МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ТЮМЕНСКИЙ ИНДУСТРИАЛЬНЫЙ УНИВЕРСИТЕТ»

ИНСТИТУТ ГЕОЛОГИИ И НЕФТЕГАЗОДОБЫЧИ

КАФЕДРА КИБЕРНЕТИЧЕСКИХ СИСТЕМ

# ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к самостоятельной работе

по дисциплине «Алгоритмы и структуры данных»

на тему: Исследование эффективности методов сортировок массива:

алгоритм сортировки вставками;

алгоритм Timsort

Вариант 10

Выполнил:

студент группы ИИПб-22-1

Войку Владислав

Проверил:

Преподаватель Лозикова И.О.

Тюмень ТИУ

2023

**Цель работы:**

1. Изучение и реализация заданных методов сортировки.
2. Экспериментальное и теоретическое исследование эффективности методов сортировки.
3. На основе исследований определить эффективный метод.

**Задание:**

1. Реализовать в виде функций методы сортировки одномерного массива по заданному варианту Таблица 1.

2. Разработать консольную программу с заданным интерфейсом в виде меню операций и набором дополнительных операций для вывода показателей эффективности. Архитектура программы - функциональная декомпозиция всех операций.

Интерфейс программы включает следующие операции:

* Изменение размера массива с шагом в заданном диапазоне.
* Формирование в массиве случайной выборки значений.
* Формирование в массиве упорядоченной выборки значений.
* Чтение/запись заданного набора данных из файла/консоль.
* Выполнения методов сортировки (по варианту задания).

Для тестирования эффективности алгоритмов сортировки программа реализует вычисления и вывод в виде таблицы показателей эффективности:

* Число выполненных сравнений в процессе сортировки.
* Число выполненных обменов в процессе сортировки.
* Вычисление системного времени работы каждого алгоритма сортировки на одинаковом наборе данных.

1. Выполнить отладку и тестирование программы с помощью меню операций. Провести тестовые испытания методов сортировки на массиве размером N=10 и проверить правильность результатов работы.
2. Выполнить сравнительное тестирование эффективности алгоритмов сортировки для лучшего, худшего и среднего случаев для массива, где размер N изменяется на отрезке [1000; 100000] с шагом dN=500. Для каждого случая фиксировать показатели: число выполненных сравнений, обменов и времени. Вывести таблицу показателей.
3. Провести анализ экспериментальных показателей эффективности алгоритмов сортировки, сравнение с теоретическими оценками эффективности методов сортировки и сделать вывод об эффективности заданных методов.

Вариант – 10.

**Метод сортировки 1 –** Алгоритм сортировки вставками

**Метод сортировки 2 –** Алгоритм сортировки Timsort

**Описание алгоритма сортировки вставками**

Основная идея алгоритма заключается в том, чтобы пройтись по массиву от второго элемента до последнего и поочередно сравнивать каждый элемент с предыдущим, сдвигая элементы на одну позицию вправо, если они больше текущего элемента, до тех пор, пока не будет найден элемент, который меньше текущего, или пока не будет достигнута граница массива.

В цикле for перебираются элементы массива от left + 1 до  right - 1. Переменная key хранит текущий элемент, который нужно вставить на свое место в уже отсортированной части массива.

Внутренний цикл while начинается с предыдущего элемента j = i - 1 и продолжается до тех пор, пока j больше или равен left и элемент arr[j] больше текущего элемента key. В цикле происходит сдвиг элементов на одну позицию.

После нахождения правильного места для текущего элемента, он вставляется в массив на позицию j + 1.

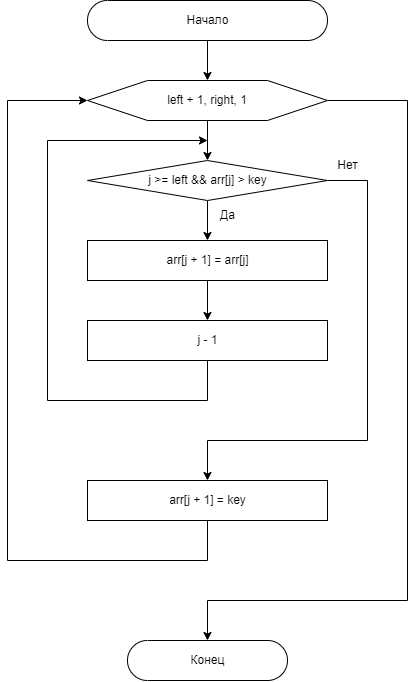


Рисунок 1. Схема алгоритма сортировки вставками

**Описание алгоритма сортировки Timsort**

Он использует комбинацию сортировки вставками и сортировки слияниями. Сначала массив разбивается на меньшие подмассивы фиксированного размера (называемые minrun), которые затем сортируются с помощью сортировки вставками. Затем отсортированные minrun объединяются с помощью сортировки сляниями.

Шаги алгоритма:

* Разбиение на minrun:
  + Определение размера minrun с помощью с функции GetMinRun().
  + Используя цикл for, разбиваем исходный массив на minrun размера от 32 до 64.
  + Каждый minrun сортируется с помощью сортировки вставками.
* Объединение отсортированных minrun:
  + С помощью вложенных циклов, объединяем отсортированные мини-беги в более крупные бега.
  + Каждый бег имеет размер 2\*minRunSize.
  + Отсортированные подмассивы объединяются с помощью Merge Sort.
* Окончательное объединение:
  + Последний бег может быть меньшего размера, чем 2\*minRunSize.
  + Оставшиеся подмассивы сливаются в конечный отсортированный массив с помощью Merge Sort.

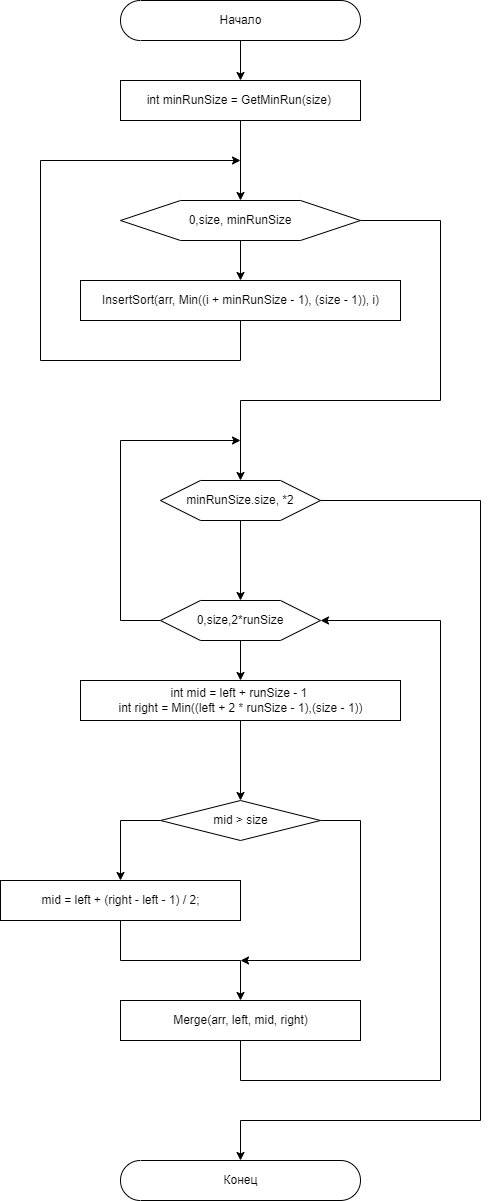


Рисунок 2. Блок-схема алогритма сортировки Timsort

**Описание массива, способы формирования исходных данных**

Для наилучшего случая массив может быть уже отсортированным. В этом случае, сортировка массива может занять минимальное время, так как не потребуется выполнение никаких операций сравнения и перестановки элементов.

Для наихудшего случая массив может быть отсортированным в обратном порядке. В этом случае, сортировка массива будет требовать максимальное количество операций сравнения и перестановки элементов.

Для среднего случая массив может быть заполнен случайными числами. В этом случае, сортировка массива потребует промежуточное количество операций сравнения и перестановки элементов. Код для формирования исходных данных представлен в Приложении 2.

**Определение переменных программы (показателей эффективности)**

Для хранения показателей (число сравнений, обменов и системное время) была описана новая структура данных, которая после всего была выведена в тестовый файл. Всего вышло 6 файлов (insertSort(best).txt; insertSort(worst).txt; insertSort(average).txt; TimSort(best).txt; TimSort(worst).txt; TimSort(average).txt).

Для подсчёта количества сравнений и обменов в функции сортировки подавались по ссылке специальные переменные а после их результат выводился в струтуру.

Для получения системного времени была использована библиотека ‘chrono’. Была использована функция steady\_clock::now() для получения временной метки в начале и в конце выполнения операции, которую необходимо измерить. Затем используется функция duration\_cast для вычисления разницы между этими временными метками в микросекундах. Далее необходимо сохранить засеченное время в переменную типа int, приведя значение метода count() к типу int с помощью static\_cast<int>. После время заносится в структуру.

Выше описанные опреации описаны в приложении 1 и 3.

**Результаты тестирования.**

1. число обменов и число сравнений для лучшего, худшего и среднего случаев для первого алгоритма сортировки (графики совмещены в одной системе координат) при разных размерах исходных данных,

2. число обменов и число сравнений для лучшего, худшего и среднего случаев для второго алгоритма сортировки (графики совмещены) при разных размерах исходных данных,

3. временной график, где показана зависимость времени выполнения от размера массива для обоих методов сортировки,

4. число сравнений алгоритмов методов сортировки для среднего случая при разных размерах исходных данных,

5 число обменов алгоритмов методов сортировки для среднего случая при разных размерах исходных данных.

**Сравнительный анализ.**

Для алгоритма сортировки вставками теоретическая оценка временной сложности указывает на то, что для лучшего случая, когда входные данные уже упорядочены, количество операций должно быть O(n), а для худшего и среднего случаев - O(n^2). Чтобы проверить эту оценку, можно написать программу, которая реализует алгоритм сортировки вставками и измеряет количество операций, которые он выполняет на разных входных данных.

Для алгоритма Timsort, который является комбинацией алгоритмов сортировки вставками и сортировки слиянием (Merge Sort), теоретическая оценка временной сложности указывает на то, что для лучшего и среднего случаев количество операций должно быть O(nlog(n)), а для худшего случая - O(nlog(n)) с константой, которая зависит от распределения элементов. Чтобы проверить эту оценку, можно также написать программу, которая реализует алгоритм Timsort и измеряет количество операций, которые он выполняет на разных входных данных.

Из приведённых выше данных можно увидеть, что теоретическая сложность алгоритмов соответствует экспериментальной сложности.

**Выводы**

Исходя из полученных результатов, можно сделать следующие выводы:

1. Алгоритм вставки демонстрирует квадратичную сложность в худшем и среднем случае, и линейную сложность в лучшем случае что соответствует теоретическим оценкам.
2. Алгоритм Timsort демонстрирует линейно-логарифмическую сложность в худшем и среднем случае, что соответствует теоретическим оценкам. В лучшем случае он также показывает линейную сложность, что является ожидаемым результатом. Таким образом, можно сделать вывод, что алгоритм Timsort показывает лучшую производительность, чем алгоритм вставки, особенно на больших наборах данных.
3. Необходимо учитывать, что проведенные тесты были выполнены на конкретном компьютере с определенной конфигурацией, что может повлиять на результаты. Поэтому для получения более точных результатов, необходимо проводить тестирование на разных компьютерах и с разными наборами данных.

Приложение 1 Исходный текст алгоритмов сортировок программы.

void InsertSort(int\*& arr, long long& countComp, long long& countEx, int right, int left = 0)

{

countComp = 0;

countEx = 0;

for (int i = left + 1; i < right; i++)

{

bool count{ false };

int key = arr[i];

int j = i - 1;

while (j >= left && arr[j] > key)

{

arr[j + 1] = arr[j];

j--;

countComp++;

countEx++;

count = true;

}

if (!count)

{

countComp++;

}

arr[j + 1] = key;

}

}

int Min(int a, int b)

{

if (a > b)

{

return b;

}

else

{

return a;

}

}

int GetMinRun(int size)

{

int r{ 0 };

while (size >= 64)

{

r |= size & 1;

size >>= 1;

}

return size + r;

}

void Merge(int\*& arr, long long& countComp, long long& countEx, int left, int mid, int right)

{

int sizeT = right - left + 1;

int\* temp{ new int[sizeT] };

int leftIndex = { left };

int rightIndex{ mid + 1 };

int tempIndex{ 0 };

while (leftIndex <= mid && rightIndex <= right)

{

if (arr[leftIndex] > arr[rightIndex])

{

temp[tempIndex] = arr[rightIndex];

rightIndex++;

}

else

{

temp[tempIndex] = arr[leftIndex];

leftIndex++;

}

tempIndex++;

countComp++;

countEx++;

}

while (leftIndex <= mid)

{

temp[tempIndex] = arr[leftIndex];

leftIndex++;

tempIndex++;

countEx++;

}

while (rightIndex <= right)

{

temp[tempIndex] = arr[rightIndex];

rightIndex++;

tempIndex++;

countEx++;

}

for (int i = left, j = 0; i <= right; i++, j++)

{

arr[i] = temp[j];

}

delete[] temp;

}

void TimSort(int\*& arr, int size, int& countComp, int& countEx)

{

int minRunSize{ GetMinRun(size) };

for (int i = 0; i < size; i += minRunSize)

{

long long tCC{ 0 };

long long tCE{ 0 };

InsertSort(arr, tCC, tCE, Min((i + minRunSize - 1), (size - 1)), i);

countComp += tCC;

countEx += tCE;

}

for (int runSize = minRunSize; runSize < size; runSize = 2 \* runSize)

{

for (int left = 0; left < size; left += 2 \* runSize)

{

long long tCC{ 0 };

long long tCE{ 0 };

int mid{ left + runSize - 1 };

int right{ Min((left + 2 \* runSize - 1),(size - 1)) };

if (mid > size)

{

mid = left + (right - left - 1) / 2;

}

Merge(arr, tCC, tCE, left, mid, right);

countComp += tCC;

countEx += tCE;

}

}

}

Приложение 2 Исходный текст функций формирования исходных наборов данных.

bool Write2Bin(int\* arr, int size, string name)

{

ofstream file(name, ios::out | ios::binary);

if (file.is\_open())

{

file.write((const char\*) arr, size \* sizeof(int));

file.close();

return true;

}

else

{

return false;

}

}

void RandValues(int\* ARR, int size)

{

random\_device seed;

mt19937 generator(seed());

uniform\_int\_distribution<int> distribution(-50000, 49999);

for (int i = 0; i < size; i++)

ARR[i] = distribution(generator);

}

//подготовка исходных данных и занесение их в бинарный файл.

case 3:

{

int size;

string name;

int countComps{ 0 };

cout << "Сколько элементов в массиве?\n"; cin >> size;

int\* arr{ new int[size] };

int j{ -50000};

for (int i = 0; i < 100000; i++,j++)

{

arr[i] = j;

}

cout << "Название: "; cin >> name;

if (Write2Bin(arr, size, name))

{

cout << "Всё норм";

}

else

{

cout << "Ошибка!!!";

}

;

}

break;

case 4:

{

int size;

string name;

int countComps{ 0 };

cout << "Сколько элементов в массиве?\n"; cin >> size;

int\* arr{ new int[size] };

int j{ 49999 };

for (int i = 0; i < 100000; i++,j--)

{

arr[i] = j;

}

cout << "Название: "; cin >> name;

if (Write2Bin(arr, size, name))

{

cout << "Всё норм";

}

else

{

cout << "Ошибка!!!";

}

}

break;

case 5:

{

int size;

string name;

int countComps{ 0 };

cout << "Сколько элементов в массиве?\n"; cin >> size;

int\* arr{ new int[size] };

if (Write2Bin(arr, size, name))

{

cout << "Всё норм";

}

else

{

cout << "Ошибка!!!";

}

}

break;

//вывод в тескстовый для проверки

case 6:

{

int size;

cout << "Сколько элементов в массиве?\n"; cin >> size;

int\* arr{ new int[size] };

ReadBin(arr, size, "averageCase.bin");

ArrOutFile(arr, size, "averageCase.txt");

ReadBin(arr, size, "bestCase.bin");

ArrOutFile(arr, size, "bestCase.txt");

ReadBin(arr, size, "worstCase.bin");

ArrOutFile(arr, size, "worstCase.txt");

}

Приложение 3. Текст программы тестирования эффективности сортировок.

struct table

{

stringstream zero;

stringstream exchanges;

stringstream comparisons;

stringstream clock;

table() : zero(";"), exchanges(""), comparisons(""), clock("") {}

};

bool ReadBin(int\* arr, int size, string name)

{

ifstream file(name, ios::in | ios::binary);

if (file)

{

file.read((char\*) arr, size \* sizeof(int));

file.close();

return true;

}

else

{

return false;

}

}

void fillStructInsert(int size, string binFileName, table& table)

{

long long countComp{ 0 }, countEx{ 0 };

int\* arr{ new int[size] };

ReadBin(arr, size, binFileName);

auto startTime{ steady\_clock::now() };

InsertSort(arr, countComp, countEx, size);

auto endTime{ steady\_clock::now() };

auto time{ duration\_cast<microseconds>(endTime - startTime) };

int elapsedTime = static\_cast<int>(time.count());

table.zero << size << ";";

table.comparisons << countComp << ";";

table.exchanges << countEx << ";";

table.clock << elapsedTime << ";";

delete[] arr;

}

void fillStructTim(int size, string binFileName, table& table)

{

int countComp{ 0 }, countEx{ 0 };

int\* arr{ new int[size] };

ReadBin(arr, size, binFileName);

auto startTime{ steady\_clock::now() };

TimSort(arr, size,countComp, countEx);

auto endTime{ steady\_clock::now() };

auto time{ duration\_cast<microseconds>(endTime - startTime) };

int elapsedTime = static\_cast<int>(time.count());

table.zero << size << ";";

table.comparisons << countComp << ";";

table.exchanges << countEx << ";";

table.clock << elapsedTime << ";";

delete[] arr;

}

void fillFile(string filename, table& table)

{

ofstream file("./results/"+filename);

file << table.zero.str() << "\n";

file << table.comparisons.str() << "\n";

file << table.exchanges.str() << "\n";

file << table.clock.str() << "\n";

file.close();

}