Санкт – Петербургский Государственный Университет

Точной Механики и Оптики.

Кафедра: «Высшая математика»

Бакалаврская работа.

Анализ алгоритмов умножения чисел неограниченной разрядности.

Студент: Кирильчук В.Е.

Научный руководитель: Кубенский А.А.

Санкт-Петербург

-2010-

**Содержание**

Введение……………………………………………………………………………………………………………….8

1.Цели и задачи…………………………………………………………….……………………………………..10

2.Представление чисел неограниченной разрядности………………………………………11

3.Реализация чисел неограниченной разрядности в Java..………………………………..12

4.Обзор существующих библиотек……………………………………………………………………..16

4.1The GNU Multiple Precision Arithmetic Library………………………………………16

4.2 FXT………………………………………………………………………………………………………….17

4.3 Arbitrary Precision Arithmetic Package…………………………………………………..17

4.4 Вывод…………………………………………………………………………………………………….18

5.Выбранные алгоритмы….………………………………………………………………………………….19

5.1 Алгоритм умножения “в столбик”……………………………………………….……….19

5.2 Алгоритм Карацубы…………………………………………………………………….…….….19

5.3 Алгоритм Toom-Cook 3-way…………………………………………………………….……22

5.4 Алгоритм с использованием БПФ………………………………………………….…….26

5.5 Алгоритм с использованием БПХ………………………………….………………….….28

6.Результаты работы…………………………………………………………………………………………….30

6.1 Выбор оптимальных параметров…………………………………………………………30

6.2 Эффективность полученной операции умножения…………………………….35

6.3 Заключение……………………….…………………………………………………………….…….37

Список литературы……………………………………………………………………………………………….38

Приложения………………………………………………………………………………………………………….39

**ВВЕДЕНИЕ**

В вычислительной технике существует такое понятие, как “длинная арифметика” – операции над числами, разрядность которых превышает длину машинного слова данной вычислительной машины. Частный случай – арифметика произвольной точности – относится к арифметике, в которой длина чисел ограничена только объёмом доступной памяти. Такие числа в данной работе будут называться числами неограниченной разрядности.

История чисел неограниченной разрядности берёт своё начало с компьютера IBM 1620, анонсированного IBM в 1959 году. Этот компьютер выполнял целочисленные операции над строками (состоящими из цифр), длина которых была ограничена только объёмом доступной памяти. В то время пределом стала длина в 60 000 десятичных знаков. Однако ранняя программная реализация чисел неограниченной разрядности связана с языком программирования Maclisp, появившимся в 70-х годах и являющегося диалектом языка Lisp.

Для чего же используется арифметика произвольной точности? Самое простое применение это, конечно же, математическое и финансовое программное обеспечение, требующее, чтобы результат вычисления на компьютере совпадал до последнего разряда с результатом вычисления на бумаге. Однако наиболее интересны всё же другие сферы применения, а именно: криптография, астрономия, «спортивные» вычисления знаменитых трансцендентных чисел (π, e и т. д.) с высокой точностью, а также высококачественные изображения фракталов.

В настоящее время, арифметика произвольной точности реализуется только программными средствами, так как, несмотря на обширную сферу применения чисел неограниченной разрядности, для решения подавляющего большинства задач хватает базовых арифметических типов, а аппаратная реализация достаточно сложна. И тут возникает вопрос об эффективности программной реализации таких чисел и эффективности операций над ними.

Некоторые современные языки программирования имеют встроенную поддержку чисел неограниченной разрядности, для других существуют библиотеки для работы с арифметикой произвольной точности. В данной работе изложение будет вестись относительно реализации таких чисел в объектно-ориентированном языке программирования Java, разработанном компанией Sun Microsystems.

Этот язык программирования выбран не случайно, дело в том, что в интернете существует сайт, ранее принадлежавший Sun Microsystems, а ныне - компании Oracle, купившей Sun Microsystems, на котором каждый может оставить сообщение об ошибке или необходимом улучшении касательно Java. В 1999 и 2003 годах разными людьми было написано несколько сообщений о необходимости улучшения операции умножения в классе BigInteger, который является стандартным классом библиотеки Java для работы с арифметикой произвольной точности. Однако качественных улучшений так и не последовало.

Собственно говоря, данная работа и посвящена анализу эффективных алгоритмов умножения чисел неограниченной разрядности и их реализации, с целью сделать операцию умножения в классе BigInteger более эффективной. Так же в работе был рассмотрен вопрос об оптимальности представления чисел неограниченной разрядности, существующего в классе BigInteger.

**1. Цели и задачи**

1. Анализ реализации класса BigInteger языка Java. Анализ необходим, чтобы понять, имеет ли смысл улучшать существующую библиотеку или имеет смысл написать новую реализацию, которая будет лишена недостатков присущих стандартной библиотеке.
2. Подбор наиболее эффективных алгоритмов с целью их реализации и создания более эффективной операции умножения чисел неограниченной разрядности.
3. Реализация выбранных алгоритмов на языке Java. Анализ их поведения, в зависимости от величины чисел. Составление операции умножения из наиболее оптимальной комбинации этих алгоритмов.
4. Сравнение эффективности полученной операции умножения с эффективностью начальной реализации. Сравнение эффективности с сторонней библиотекой для работы с числами неограниченной разрядности.

**2. Представление чисел неограниченной разрядности**

Неотрицательное число N представляется в виде:

где, а BASE – основание системы счисления.

Например, число

Такое представление N является частным случаем многочлена n-ой степени

Основание системы счисления BASE обычно зависит от максимального размера базового типа данных, предоставляемого языком программирования (если мы говорим о программной реализации) или аппаратным обеспечением (если мы говорим об аппаратной реализации). Везде в дальнейшем будет говориться именно о программной реализации.

Исходя из того, что написано выше, для программной реализации возникает несколько главных вопросов:

1. Как хранить коэффициенты, какую структуру данных для этого выбрать?
2. В каком порядке хранить коэффициенты?

Существует два широко распространённых варианта:

А) Little-endian нотация, когда младший коэффициент хранится в начале структуры, а старший в конце.

Б) Big-endian нотация, когда младший коэффициент хранится в конце структуры, а старший в начале.

1. Какое выбрать основание BASE?

Ответы на эти вопросы будут даны в следующей главе при рассмотрении реализации чисел неограниченной разрядности в языке Java.

**3. Реализация чисел неограниченной разрядности в Java**

В стандартной библиотеке Java есть класс BigInteger, который предназначен для работы с числами неограниченной разрядности. Он имеет все необходимые функции для работы с ними, однако, в базе данных запросов об улучшениях и исправлениях для языка Java, можно найти около десятка запросов на улучшение эффективности некоторых из этих функций. В частности, половина этих запросов связана с тем, что алгоритмы умножения и возведения в степень в классе BigInteger реализованы недостаточно эффективно.

Одной из целей работы как раз является улучшение эффективности операции умножения в классе BigInteger. Было выдвинуто предположение, что добиться этого можно тремя способами:

1. Изменить представление чисел таким образом, чтобы операции выполнялись более эффективно, чем в текущей реализации.
2. Реализовать более быстрый алгоритм умножения, возможно, являющийся комбинацией нескольких разных алгоритмов, и использовать его для умножения.
3. Совмещение первого и второго способа.

Первый способ подразумевает под собой изучение текущего представления. О чём сейчас и пойдёт речь. В случае если окажется, что изменение представления нецелесообразно, для улучшения эффективности останется только второй способ.

Итак, в классе BigInteger для хранения коэффициентов выбрана стандартная встроенная в Java структура – массив. Возможно ли использование вместо массива какой-либо более эффективной структуры? Ответ – нет. Даже, если использовать для каждого коэффициента отдельную переменную примитивного типа (что само по себе очень не разумно), это окажется менее эффективно. Почему? Дело в том, что современные процессоры осуществляют доступ к памяти не побайтно, а небольшими блоками, которые называют строками кэша. Обычно размер строки составляет 64 байта. Когда вы читаете какое-либо значение из памяти, в кэш попадает как минимум одна строка кэша. Последующий доступ к какому-либо значению из этой строки происходит очень быстро. Отдельные переменные не обязательно будут расположены в памяти друг за другом, а вот элементы массива всегда располагаются в памяти последовательно, а, следовательно, при доступе к первому элементу, в кэше окажутся несколько последующих элементов и доступ к ним будет быстрее. Другое предложение – это использование более сложных структур данных, в основе которых лежит массив, но, если нам не требуется ничего кроме доступа к коэффициентам, то это не даёт никаких преимуществ, и, более того, может ухудшить эффективность операций. Таким образом, структура для хранения коэффициентов в BigInteger оптимальна.

Рассмотрим вопрос о порядке хранения коэффициентов. Как уже было сказано в предыдущей главе, существует два широко распространённых способа расположения коэффициентов в структуре: от младшего коэффициента к старшему и от старшего - к младшему. В BigInteger используется big-endian нотация. Основное преимущество такой записи в отличие от little-endian – естественность записи, ведь мы записываем арабские числа именно таким образом. С другой стороны, большинство операций с числами выполняется от младших разрядов к старшим, а, значит, при таком порядке придётся идти с конца массива в начало, не повлияет ли это на кэширование? На трёх разных системах была проведена проверка этого предположения, и, как оказалось, это абсолютно не влияет на эффективность операций. Таким образом, порядок коэффициентов существенной роли не играет.

Осталось рассмотреть последний пункт – выбор основания. В BigInteger коэффициенты принадлежат стандартному примитивному типу языка Java - int, что соответствует основанию системы счисления, равному 2^32. На следующей странице приведен график, на котором отображены кривые, показывающие зависимость времени выполнения операции умножения методом “в столбик” от размера исходных данных при различных основаниях, соответствующих 2^8(byte), 2^16(short), 2^32(int). Стоит отметить, что чем больше основание, тем меньше коэффициентов содержит представление. То есть если размер исходных данных составляет 1024 байта, то для основания 2^8 мы получим 1024 коэффициента, для 2^16 – уже 512 коэффициентов, а для 2^32 – всего 256 коэффициентов. Умножение выполняется над числами одинаковой длины, то есть с одинаковым количеством коэффициентов.

Результаты получены на системе с процессором Intel Core 2 Duo E7200 с 2 гигабайтами оперативной памяти.

Как видно из графика, чем большее основание мы выбираем, тем быстрее выполняется умножение. Обеспечивается это как раз тем, что чем больше основание, тем меньше коэффициентов содержит представление исходных чисел, а умножение коэффициентов одного типа(byte, short или int) занимает одинаковое количество времени. По крайней мере, это верно для процессоров с разрядностью 32 бита и выше.

Однако в языке Java существует ещё один примитивный тип long, что соответствует основанию системы счисления, равному 2^64. Почему бы не выбрать его в качестве типа коэффициентов? Дело в том, что при умножении двух коэффициентов, мы можем получить так называемое переполнение, когда результат умножения не помещается в используемый нами тип. Для контроля такого переполнения необходимо использовать более вместительный тип данных. Для byte таким типом данных будет short, для short – int, для int – long, а вот для long подходящего типа уже нет. Поэтому нельзя использовать long.

Таким образом, можно сказать, что представление чисел неограниченной разрядности в Java является оптимальным. Следовательно, для улучшения эффективности операции умножения в классе BigInteger остаётся только второй способ – использование более эффективного алгоритма. А какой алгоритм используется в существующей реализации и можно ли улучшить его?

На рис.1 не случайно приведёно время выполнения операции умножения методом “в столбик”. Именно этот алгоритм и используется в BigInteger для перемножения чисел неограниченной разрядности. Он также написан оптимально и его улучшение нецелесообразно. Асимптотическая оценка скорости этого алгоритма в случае, когда длины множителей равны, есть *O(n^2).* Однако существуют алгоритмы умножения чисел неограниченной разрядности с лучшей асимптотической оценкой. В работе была предпринята попытка использовать некоторые из них, чтобы добиться улучшения эффективности операции умножения класса BigInteger.

Перед предоставлением результатов стоит сначала рассмотреть, какие же алгоритмы умножения используются в специализированных библиотеках для работы с числами неограниченной разрядности, рассказать о том, какие алгоритмы были выбраны для реализации и, наконец, показать, в чём состоит суть каждого из выбранных алгоритмов.

**4. Обзор существующих библиотек**

Далее будут перечислены наиболее распространённые библиотеки для работы с числами неограниченной разрядности, дано их краткое описание и представлен список использующихся в них алгоритмов для умножения. Исходя из этих списков, и были выбраны алгоритмы, которые использовались в работе. Однако в данной главе суть алгоритмов раскрываться не будет, будут даны лишь названия. Более подробно об алгоритмах, которые были реализованы в работе, можно прочитать в главе, посвящённой алгоритмам.

* 1. **The GNU Multiple Precision Arithmetic Library (GMP)**

Описание: GMP - это свободно распространяемая библиотека для арифметических операций произвольной точности со знаковыми целыми, рациональными числами и числами с плавающей точкой. Предел точности не ограничен, за исключением ограничений, вытекающих из размера доступной памяти. GMP содержит богатый набор функций, оснащенных стандартизованным интерфейсом. Разработчики ставили целью высокую скорость, как для маленьких операндов, так и больших. Скорость работы достигается путем использования быстрых алгоритмов с оптимизированным под большинство процессоров ассемблерным кодом для большинства общих внутренних циклов и благодаря основному акценту на скорость (вместо простоты и элегантности). Считается, что GMP работает быстрее, чем любая другая подобная библиотека. Библиотека широко используется в криптографических целях и для компьютерных вычислений.

Библиотека написана на языках C\C++ с использованием ассемблерных вставок и считается самой быстрой и эффективной. Кроме того, она является самой известной из описанных в этой главе библиотек. Рассмотрим алгоритмы, которые используются в GMP для умножения чисел неограниченной разрядности.

1. Умножение в “столбик”
2. Умножение по алгоритму Карацубы
3. Умножение по алгоритму Toom-Cook 3-way
4. Умножение по алгоритму Toom-Cook 4-way
5. Умножение по алгоритму Шенхаге-Штрассена
   1. **FXT: a library of algorithms**

Описание: fxt - это свободно распространяемая библиотека, которая является сборником различных алгоритмов не только для чисел неограниченной разрядности, но и для битовых операций, некоторых абстрактных структур данных, вычисления некоторых элементарных функций при ограниченных ресурсах и многого другого…

Библиотека написана на языках C\C++. Рассмотрим алгоритмы, которые используются в этой библиотеке для умножения чисел неограниченной разрядности.

1. Умножение в столбик
2. Умножение по алгоритму Карацубы
3. Умножение по алгоритму Toom-Cook 3-way
4. Умножение по алгоритму Toom-Cook 4-way
5. Умножение по алгоритму Toom-Cook 5-way
6. Умножение с использование Быстрого Преобразования Фурье
7. Умножение по алгоритму Шенхаге-Штрассена
8. Умножение с использование Теоретико-Численного Преобразования

Так же следует отметить, что в этой библиотеке есть подраздел, связанный с быстрыми преобразованиями, в которых есть алгоритмы Быстрого Преобразования Фурье, Быстрого преобразования Хартли, а также Теоретико-Численного Преобразования. Для умножения Быстрое Преобразование Хартли в данной библиотеке не используется. Это важно, так как в работе одним из алгоритмов умножения был выбран именно алгоритм, использующий Быстрое Преобразование Хартли.

* 1. **Arbitrary Precision Arithmetic Package (APFLOAT)**

Описание: является высокопроизводительной библиотекой для работы с арифметикой произвольной точности. Помимо основных операций с числами неограниченной разрядности, включает в себя также функцию нахождения числа π с нужным количеством десятичных знаков.

Изначально библиотека написана на языке C++. Затем была адаптирована к Java. Поддержка версии написанной на С++ была прекращена в апреле 2006 года. Последняя версия на Java датируется 2009 годом.

Считается медленной. Но, всё же, рассмотрим алгоритмы, которые используются в этой библиотеке для умножения чисел неограниченной разрядности.

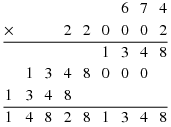
1. Умножение в столбик
2. Умножение по алгоритму Карацубы
3. Умножение с использование Теоретико-Численного Преобразования
   1. **Вывод**

Видно, что во всех этих библиотеках используется не один алгоритм, а несколько. То есть для умножения чисел неограниченной разрядности в них используется некая комбинация этих алгоритмов. Также, видно, что для всех перечисленных библиотек общим является умножение по алгоритму Карацубы и “в столбик”, следовательно, именно эти алгоритмы стоит реализовать в первую очередь (метод умножения в столбик уже реализован в классе BigInteger). Также были реализованы алгоритм Toom-Cook 3-way, алгоритм умножения с использованием Быстрого Преобразования Фурье и алгоритм умножения с использованием Быстрого Преобразования Хартли. В следующей главе дано подробное описание каждого из этих алгоритмов.

**5. Выбранные алгоритмы**

**5.1 Алгоритм умножения “в столбик”**

Иногда используется другие названия – “школьный” или “наивный” алгоритм. Этот алгоритм уже есть в классе BigInteger, поэтому не понадобилось его реализовывать. Сам алгоритм аналогичен тому, который мы используем при умножении чисел в десятичной системе счисления с таким же названием. Далее приведён пример умножения двух длинных чисел (длины n и m) с основанием системы счисления равным 10, с коэффициентами в big-endian нотации. (n равно 3, m = 6)



Асимптотическая оценка скорости данного алгоритма O(n\*m).

**5.2 Алгоритм Карацубы**

Умножение Карацубы — один из быстрых алгоритмов: метод быстрого умножения, который позволяет перемножать два N-значных числа со сложностью вычисления:

Перемножение двух N-значных целых чисел обычным школьным методом “в столбик” сводится, по сути, к сложению N-значных чисел N раз. Поэтому для сложности «школьного» или «наивного» метода имеем оценку сверху:

В 1956 году А. Н. Колмогоров сформулировал гипотезу, что нижняя оценка для M() при любом методе умножения есть также величина порядка (так называемая «гипотеза Колмогорова »). На правдоподобность гипотезы указывал тот факт, что метод умножения «в столбик» известен не менее четырёх тысячелетий (например, этим методом пользовались шумеры), и если бы был более быстрый метод умножения, то он, вероятно, уже был бы найден. Однако в 1960 году Анатолий Карацуба нашёл новый метод умножения двух N-значных чисел с оценкой сложности

и тем самым опроверг «гипотезу ». Впервые метод был опубликован в 1962 году.

Впоследствии метод Карацубы был обобщён до парадигмы «Разделяй и властвуй», другими важными примерами которой являются метод двоичного разбиения (англ. binary splitting), двоичный поиск, метод бисекции и др.

В работе был использован алгоритм Карацубы, который основывается на следующей формуле:

Пусть, , , A и B — два N-значных числа. Представляя A и B в виде:

где ,,, это k-значные числа, по указанной выше формуле находим:

(1)

Таким образом, четыре умножения заменились на три умножения, шесть сумм и две операции сдвига (выполняются за линейное время, если основание системы исчисления кратно двум). Причём для каждого из этих трёх умножений мы снова можем применить нашу формулу.

Асимптотическая сложность.

Обозначим символом количество операций, достаточное для умножения двух N-значных чисел по формуле (1). Видно, что выполняется следующее неравенство:

где > 0 есть абсолютная константа. Последовательно применяя эту формулу и, приняв во внимание, что

Получаем:

Пусть , если это не так, то можно дополнить число нулевыми разрядами до выполнения этого условия. Тогда получим:

То есть асимптотическая сложность перемножения двух N-значных чисел по алгоритму Карацубы оценивается как Однако из-за накладных расходов умножение метод Карацубы оказывается медленнее, чем умножение «в столбик» для небольших чисел. Типичным решением является использование умножения «в столбик» в качестве базы рекурсии данного алгоритма. Одной из задач работы стало нахождение оптимальной базы рекурсии для этого алгоритма.

Если разбивать числа не на два, а на большее количество слагаемых, то можно получать асимптотически лучшие оценки сложности вычисления произведения. Далее будет рассмотрен алгоритм Toom-Cook 3-way, который разбивает число на 3 части и имеет асимптотическую оценку

**5.3 Алгоритм Toom-Cook 3-way**

Вообще говоря, Toom-Cook – это целое семейство алгоритмов, поэтому обычно конкретизируют, какой именно алгоритм используется, указывая после названия k-way. Алгоритм “Toom-Cook k-way” – это метод умножения двух больших чисел, основанный на разбиении этих чисел на k слагаемых, названный в честь Андрея Тоома и Стефана Кука. Так же, как и в алгоритме умножения Карацубы, суть этого метода заключается в выполнении меньшего количества умножений для этих слагаемых, чем потребовалось бы при прямом перемножении.

В работе был применён алгоритм Toom-Cook 3-way, который разбивает число на три слагаемых и выполняет вместо 9 умножений всего 5 умножений втрое меньшей длины (и несколько операций за линейное время), которые также выполняются с использованием этого алгоритма.

Впервые этот метод был описан русским математиком Андреем Тоомом в 1963 году, в то время как Стефан Кук улучшил алгоритм и опубликовал результат своей работы в 1966 году. Стоит отметить, что этот метод является обобщением метода умножения Карацубы, для которого k=2. В общем случае, асимптотическая сложность k-way алгоритма оценивается как , где . С ростом k, показатель e стремится к единице, однако функция C(k) достаточно быстро растёт. Поэтому стратегия увеличения параметра k не приводит к желаемому результату.

Вследствие накладных расходов данного метода умножения, он также, несмотря на хорошую асимптотическую сложность, проигрывает другим алгоритмам (Карацубы и умножение «в столбик») на относительно малых числах. Обычно данный алгоритм используется для перемножения чисел среднего размера вплоть до тех пор, пока использование методов с лучшей асимптотической сложностью не становится целесообразным.

Рассмотрим идею алгоритма, который был использован в работе. Так как k=3, то мы получаем из двух чисел (A и B) следующее представление в виде полиномов:

где для нахождения коэффициентов выбирается x, исходя из основания системы счисления b и количества коэффициентов в числах (m и n), следующим образом:

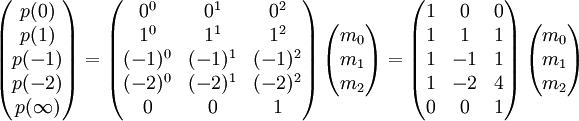
Теперь, если бы мы просто перемножили два полученных многочлена, то, конечно же, получили бы ответ, однако для этого нам пришлось бы выполнить 9 умножений. Ключевая идея алгоритма заключается в том, что можно вычислить эти полиномы в некоторых точках, перемножить получившиеся значения поточечно, получив при этом набор значений, а затем интерполировать и получить полином, который будет, по сути, являться результатом умножения.

Известно, что существует единственный интерполяционный многочлен степени не превосходящий , который принимает в точках значения . То есть для того, чтобы получить результат нам понадобится при k равном трём всего пять значений. Их можно получить, вычислив исходные полиномы в пяти точках и перемножив их поточечно. Важным вопросом является выбор этих точек. В работе использовалась последовательность точек 0; 1; -1; -2;

Итак, для некоторого многочлена :

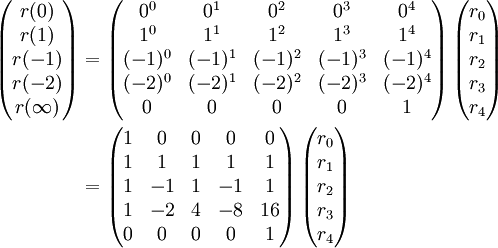
*, где под понимается*

Если записать это в матричном виде, то:

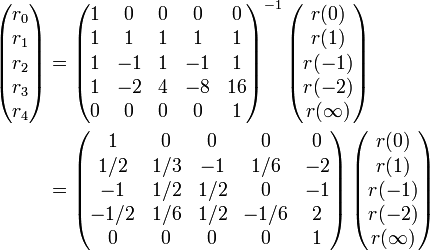
**

Перемножая поточечно значения, полученные для многочленов A(x)и B(x), получим:

Теперь надо понять, как по этим значениям «восстановить» результирующий многочлен. Рассмотрим сначала, как, зная результирующий полином, могли быть получены данные значения:



А теперь воспользуемся тем, что квадратная матрица, стоящая в правой части обратима. Тогда коэффициенты результирующего полинома могут быть получены как:



Всё, что осталось – это посчитать матрично-векторное произведение и восстановить число. Результирующие коэффициенты целые числа, так что все необходимые операции могут быть выполнены при помощи целочисленной арифметики. При этом единственные нелинейные по времени операции умножения возникают при получении значений r(x). Таким образом, вместо 9 умножений мы получили 5 втрое меньшей длины (плюс операции за линейное время).

Вывод асимптотической оценки для этого алгоритма аналогичен выводу оценки для алгоритма Карацубы. Приведём лишь саму оценку: .

**5.4 Алгоритм с использованием Быстрого Преобразования Фурье**

Следующий наиболее известный алгоритм, использующийся для быстрого перемножения двух чисел - это быстрый метод умножения двух чисел с использованием так называемого ”Быстрого Преобразования Фурье”.

Основные идеи Быстрого Преобразования Фурье были известны уже в 1805 году Карлу Фридриху Гауссу, однако, на долгое время вплоть до 1965 года эти принципы были забыты, пока Джон Туки в соавторстве с Джимом Кули не описали алгоритм для вычисления в цифровой форме преобразования Фурье. В настоящее время существует достаточно много алгоритмов Быстрого Преобразования Фурье, однако в работе использовался именно алгоритм Кули-Туки.

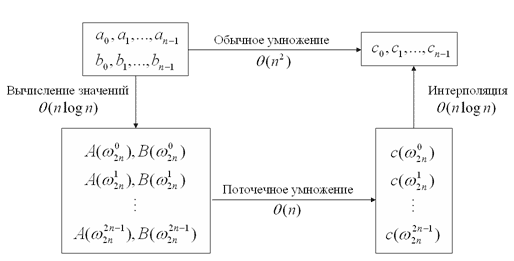
Вообще говоря, Быстрое Преобразование Фурье — это быстрый алгоритм вычисления Дискретного Преобразования Фурье. Иногда под БПФ понимается один из быстрых алгоритмов, называемый алгоритмом прореживания по частоте/времени или алгоритмом по основанию 2, имеющего сложность . Это преобразование широко применяется в алгоритмах цифровой обработки сигналов: в сжатии звука в MP3, сжатии изображений в JPEG и др., а также в других областях, связанных с анализом частот в дискретном (к примеру, оцифрованном аналоговом) сигнале.

Так, как же можно использовать Быстрое Преобразование Фурье для перемножения двух чисел? Дело в том, что одно из главных свойств Дискретного Преобразования Фурье (а также БПФ) состоит в выполнении теоремы о свёртке:

где под свёрткой можно понимать произведение наших чисел.

Таким образом, для того, чтобы вычислить произведение двух чисел, достаточно выполнить обратное преобразование от произведения преобразований этих чисел. Причём, произведение преобразований есть покоординатное умножение двух векторов, которое выполняется за линейное время.

Идея алгоритма в чём-то похожа на метод Toom-Cook. Однако числа не разбиваются на несколько частей, вместо этого они сразу интерпретируются, как многочлены вида , где это коэффициенты соответствующего числа. Затем, вычисляются значения этих многочленов при помощи Быстрого Преобразования Фурье, которое выполняется за время . Получившиеся значения перемножаются поточечно, а затем при помощи интерполяции, которая является обратным Быстрым Преобразованием Фурье и также выполняется за время , получаем результат. Ниже представлена схема, которая иллюстрирует вышесказанное:



где ω2n- комплексный корень из 1, степени 2n.

Стоит отметить, что при умножении двух чисел, длины n и m, для интерполяции необходимо n+m вычисленных значений многочленов, кроме того, для Быстрого Преобразования Фурье, которое использовалось в работе, необходимо, чтобы длина была степенью двойки. Поэтому исходные числа дополняются нулевыми разрядами до длины N - ближайшей к n+m степени двойки. Соответственно, в большинстве случаев данный алгоритм увеличивает размерность задачи. Кроме того, этот алгоритм использует комплексные числа, что сопряжено с проблемами точности результата. Асимптотическая оценка для этого алгоритма

**5.5 Алгоритм с использованием Быстрого Преобразования Хартли**

Преобразование Хартли это аналог Преобразования Фурье. Его главное отличие от Преобразования Фурье состоит в том, что Преобразование Хартли оперирует с вещественными, а не комплексными числами. Впервые алгоритм был предложен Рональдом Брейсуэллом в 1983 году, как более эффективный вычислительный инструмент, нежели Преобразование Фурье, для задач, где данные являются вещественными. Однако существуют специальные версии Преобразования Фурье, которые также могут работать с вещественными данными и даже используют при этом на несколько операций меньше, чем аналогичные версии Преобразований Хартли.

Алгоритм Преобразования Хартли долгое время находился под защитой патента, и, только в 1994 году стал общедоступен. Это сыграло свою роль и данный алгоритм не получил широкого распространения. Однако он имеет более простую структуру. А, соответственно, его легче реализовать.

Аналогичное Преобразованию Фурье выражение для свёртки выглядит следующим образом:

где f = DFH(a); g = DFH(b); Индексы считаются по модулю N, то есть вместо элементов с индексом N, надо брать элементы с индексом ноль. Под свёрткой можно понимать произведение наших чисел.

Таким образом, для того, чтобы вычислить произведение двух чисел, достаточно выполнить обратное преобразование от чуть более сложного выражения, нежели в Преобразовании Фурье.

Так, как Преобразование Хартли не столь широко известно, как Преобразование Фурье, приведём основную формулу для данного преобразования.

Опр.: Преобразованием Хартли вектора () называется вектор

f =(, где

Так же, как и преобразование Фурье, Преобразование Хартли является обратимым. При этом обратное преобразование отличается от прямого преобразования только множителем .

Точно так же, как и для алгоритма, использующего преобразование Фурье, исходные числа дополняются нулевыми разрядами до длины N - ближайшей к n+m степени двойки, где n – количество коэффициентов первого числа, m - второго. Данный алгоритм использует вещественные числа, что сопряжено с проблемами точности результата.

Быстрое Преобразование Хартли тоже аналогично Быстрому Преобразованию Фурье и вычисляет Дискретное Преобразование Хартли за время , вместо Рисунок, приведённый при рассмотрении алгоритма с использованием Быстрого Преобразования Фурье, так же подходит и для алгоритма с использованием Быстрого Преобразования Хартли, только значения вычисляются в других точках. Поэтому асимптотическая оценка данного алгоритма так же .

**6. Результаты работы**

**6.1 Выбор оптимальных параметров**

Результаты получены на системе с процессором Intel Core2 Duo E7200 и 2 гигабайтами оперативной памяти. Операционная система: Windows Vista.

Алгоритм умножения в столбик не пришлось реализовывать, так как он уже был написан в классе BigInteger. Поэтому реализация алгоритмов началась с умножения по методу Карацубы. Как было сказано в пункте, посвящённом этому алгоритму, на небольших числах, вследствие накладных расходов на рекурсию и некоторые другие операции этот алгоритм работает менее эффективно, чем метод умножения “в столбик”. Но, так как алгоритм рекурсивный, то в качестве базы рекурсии мы можем выбрать некоторую величину числа, при которой умножение будет производиться по алгоритму “в столбик”.

На рис.2 отображены кривые, соответствующие различным базам рекурсии:

Умножение на рис.2 выполняется над числами с равным количеством коэффициентов. База рекурсии указана в байтах.

Показаны кривые для трёх характерных баз рекурсии. Видно, что эффективность алгоритма при увеличении базы от 128 до 256 байт растёт. Стоит сказать, что уже при 178 байтах увеличение эффективности было едва заметно. При дальнейшем увеличении наблюдается обратный результат, при выборе основания в 512 байт эффективность алгоритма сильно ухудшается. Это связано с тем, что чем больше база рекурсии, тем больше работает медленное умножение “в столбик”. Итак, была выбрана база рекурсии в 256 байт.

Рассмотрим рис.3 - сводный график для умножения “в столбик” и для реализованного умножения по методу Карацубы:

Видно, что примерно при 256 байтах эффективность алгоритмов совпадает, это связно с тем, что если число меньше порога, и там и там работает один и тот же алгоритм – умножение “в столбик”. После порогового значения алгоритм с использованием умножения по методу Карацубы начинает выигрывать.

Следующий алгоритм, который был реализован, это алгоритм Toom-Cook 3-way. Это также рекурсивный алгоритм. Но теперь в качестве базы рекурсии мы можем брать уже не умножение в столбик, а полученный алгоритм для умножения по методу Карацубы. Оптимальной базой рекурсии был выбран размер числа, соответствующий 400 байтам. Таким образом, если перемножаются числа с большим значением, то используется алгоритм Toom-Cook, меньше – алгоритм Карацубы, ещё меньше, столбик. На рис. 4 приведён сводный график полученного алгоритма и комбинированного с методом “в столбик” алгоритма Карацубы:

До 400 байт эффективность алгоритмов совпадает. Затем алгоритм Toom-Cook, комбинированный с алгоритмом Карацубы, начинает выигрывать. Ситуация идентична предыдущей.

Следующий алгоритм, это алгоритм, использующий Быстрое Преобразование Фурье. К сожалению, он уже не может использовать полученные алгоритмы, так как работает по совершенно иному принципу.

Во-первых, реализованная версия алгоритма оказалась менее эффективной, чем даже простое умножение “в столбик”. Существует множество различных версий алгоритмов умножения с применением Быстрого Преобразования Фурье, в работе была использована не самая простая версия, но и не самая сложная. Одной из причин такой низкой скорости, является то, что она не была специализирована для вещественных входных данных. Но это абсолютно не значит, что все алгоритмы этого типа неприменимы на практике!

Во-вторых, интересна сама форма кривой для этого алгоритма. В пункте, описывающем данный алгоритм, говорилось, что исходные числа дополняются нулевыми разрядами до длины N - ближайшей к n+m степени двойки, где n – количество коэффициентов первого числа, m – второго. Этим объясняется наличие практически параллельных горизонтальной оси линий. Практически параллельных потому, что если присмотреться, то видно, что они имеют небольшой наклон. Этот наклон как раз и связан с тем, что векторы дополняются до необходимой длины и на это тратится некоторое время.

На рис.6 представлен сводный график умножения с использованием Быстрого Преобразования Хартли и умножения по методу Toom-Cook. Реализация умножения с использованием Быстрого Преобразования Хартли показала гораздо более хороший результат, в сравнении с умножением, использующим Быстрое Преобразование Фурье, так как изначально предполагала работу с вещественными входными данными.

Реализованный алгоритм с применением Быстрого Преобразования Хартли показал не просто хороший результат, а отличный. При числах, соответствующих 1500 байтам он оказался эффективнее алгоритма Toom-Cook. В пункте, где описывался данный алгоритм, было сказано, что он использует вещественные числа, что может приводить к неточным результатам из-за погрешности вычислений. Границей точности является примерно длина чисел, соответствующая 2^17 байт, поэтому по достижению этой границы снова начинает использоваться метод Toom-Cook.

Из комбинации реализованных алгоритмов была составлена новая операция умножения для класса BigInteger.

**6.2 Эффективность полученной операции умножения**

Результаты получены на системе с процессором Intel Core2 Duo E7200(2.53 Ghz) и 2 гигабайтами оперативной памяти. Операционная система – Windows Vista.

На рис. 7 представлено сравнение полученной операции умножения, составленной из оптимальной комбинации реализованных алгоритмов, с начальной реализацией в классе BigInteger.

В результате работы была реализована операция умножения, являющаяся, по сути, комбинацией нескольких алгоритмов умножения, которая, начиная с чисел, величина которых соответствует 256 байтам, работает более эффективно. Причём чем больше числа, тем ощутимей разница. Таким образом, одна из основных целей работы достигнута.

Теперь, сравним полученный результат с библиотекой APFLOAT. Для сравнения выбрана именно эта библиотека, так как она также написана на Java.

Умножение в библиотеке APFLOAT сначала оказывается менее эффективным, чем стандартная реализация в классе BigInteger, но затем, за счёт использования асимптотически лучших алгоритмов, начинает выигрывать по скорости. Проигрыш вначале связан с тем, что представление чисел в этой библиотеке сильно усложнено, вместо массива используется более сложная структура, которая содержит в себе массив, это и даёт потерю эффективности. Но полученная в результате работы реализация операции умножения оказалась эффективней, чем APFLOAT при любой величине числа.

На рис. 9 представлен график, аналогичный рис.7, но с результатами, полученными на другой системе. Результаты получены на системе с процессором AMD Athlon64 X2 TL-52(1.6Ghz) и 1 гигабайтом оперативной памяти. Операционная система: Linux Debian.

В целом, результат похож на предыдущие, с той лишь разницей, что заметный рост скорости происходит несколько позже, чем у системы, для которой подбирались оптимальные параметры.

**6.3 Заключение**

В результате работы были реализованы несколько существующих алгоритмов умножения чисел неограниченной разрядности на языке Java, все, кроме одного, уже имели различные реализации на языке Java, однако алгоритм умножения чисел неограниченной разрядности с использованием Быстрого Преобразования Хартли не имел до этого реализации на языке Java. Из реализованных алгоритмов была составлена новая операция умножения для класса BigInteger, которая оказалась более эффективной, чем стандартная операция умножения класса BigInteger. Также, полученная операция умножения оказалась эффективней сторонней библиотеки написанной на языке программирования Java – APFLOAT.

**Список литературы**

[1] Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы. Построение и анализ. 2-е издание. Пер. с англ. 2005г. —863с.

[2] Дональд Кнут. Искусство программирования, том 2. Получисленные алгоритмы. 3-е изд. Пер. с англ. 2007г. — С. 832.

[3] Нуссбаумер Г.Быстрое Преобразование Фурье и алгоритмы вычисления свёрток. Пер. с англ. 1985г. — С. 235.

[4] Fateman R. The finite field Fast Fourier Transform. —2000 —p. 10.

[5] Jorg Arndt. Matters computational ideas, algorithms, source code. —2009 —p. 998

[6] Martin Furer. Faster Integer Multiplication. —2009 —p. 27

[7] M. Bodrato. Toward Optimal Toom–Cook Multiplication, —2007. —p. 20

[8] Henry S. Warren, Jr. Hacker's Delight. —2002 — p. 320

[9] Брюс Эккель. Философия Java 2-е издание. Пер. с англ. 2006г. —976с.

[10] Википедия. Свободная энциклопедия. [Электронный ресурс]. – Режим доступа : <http://ru.wikipedia.org>

[11] Wikipedia. The free encyclopedia. [Электронный ресурс]. – Режим доступа : <http://en.wikipedia.org>

[12] Gnu multiple precision arithmetic library. [Электронный ресурс]. – Режим доступа : <http://gmplib.org/gmp-man-5.0.1.pdf>