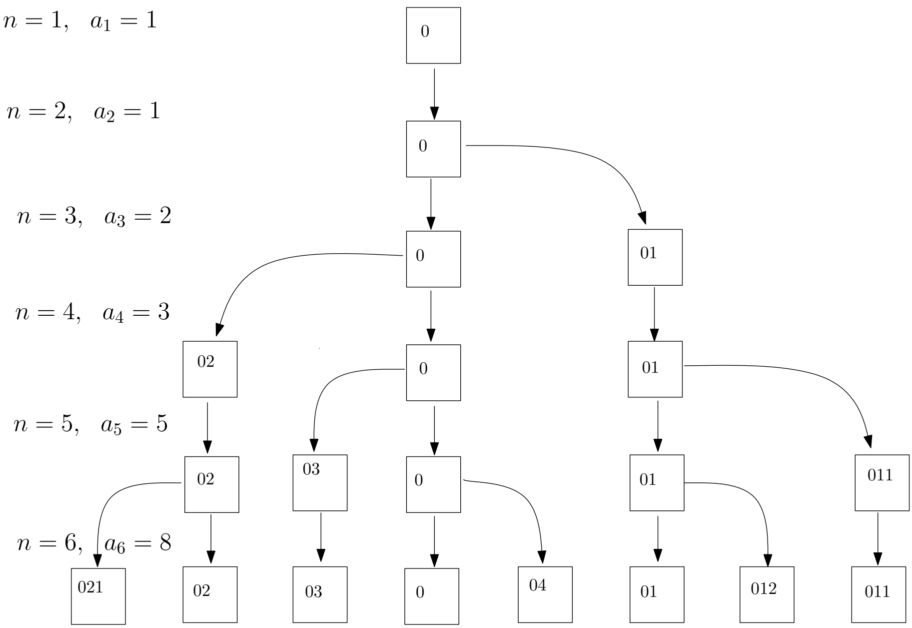

Рис. 1. Графическая интерпретация последовательности чисел Фибоначчи

родители (`03`) и первая пара крольчат (`011`), следовательно, будет две пары приплода и три уже имеющихся пары, т. е., пять пар кроликов. Получается, что в пятый месяц приплод принесут те пары кроликов, которые существовали в третий месяц. Количество пар в пятый месяц - это приплод (равный количеству пар кроликов в третий месяц (a_3)) + пары, уже бывшие в предыдущем месяце (a_4). Тогда $a_5 = a_3 + a_4$.

Таким образом, в n-ый месяц приносят приплод все пары кроликов, которые существовали в $n-2$-ом месяце. Тогда, общее число пар кроликов в n-ый месяц - это количество пар кроликов в $n-1$ месяце + приплод, равный количеству пар в $n-2$ месяце.

Т. е., число пар кроликов в n-ый месяц можно определить через рекуррентное соотношение вида:

$$a_n = a_{n-1} + a_{n-2}, \tag{2}$$

где $n = 3, 4, \ldots$ и $a_1 = a_2 = 1$.

В данном случае для того, чтобы найти `n`-ый член последовательности Фибоначчи, необходимо хранить в памяти значения двух предыдущих чисел, поэтому в программе будем использовать 3 переменных: `a1`, в которой будем хранить значение `n-2`-ого члена последовательности, `a2` — для `n-1`-ого числа и `a` — для текущего значения.

Например, при `i = 4` значение `a1 = 1 (i - 2` итерация), значение `a2 = 2 ( i - 1` итерация), после выполнения соотношения 2 значение `a = 3`.

После выполнения `4`-ой итерации значение переменной `a1` соответствует `2`-ой итерации, значение переменной `a2` соответствует `3`-ой итерации, значение переменной `a` соответствует `4`-ой итерации,

На следующем шаге необходимо найти сумму значений переменных, соответствующих `3`-ой итерации и `4`-ой итерации. Ввод новых переменных не возможен, поэтому надо переопределить уже имеющиеся переменные.

Таблица 1. Числа Фибоначчи

Номер итерации	**Шаг итерации**		
	$a1$	$a2$	a
i	$i-2$	$i-1$	i
$i+1$	$i-3$	$i-2$	$i-1$

Как видно из таблицы 1, после вычисления `i`-ого значения числа Фибоначчи необходимо переопределить переменные для `i + 1`-ого шага следующим образом:

$$\begin{aligned} a1 &= a2 \\ a2 &= a \end{aligned}$$

Например, после `4`-ой итерации значения переменных

`a1 = 1, a2 = 2, a = 3`.

После переопределения переменных значения переменных:

`a1 = 2, a2 = 3.`

В итоге, на `5`-ой итерации значение переменной `a`:

`a = a1 + a2 = 5.`

Пример 0.2. Рассмотрим пример программы, вычисляющей `n`-ое число Фибоначчи:

Листинг 2.

```
#include <iostream>
using namespace std;
int main(){
  setlocale (LC_ALL,"Russian");
  int a1 = 1, a2 = 1, a, n;
```

```
  cout << "Введите номер числа Фибоначчи\n";
  cout << "n="; cin >> n;
  if (n > 2){                        //первые два числа уже известны
    for ( int i = 3; i <= n; i++){
      a = a1 + a2;                    //вычисляем i-ое число Фибоначчи
      a1 = a2;                       //a1 - i-2-ое число на следующем шаге,
      a2 = a;                        //a2 - i-1-ое число на следующем шаге,
    }
    cout << "a" << n << "=" << a << endl; //n-ое число Фибоначчи
  }
  else cout << "a" << n << "=1\n";
  return 0;
}
```

Результат работы программы при `n = 6`:

i	$a1$	$a2$	a
1	1		
2	1	1	
3	1	1	2
4	1	2	3
5	2	3	5
6	3	5	8

□

Также в виде рекуррентного соотношения можно представить такие математические понятия, как факториал и возведение в степень натурального числа.

2. Вычисление сумм ряда

Рассмотрим теперь задачи, связанные с вычислением сумм или произведений функциональных рядов. Сумма ряда — это выражение вида

$$S = \sum_{k=1}^{n} a_k.$$

На каждом шаге к сумме добавляется значение a_k, где k — номер шага. Тогда сумму ряда на k-ом шаге можно представить в виде рекуррентного соотношения

$$S_k = S_{k-1} + a_k.$$

2.1. Простые ряды

Под простым рядом понимается ряд, каждый член которого не зависит от предыдущего значения.

Например, необходимо найти сумму следующего ряда:

$$S = 1 + 2^2 + 3^2 + 4^2 + 5^2 + \ldots + n^2.$$

На каждом шаге итерации к сумме добавляется i^2. Алгоритм для решения подобных задач похож на алгоритм нахождения арифметической прогрессии, только на каждом шаге итерации к сумме добавляется различные числа, а не одинаковые, как в случае арифметической прогрессии.

Например, для `n = 6` результат работы программы:

i	$a = i * i$	$S+ = a$
1	1	1
2	4	5
3	9	14
4	16	30
5	25	55
6	36	91

Алгоритм действий следующий:

1. На каждом шаге определяется значение `a`.

2. Полученное значение добавляется к значению переменной `S`.

Пример 0.3. Например, найдем сумму ряда

$$S = \frac{1}{\sqrt{|\sin 1|}} + \frac{1}{\sqrt{|\sin 2|}} + \frac{1}{\sqrt{|\sin 3|}} + \ldots + \frac{1}{\sqrt{|\sin n|}}.$$

Сокращенно эту сумму можно представить в виде

$$S = \sum_{k=1}^{n} \frac{1}{\sqrt{|\sin k|}}.$$

Тогда $a_k = \frac{1}{\sqrt{|\sin k|}}$ и для этой суммы на k-ом шаге можно записать рекуррентное соотношение вида

$$S_k = S_{k-1} + \frac{1}{\sqrt{|\sin k|}}.$$

Начальное значение суммы $S_0 = 0$.

Листинг 3.

```
#include <iostream>
#include <math.h>
using namespace std;
int main(){
  setlocale (LC_ALL,"Russian");
  float S=0, n;                    //начальное значение
  cout << "n="; cin >> n;
  for (float i = 1; i <= n; i++)
    S += 1 / sqrt(abs(sin(i)));  //сумма ряда
  cout << "S=" << S << endl;
   return 0;
}
```

Результат работы программы при `n = 6`:

i	a	S
1	1.09014	1.09014
2	1.04869	2.13883
3	2.66199	4.80081
4	1.1495	5.95031
5	1.02119	6.9715
6	1.8918	8.8633

□

2.2. Сложные ряды

В случае, когда члены функционального ряда представляют собой простые функции, проблем с вычислением суммы или произведения ряда не возникает. Более сложной является задача, когда члены ряда — это функции, содержащие либо факториалы, либо возведение в степень натурального числа. Например, необходимо найти сумму следующего ряда:

$$\sum_{n=1}^{k} \frac{(-1)^n x^{2n+1}}{(2n+1)!} \tag{3}$$

Факториал числа и возведение в степень, как было рассмотрено в предыдущих разделах, являются рекуррентными соотношениями. Соответственно, каждое из множителей ряда (3) необходимо вычислять через рекуррентные соотношения, это, во-первых, загромождает программу, во-вторых, нерационально, так как три цикла — это затраты времени, в-третьих, факториал — быстрорастущая функция, определенная только для `n <= 12`. Однако, выражение $\frac{x^n}{n!}$ определено для `n > 12`, если сразу вычислять отношение, а не определять по отдельности сначала числитель, потом знаменатель, а потом вычислять их частное. Поэтому рассмотрим прием, который облегчает решение таких задач.

Распишем сумму (3):

$$\sum_{n=1}^{k} \frac{(-1)^n x^{2n+1}}{(2n+1)!} = -\frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

Рассмотрим по шагам как будут вычисляться члены ряда.

Таблица 2. Вычисление суммы ряда (3)

i	a_i	соотношение (число)	соотношение (символ.)
1	$-\frac{x^3}{3!}$		
2	$\frac{x^5}{5!}$	$-\frac{x^3}{3!} \cdot \frac{-x^2}{4 \cdot 5}$	$a_1 \cdot \frac{-x^2}{2i(2i+1)}$
3	$-\frac{x^7}{7!}$	$\frac{x^5}{5!} \cdot \frac{-x^2}{6 \cdot 7}$	$a_2 \cdot \frac{-x^2}{2i(2i+1)}$
4	$\frac{x^9}{9!}$	$-\frac{x^7}{7!} \cdot \frac{-x^2}{8 \cdot 9}$	$a_3 \cdot \frac{-x^2}{2i(2i+1)}$
...			
$k-1$	$\frac{(-1)^{k-1} x^{2k-1}}{(2k-1)!}$	$\frac{(-1)^{k-2} x^{2k-3}}{(2k-3)!} \cdot \frac{-x^2}{2(k-1) \cdot (2(k-1)+1)}$	$a_{k-2} \cdot \frac{-x^2}{2i(2i+1)}$
k	$\frac{(-1)^k x^{2k+1}}{(2k+1)!}$	$\frac{(-1)^{k-1} x^{2k-1}}{(2k-1)!} \cdot \frac{-x^2}{2k \cdot (2k+1)}$	$a_{k-1} \cdot \frac{-x^2}{2i(2i+1)}$

Как видно из табл. 2 на каждом шаге член ряда умножается на один и тот же параметр, соответственно, член ряда на текущем шаге можно рекуррентно выразить через предыдущий по следующей формуле

$$a_n = a_{n-1} * d \tag{4}$$

Для (3) значение параметра $d = -\dfrac{x^2}{2n(2n+1)}$ для $n = 1, 2, \ldots, k$. Значение этого параметра можно определить, не вычисляя члены ряда, из формулы (4) $d = \dfrac{a_n}{a_{n-1}}$. Рассмотрим, как определить значение этого параметра для вычисления суммы ряда (3):

$$\begin{aligned} d &= \frac{a_n}{a_{n-1}}, \\ a_n &= \frac{(-1)^n x^{2n+1}}{(2n+1)!}, \\ a_{n-1} &= \frac{(-1)^{n-1} x^{2(n-1)+1}}{(2(n-1)+1)!} = \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!}. \end{aligned}$$

Тогда

$$d = \frac{(-1)^n x^{2n+1}(2n-1)!}{(-1)^{n-1} x^{2n-1}(2n+1)!} = \frac{-x^2(2n-1)!}{(2n-1)! * 2n * (2n+1)} = -\frac{x^2}{2n(2n+1)}.$$

Мы получили результат, аналогичный определенному из таблицы (2).

Таким образом, при вычислении сумм ряда, подобного (3), можно воспользоваться двумя рекуррентными соотношениями:

$$\begin{aligned} a_n &= a_{n-1} * d \\ S_n &= S_{n-1} + a_n \end{aligned} \tag{5}$$

Так как рекуррентных соотношений два, то и начальные значения необходимо определять как для переменной `a`, так и для `S`.

Итак, для решения задач, подобных (3), необходимо воспользоваться следующим алгоритмом:

1. Из формулы (4) определить параметр `d`.
2. Определить начальные значения `S` и `a`.
3. По формулам (5) определить сумму ряда.

Пример 0.4. Рассмотрим пример программы, вычисляющей сумму ряда (3). Замечание. Для наглядности будем выводить значения переменных на каждом шаге итерации:

Листинг 4.

```
#include <iostream>
#include <math.h>
using namespace std;
int main(){
```

```
  setlocale(LC_ALL,"Russian");
  int n;
  cout << "n="; cin >> n;
  float x;
  cout << "x="; cin >> x;
  float a = -x*x*x / 6, S = a;              //начальные значения
  cout << "i\ta\tS\n";                      //заголовок таблицы
  cout << 1 << "\t| " << a << "\t| " << S << endl; //вывод 1-ого шага
  for (int i = 2; i <= n; i++){
    a *= -x*x / ( 2*i * (2*i + 1));         //i - ый член ряда
    S += a;                                 // сумма
    cout << i << "\t| " << a << "\t| " << S << endl;
  }
  return 0;
}
```

Результат работы программы при `n = 5, x = 2`:

i	a	S
1	-1.33333	-1.33333
2	0.266667	-1.06667
3	-0.0253968	-1.09206
4	0.00141093	-1.09065
5	-5.13067e-005	-1.0907
6	1.8918	8.8633

2.3. Бесконечные суммы

Очень часто в математике возникают задачи нахождения бесконечных сумм. Аналитически можно определить предел, к которому сходится подобный ряд. Компьютер должен работать с конечными задачами, соответственно, необходимо определить условие прекращения вычисления суммы.

Из математического анализа известно, что ряд является сходящимся, если существует такое $0 < \varepsilon \ll 1$, что $|S_n - S_{n-1}| < \varepsilon$. Из рекуррентного соотношения относительно суммы известно, что $S_n = S_{n-1} + a_n$. Тогда условие прекращения цикла имеет следующий вид:

$$|S_n - S_{n-1}| = |S_{n-1} + a_n - S_{n-1}| = |a_n| < \varepsilon \tag{6}$$

Возможна ситуация, когда ряд будет расходящимся. В этом случае получится бесконечный цикл. Тогда для того, чтобы решить такую задачу, необходимо добавить проверку на количество итераций, то есть, если прошло `N` итераций, а выражение (6) ложное, то вычисление суммы прекращается.

Таким образом, условие завершения вычисления бесконечной суммы имеет следующий вид:

```
abs(a)> eps && n < N
```

где `a` — `n`-ый член ряда.

Итак, для решения задач, связанных с бесконечными суммами, необходимо воспользоваться следующим алгоритмом:

1. Из формулы (4) определить параметр `d`.

2. Определить точность вычислений (переменная `eps`) и условие выхода из цикла (если ряд расходящийся) (переменная `N`).

3. Определить начальные значения `S` и `a`.

4. По формулам (5) определить сумму ряда до тех пор, пока `a >= eps` и `i < N`.

Пример 0.5. Найти сумму

$$S = \sum_{k=1}^{\infty} \frac{x^k}{k!}.$$

Решение задачи ищется по следующему алгоритму:

1. Сначала определим

$$d = \frac{a_n}{a_{n-1}} = \frac{x^n(n-1)!}{x^{n-1}n!} = \frac{x}{n}$$

2. Определяем начальные условия $a_1 = x$; $S_1 = a_1$

3. Суммы вычисляем с помощью следующих рекуррентных соотношений:

```
for (int i = 2; abs(a)< eps && i < N; i++){
  a *= x / i;
  S += a;
}
```

Листинг 5.

```cpp
#include <iostream>
#include <math.h>
#include <iomanip>
using namespace std;
int main(){
  setlocale (LC_ALL,"Russian");
  int N = 100;
  float x, eps;
  cout << "x="; cin >> x;
  cout << "eps="; cin >> eps;
  float a = x, S = a;                       //начальные условия
  cout << left << setw(3) << "i\t" << setw(10) << "a\t"; //заголовок
  cout << setw(10) << "S" << endl;
  cout << left << setw(3) << 1 << "\t" << setw(10) << a; //1 шаг
  cout << "\t" << setw(10) << S << endl;
  for (int i = 2; abs(a) > eps && i <= N; i++){         //i-ый шаг
    S += a;
    a*= x/i;
    cout << left << setw(3) << i << "\t" << setw(10) << a;
    cout << "\t" << setw(10)<< S << endl;
  }
  return 0;
}
```

Результат работы программы для `x = 2, N = 100, eps = 0.01`:

i	a_i	S_i
1	2	2
2	2	4
3	1.33333	5.3333
4	0.666667	6
5	0.266667	6.26667
6	0.0888889	6.35556
7	0.0253968	6.38095
8	0.00634921	6.3873

Видно, что программа завершает работу, как только значение переменной `a` становится больше значение переменной `eps`. □