МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра математического обеспечения и применения ЭВМ

ОТЧЁТ

по лабораторной работе №3

по дисциплине «Алгоритмы и структуры данных»

Тема: Поразрядная сортировка (LSD и MSD)

Студент гр. 8381 _____ Почаев Н.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

Цель работы.

Изучить алгоритм работы и способы реализации поразрядной / цифровой сортировки (англ. Radix Sort), оценить сложность данного алгоритма в созданной реализации. Разработать программу с графическим интерфейсом при помощи использования фреймворка Qt. Освоить работу с шаблоном ”наблюдатель”, представленного механизмом сигналов и слотов.

Основные теоретические положения.

Исходно алгоритм предназначен 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 (для этого зачастую используют MSD реализацию), и вообще произвольные значения в памяти, представленные в виде набора байт.

Сравнение производится поразрядно: сначала сравниваются значения одного крайнего разряда, и элементы группируются по результатам этого сравнения, затем сравниваются значения следующего разряда, соседнего, и элементы либо упорядочиваются по результатам сравнения значений этого разряда внутри образованных на предыдущем проходе групп, либо переупорядочиваются в целом, но сохраняя относительный порядок, достигнутый при предыдущей сортировке. Затем аналогично делается для следующего разряда, и так до конца.

Так как выравнивать сравниваемые записи относительно друг друга можно в разную сторону, на практике существуют два варианта этой сортировки. Для чисел они называются в терминах значимости разрядов числа, и получается так: можно выровнять записи чисел в сторону менее значащих цифр (по пра-

вой стороне, в сторону единиц, *least significant digit*, **LSD**) или более значащих цифр (по левой стороне, со стороны более значащих разрядов, *most significant digit*, **MSD**).

Корректность алгоритма LSD-сортировки.

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

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

Переход: Пусть для $n = k$ алгоритм правильно отсортировал последовательности по k младшим разрядам. Покажем, что в таком случае, при сортировке по $(k + 1)$-му разряду, последовательности также будут отсортированы в правильном порядке.

Вспомогательная сортировка разобьет все объекты на группы, в которых $(k + 1)$-й разряд объектов одинаковый. Рассмотрим такие группы. Для сортировки по отдельным разрядам мы используем устойчивую сортировку, следовательно порядок объектов с одинаковым $(k + 1)$-м разрядом не изменился. Но по предположению индукции по предыдущим k разрядам последовательности были отсортированы правильно, и поэтому в каждой такой группе они будут отсортированы верно. Также верно, что сами группы находятся в правильном относительно друг друга порядке, а, следовательно, и все объекты отсортированы правильно по $(k + 1)$-м младшим разрядам.

Постановка задачи.

Программа запускается из консоли: в зависимости от передачи флага `console` запускается графический или консольный интерфейс программы. Далее через

файл или через строку ввода (консоль) вводится набор чисел типа `int32_t`. Далее пользователю даётся возможность выбрать пошаговый режим выполнения программы, после чего происходит сортировка отмеченным ранее алгоритмом.

Выполнение работы.

Написание работы производилось на базе операционной системы Linux Manjaro в среде разработки Qt Creator с использование фреймворка Qt.

Для реализации графического интерфейса в стиле Material Design была использована сторонняя библиотека `laserpants/qt-material-widgets` по открытой лицензии с GitHub.

Для выполнения сортировки и сопутствующих функций был создан класс `radixSort`, для работы в консольном режиме - `Console`. Все графические элементы размещены в классе `mainWindow`.

Реализация функций LSD и MSD сортировок представлена в коде, размещённом в приложении А.

Оценка сложности работы алгоритма.

Сложность LSD-сортировки.

Пусть m - количество разрядов, n - количество объектов, которые нужно отсортировать, $T(n)$ - время работы устойчивой сортировки. В реализации, представленной в данной лабораторной работе, используются $m = 32$. Цифровая сортировка выполняет k итераций, на каждой из которой выполняется устойчивая сортировка и не более $O(1)$ других операций. Следовательно время работы цифровой сортировки - $O(kT(n))$.

Рассмотрим отдельно случай сортировки чисел. Пусть в качестве аргумента сортировке передается массив, в котором содержатся n m-значных чисел, и каждая цифра может принимать значения от 0 до $k - 1$. Тогда цифровая сортировка позволяет отсортировать данный массив за время $O(m(n+k))$, если

устойчивая сортировка имеет время работы $O(n+k)$. Если k небольшое, то оптимально выбирать в качестве устойчивой сортировки сортировку подсчетом. В текущей реализации из-за фиксированного подсчёта и оценки исключительно алгоритма Radix Sort данные оптимизации не используются. Для каждого набора входных данных LSD выполняет фиксированное m количество итераций побитового сравнения.

Если количество разрядов — константа, а $k = O(n)$, то сложность цифровой сортировки составляет $O(n)$, то есть она линейно зависит от количества сортируемых чисел.

Сложность MSD-сортировки.

Пусть значения разрядов меньше b, а количество разрядов — k. При сортировке массива из одинаковых элементов MSD-сортировкой на каждом шаге все элементы будут находится в неубывающей по размеру корзине, а так как цикл идет по всем элементам массива, то получим, что время работы MSD-сортировки оценивается величиной $O(nk)$, причем это время нельзя улучшить. Хорошим случаем для данной сортировки будет массив, при котором на каждом шаге каждая корзина будет делиться на b частей. Как только размер корзины станет равен 1, сортировка перестанет рекурсивно запускаться в этой корзине. Таким образом, асимптотика будет $\Omega(n \log_b n)$. Это хорошо тем, что не зависит от числа разрядов.

Финальная сложность реализованного алгоритма

Для реализации внутренней побитовой сортировки использовались функции STL библиотеки: `stable_partition()` для LSD и `partition()` для MSD соответственно. Оба алгоритма сортировки выполняются с гарантированной сложностью $O(n \log N)$ для `swap` операций и $O(n)$ для применения предиката, что удовлетворяет условию внутренних операций, оговоренному выше.

Выводы.

В ходе выполнения данной лабораторной работы была изучена, реализована и протестирована поразрядная сортировка. Был реализован графический интерфейс, позволяющий применять её для указанных наборов чисел. Математически была доказан сложность данного алгоритма, в том числе, применительно к выполненной реализации.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: radixSort.cpp

```
#include ”basicheaders.h”
#include ”radixSort.h”
#include ”console.h”

radixSort::radixSort(bool sortFlag, bool stepFlag, bool consoleMode)
{
    this->stepFlag = stepFlag;
    this->sortFlag = sortFlag;
    this->consoleMode = consoleMode;
    this->printResFlag = false;

    lsb = 0;
}

//  Least significant digit radix sort
void radixSort::lsd_radix_sort()
{
    QFile log(”../log.txt”);
    log.open(QIODevice::Append);
    QTextStream logStream(&log);

    if(lsb == ITERCOUNT) {sendDeactivateNextStepBut(); return;}

    if (stepFlag == true)
    {
        logStream << ”Step ” << lsb << ”: ” << endl;
        if(consoleMode == false) emit \
        strToPrintInWindow(”Step ” + QString::number(lsb) + ”: \n”);
        for (auto i : inpArr)
        {
            logStream << i << ’ ’;
            if(consoleMode == false) emit \
            strToPrintInWindow(QString::number(i) + ” ”);
        }
        logStream << endl;
        if(consoleMode == false) emit strToPrintInWindow(”\n”);
    }
    std::stable_partition(inpArr.begin(), inpArr.end(), radix_test(lsb));
```

```
    ++lsb;
    log.close();
}

// Most significant digit radix sort (recursive)
void radixSort::msd_radix_sort(std::vector<int32_t>::iterator first, \
                        std::vector<int32_t>::iterator last, int msb = 31)
{
    if (first != last && msb >= 0)
    {
        std::vector<int32_t>::iterator mid = std::partition(first, \
                                             last, radix_test(msb));
        msb--; // decrement most-significant-bit
        msd_radix_sort(first, mid, msb); // sort left partition
        msd_radix_sort(mid, last, msb); // sort right partition
    }
}

// write input array to radix sort class object
void radixSort::writeData(std::vector<int32_t> inpArr)
{
    this->inpArr.reserve(inpArr.size());
    std::copy(inpArr.begin(), inpArr.end(), \
    std::back_inserter(this->inpArr));
}

void radixSort::mainSortFunc()
{
    QFile log("../log.txt");
    log.open(QIODevice::Append);
    QTextStream logStream(&log);

    if (sortFlag == true)
    {
        if(stepFlag == true)
        {
            if (lsb == 1)
            {
                logStream << "LSD radix sort process:" << endl;
                if(consoleMode == false) \
                emit strToPrintInWindow("LSD radix sort process:\n");
            }
```

```
        } else
        {
            for(int i = 0; i < ITERCOUNT; ++i)
                lsd_radix_sort();
        }

        lsd_radix_sort();
    } else
    {
        msd_radix_sort(inpArr.begin(), inpArr.end());
    }

    if ((lsb == ITERCOUNT || sortFlag == false) \
    && printResFlag == false)
    {
        resPrint();
        printResFlag = true;
    }

    log.close();
}

void radixSort::implementForFile(QString fileName)
{
    QFile file(fileName);
    std::vector<int32_t> inpArr;

    // creating and naming log file
    QFile log("../log.txt");
    log.open(QFile::WriteOnly|QFile::Truncate);
    QTextStream logStream(&log);
    logStream << "***********\n";
    logStream << "* LOG FILE *\n";
    logStream << "***********\n";
    log.close();

    if (file.open(QIODevice::ReadOnly | QIODevice::Text)) {
        QTextStream stream(&file);
        foreach (QString i,\
        QString(stream.readAll()).split(\
        QRegExp("[\r\n]"),QString::SkipEmptyParts))
        {
```

```
            inpArr.push_back(static_cast<int32_t>(i.toInt()));
        }
    }
    file.close();

    writeData(inpArr);
    mainSortFunc();
}

void radixSort::implementForLine(std::vector<int32_t> inpArr)
{
    // creating and naming log file
    QFile log("../log.txt");
    log.open(QFile::WriteOnly|QFile::Truncate);
    QTextStream logStream(&log);
    logStream << "************\n";
    logStream << "* LOG FILE *\n";
    logStream << "************\n";
    log.close();

    writeData(inpArr);
    mainSortFunc();
}

void radixSort::resPrint()
{
    if(DEBUG) std::cout << "Result printing, lsb = " << lsb << std::endl;
    QFile log("../log.txt");
    log.open(QIODevice::Append);
    QTextStream logStream(&log);
    logStream << "Results of radix sort:" << endl;
    if(consoleMode == false) \
    emit strToPrintInWindow("Results of radix sort:\n");
    else std::cout << "Results of radix sort:" << std:: endl;
    for (auto i : inpArr)
    {
        logStream << i << endl;
        if(consoleMode == false) \
        emit strToPrintInWindow(QString::number(i) + "\n");
        else std::cout << i << std:: endl;
    }
}
```

```
void radixSort::callWorkSortFunc()
{
    mainSortFunc();
}

radixSort::~radixSort()
{
}
```