Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

**Отчет по лабораторной работе**

**«Сортировки»**

**Выполнил**:

студент группы 382003-1

Филатов Андрей Александрович

**Проверил**:

ассистент каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2020

**Содержание**

1. [Постановка задачи 3](#_Toc26962562)
2. Сортировки……………………..…………………………………………………….………….4
   1. Сортировка вставками (*Selection sort*)…………………………………………………….4
      * 1. Метод решения и оценка сложности алгоритма…………………………..4
        2. Результаты экспериментов………………………………………………….5
   2. Сортировка Шелла (*Shell sort)………………………………………..*…………………………6
      * 1. Метод решения и оценка сложности алгоритма…………………………..6
        2. Результаты экспериментов………………………………………………….7
   3. Сортировка слиянием *(Merge sort*)………………………………………………………..8
      * 1. Метод решения и оценка сложности алгоритма…………………………..8
        2. Результаты экспериментов………………………………………………….9
   4. Поразрядная сортировка *(Radix sort)*…………………………………………………….10
      * 1. Метод решения и оценка сложности алгоритма…………………………10
        2. Подтверждение корректности……………………………………………..11
        3. Особенности программной реализации…………………………………..11
        4. Результаты экспериментов………………………………………………...12

**3.**  [Руководство пользователя………………………………………………………………………………………………………………….](#_Toc26962564)13

**4.** [Описание программной реализации](#_Toc26962565)……………………………………………………………………………………………………14

[Заключение](#_Toc26962568) 15

[Приложение](#_Toc26962569) 17

# Постановка задачи

Реализовать алгоритмы сортировки чисел типа double на языке С.

Реализованы:

* сортировка вставками (*Selection sort*);
* сортировка Шелла (*Shell sort*);
* сортировка слиянием *(Merge sort*);
* поразрядная сортировка *(Radix sort).*

# Метод решения

* 1. **Сортировка вставками (*Selection sort*);**

**2.1.1 Метод решения и оценка сложности алгоритма**

Для сортировки выбором проходим по массиву в поисках минимального элемента. Найденный минимум меняем местами с первым элементом. Неотсортированная часть массива уменьшилась на один элемент (не включает первый элемент, куда переставили найденный минимум). К этой неотсортированной части применяем те же действия — находим минимум и ставим его на первое место в неотсортированной части массива. И так продолжаем до тех пор, пока неотсортированная часть массива не уменьшится до одного элемента.

Посчитаем сложность данного алгоритма. Пусть список содержит n элементов. Сначала нужно найти минимум среди n элементов списка, что потребует n операций. Потом нужно найти наименьший из n-1 элемента, на это нужно n-1 операция. Потом нужно n-2 операции и т. д. Общее число операций равно

Таким образом, сортировка выбором — квадратичный алгоритм, время его работы пропорционально квадрату от размера списка.

Сортировка выбором не требует дополнительной памяти для работы – дополнительная память .

Среднее количество обменов, необходимое для выполнения сортировки массива .

**2.1.2 Результаты экспериментов**

По данным экспериментов видно, что сортировка выбором — квадратичный алгоритм, время его работы пропорционально квадрату от размера сортируемого списка.

**Рис. 1** Зависимость времени работы алгоритма от количества элементов сортировки.

**Рис. 2** Зависимость количества обменов и сравнений от количества элементов сортировки.

Тесты проводились на ноутбуке Aser Intel Core i5-7200U CPU 2.50 GHz

# 2.2 Сортировка Шелла (*Shell sort*);

**2.2.1 Метод решения и оценка сложности алгоритма**

Сортировка Шелла – это модификация алгоритма сортировки вставками, сравнивающая элементы на расстоянии . Шелл предлагал использовать .

Используемый алгоритм для решения:

1. Начало.
2. **Шаг 0.**
3. **Шаг 1.** Разобьем массив на списки элементов, отстающих друг от друга на . Таких списков будет .
4. **Шаг 2.** Отсортируем элементы каждого списка.
5. **Шаг 3.** Объединим списки обратно в массив. Уменьшим *i*. Если *i*неотрицательно — вернемся к **Шаг 1.**
6. Конец.

Сложность алгоритма будет зависеть от выбора шага. Посчитаем сложность алгоритма для указанного выше шага. Пусть список содержит n элементов, тогда общее число операций равно

Таким образом, сортировка Шелла с выбранными шагами – алгоритм, имеющий время в худшем случае .

Сортировка Шелла не требует дополнительной памяти для работы – дополнительная память .

Среднее количество обменов, необходимое для выполнения сортировки массива .

**2.2.2 Результаты экспериментов**

По данным экспериментов видно, что сортировка выбором — квадратичный алгоритм, время его работы пропорционально квадрату от размера сортируемого списка.

**Рис. 1** Зависимость времени работы алгоритма от количества элементов сортировки.

**Рис. 2** Зависимость количества обменов и сравнений от количества элементов сортировки.

Тесты проводились на ноутбуке Aser Intel Core i5-7200U CPU 2.50 GHz

# 2.3 Сортировка слиянием (*Merge sort*);

**2.3.1 Метод решения и оценка сложности алгоритма**

Алгоритм сортировки слиянием:

1. Если в рассматриваемом массиве один элемент, то он уже отсортирован — алгоритм завершает работу.
2. Иначе массив рекурсивно разбивается пополам, и каждая из половин делиться до тех пор, пока размер очередного подмассива не станет равным единице;
3. После сортировки двух частей массива к ним применяется процедура слияния, которая по двум отсортированным частям получает исходный отсортированный массив.

Алгоритм слияния:

1. Сравниваем элементы массивов (начиная с начала) и меньший из них записываем в финальный.
2. В массиве, у которого оказался меньший элемент, переходим к следующему элементу и сравниваем теперь его.
3. В конце, если один из массивов закончился, мы просто дописываем в финальный оставшийся массив.

Посчитаем сложность данного алгоритма. Пусть список содержит n элементов.

Тогда за его можно разделить на две части и после сортировки слить их вместе. Каждая из этих двух частей имеет размер n/2, и за  шагов каждую из них можно поделить на две части размером n/4 и затем после сортировки слить их вместе. Аналогично, четыре части размером n/4 за суммарное  шагов делятся на части размером n/8 и сливаются вместе. Этот процесс «в глубину» продолжается столько раз, сколько раз можно число n делить на 2, до тех пор, пока размер части не станет равен 1, то есть . Итого, общая сложность этого алгоритма равна

Одним из главных недостатков сортировки слиянием то, что данный алгоритм требует дополнительную память для работы – дополнительная память .

Количество обменов, необходимое для выполнения сортировки массива .

**2.3.2 Результаты экспериментов**

По данным экспериментов видно, что сортировка слиянием — логарифмический алгоритм, время его работы пропорционально произведению размера сортируемого списка на его логарифм.

**Рис. 1** Зависимость времени работы алгоритма от количества элементов сортировки.

**Рис. 2** Зависимость времени работы алгоритма от количества элементов сортировки.

Тесты проводились на ноутбуке Aser Intel Core i5-7200U CPU 2.50 GHz

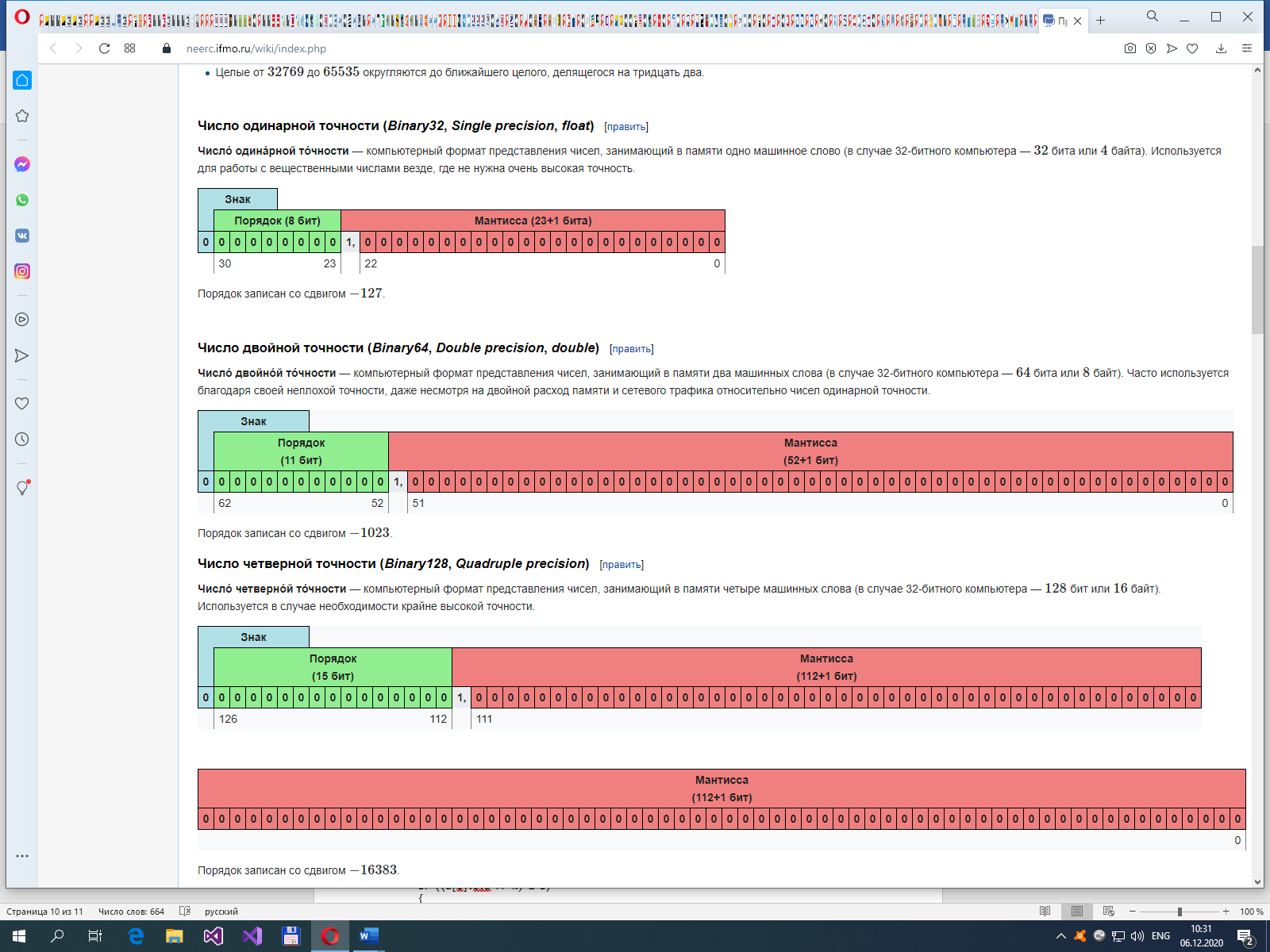
# 2.4 Поразрядная сортировка (*Radix sort*);

**2.4.1 Метод решения и оценка сложности алгоритма**

Изначально поразрядная сортировка предназначалась для сортировки целых чисел, записанных цифрами. Но так как в памяти компьютеров любая информация записывается целыми числами, алгоритм пригоден для сортировки любых объектов, запись которых можно поделить на «разряды», содержащие сравнимые значения. Например, так сортировать можно не только числа, записанные в виде набора цифр, но и строки, являющиеся набором символов, и вообще произвольные значения в памяти, представленные в виде набора байт.

**Число двойной точности –** это компьютерный формат представления чисел, занимающий в памяти два машинных слова (в случае 32-битного компьютера — 64 бита или 8 байт).

 В наиболее распространённом формате (стандарт IEEE 754) число с плавающей запятой представляется в виде набора битов, часть из которых кодирует собой мантиссу числа, другая часть — показатель степени, и ещё один бит используется для указания знака числа (0 — если число положительное, 1 — если число отрицательное). При этом порядок записывается как целое число, а мантисса- своей дробной частью в двоичной системе счисления.



 Запишем код числа «−312,3125» в машинном представлении

1. Двоичная запись модуля этого числа имеет вид 100111000,0101
2. Имеем 100111000,0101 == 1,001110000101×28.
3. Получаем смещенный порядок 8 + 1023 == 1031. Далее имеем 1031(10) == 100000001111(2).
4. Окончательно

|  |  |  |
| --- | --- | --- |
| 1 | 10000000111 | 0011100001010000000000000000000000000000000000000000 |
| 63 | 62..52 | 51..0 |

Данное представление будем использовать для поразрядной сортировки. Сам алгоритм поразрядной сортировки состоит в последовательной сортировке по каждому разряду, в порядке от старшего разряда к младшему, после чего последовательности будут расположены в требуемом порядке.

Посчитаем сложность данного алгоритма. Пусть список для сортировки содержит n w-значных чисел, и каждая цифра может принимать значения 0 и 1.

Идём двумя указателями с двух сторон массива, меняя элементы, стоящие не на своём месте. Проход по массиву . Для w разрядов

Т.к. количество разрядов — константа, то сложность поразрядной сортировки составляет , то есть она линейно зависит от количества сортируемых чисел.

Поразрядная сортировка не требует дополнительной памяти для работы – дополнительная память .

Количество обменов – в лучшем случае 0 в худшем .

**2.4.2 Подтверждение корректности.**

Докажем, что данный алгоритм работает верно, используя метод математической индукции по номеру разряда. Пусть w — количество разрядов в сортируемых объектах.

**База**: w=1. Очевидно, что алгоритм работает, верно, потому что в таком случае мы просто сортируем старшие разряды какой-то заранее выбранной устойчивой сортировкой.

**Переход**: пусть для w=k алгоритм правильно отсортировал последовательности по k старшим разрядам.

В таком случае массив будет разделён на групп. В первой группе находится элементы, которые можно представить в двоичном виде, как 64 единицы, а в последней, как 0 и 63 единицы.

Таким образом доказано, что данный алгоритм работает корректно для чисел с любым количеством разрядов.

**2.4.3 Особенности программной реализации.**

Будем использовать побитное представление числа (в стандарте IEEE 754 с двойной точностью) для поразрядной сортировки. Для доступа к битовому представлению числа будем использовать объединение (**union**), поскольку его члены ссылаются на общую память, поэтому **union** может быть использован для упаковки/распаковки данных. Это конкретное применение объединений включает в себя запись значения в один член объединения и чтение другого его члена.

**2.4.4 Результаты экспериментов**

По данным экспериментов видно, что поразрядная сортировка — линейный алгоритм, время его работы пропорционально размеру сортируемого списка.

**Рис. 1** Зависимость времени работы алгоритма от количества элементов сортировки.

**Рис. 2** Зависимость времени работы алгоритма от количества элементов сортировки.

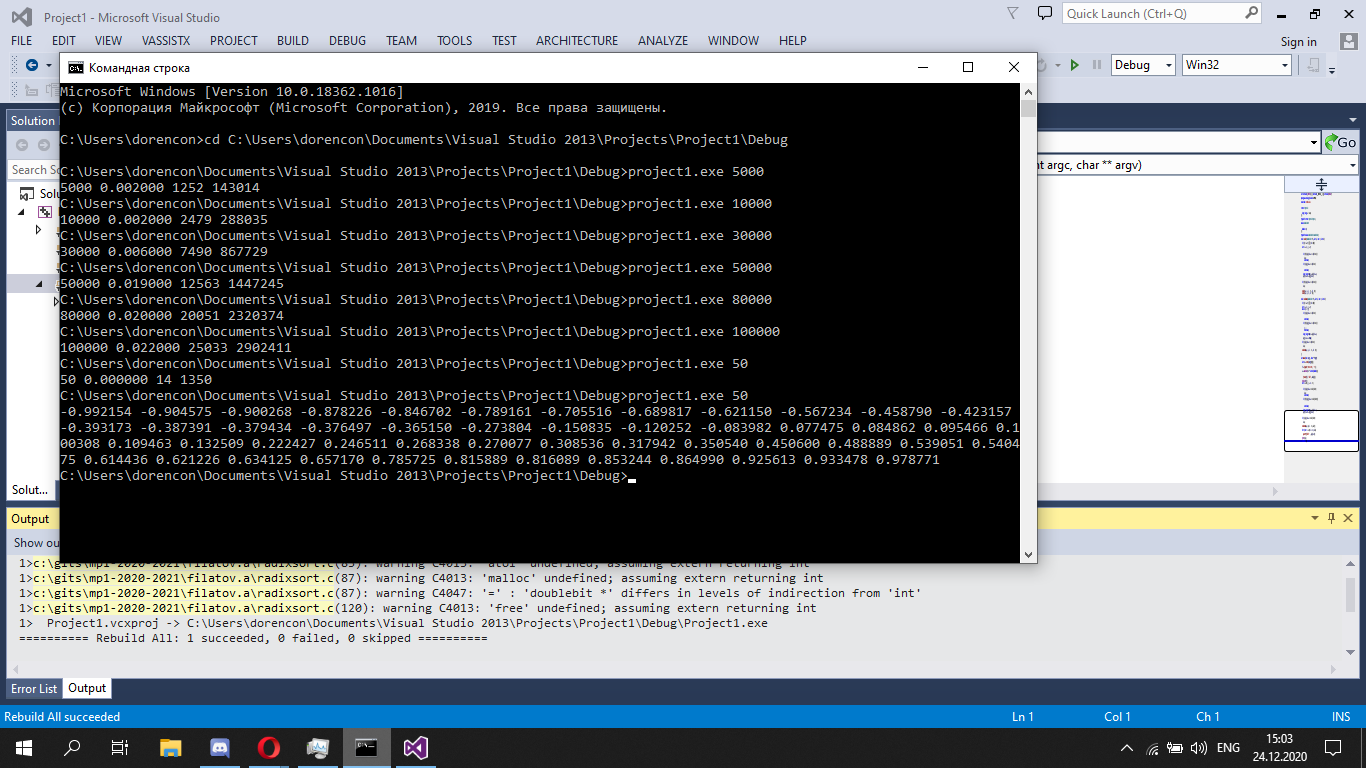
Тесты проводились на ноутбуке Aser Intel Core i5-7200U CPU 2.50 GHz

# 3. Руководство пользователя

Для каждой сортировки существует файл SelSort.c, ShellSort.c, MergeSort.c, RadixSort.c. Для использования программы пользователь должен иметь файл с именем test.txt, содержащий неотсортированный массив.

Для удобства тестирования, программа запускается из командной строки с одним аргументом. Аргумент – сколько последовательных чисел прочитать из файла.

Пример командной строки



Результат работы программы – отсортированный массив, будет выведен на экран.

# Описание программной реализации

Сортировка выбором и сортировка Шелла реализованы в функции main. Сортировка слиянием и поразрядная сортировка реализованы с помощью рекурсии соответсвующих функций.

Void merge(double\*a, double\*b, int l, int r)  
 Функция сортировки слиянием. Получает на вход массив a, который надо отсортировать, дополнительный массив b, левую и правую границы сортировки l и r. Если границы совпадают ничего не происходит. В противном случае массив делится пополам и для обеих половин запускается merge, затем они сливаются.

Void radix(doublebit\* a, int l, int r, int k)  
 Функция поразрядной сортировки. Получает на вход массив a, который надо отсортировать, левую и правую границы сортировки l и r, количество сортируемых разрядов k. Сортирует массив по k разряду, затем по k-1...

Void radixmn(doublebit\* a, int l, int r, int k)  
 Функция аналогичная radix, для отрицательных чисел.

# Заключение

В этой лабораторной работе я научился реализовывать различные виды сортировок.

Рассмотрим сравнение времени работы сортировок.

Сортировка выбором – это простая в реализации сортировка, но на больших массивах данных будет работать очень медленно. Это ограничивает его применимость в реальных задачах.

Сортировка Шелла – это модифицированная сортировка вставками, зависящая от выбора шага.

Классический выбор шага, предложенный Шеллом, на некоторых массивах имеет время . Это ограничивает его применимость в реальных задачах.

Однако, современные модификации данного алгоритма улучшают его скоростные характеристики. Например, А.А. Папернов, Г.В. Стасевич предложили выбор шага как , что приводит к времени сортировки .

Дальнейшее улучшение было получено В. Праттом. Если все шаги при сортировке выбираются из множества чисел вида , то время выполнения алгоритма будет .

Сортировка слиянием – это сортировка, работающая по принципу “Разделяй и властвуй”. Алгоритм был изобретён [Джоном фон Нейманом](https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D0%BD_%D0%9D%D0%B5%D0%B9%D0%BC%D0%B0%D0%BD,_%D0%94%D0%B6%D0%BE%D0%BD) в [1945 году](https://ru.wikipedia.org/wiki/1945_%D0%B3%D0%BE%D0%B4). Достаточно быстрый алгоритм,

общая сложность этого алгоритма равна

Одним из главных недостатков сортировки слиянием то, что данный алгоритм требует дополнительную память для работы – дополнительная память . Однако, существуют модифицированные алгоритмы слияния, которые при тех же временных затратах, требуют только (1 дополнительной памяти.

Изначально поразрядная сортировка предназначалась для сортировки целых чисел, записанных цифрами. Но так как в памяти компьютеров любая информация записывается целыми числами, алгоритм пригоден для сортировки любых объектов, запись которых можно поделить на «разряды», содержащие сравнимые значения. Например, так сортировать можно не только числа, записанные в виде набора цифр, но и строки, являющиеся набором символов, и вообще произвольные значения в памяти, представленные в виде набора байт.

# Приложение

**SelSort**  
for (int i = 0; i < n; i++)

{

m = i;

for (int j = i + 1; j < n; j++)

{

if (a[j] < a[m])

m = j;

}

temp = a[i];

a[i] = a[m];

a[m] = temp;

}

**ShellSort**  
for (int s = n / 2; s > 0; s /= 2)

{

for (int i = s; i < n; i++)

{

for (int j = i - s; (j >= 0) && (a[j] > a[j + s]); j -= s)

{

temp = a[j];

a[j] = a[j + s];

a[j + s] = temp;

}

}

}

**MergeSort**  
void merge(double\* a, double\* b, int l, int r)

{

if (l >= r)

{

return;

}

int m = (l + r) / 2;

merge(b, a, l, m);

merge(b, a, m + 1, r);

int i = l, j = m + 1;

while ((i <= m) || (j <= r))

{

if ((j <= r) && ((i > m) || (b[i] > b[j])))

{

a[i + j - m - 1] = b[j];

j++;

}

else

{

a[i + j - m - 1] = b[i];

i++;

}

}

return;

}

**RadixSort**  
struct bytes8

{

long long a : 64;

};

typedef struct bytes8 bytes;

union doublebit

{

bytes b;

double d;

};

typedef union doublebit doublebit;

void radix(doublebit \*a, int l, int r, int k)

{

if ((l >= r) || (k < 0))

return;

int i = l, j = r;

while (i < j)

{

if (!((a[i].b.a >> k) & 1))

{

i++;

continue;

}

if ((a[j].b.a >> k) & 1)

{

j--;

continue;

}

long long temp = a[i].b.a;

a[i].b.a = a[j].b.a;

a[j].b.a = temp;

}

if (!((a[i].b.a >> k) & 1))

{

i++;

}

radix(a, l, i - 1, k - 1);

radix(a, i, r, k - 1);

}

void radixmn(doublebit \*a, int l, int r, int k)

{

if ((l >= r) || (k < 0))

return;

int i = l, j = r;

while (i < j)

{

if ((a[i].b.a >> k) & 1)

{

i++;

continue;

}

if (!((a[j].b.a >> k) & 1))

{

j--;

continue;

}

long long temp = a[i].b.a;

a[i].b.a = a[j].b.a;

a[j].b.a = temp;

}

if ((a[i].b.a >> k) & 1)

{

i++;

}

radixmn(a, l, i - 1, k - 1);

radixmn(a, i, r, k - 1);

}