Глава 1

Алгоритмические задачи

Задачи данной главы можно условно разделить на четыре группы. Простейшие задачи предназначены для освоения базовых конструкций языка C; задачи на обработку последовательностей — для изучения однопроходных (рекуррентных, индуктивных) алгоритмов; задачи на работу с массивами — для получения навыков работы с динамически выделяемой памятью и отработки разнообразных алгоритмов перестановки элементов в рамках заданного участка памяти.

Последующие разделы рассматривают другие важные классы алгоритмов и приемы их реализации, часто встречающиеся в прикладных задачах.

1.1 Простейшие задачи

При решении задач данного раздела условимся, что ввод необходимых входных данных выполняется с клавиатуры (по соответствующим указаниям на экране). Программа вычисляет искомые величины (и сохраняет эти значения в отдельных переменных), а затем выводит (печатает на экран с пояснениями) полученный результат.

1.1. Найти минимум из двух целых чисел A и B.
Решение.

```
#include<stdio.h> //  Включение содержимого файла stdio.h,
                           // содержащего прототипы функций  printf(), scanf().
int main(void)
{
  int A, B, min;
  printf( ‘‘Введите A ‘‘ );   // Печать "приглашения" к вводу
  scanf( "%d", &A );         // ввод значения A с клавиатуры
  printf( ‘‘Введите B ‘‘ );
  scanf( ‘‘%d’’, &B );
  if( A < B ) { min = A; } else { min = B; }
  printf( "Минимум из A=%d и B=%d равен %d\n", A, B, min );
  return 0;
```

```
}
```

Однако этот вариант, хотя он формально решает поставленную задачу, нельзя принять за образец, поскольку он написан без оглядки на принцип процедурного программирования — вся работа реализована в единственной функции `main()`, что считается неправильным и неграмотным даже для случая простейших программ. При решении любой задачи в ней нужно выделять отдельные логические и алгоритмические части, которые отвечают за разные этапы решения, и оформлять эти части в виде отдельных функций. В данном тривиальном примере тоже можно выделить две части — ввод данных и вывод результата (т.е. общение с пользователем, интерфейсная часть) и собственно вычисление минимума (вычислительная часть). Поэтому решим данный пример заново, выделив вычислительную часть в отдельную функцию. Заодно продемонстрируем и другие конструкции и возможности языка. Решения всех последующих задач следует оформлять с учетом этого замечания, т.е. выделяя отдельные более-менее независимые части алгоритма в отдельные функции.

```
#include<stdio.h>
int Min (int x, int y);  // прототип нашей функции
int main(void)
{
  int A, B;
  printf( ‘‘Введите A и B ‘‘ );
  scanf( "%d%d", &A, &B );
  printf( " min(%d,%d) = %d\n", A, B, Min( A, B ) );
  return 0;
}
int Min (int x, int y)
{
    int z;
    z = ( x < y ) ? x : y;
    return z;
}
```

Замечание. В данном примере в печатаемый на экране текст добавлен управляющий символ новой строки `\n`, выдача которого на экран переводит каретку на следующую строку. Отметим, что `\n` *всегда следует* добавлять в конец сообщений при поиске ошибок в программе методом отладочной печати. Дело в том, что отвечающие за вывод библиотечные функции, предварительно накапливают некоторое достаточно большое количество символов в специальном буфере, и только затем передают на устройство вывода. И если программа аварийно завершается, то содержимое буфера теряется. Символ `\n` почти всегда обеспечивает досрочное проталкивание буфера печати — выдачу его содержимого на экран независимо от степени его заполнения. Гарантированное проталкивание обеспечивает функция `int fflush(stdout)`, отключить буферизацию можно с помощью функции `setbuf(stdout,NULL)`.

1.2. Найти максимум из целых чисел A и B.

1.3. Вычислить модуль целого числа A.
Указание.

```
int imod (int x){
 if(x<0)
  return (-x);
else
  return x;
}
```

1.4. Заменить содержимое A на $|A|$.
Указание. Например,

```
if(A<0)A=-A; или A = ( A < 0 ) ? (-A) : A;
```

1.5. Найти знак целого числа A, т.е. вычислить $sign(A) = \begin{cases} 1 & A > 0, \\ 0 & A = 0, \\ -1 & A < 0. \end{cases}$
Результат сохранить в переменной `int sgn`.

1.6. Для заданного действительного A найти наибольше из чисел $(10A+7)$ и $(A^2 - 2A + 1)$.
Решение.

```
#include<stdio.h>
double FindMax( double , double );        // описание прототипа функции
double F1( double A ) { return 10*A+7; }  // определение "простых" функций
double F2( double A ) { return A*A -2*A+1; }
int main(void){
  double A, Max;
  printf("Input A ");
  scanf("%lf",&A);
  Max = FindMax( F1(A), F2(A) );
  printf("Максимум F1(A) и F2(A) для A=%lf равен %lf\n", A, Max);
  return 0;
}
double FindMax ( double Y1, double Y2 )
{
  return ( Y1 > Y2 ) ? Y1 : Y2;
}
```

1.7. Для данного A найти наибольшее из трех действительных чисел $(A^2 - 10), (A-1)(A+2), (7A+1)$.

1.8. Проверить принадлежит ли вещественное число x отрезку $[a, b]$.
Решение. Приведем в качестве решения только искомую функцию. При этом ответом формально является не число, а утверждение — принадлежит или нет. В таких случаях нужно как-то представить результат работы функции в виде условных числовых значений — кода возврата. Например, в нашем случае мы можем договориться, что число 0 соответствует "попаданию в отрезок", а значения ± 1 показывают слева или справа от отрезка расположилось число x.

```
int Inside(double x, double a, double b)
{
  int answer = 0;
  if ( x > b ) answer = 1;
  if ( x < a ) answer = -1;
  return answer;
}
```

Отметим, что функция из задачи 1.5 имеет аналогичную структуру.

Замечание. Следует помнить, что в большинстве современных ЭВМ действительные числа хранятся в формате *с плавающей точкой*, что имеет свою специфику (см. далее 2). Например, значения, вычисленные по математически эквивалентным, но разным по структуре формулам могут отличаться в младших разрядах, т.е. окажутся формально разными: в данном случае при $a = 3.0$, $b = 7.0$ и введенном с клавиатуры $x = 3.0$ не гарантируется, что $x \in [a, b]$, т.е. функция вернет `answer=0`.

Также сравнение переменных типа `double` на равенство не является корректным действием, а поэтому запрещено: многие компиляторы выдают предупреждения по поводу сравнения вещественных чисел на равенство/неравенство, а иногда такая операция вообще запрещается настройками компилятора.

Отметим, что не рекомендуется использовать конструкции типа

```
 if((a-x)*(b-x)<=0){
// x из [a,b]
 }
 else{
// x вне [a,b]
 }
```

1.9. Проверить принадлежит ли число x объединению отрезков [1, 11], [101, 1001].

1.10. Проверить принадлежит ли число x объединению и/или пересечению отрезков $[A1, B1]$, $[A2, B2]$.

1.11. Проверить принадлежит ли число x интервалам $(A1, B1)$, $(A2, B2)$, $(A3, B3)$, и если принадлежит, то указать номер каждого такого интервала.

Указание. Решение оформить в виде трехкратного последовательного вызова функции.

1.12. Вычислить $|A| + |B|$, $||A| - |B||$, $||A + B| - |A - B||$.

Указание. Решение оформить в виде последовательных и вложенных вызовов функции модуля (для целых или вещественных чисел).

1.13. Заменить содержимое A на значение $(A + B)$, содержимое B на значение $(A - B)$ без использования дополнительных переменных.

1.14. Поменять содержимое A и B местами без использования дополнительных переменных.

Указание. См. пред. задачу 1.13. Отметим, что не стоит без необходимости использовать данный прием — это не только делает код "непрозрачным",

но и может, например, при больших значениях A, B, привести к ошибочному ответу в результате переполнения.

1.15. Найти сумму $1 + \ldots + n$ с использованием конструкции цикла `for()`.
Решение.

```
int NatSum (int n)
{
  int sum=0,i;
  for(i=1; i<=n; i++) sum += i;
  return sum;
}
```

1.16. Найти величину $n! = 1 \cdot 2 \cdot \ldots \cdot n$. Расчеты провести со следующими типами данных, печатая результат по соответствующему формату:

```
short int           (%hd);
unsigned short int  (%hu);
int                 (%d);
unsigned int        (%u);
long int            (%ld)
unsigned long int   (%lu)
long long int          (%lld)
unsigned long long int    (%llu)
```

В зависимости от архитектуры используемой ЭВМ и компиллятора, для переменных каждого типа выделяется свое количество байт, узнать которое можно с помощью оператора `sizeof(имя типа)`. Это позволяет для каждого типа a'priori найти наибольшее значение n, для которого программа вычисления $n!$ выдаст правильный ответ. Экспериментально проверьте правильность теоретических оценок.

1.17. Для заданных $n > 1$ и Max найти величину $sum = 0 + n^0 + n^1 + \ldots + n^k$ пока $n^k < Max$.

Указание. Если считать, что допустимо решение со сложностью $O(k)$ действий, то можно использовать конструкцию цикла $while()$:

```
double BoundSum (int n, double Max)
{
  double sum = 0;
  double nk = 1;
  while ( nk < Max ) {
    sum += nk;
    nk=nk*n;
  }
  return sum;
}
```

1.18. Для заданных $m, n > 0, Max$ найти величину $sum = m + (m + n) + \ldots + (m + k \cdot n)$ пока $sum < Max$. Обобщить алгоритм на случай $n \leq 0$.

1.19. Вычислить значение так называемой функции $n!!$, т.е. для нечетного n найти произведение $1 \cdot 3 \cdot \ldots \cdot n$, а для четного — произведение $2 \cdot 4 \cdot \ldots \cdot n$.

1.20. Для положительного вещественного числа n найти ближайшее к нему целое число вида $2k$ (т.е. четное), распечатать его значение и значение соответствующего k.

1.21. Найти наибольший делитель целого числа n.

Указание.

```
for( i=n/2; i>1; i--) {
  if( n%i==0 ) break;
}
if(i==1) { printf("Число %d является простым\n",n); }
else { printf("Наибольший делитель числа %d равен %d\n",n,i); }
```

1.22. Проверить, является ли введенное число n простым.

Идеи реализации. Последовательно проверять делимость n на числа $2, 3, \ldots, k$ пока $k * k \leq n$, т.е. для $k \leq \sqrt{n}$.

1.23. Распечатать все делители числа n.

1.24. Найти n–е простое число.

1.25. Распечатать все простые делители числа n.

1.26. Распечатать разложение числа n на простые множители.

1.27. Найти все простые числа, не превосходящие n.

Идеи реализации. Простейший способ решения — последовательно проверять числа $1, 2, \ldots$ на простоту, см. задачу 1.22. Для больших значений n лучше воспользоваться алгоритмом “решето Эратосфена”. Также см. далее 1.200.

1.28. Имеется n различных ненулевых цифр. Вычислить количество различных k-значных чисел, которые могут быть из них составлены, т.е. найти значение т.н. функции $A_n^k = n!/k!$. При реализации учесть возможность переполнения величины $n!$ для корректных значений A_n^k.

1.29. Имеется множество из n различных предметов. Вычислить количество различных k-подмножеств, которые могут быть из него выбраны, т.е. найти $C_n^k = n!/(n-k)!/k!$. Учесть возможность переполнения $n!$ для корректных значений C_n^k.

1.30. Найти N-е число Фибоначчи (Леонардо Пизанского), определяемое рекуррентной формулой: $x_1 = 1, x_2 = 1, x_n = x_{n-1} + x_{n-2}, n = 3, \ldots, N$.

Указание. Можно использовать реализации на основе цикла, рекурсии и явной формулы, основанной на величине $\frac{1+\sqrt{5}}{2}$, называемой “золотым сечением”.

```
unsigned long int Fibonacci_recurrent( int N )
{
  unsigned long  xn_2=1, xn_1=1;
  unsigned long  xn;
  int i;
```

```
  for ( i=3; i<=N; i++ ) {
    xn=xn_1+xn_2;
    xn_2=xn_1; xn_1=xn;
  }
  return xn;
}
unsigned long int FiBonacci_recursion( int N )
{
  unsigned long int xn;
  if ( N==1 || N==2 ) return 1;
  xn = FiBonacci_recursion(N-2) + FiBonacci_recursion(N-1);
  return xn;
}
double pow(double, double);
unsigned long int FiBonacci_explicit( unsigned long int N )
{
  double sq5 = sqrt(5.);
  double xn=1./sq5*( pow((1.+sq5)/2., (double)N )
                  - pow((1.-sq5)/2., (double)N));
  return lround(xn);
}
```

Отметим, что явная формула также может быть реализована на основе `long double sq5, xn` и библиотечных функциях `sqrtl(),powl(), lroundl()`.

Полезно сравнить эффективность данных подходов, выяснив диапазон допустимых для N значений. Также важно измерить время работы каждой функции при больших N. Для этого можно вызвать функцию `clock()` из библиотеки `time.h`, возвращающую переменную типа `clock_t`, содержащую количество тактов, совершенных от начала выполнения программы, и поделить на именованную константу `CLOCKS_PER_SEC` для пересчета в секунды.

```
#include<time.h>
#include<stdio.h>
#include<math.h>
unsigned long int FiBonacci_explicit( unsigned long int N );
unsigned long int FiBonacci_recursion( unsigned long int N );
int main(){
 clock_t begin, end;
 time_t t_begin, t_end;
 double T;
 unsigned long int N=47,NF;
 begin = clock();
   t_begin=time(NULL);
     NF=FiBonacci_explicit(N);
   t_end=time(NULL);
   T = difftime(t_end, t_begin);
printf("time = %.0lf NF = %lu: \n",T, NF);
  t_begin=time(NULL);
    NF = FiBonacci_recursion(N);
```

```
  t_end=time(NULL);
  T = difftime(t_end, t_begin);
printf("time = %.0lf NF = %lu: \n",T, NF);
 end = clock();
printf("time = %.0lf NF = %lu %lf : \n",T, NF,
                  (double)(end-begin)/ CLOCKS_PER_SEC);
 return 0;
}
```

В данном примере функция `time(NULL)` возвращает *целое* число типа `time_t`, соответствующее текущему времени (количество секунд, прошедших с 00:00:00 UTC 1 января 1970 года — т.н. даты начала эры Юникса (Unix Epoch)). Следует помнить, что при работе программы менее секунды ответ будет нулевым.

В ОС Linux имеется системная утилита time, вызов которой в формате `time ./a.exe` выдает по завершении работы более детальную информацию о временных ресурсах, затраченных при выполнении `a.exe`.

1.31. Проверить являются ли три введенных числа a_1, a_2, a_3 последовательными элементами арифметической прогрессии, т.е. удовлетворяют формуле $a_n = a_{n-1} + d$, $n = 2, 3$.

1.32. Проверить являются ли три введенных числа $b1, b2, b3$ последовательными элементами геометрической прогрессии, т.е. удовлетворяют формуле $b_n = qb_{n-1}$, $n = 2, 3$.

1.33. Проверить являются ли три введенных числа a_i, a_j, a_k последовательными (возможно, неупорядоченными) элементами арифметической прогрессии.

1.34. Проверить являются ли три введенных числа b_i, b_j, b_k последовательными (возможно, неупорядоченными) элементами геометрической прогрессии.

1.35. Найти с помощью алгоритма Евклида наибольший общий делитель положительных чисел m и n, т.е. $nod = \text{НОД}(m, n)$.

Указание. Алгоритм основан на очевидном равенстве $\text{НОД}(m, n) = \text{НОД}(m - n, n)$ при $m > n > 0$.

```
int E1( int m, int n)
{
  int nod;
  while(1) {
    if( m == n ) break;
    if( m > n ) {
      m = m - n;
    } else {
      n = n - m;
    }
  }
  nod=m;
  return nod;
}
```

Следующая реализация является оптимизированной версией `E1()`. В данном случае также считаем, что $m, n > 0$.

```
int E2( int m, int n )
{
  int c, nod;
  if( m < n ) { c = m; m = n; n = c; }
  while( m != n ) {
    c = m%n;
    if(c==0) break;
    n = m;
    m = c;
  }
  nod = n;
  return nod;
}
```

1.36. Найти хотя бы одно решение (x, y) уравнения в целых числах $x \cdot m + y \cdot n = \text{НОД}(m, n)$.

Указание. Например, можно реализовать алгоритм перебора. Полное решение обычно ищется методом "расширенного алгоритма Евклида", см. далее 1.197.

1.37. Найти наименьшее общее кратное чисел m и n, т.е. $\text{НОК}(m, n)$.

Указание. Применить формулу $\text{НОД}(m, n)\text{НОК}(m, n) = m \cdot n$.

1.38. По заданным значениям параметров (размер изображений, углы наклона) вывести на экран псевдографические картинки следующего вида:

```
***********    *                     *    *************     *       *
*         *     *                   *       ********          *   *
*         *      *               *          ***                 *
*         *       *           *                                 *    *
***********        *       *                                   *       *
  Рис. 1         Рис. 2     Рис.3           Рис. 4              Рис. 5
```

Указание.

```
void Pict1( int nx, int ny )
{
  int i,j;
  for( j=0; j<nx; j++)  printf(‘‘*’’);
  printf(‘‘\n’’);
  for( i=1; i<ny-1; i++){
    printf(‘‘*’’);
    for( j=1; j<nx-1; j++) printf(‘‘ ’’);
    printf(‘‘*\n’’);
  }
  for( j=0; j<nx; j++) printf(‘‘*’’);
  printf(‘‘\n’’);
}
```

Замечание. Выяснить, что напечатает функция для неположительных значений nx, ny.

1.39. Реализовать консольный калькулятор для целых чисел на основе набора функций так, чтобы функция `main()` отвечала только за ввод чисел и кода операции, вызов соответствующей функции и печать результата.

Правдивая история
— *Так что же делает Ваша программа?*
— *Работает!!*
— *А точнее?*
— *Считает!*
— *Что считает?*
— *Числа.*
— *Какие числа?*
— *Типа double, кажется ...*

1.2 Задачи на обработку последовательности

Общая постановка задач этого раздела выглядит следующим образом: имеется некоторое количество однотипных данных (например, чисел), и требуется вычислить некоторую характеристику этого набора (функцию от этих данных). Специфика задачи состоит в том, что мы не можем сохранить весь набор данных в памяти, поскольку обычно нам заранее неизвестно общее количество элементов. Поэтому нужно построить алгоритм, вычисляющий необходимую характеристику за один проход (просмотр) последовательности. Допускается сохранение и пересчет лишь конечного набора промежуточных значений, на основе которых в любой момент можно вычислить требуемую характеристику.

На практике алгоритмы решения подобных задач сводятся к построению некоторых рекуррентных соотношений между искомыми значениями, а также, возможно, и другими промежуточными результатами вычислений, чтобы можно было их пересчитывать, продвигаясь шаг за шагом от начала к концу последовательности

Подробное изложение вопросов, связанных с возможностью построения подобных алгоритмов, см. в [1].

1.40. С клавиатуры вводится положительное целое число n, а далее — последовательность, состоящая ровно из n целых чисел. Требуется найти и распечатать сумму элементов последовательности.
Решение.

```
#include <stdio.h>
int  main (void)
{
  double  x, sum;
  int  i, n;
  printf("  ----Вычисление суммы действительных чисел---- \n");
```