Министерство науки и высшего образования Российской Федерации Федеральное государственное автономное образовательное учреждение высшего образования  
«Уральский федеральный университет имени первого Президента России Б.Н.Ельцина»

Институт радиоэлектроники и информационных технологий – РТФ

Алгоритм RSA-шифрования

Отчёт по лабораторной работе  
по дисциплине «Алгоритмы, структуры данных и анализ сложности»

Выполнил: студент группы, Аристов В.Е.

Преподаватель: доцент, к.ф.-м.н. Трофимов С.П.

2023

**Оглавление**

**Задание**

1. Написать класс, который содержит целое число со знаком в виде массива однобайтовых элементов. Реализовать конструкторы, деструктор, перегрузить операции: аддитивные (+, –), мультипликативные (\*, /, %), сравнения (==, !=, <, >), взятие обратного по заданному модулю.
2. Написать функцию шифрования строки с помощью алгоритм RSA.
3. Зашифровать/расшифровать текстовый файл с помощью открытого RSA-ключа.

**Теоретическая часть**

Алгоритм RSA (Rivest-Shamir-Adleman) является одним из самых широко используемых асимметричных алгоритмов шифрования и основан на сложности задачи факторизации больших простых чисел.

Процесс шифрования и дешифрования в алгоритме RSA основывается на использовании пары ключей: открытого и закрытого. Открытый ключ используется для шифрования сообщений, в то время как закрытый ключ необходим для их дешифрования.

Вот основные шаги алгоритма RSA:

1. Генерация ключей:
   1. Выбор двух больших простых чисел p и q.
   2. Вычисление их произведения n = p \* q. Число n называется модулем.
   3. Вычисление функции Эйлера φ(n) = (p - 1) \* (q - 1).
   4. Выбор открытой экспоненты e, которая должна быть взаимно проста с φ(n) и меньше φ(n).
   5. Вычисление закрытой экспоненты d, которая является обратной к e по модулю φ(n). То есть d \* e ≡ 1 (mod φ(n)).
   6. Пара (e, n) является открытым ключом, а пара (d, n) - закрытым ключом.
2. Шифрование:
   1. Предположим, что у нас есть открытый ключ (e, n) и сообщение M, которое мы хотим зашифровать.
   2. Представление сообщения M в виде целого числа m, где 0 ≤ m < n.
   3. Вычисление зашифрованного сообщения C по формуле: C ≡ m^e (mod n).
3. Дешифрование:
   1. Предположим, что у нас есть закрытый ключ (d, n) и зашифрованное сообщение C, которое мы хотим дешифровать.
   2. Вычисление исходного сообщения m по формуле: m ≡ C^d (mod n).

Алгоритм RSA основывается на математической сложности факторизации больших чисел. Предположительно, чтобы восстановить закрытый ключ по открытому ключу, необходимо выполнить факторизацию числа n на простые множители p и q, что является вычислительно сложной задачей при использовании достаточно больших простых чисел.

RSA широко применяется для шифрования данных, подписи цифровых документов, обеспечения безопасности в сети, аутентификации и других задач, где требуется обмен информацией с конфиденциальностью и целостностью. Однако, важно учитывать, что безопасность RSA основана на сложности факторизации больших чисел, и с появлением квантовых компьютеров, способных решать эту задачу эффективно, может потребоваться переход к квантовоустойчивым алгоритмам шифрования.

Алгоритм RSA обладает несколькими особенностями, которые делают его уникальным и позволяют использовать его в широком спектре приложений:

1. Асимметричное шифрование: RSA является асимметричным алгоритмом шифрования, что означает, что для шифрования и дешифрования используются разные ключи. Открытый ключ используется для шифрования сообщений, в то время как закрытый ключ необходим для их дешифрования. Это обеспечивает простоту и безопасность обмена зашифрованными данными.
2. Безопасность основана на сложности факторизации: Основная безопасность RSA основана на математической сложности факторизации больших простых чисел. Чтобы взломать RSA, необходимо выполнить факторизацию числа n на простые множители p и q. Сложность этой задачи растет экспоненциально с увеличением размера простых чисел. Поэтому использование достаточно больших ключей делает атаку путем перебора неэффективной.
3. Эффективность шифрования небольших объемов данных: RSA хорошо подходит для шифрования небольших объемов данных, таких как ключи шифрования симметричных алгоритмов или хэш-значения. Он обычно не используется для шифрования больших объемов данных, так как процесс шифрования и дешифрования RSA является вычислительно сложным и требует большого объема вычислительных ресурсов.
4. Цифровая подпись: RSA также используется для создания и проверки цифровых подписей. Цифровая подпись позволяет аутентифицировать отправителя и гарантировать целостность сообщения. Отправитель создает подпись, применяя свой закрытый ключ к хэшу сообщения, а получатель проверяет подпись, используя открытый ключ отправителя.
5. Генерация ключей: Генерация безопасных ключей в RSA включает выбор больших простых чисел и вычисление связанных значений. Это процесс, требующий случайности и вычислительных ресурсов. Безопасность RSA напрямую зависит от генерации ключей, и неправильная реализация генерации ключей может привести к уязвимостям.

Несмотря на свои преимущества, RSA также имеет некоторые ограничения. Один из них - относительная медлительность процесса шифрования и дешифрования, особенно для больших объемов данных. Кроме того, с развитием квантовых компьютеров, RSA может стать уязвимым для атак, поскольку они могут эффективно факторизовывать большие числа. Поэтому сейчас проводятся исследования по разработке квантовоустойчивых алгоритмов шифрования для замены RSA в будущем.

Сложность RSA зависит от нескольких факторов, включая размер ключа, операции шифрования/дешифрования и операции генерации ключей. Основная вычислительная сложность RSA связана с факторизацией больших чисел.

1. Генерация ключей: Сложность генерации ключей RSA зависит от размера простых чисел, выбранных для создания ключей. Чем больше размер ключа (т.е. более длинные простые числа), тем больше времени и вычислительных ресурсов требуется для их генерации. Генерация ключей может занять значительное время, особенно для больших ключей.
2. Шифрование и дешифрование: Операции шифрования и дешифрования в RSA требуют выполнения операции возведения в степень по модулю. Сложность этих операций зависит от размера ключа (длины модуля). Обычно время, необходимое для шифрования или дешифрования, пропорционально квадрату длины модуля.
3. Факторизация: Основная вычислительная сложность RSA связана с факторизацией модуля n на простые множители p и q. Сложность факторизации увеличивается экспоненциально с ростом размера модуля (длины ключа). Чем больше размер ключа, тем сложнее выполнить факторизацию и взломать RSA.

Использование достаточно больших ключей является важным для обеспечения безопасности RSA. В настоящее время рекомендуется использовать ключи длиной не менее 2048 бит для надежной защиты данных. Однако с развитием квантовых компьютеров, способных эффективно факторизовывать большие числа, размеры рекомендуемых ключей могут увеличиться в будущем.

Алгоритм RSA не имеет линейной сложности O(n), где n - размер входных данных. Сложность RSA зависит от различных факторов, таких как размер ключа, операции шифрования/дешифрования и факторизации.

Генерация ключей RSA имеет сложность, которая зависит от размера простых чисел, выбранных для создания ключей. Эта операция не является линейной по размеру входных данных.

Операции шифрования и дешифрования в RSA, в основном, зависят от длины модуля (размера ключа). Обычно время, необходимое для выполнения операций шифрования и дешифрования, пропорционально квадрату длины модуля. Это означает, что сложность RSA не является линейной по размеру входных данных.

Самая вычислительно сложная операция в RSA - факторизация модуля на простые множители p и q. Сложность факторизации увеличивается экспоненциально с ростом размера модуля (длины ключа). Факторизация не имеет линейной сложности и является одной из основных вычислительных сложностей RSA

Итак, сложность алгоритма RSA не может быть оценена как O(n), так как она зависит от нескольких факторов и не является линейной по размеру входных данных.

**Инструкция пользователя**

При запуске index (главного файла программы) открывается консоль, где пользователю предлагается перед сортировкой собственного массива провести эксперимент и оценить производительность алгоритма. Если он согласен, то на консоль будут выведены результаты эксперимента.

A screenshot of a computer program

Description automatically generated with low confidence

Рисунок 1 – Проведение эксперимента

Далее, независимо от того, производился опыт или нет, пользователю предлагается ввести собственный массив строк, разделяя его элементы пробелом. После ввода на консоль будет выведен уже отсортированный массив.

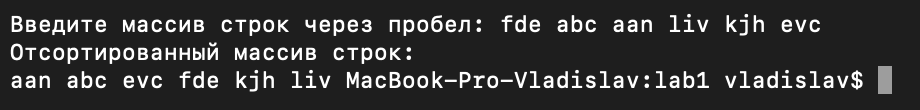


Рисунок 2 – Сортировка пользовательского массива строк

**Инструкция программиста**

Далее, независимо от того, производился опыт или нет, пользователю предлагается ввести собственный массив строк, разделяя его элементы пробелом. После ввода на консоль будет выведен уже отсортированный массив.

Программа написана на языке C. Вся программа разбита для удобного редактирования на несколько файлов .c. Каждый отдельный файл это отдельная функция. Главный функция находится в index.c. Тесты находятся в test.c. Компиляция происходит посредством компилятора gcc и ввода в командную строку:

**gcc index.c compare\_strings.c swap.c my\_qsort.c experiment.c xmlReader.c -o ../index -lxml2**

Также предварительно требуется установить библиотеку lxml12. В разных операционных системах это происходит по-разному.

Windows (требует установки MSYS2):

**pacman -S mingw-w64-x86\_64-libxml2**

macOS:

**brew install libxml2**

linux:

**sudo apt-get install libxml2-dev**

Используемые функции:

1. int main() - Главная функция приложения
2. int my\_qsort(void\* base, size\_t nmemb, size\_t size, cmp\_t cmp) – Моя реализация алгоритма qsort
3. int compare\_strings(const void\* a, const void\* b) – Свойство по которому сравниваются строки
4. int xmlReader() – Функция, считывающая с накопителя xml файл
5. int swap(void\* a, void\* b, size\_t size) – Функция, которая производит замены при сортировке
6. void experiment(int count, int length, int max) – Функция, проводящая эксперимент
7. void parseNode(xmlNode \*node) - Функция, разбивающая xml на куски и запускающая эксперименты для каждого отдельного Node

**Тестирование**

Для того чтобы провести тесты необходимо запустить test. Данная программа содержит тесты, проверяющий корректность работы алгоритма сортировки и выполняется в момент запуска программы.

A screenshot of a computer program

Description automatically generated with medium confidence

Рисунок 3 – Тестирование алгоритма

Помимо тестирования, программа предоставляет возможность провести эксперимент с пирамидальной сортировкой на больших массивах данных. Для этого в xml-файле experiment был создан план эксперимента. Он включает в себя несколько элементов nodes, каждый из которых описывает свою часть эксперимента: какое количество массивов будет сгенерировано, какой длины, с какими максимальными значениями. Пользователь сам может отредактировать xml-файл для проведения желаемого эксперимента и добавить или убрать какие-либо из тестов.

Полученные результаты (длины массивов и кол-во операций для их сортировки) были внесены в таблицу Excel.

**Выводы**

В данной работе мы познакомились с одним из алгоритмов сортировки – пирамидальной сортировкой QSort, написали программу для сортировки массивов строк с его использованием, а также провели эксперимент для оценки сложности данного алгоритма. Полученная программа работает исправно и позволяет достаточно быстро сортировать массивы из тысяч строк. Это же подтверждается и результатами эксперимента: алгоритм имеет сложность вида O (n \* log(n)).

QSort имеет несколько преимуществ по сравнению с другими алгоритмами сортировки:

1. Высокая скорость: qsort реализует алгоритм быстрой сортировки (quick sort), который, в среднем случае, имеет сложность O(n log n). Quick sort обычно является одним из самых быстрых алгоритмов сортировки для общего случая. Он эффективно справляется с большими массивами данных и обеспечивает высокую скорость выполнения.
2. Универсальность: qsort может быть использован для сортировки различных типов данных, включая целочисленные, символьные и пользовательские структуры данных. Это делает его универсальным инструментом, который может быть применен для широкого спектра сортировочных задач.
3. Гибкость: qsort позволяет определить пользовательскую функцию сравнения, которая определяет порядок сортировки элементов. Это дает программисту гибкость в определении пользовательских правил сортировки. Например, можно определить собственный порядок сортировки для пользовательской структуры данных или определить сортировку в обратном порядке.
4. Простота использования: qsort является стандартной функцией в языке C и имеет простой интерфейс. Она требует от программиста только предоставление указателя на массив, количество элементов и размер каждого элемента. Это делает qsort простым в использовании и не требует сложной настройки или наличия дополнительного кода.

Тем не менее, алгоритм не идеальный и имеет некоторые недостатки. Например:

1. Неустойчивость: qsort не гарантирует сохранение относительного порядка элементов с одинаковыми значениями. Если вам важно сохранить порядок элементов с одинаковыми значениями, вам может потребоваться использовать другой алгоритм сортировки.
2. Неэффективность для небольших массивов: qsort использует алгоритм сортировки QuickSort, который имеет лучший случайный временной сложность O(n log n). Однако для небольших массивов может быть более эффективным использовать другие алгоритмы, такие как сортировка вставками (Insertion Sort) или сортировка пузырьком (Bubble Sort).
3. Необходимость передачи функции сравнения: qsort требует передачи пользовательской функции сравнения, которая определяет порядок сортировки. Это может быть неудобно и более громоздко в использовании по сравнению с алгоритмами сортировки, которые не требуют такого дополнительного шага.
4. Отсутствие поддержки оптимизаций: В реализации qsort могут отсутствовать оптимизации, которые могут быть применены в специфических случаях сортировки. Некоторые алгоритмы сортировки, такие как сортировка пузырьком, могут быть оптимизированы для лучшей производительности в специфических случаях, например, когда массив уже частично отсортирован.

**Литература**

1. qsort – программирование на C и C++ [Электронный ресурс]: URL: <http://www.c-cpp.ru/content/qsort>
2. Левитин А. В. Глава 4. Метод декомпозиции: Быстрая сортировка // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006. — С. 174—179. — 576 с.
3. Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Глава 7. Быстрая сортировка // Алгоритмы: построение и анализ = Introduction to Algorithms / Под ред. И. В. Красикова. — 2-е изд. — М.: Вильямс, 2005. — С. 198—219.
4. Быстрая сортировка - Википедия [Электронный ресурс]: URL: [https://ru.wikipedia.org/wiki/Быстрая\_сортировка](4.%09https:/ru.wikipedia.org/wiki/Быстрая_сортировка)

**Приложение**

int main() {

char ans;

printf("Хотите провести эксперимент перед началом программы?: (y/n) ");

scanf(" %c", &ans);

getchar();

if (ans == 'y' || ans == 'Y') {

xmlReader();

getchar();

}

char input[1000];

printf("\nВведите массив строк через пробел: ");

fgets(input, sizeof(input), stdin);

int length = strlen(input);

if (input[length - 1] == '\n') {

input[length - 1] = '\0';

}

char\* substrings[100];

int count = 0;

char\* token = strtok(input, " ");

while (token != NULL) {

substrings[count++] = token;

token = strtok(NULL, " ");

}

my\_qsort(substrings, count, sizeof(char\*), compare\_strings);

printf("Отсортированный массив строк:\n");

for (int i = 0; i < count; i++) {

printf("%s ", substrings[i]);

}

return 0;

}

int my\_qsort(void\* base, size\_t nmemb, size\_t size, cmp\_t cmp) {

int swaps = 0;

char\* arr = base;

char\* pivot = &arr[(nmemb / 2) \* size];

size\_t left = 0, right = nmemb - 1;

while (left <= right) {

while (cmp(&arr[left \* size], pivot) < 0) {

++left;

}

while (cmp(&arr[right \* size], pivot) > 0) {

--right;

}

if (left <= right) {

swap(&arr[left \* size], &arr[right \* size], size);

swaps += 1;

++left;

--right;

}

}

if (right >= 1) {

swaps += my\_qsort(arr, right + 1, size, cmp);

}

if (left < nmemb - 1) {

swaps += my\_qsort(&arr[left \* size], nmemb - left, size, cmp);

}

return swaps;

}

int compare\_strings(const void\* a, const void\* b) {

return strcmp(\*(const char\*\*)a, \*(const char\*\*)b);

}

int swap(void\* a, void\* b, size\_t size) {

char\* p = a, \* q = b, tmp;

for (size\_t i = 0; i < size; ++i) {

tmp = p[i];

p[i] = q[i];

q[i] = tmp;

}

return 0;

}

int xmlReader() {

const char\* filePath = "./code/experiment.xml";

xmlDoc \*doc = xmlReadFile(filePath, NULL, 0);

xmlNode \*root = NULL;

if (doc == NULL) {

printf("Failed to parse XML document.\n");

return 1;

}

root = xmlDocGetRootElement(doc);

if (root == NULL) {

printf("Empty XML document.\n");

xmlFreeDoc(doc);

return 1;

}

parseNode(root);

xmlFreeDoc(doc);

xmlCleanupParser();

return 0;

}

void parseNode(xmlNode \*node) {

xmlNode \*curNode = NULL;

for (curNode = node; curNode; curNode = curNode->next) {

if (curNode->type == XML\_ELEMENT\_NODE) {

if (xmlStrcmp(curNode->name, (const xmlChar \*)"nodes") == 0) {

xmlChar \*countAttr = xmlGetProp(curNode, (const xmlChar \*)"count");

xmlChar \*lengthAttr = xmlGetProp(curNode, (const xmlChar \*)"length");

xmlChar \*maxAttr = xmlGetProp(curNode, (const xmlChar \*)"max");

if (countAttr && lengthAttr && maxAttr) {

int count = atoi((const char \*)countAttr);

int length = atoi((const char \*)lengthAttr);

int max = atoi((const char \*)maxAttr);

printf("\nCount: %d, Length: %d, Max: %d", count, length, max);

experiment(count, length, max);

}

xmlFree(countAttr);

xmlFree(lengthAttr);

xmlFree(maxAttr);

}

parseNode(curNode->children);

}

}

}

void experiment(int count, int length, int max) {

int total[count];

srand(time(NULL));

for (int k = 0; k < count; k++) {

int array[length];

for (int i = 0; i < length; i++) {

int random\_int = rand() % max + 1;

array[i] = random\_int;

}

total[k] = my\_qsort(array, length, sizeof(int), (cmp\_t) &strcmp);

}

int average = 0;

for (int i = 0; i < count; i++) {

average += total[i];

}

average = average / count;

printf("\nПолучившееся среднее количество swap: %d\n", average);

}