УДК. 681.3

Осепянц О.А., Исмаилов Ш-М.А

Методика генерации оптимального основания для представления чисел в системе остаточных классов

(*Дагестанский государственный технический университет, кафедра вычислительной техники*)

При решении практических задач на ЭВМ часто необходимо делать выбор между быстродействием и затратами памяти. В статье рассматривается разработанная авторами методика выбора модулей системы остаточных классов, обеспечивающих оптимальное соотношение быстродействие/затраты памяти для некоторых алгоритмов машинной арифметики системы остаточных классов.

Одной из важнейших задач при создании программного обеспечения (*далее – ПО*) является нахождение компромисса между быстродействием и затратами аппаратных ресурсов, причём решение этой задачи существенно зависит от конкретных условий применения. Замена позиционных систем счисления (*далее – СС*) на систе-

му остаточных классов (*СОК*) позволяет оперативно оптимизировать эти параметры в каждом конкретном случае путём подбора векторного основания СС. То есть при создании ПО программист может вводить в систему процедуру оптимизации модулей, которая позволит автоматически (незаметно для пользователя) либо через системные настройки выбирать оптимальное для каждого конкретного случая соотношение между быстродействием и затратами ресурсов. Более того, такой алгоритм может быть встроен в операционную систему и функционировать незаметно как для пользователя, так и для программиста.

1. Анализ алгоритмов модульной арифметики

Рассмотрим следующий алгоритм сложения, наиболее простой при аппаратной реализации.

1) *В счётчик A_m (имеется в виду: счётчик A по модулю m, то есть производящий подсчёт в диапазоне от 0 до m-1) заносится число a;*
2) *В двоичный счётчик B заносится b;*
3) *Пока $B > 0$ делать:*
   a) *Увеличить значение A_m на 1,*
   b) *Уменьшить значение B на 1,*
4) *Конец (в A_m – сумма чисел).*

Данный алгоритм предельно детализирован (1 команда = 1 такт процессора) с целью упростить дальнейший анализ.

Проведём анализ этого алгоритма согласно [2][1].

Прежде всего определим параметр, по которому будем оценивать сложности алгоритма – размер исходной задачи. Как уже упоминалось, длительность вычислений определяется значениями исходных чисел. Поэтому в качестве размера задачи следует использо-

[1] Следует учитывать, что все последующие рассуждения и выкладки приведены применительно к описанным здесь алгоритмам. Для других, отличных от них алгоритмов анализ проводится аналогично (методика описана в [2]).

вать значение модуля m как меры максимального представимого числа. В общем случае при применении векторного основания $b = [m_1, m_2, ..., m_n]$ и параллельном вычислении по всем модулям в качестве размера задачи берётся $m_{\max}$ – максимальный из модулей.

Итак, при нахождении временной сложности $g_t^+(m)$ в приведённом алгоритме будет доминировать цикл *3)*, при чём, поскольку при практической реализации шаг *a)* может выполняться параллельно с шагом *b)*, то каждый этап цикла потребует один такт процессора. Всего же цикл повторится b раз (в предельном случае – m_{max} раз). Таким образом, временную сложность этого алгоритма можно записать в виде:

$$g_t^+(m) = O(m), \qquad (1.1)$$

(читается: временная сложность порядка m).

Поскольку при выполнении этого и всех последующих алгоритмов память расходуется только на хранение исходных чисел и результата, оценка емкостной сложности будет приведена ниже для всех алгоритмов сразу.

Вычитание. Рассмотрим алгоритм вычитания.

1) *В счётчик A_m заносится b;*
2) *В счётчик B заносится m;*
3) *Пока $A_m > 0$ делать:*
   a) *Уменьшить значение B на 1,*
   b) *Уменьшить значение A на 1,*
4) *В счётчик A_m заносится a;* *//На этом этапе в счётчике B находится значение $\underline{b}$*
5) *Пока $B > 0$ делать:*
   a) *Увеличить значение A_m на 1,*
   b) *Уменьшить значение B на 1,*
6) *Конец (в A_m – разность чисел).*

Вторая часть алгоритма полностью соответствует вышеприведённому алгоритму сложения.

В данном алгоритме присутствует два цикла. Один этап каждого из них также выполняется за 1 такт процессора. Однако следует заметить, что цикл *3)* выполнится b раз, а цикл *5)* – *(m-b)* раз. Таким образом, при любом b циклы *3)* и *5)* в сумме выполнятся m раз и, следовательно, временная сложность $g_t^-(m)$ этого алгоритма также определяется выражением (1.1)

Умножение $a \cdot b$ сводится к вычислению $|ab|_m$. При реализации умножения на счётчиках, оно также сводится к сложению: $|ab|_m = \left|\sum_{i=1}^{b} |a|_m\right|_m$. При b=0 соответственно и $|ab|_m = 0$.

Сформулируем этот алгоритм:

1) *В счётчик A_m заносится значение* 0*;*
2) *В счётчик B заносится b;*
3) *Пока $B > 0$ делать:*
   a) *В счётчик C заносится число a,*
   b) *Пока $C > 0$ делать:*
      i) *Увеличить значение A_m на 1,*
      ii) *Уменьшить значение C на 1,*
   c) *Уменьшить значение B на 1,*
4) *Конец (в A_m – произведение чисел).*

Как видно из алгоритма, для операции умножения потребуется третий счётчик. Дополнительное удобство заключается в том, что результат всех трёх операций (сложение, вычитание и умножение) получается в одном и том же счётчике A_m, что также значительно упрощает аппаратную реализацию.

В данном алгоритме имеет место вложенный цикл. Цикл *3)* состоит из 3-х этапов, первый и третий из которых выполняются за 1 такт процессора, а второй (вложенный цикл) – за a тактов. Сам же цикл

3) выполнится b раз. Значит, общее время выполнения цикла 3) составит $b(a+2)$, а в предельном случае - $m \times (m+2) \approx m^2$. Отсюда временная сложность алгоритма умножения:

$$g_t^{\times}(m) = O(m^2) \qquad (1.2)$$

Деления. Операция деления сводится к нахождению произведения делимого на мультипликативный обратный делителя по модулю системы остаточных классов Для такого деления наиболее эффективным является матричных метод. Алгоритм такого деления практически совпадёт с вышеприведённым алгоритмом умножения за исключением строки 2, где в счётчик B вместо числа b заносится полученное с помощью ПЗУ значение b^{-1}.

Таким образом, временная сложность $g_t^{:}(m_{\max})$ деления также определяется из (1.2)

Из всего вышесказанного следует, что при уменьшении m_{max} *(а добиться этого можно увеличивая количество модулей)* происходит увеличение быстродействия.

Найдём теперь емкостную сложность вышеприведённых алгоритмов.

Поскольку, как отмечалось ранее, при выполнении этих алгоритмов память расходуется только на хранение исходных данных и результата, следовательно, емкостная сложность определяется исключительно формой представления чисел, то есть векторным основанием $b = [m_1, m_2, ..., m_n]$. Предположим, что для хранения всех разрядов представления числа используются ячейки памяти одинакового размера. В этом случае целесообразно использовать равномерный весовой критерий. Тогда емкостная сложность алгоритмов окажется пропорциональной числу модулей и составит:

$$g_{mem}(m_{\max}) = O(n) \qquad (1.3)$$

2. Основные требования к векторному основанию

Пусть $\{x_1, x_2, ..., x_l\}$ – множество входных данных; $\{y_1, y_2, ..., y_k\}$ – множество выходных данных вычислительного процесса. Тогда сам вычислительный процесс можно представить в следующем виде:

$$\begin{cases} y_1 = f_1(x_1, x_2, ..., x_l) \\ y_2 = f_2(x_1, x_2, ..., x_l) \\ \quad ..... \\ y_k = f_k(x_1, x_2, ..., x_l) \end{cases}, \text{ или } \vec{Y} = F(\vec{X}) \qquad (2.1)$$

Векторное основание $b = [m_1, m_2, ..., m_n]$ для проведения вычислений (2.1) должно обеспечивать однозначное представление всех входных и выходных данных. Как показано в [1], промежуточные данные на выполнение этого условия контролировать не обязательно.

Пусть $W : x_i \in W, y_j \in W, i = \overline{1..l}, j = \overline{1..k}$ – область значений входных и выходных данных, w_{max} – максимальный по модулю элемент из W. Перечислим требования, выполнение которых позволит однозначно представить все элементы W по модулю $b = [m_1, m_2, ..., m_n]$:

1. Для обеспечения однозначного соответствия любого $w_i \in W$ и его представления $|w_i|_b$, согласно [1], необходимо выполнение следующего условия:

$$(m_i, m_j) = 1, \quad i \neq j \qquad (2.2)$$

Здесь (m_i,m_j) – наибольший общий делитель чисел m_i и m_j

2. Для выполнения операции деления a/b необходимо существование элемента, обратного делителю по векторному основанию β, т.е. $b^{-1}(b)$. Согласно [1], необходимым и достаточным условием этого является:

$$b \neq 0, \quad (b, m_i) = 1, \ i = \overline{1..n} \qquad (2.3)$$

Следовательно, выбор в качестве $m_1, m_2, \ldots, m_n$ простых чисел гарантирует выполнение условий 1 и 2.

3. Для исключения псевдопереполнения при представлении исходных и конечных данных, согласно [3], должно быть выполнено следующее условие:

$$\begin{cases} M > w_{\max} & \text{при } W \subset I_+ \\ M > 2w_{\max} & \text{при } W \subset I \\ \left.\begin{array}{l} M \geq 2w_{\max}^2 + 1 \\ M \geq 2/e^2 + 1 \end{array}\right\} & \text{при } W \subset Q \end{cases} \qquad (2.4)$$

Здесь:

$$M = \prod_{i=1}^{n} m_i ;$$

ε *– точность представления рациональных чисел*

Q *– множество рациональных чисел*

I *– множество целых чисел*

I_+ *– множество целых неотрицательных чисел*

Для дальнейшего удобства перепишем (2.4) в следующем виде:

$$M \geq M_0, \text{ где } M_0 = \begin{cases} w_{\max} & \text{при } W \subset I_+ \\ 2w_{\max} & \text{при } W \subset I \\ \max\left(2w_{\max}^2 + 1,\ 2/e^2 + 1\right) & \text{при } W \subset Q \end{cases} \qquad (2.5)$$

Запись max(…) *означает максимальное из перечисленных значений.*

Таким образом алгоритм нахождения векторного основания β сводится к генерации последовательности простых чисел $m_1 < m_2 < \ldots < m_n$, произведение которых больше или равно M_0.

3. Задача оптимизации векторного основания

Оптимизация основания производится по максимуму быстродействия и по минимуму расхода памяти. Но поскольку алгоритмы сложения, вычитания, умножения и деления имеют разные временные сложности, а при решении различных задач могут доминировать любые из них, следовательно целевая функция должна учитывать их реальное соотношение в решаемой задаче. Сведя задачу макси-

мума быстродействия к задаче минимума временных затрат, получим целевую функцию следующего вида:

$$Z = k_+ a_+ + k\_a\_ + k_\times a_\times + k_\div a_\div + k_{mem} a_{mem} \qquad (3.1)$$

Здесь: k_+, $k\_$, $k_\times$, $k_\div$ *– весовые коэффициенты операций сложения, вычитания, умножения и деления, зависящие от среднего числа соответствующих операций, выполняемых в ходе решения задачи;*

k_{mem} *– весовой коэффициент затрат памяти;*

a_+, $a\_$, $a_\times$, $a_\div$ *– составляющие, учитывающие временные затраты на операции сложения, вычитания, умножения и деления;*

a_{mem} *– составляющая, учитывающая затраты памяти;*

Таким образом, варьируя значения весовых коэффициентов в зависимости от конкретных условий и желаемого результата, а затем решая задачу минимума *Z*, можно получить искомое оптимальное основание.

Подставив в (3.1) выражения (1.1), (1.2) и (1.3) для временной и емкостной сложностей алгоритмов, получим:

$$Z = k_+ m_{\max} + k\_ m_{\max} + k_\times m_{\max}^2 + k_\div m_{\max}^2 + k_{mem} n \qquad (3.2)$$

где $m_{\max}$ *– максимальный модуль основания* $b = [m_1, m_2, ..., m_n]$ *(в случае одномодульного основания вместо* $m_{\max}$ *следует использовать m.)*

4. Генерация векторного основания

С учётом вышесказанного, генерация векторного основания при известном M_0 сводится к нахождению такого $M \geq M_0$, при разложении которого на простые множители $m_1 < m_2 < ... < m_n$ выражение (3.2) принимает минимальное значение. Сделать это можно методом пассивного поиска, последовательно разлагая на множители числа M_0, M_0+1, M_0+2,…, найти такое *M*, которое удовлетворяет вышеназванному условию.

Следствие: составное M не разделится на составное b, если до этого M было максимально возможное число раз разделено на все простые делители числа b.

Таким образом, если последовательно максимально возможное число раз делить M на натуральные числа в порядке возрастания, начиная с числа 2, то на момент деления на составное b, M уже окажется разделённым на все простые делители числа b. То есть, согласно теореме 4.1, b не войдёт в число делителей M, так как деление произойдёт с остатком, причём такое разложение будет состоять не более, чем из $r + M - 1$ операций деления (r – *количество простых делителей числа* M).

Этот метод можно ускорить, уменьшив число операций деления следующим образом: согласно [4], если M не имеет делителей в диапазоне $2...\sqrt{M}$, то M – простое число и, следовательно, не имеет нетривиальных делителей. В этом случае число операций по разложению M на простые делители не превысит $r + \sqrt{M} - 1$.

Поскольку решено использовать метод пассивного поиска, следует определить границы поиска для M: нижняя граница – это, как уже упоминалось, значение M_0 из (2.5), а верхняя граница должна определяться аппаратными возможностями, то есть максимально возможным значением модулей m_i и их максимальным числом. Тогда верхняя граница поиска будет определена по формуле:

$$M_{\max} = (m_{i\max})^{n_{\max}} \qquad (4.1)$$

где: $n_{\max}$ – *максимально допустимое в системе число модулей;*

$m_{i\max}$ – *максимально допустимое в системе значение модуля.*

Однако в этом случае число итераций неоправданно возрастёт (например, для $n_{\max}$=16 и $m_{i\max}$=255, число итераций составит: 3×10^{38}), что потребует нереально больших временных затрат. Практика же показала, что в большинстве случаев оказывается достаточным гораздо меньшее число итераций. Проводя эксперименты над описанным здесь алгоритмом в системе программирования Delphi 6 (в диапазоне значений M_0 от 10 до 10^9), я предпочёл выбрать более гибкий вариант нахождения $M_{\max}$:

$$M_{\max} = 300 \times 2^{\lg M} \qquad (4.2)$$

Программа с использованием (4.2) вполне удовлетворяла требованиям, предъявляемым к экспериментальным программам в указанном диапазоне значений, однако эта формула не является оптимальной. Оптимизировав значение $M_{\max}$, мы получим существенное увеличение быстродействия.

В процессе поиска каждое найденное разложение следует подвергать проверке на выполнение условия (4.3), вытекающего из сформулированных выше требований к основанию СОК:

$$\begin{cases} n \le n_{\max} \\ m_i \le m_{i\max}, i = 1..n, \\ m_1 < m_2 < ... < m_n \end{cases} \qquad (4.3)$$

Невыполнение этого условия означает, что данное разложение нельзя использовать в качестве основания.

Итак, теперь мы готовы сформулировать окончательный алгоритм генерации оснований.

5. Алгоритм генерации оптимального основания СОК

Входные данные:

$w_{\max}$ – максимальное по модулю значение из области определения входных и выходных данных решаемой задачи;

ε – точность (для рациональных чисел). Условимся, что при ε=0 мы имеем дело с множеством целых неотрицательных чисел, при ε=-1 – с множеством целых чисел, а при 0<ε<1 – с множеством рациональных чисел;

k_+, k_-, $k_\times$, $k_\div$, k_{mem} – весовые коэффициенты операций сложения, вычитания, умножения и деления в решаемой задаче, а также весовой коэффициент затрат памяти – задаются исходя из специфики решаемой задачи и имеющихся аппаратных ресурсов;

$n_{\max}$ – максимально возможное в системе число модулей;

$m_{i\max}$ – максимально возможное в системе значение каждого модуля.

Выходные данные: $b = [m_1, m_2, ..., m_n]$ – векторное основание СОК.

Алгоритм:

1) Ввод значений $w_{\max}$, ε, k_+, k_-, $k_\times$, k_+, k_{mem}, $n_{\max}$, $m_{i\max}$.

2) $$M_0 = \begin{cases} w_{\max} & \text{при } e = 0 \\ 2w_{\max} & \text{при } e = -1 \\ \max\left(2w_{\max}^2 + 1,\ {}^{2}\!/_{e^2} + 1\right) & \text{при } 0 < e < 1 \end{cases}$$

//Нижняя граница поиска

3) $M_{\max} = 300 \times 2^{\lg M_0}$ *//Верхняя граница поиска*

4) $M=M_0$, $M_X=M$, $Z=$ любое достаточно большое число

5) $x = M_x$, $b = [\ ]$, $n = 0$ *//Инициализация переменных*

6) Для $i = \overline{2..M_x}$ делать: *//Разложение M_x на простые множители*

   a) Если $i > \sqrt{M_x}$ и $n = 0$ то перейти к п. 7

   //Если M_x – простое

   b) Если $x = 1$ то перейти к п.8

   //Исключение лишних итераций

   c) $y = {}^{x}\!/_{i}$ *//Пробное деление*

   d) Если y не имеет дробной части, то:

      i) $x = y$ *//Запоминание результата деление*

      ii) $n = n + 1$

      iii) $b_n = i$ *//Запоминание найденного делителя*

iv) Перейти к п. 6b

7) Если $n = 0$ то: //Если M_x – простое…

a) $n = n + 1$

b) $b_n = M_x$

8) $m_{\max} = \max(b)$ *//Выделение наибольшего делителя*

9) Если $n \le n_{\max}$, $m_{\max} \le m_{i\max}$ и $b_j \ne b_l$ $(j \ne l)$ то:

a) Если $Z > k_+ m_{\max} + k_- m_{\max} + k_\times m_{\max}^2 + k_\div m_{\max}^2 + k_{mem} n$ то: *//Сравнение значений целевой функции*

i) $Z = k_+ m_{\max} + k_- m_{\max} + k_\times m_{\max}^2 + k_\div m_{\max}^2 + k_{mem} n$

//Запоминание значения целевой ф-и

ii) $M = M_x$ *//Запоминание промежуточн. значения M*

10) $M_x = M_x + 1$

11) Если $M_x \le M_{\max}$ то перейти к п. 5

//Проверка следующего M

12) $x = M$, $b = [\ ]$, $n = 0$

//Инициализация переменных

13) Для $i = \overline{2..M}$ делать: *//Разложение M на простые множители*

a) Если $i > \sqrt{M}$ и $n = 0$ то перейти к п. 14

//Если M_x – простое

b) Если $x = 1$ то перейти к п.8

//Исключение лишних вычислений

c) $y = x / i$ *//Пробное деление*

d) Если y не имеет дробной части, то:

i) $x = y$ *//Запоминание результата деления*

ii) $n = n + 1$

iii) $b_n = i$ *//Запоминание найденного модуля*

iv) Перейти к п. 13b

14) Если $n = 0$ то: //Если M – простое…

a) $n = n + 1$

b) $b_n = M$

15) Вывод $b = [m_1, m_2, ..., m_n]$

16) Конец

Примечание: в случае применения алгоритмов модульной арифметики, отличных от описанных здесь (см. «Анализ алгоритмов модульной арифметики»), целевую функцию (3.2) следует находить исходя из их анализа аналогично описанной здесь методике. В остальном же приведённый алгоритм пригоден для любых вариантов реализации операций модульной арифметики.

6. Анализ алгоритма и исключение избыточных итераций

Главным недостатком полученного алгоритма следует считать резкое увеличение числа итераций при больших значениях M_0, согласно (4.2), что в совокупности с достаточно медленной процедурой разложения на простые множители даёт резкое снижение быстродействия. Частично справиться с этим недостатком можно уменьшив число итераций до минимально необходимого, но обеспечивающего нахождение минимума целевой функции.

Итак, проанализируем зависимость $Z(M)$.

На рисунке 6.1 представлен график функции $Z(M)$ (для $k_+ = 1$, $k_- = 1$, $k_\times = 1$, $k_\div = 1$, $k_{mem} = 400$, $n_{\max} = 16$, $m_{i\max} = 255$), из которого видно, что данная зависимость имеет хаотический характер, определяемый формой разложений M на простые множители,

то есть параметрами m_{max} и n. Не смотря на это, в области максимумов наблюдается чёткая зависимость, определяемая формой целевой функции и значениями её коэффициентов. Однако, поскольку в нашем случае решается задача минимума, проанализируем область минимумов данной функции, для чего программным путём построим для её графика огибающую по минимальным значениям. Для этого для каждого M найдём минимальное значение целевой функции на интервале $[M - \Delta M; M + \Delta M]$, а полученные минимумы соединим на графике отрезками.

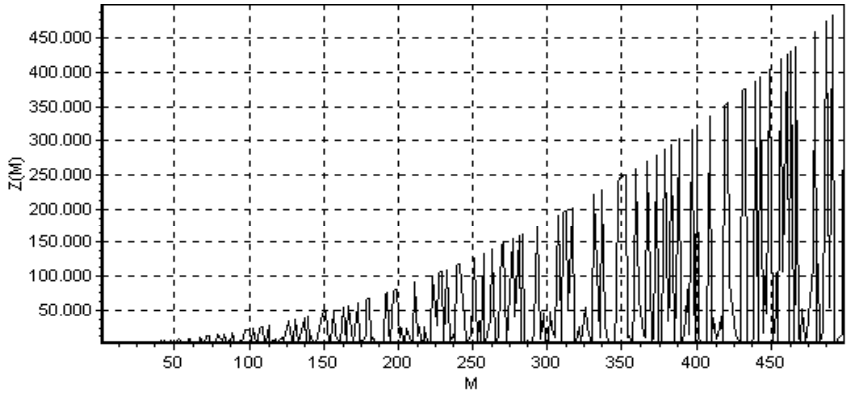

Рисунок 6.1.

С увеличением M значения функции в минимумах также возрастают. Аналогичный результат был получен и для других значений коэффициентов. Следовательно, на любом интервале [M_0; M_{max}] минимум функции $Z(M)$, вероятнее всего, будет находиться ближе к нижней границе интервала. Однако полученный алгоритм проходит весь интервал [M_0; M_{max}] от начала до конца и, следовательно, выполняет излишний объём вычислений.

Наиболее очевидным решением здесь является выбор значения M_{max}, исключающего лишние этапы вычислений. Однако с уменьшением значения M_{max} повышается вероятность выхода минимума целевой функции за границы интервала поиска. Существует и другой вариант, более гибкий: ввести в основной цикл алгоритма проверку условия, был ли уже найден минимум. Если минимум найден → выйти из цикла. Но здесь возникает другая проблема: как определить момент нахождения минимума?

Как видно из рис. 6.1, значение функции $Z(M)$ изменяется скачкообразно. Так, согласно графику, на интервале [100; 200] её производная меняет знак 39 раз, причём на разных участках графика для интервалов одинаковой ширины это значение в среднем остаётся постоянным. Таким образом, учитывая вышесказанное относительно огибающей ф-и $Z(M)$, можно утверждать, что если за предыдущие ΔM итераций новый минимум не найден, то он не будет найден и при последующих итерациях и, следовательно, процесс поиска можно завершить. В этом случае произойдёт ровно столько итераций, сколько требуется для отыскания минимума. Тогда роль параметра $M_{\max}$ сведётся к ограничению максимального числа итераций и, следовательно, его значение следует выбирать с запасом, чтобы исключить выход минимума функции за пределы интервала. На быстродействие это не повлияет, так как реальное число итераций теперь не будет зависеть от $M_{\max}$.

Для реализации этого метода в Алгоритм генерации оптимального основания СОК следует внести следующие изменения:

1) Вводятся две дополнительные переменные: *C* (счётчик нерезультативных итераций), *ΔM* (максимально допустимое число следующих подряд нерезультативных итераций)

2) В п. 5 вводится инициализация переменных *C* и *ΔM*:

C=0; *ΔM =заданное по умолчанию или введённое пользователем значение;*

3) Добавить П. 9.a.iii:

C=0; *//если итерация результативна, то сбросить счётчик*

4) Между пп. 8 и 9 добавить п. 8.1:

C=*C*+1; *//если итерация нерезультативна – инкремент счётчика*

5) Между пп. 10 и 11 добавить п. 10.1:

Если *C*> *ΔM*, перейти к п. 12

6) В п. 3 задать для $M_{\max}$ достаточно большое значение, гарантирующее попадание минимума целевой функции в интервал поиска.

В предельном случае можно совсем исключить из алгоритма операции с $M_{\max}$, заменив в п. 11 проверку условия $M_x \le M_{\max}$ на безусловный переход. В этом случае алгоритм упростится, однако максимальное число итераций не будет иметь ограничений, что может повлечь за собой ошибки переполнения разрядной сетки ЭВМ.

Для испытания полученного алгоритма в среде Borland Delphi 6 была создана программа, реализующая его. Испытания проводились на ЭВМ со следующими параметрами:

Таблица 6.1

Параметр	**Значение**
Центральный процессор	Intel Celeron 600
Материнская плата	Zida T810
Тактовая частота	600 МГц
Объём оперативной памяти	64 МБ
Операционная система	Microsoft Windows 98

При входных данных: $e = 0$, $k_+ = 1$, $k_- = 1$, $k_\times = 1$, $k_\div = 1$, $k_{mem} = 400$, $n_{\max} = 16$, $m_{i\max} = 255$, были получены следующие значения длительностей вычислительного процесса (таблица 6.2):

Как видно из таблицы 6.2, в результате принятых мер быстродействие существенно возросло. Однако при $w_{\max} > 10^5$ в обоих случаях быстродействие алгоритма резко падает, что является ограничивающим фактором в практическом применении алгоритма.

В случае встраивания данного алгоритма в операционную систему возможны случаи, когда процесс оптимизации основания потребует гораздо больше машинного времени, чем сами вычисления. Выходом из положения является хранение в памяти ЭВМ заранее сгенерированного с помощью данного алгоритма массива оптимизированных оснований для различных фиксированных значений весовых коэффициентов. Работа генератора оснований в этом случае сводится к выбору наиболее подходящего основания из списка. Также в систему может быть включена реализация данного алгоритма, ориентированная на решение задач малого размера.

Таблица 6.2

w_{max}	t, мс	
	без исключения избыточных итераций (M_{max} согласно (4.2))	*с исключением избыточных итераций (ΔM=100)*
10	0	0
10^2	0	0
10^3	0	0
10^4	50	30
10^5	2.360	1.590
10^6	26.920	19.440
10^7	373.430	316.540

Литература:

1. Грегори Р., Кришнамурти Е. Безошибочные вычисления. Методы и приложения. – М.: «Мир», 1988
2. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. – М.: Мир, 1979
3. Осепянц О.А. Проблема выбора модулей при проведении вычислений в системе остаточных классов. Сборник «Современные информационные технологии в управлении. Всероссийская научно-техническая конференция». – Махачкала, 2003 г. С. 87-90.
4. Оре О. Приглашение в теорию чисел. – М.: Наука, 1980
5. Виноградов И.М. Основы теории чисел. – М.: Наука, 1981