Государственное образовательное учреждение высшего профессионального образования «Московский Государственный Технический Университет имени Н.Э. Баумана»

**ОТЧЕТ**

По лабораторной работе №2

По курсу «Анализ алгоритмов»

Тема: «Алгоритмы умножения матриц»

Студент: Жарова Е.А.

Группа ИУ7-51

Москва, 2017

# Оглавление



# Постановка задачи

Необходимо изучить алгоритмы умножения матриц: стандартный («в лоб»), алгоритм Винограда и модифицированный алгоритм Винограда. Реализовать и сравнить алгоритмы.

# Описание модели вычислений

При оценке трудоёмкости мы будем пользоваться следующей моделью вычисления:

1. Операции, которые имеют трудоемкость “0”:
   - логический переход по ветвлению;
   - операции обращения к полю структуры/класса (->, .);
   - объявление переменных
2. Операции, которые имеют трудоёмкость «1»:
   - Арифметические операции {сложение(+), вычитание(-), умножение(*), деление(/), битовый сдвиг(<<, >>), деление нацело, взятие остатка(%) };
   - логические операции { и(&&), не(!), или(||) };
   - операции сравнения {<, >, =, !=, >=, <=};
   - операции присваивания {=, +=, -=, *=, /=, %=};
   - операция взятия индекса ([]);
   - операции побитового И(&) и ИЛИ(|)
   - унарный плюс и минус
   - операции инкремента и декремента(постфиксные и префиксные) (++, --).

# Стандартный алгоритм

1. Описание

Для того, чтобы вычислить произведение двух матриц A размерностью N×M и B размерностью M×K необходимо каждую строчку матрицы A умножить на каждый столбец матрицы B. Затем подсчитываем сумму таких произведений и записываем её в соответствующую ячейку результирующей матрицы.

$R[i][j] = \sum_{k=0}^{M} A[i][k] * B[k][j]$, где i,j – пробегают все значения i = 0..N-1, j = 0..K-1

2. Реализация

```
1. def base(m1, m2):
2.     m = []
3.     a, b, c = len(m1), len(m2), len(m2[0])
4.     if b != len(m1[0]):
5.         print("Error")
6.         return
7.     for i in range(a):
8.         m.append([0 for j in range(c)])
9.     for i in range(a):
10.          for j in range(c):
11.              for k in range(b):
12.                  m[i][j] += m1[i][k]*m2[k][j]
13.      return m
```

3. Теоретическая оценка

Трудоемкость алгоритма подсчитаем по строкам 9-12, так как проверка корректности, создание результирующей матрицы - не зависит от реализации алгоритма и выполняется для каждого и рассмотренных алгоритмов. Сложность данного алгоритма будет равна:

f = 2 + a(2 + 2 + c(2 + 2 + b(2+ 8))) = 2 + 4a + 4ac + 10abc

# Алгоритм Винограда

1. Описание

Алгоритм Винограда считается эффективнее за счет сокращения количества операций умножения. Результатом умножения двух матриц является скалярное произведение соответствующих строки и столбца. Такое умножение позволяет выполнить заранее часть работы.

Рассмотрим два вектора $V = (v1, v2, v3, v4)$ и $W = (w1, w2, w3, w4)$, скалярное произведение которых равно: $V \bullet W = v1w1 + v2w2 + v3w3 + v4w4$.

Данное равенство можно переписать следующим образом: $V \bullet W = (v1 + w2)(v2 + w1) + (v3 + w4)(v4 + w3) - v1v2 - v3v4 - w1w2 - w3w4$. Умножения v1v2, v3v4, w1w2, w3w4 можно рассчитать заранее.

Выражение $v1w1 + v2w2 + v3w3 + v4w4$ имеет большую трудоёмкость, чем выражение $(v1 + w2)(v2 + w1) + (v3 + w4)(v4 + w3)$.

2. Реализация

```
1. def vinograd(m1, m2):
2.     a, b, c = len(m1), len(m2), len(m2[0])
3.     if b != len(m1[0]):
4.         print("Error")
5.         return
6.     d = b // 2
7.     row_factor = [0 for i in range(a)]
```

```
    col_factor = [0 for i in range(c)]

    for i in range(a):
        for j in range(d):
            row_factor[i] = row_factor[i] + m1[i][2 * j] * m1[i][2 * j + 1]

    for i in range(c):
        for j in range(d):
            col_factor[i] = col_factor[i] + m2[2 * j][i] * m2[2 * j + 1][i]

    m = [[0 for i in range(c)] for j in range(a)]

    for i in range(a):
        for j in range(c):
            m[i][j] = - row_factor[i] - col_factor[j]
            for k in range(d):
                m[i][j] = m[i][j] + ((m1[i][2 * k] + m2[2 * k + 1][j]) * (m1[i][2 * k + 1] + m2[2 * k][j]))
    if b % 2:
        for i in range(a):
            for j in range(c):
                m[i][j] = m[i][j] + m1[i][b - 1] * m2[b - 1][j]
    return m
```

### 3. Теоретическая оценка

У нас 4 цикла (внутри которых тоже есть циклы):

f1 = 2 + a ∗ (2 + 2 + b/2 ∗ (2 + 12)) = 2 + 4a + 7ab

f2 = 2 +c ∗ (2 + 2 + b/2 ∗ (2 + 12)) = 2 + 4c + 7bc

f3 = 2 +a ∗ (2 + 2 +c ∗ (2 + 7 + 2 + b/2 ∗ (2 + 23))) = 2 + 4a + 11ac + 12.5abc

f4 = 2 +a ∗ (2 + 2 +c * (2 + 13)) = 2 + 4a + 15ac

Лучший случай:

fл= 2 + 4a + 7ab + 2 + 4c + 7bc + 2 + 4a + 11ac + 12.5abc = 12.5abc + 11ac + 7bc + 4c + 7ab + 8a + 4

Худший случай:

fx= 2 + 4a + 7ab + 2 + 4c + 7bc + 2 + 4a + 11ac + 12.5abc + 2 + 4a + 15ac = 12.5abc + 26ac + 7bc + 4c + 7ab + 12a + 6

Трудоемкость в среднем:

f = (fл + fx ) / 2 = 5 + 10a + 7ab + 4c + 7bc + 18.5ac +12.5abc

# Модифицированный алгоритм Винограда

### 1. Описание

Модифицируем алгоритм Винограда следующим образом:

- Используем составное присваивание.
- Вычисляем заранее индексы и выражения, которые часто используются.
- В случае нечетной размерности перенос последнего цикла внутрь основного цикла.

### 2. Реализация

```
def vinograd_mod(m1, m2):
    a, b, c = len(m1), len(m2), len(m2[0])
    if b != len(m1[0]):
        print("Error")
        return
```

```
6.      flag = (b % 2 == 1)
7.      row_factor = [0 for i in range(a)]
8.      col_factor = [0 for i in range(c)]
9.
10.       for i in range(a):
11.           for j in range(1, b, 2):
12.               row_factor[i] -= m1[i][j - 1] * m1[i][j]
13.
14.       for i in range(c):
15.           for j in range(1, b, 2):
16.               col_factor[i] -= m2[j - 1][i] * m2[j][i]
17.
18.       m = [[0 for i in range(c)] for j in range(a)]
19.
20.       for i in range(a):
21.           for j in range(c):
22.               m[i][j] += row_factor[i] + col_factor[j]
23.               for k in range(1, b, 2):
24.                   m[i][j] += ((m1[i][k - 1] + m2[k][j]) * (m1[i][k] +
    m2[k - 1][j]))
25.                   if flag:
26.                       m[i][j] += m1[i][b - 1] * m2[b - 1][j]
27.       return m
```

### 3. Теоретическая оценка

У нас 3 цикла (внутри которых тоже есть циклы):
f1 = 2 +a∗(2 + 2 +b/2∗(2 + 8)) = 2 + 4a + 5ab
f2 = 2 +c∗(2 + 2 +b/2∗(2 + 8)) = 2 + 4c + 5bc
f3 = 2 +a∗(2 + 2 +c∗(2 + 6 + 2 +b/2∗(2 + (16or26)))) = 2 + 4a + 10ac + abc + abc(16or26)

Лучший случай:
fл = 2 + 4a + 5ab + 2 + 4c + 5bc + 2 + 4a + 10ac + 8abc = 6 + 8a + 4c + 5ab + 5bc + 10ac + 9abc

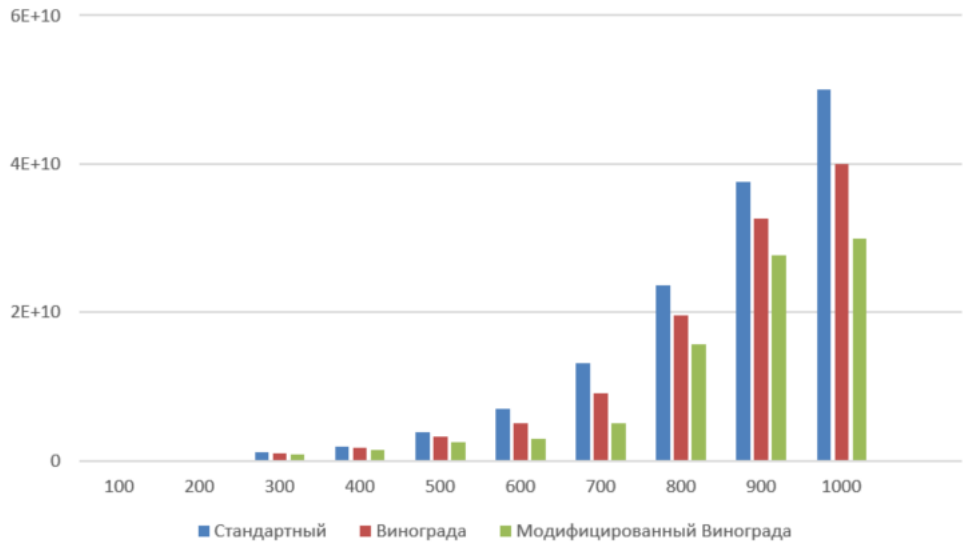
Худший случай:
fx = 2 + 4a + 5ab + 2 + 4c + 5bc + 2 + 4a + 10ac + 14abc = 6 + 8a + 4c + 5ab + 5bc + 10ac + 14abc

Трудоемкость в среднем:
f = (fл + fx ) / 2 = = 6 + 8a + 4c + 5ab + 5bc + 10ac + 11.5abc

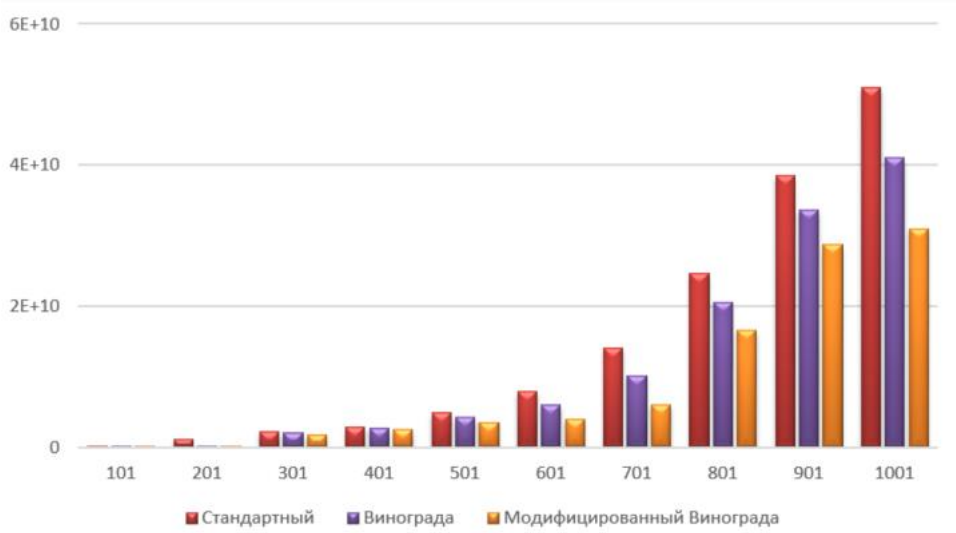
## Сравнение алгоритмов

Чтобы сравнить алгоритмы было посчитано время работы для матриц размерностью 100×100, 200×200, …, 1000×1000 и 101×101, 202×202, …, 1001×1001.

Анализ быстродействия алгоритмов для матриц четных размерностей




Анализ быстродействия алгоритмов для матриц нечетных размерностей




Выводы:

- Чем больше размер матрицы, тем выигрышнее использовать алгоритм Винограда
- Алгоритм Винограда работает быстрее стандартного алгоритма, а модифицированный алгоритм Винограда работает быстрее алгоритма Винограда
- Обнаружены несовпадения результатов теоретической оценки и результатов эксперимента. Это связано с тем, что в случае алгоритма Винограда компилятор оптимизирует большинство одинаковых вычислений в цикле, это позволяет достичь трудоёмкости O(n, m, k) = 9nmk. Тогда как реализация стандартного и модифицированного алгоритма написана уже примерно с учетом возможных оптимизаций.

# Заключение

Были изучены и реализованы алгоритмы умножения матриц: стандартный алгоритм, алгоритм Винограда и модифицированный алгоритм Винограда. Так же было произведено сравнение этих методов.