МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Алгоритмы сортировки

Студент гр. 9303 _____ Куршев Е.О.

Преподаватель _____ Филатов Ар.Ю.

Санкт-Петербург
2020

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Куршев Евгений Олегович

Группа 9303

Тема работы: продвинутые алгоритмы сортировки

Исходные данные:

Количество массивов

Размер этих массивов

Границы, в которых будут задаваться случайные числа

Содержание пояснительной записки:

Содержание

Введение

Разработка плана

Реализация алгоритма плавной сортировки

Анализ результатов

Заключение

Предполагаемый объем пояснительной записки:

Не менее 10 страниц.

Дата выдачи задания: 06.11.2020

Дата сдачи реферата: 25.12.2020

Дата защиты реферата: 25.12.2020

Студент гр. 9303 _____ Куршев Е.О.

Преподаватель _____ Филатов Ар.Ю.

АННОТАЦИЯ

Плавная сортировка. Сравнительное исследование с другим алгоритмом сортировки: алгоритм быстрой сортировки с процедурой трёхчастного деления. Курсовая работа представляет собой программу, предназначенную для сортировки массива методом плавной сортировки. Код программы написан на языке программирования C++. В написании кода программы использовались функции стандартных библиотек языка C++. Для проверки работоспособности программы проводилось тестирование. Код программы и результаты тестирования представлены в приложениях.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы - провести исследование алгоритма плавной сортировки, протестировать его на случайных входных данных, а также, сравнить с другим алгоритмом: быстрая сортировка с процедурой трёхчастного деления. В рамках работы поставлены следующие условия:

1. Анализ задачи, цели, составление технологии проведения и планв экспериментального исследования.
2. Генерация представительного множества реализаций входных данных .
3. Выполнение исследуемых алгоритмов на сгенерированных наборах данных.
4. Фиксация результатов испытаний алгоритма, накопление статистики.
5. Представление результатов испытаний, их интерпретацию и сопоставление с теоретическими оценками.

1. РАЗРАБОТКА ПЛАНА

1.1. Анализ задачи и целей

Была поставлена задача: реализовать алгоритм плавной сортировки и сравнить его с другим алгоритмом.

Вначале был изучена плавная сортировка: все её особенности, были отмечены отличия этой сортировки от сортировки бинарной кучей.

Исходя из этих условий была поставлена цель: найти сильные и слабые стороны этого алгоритма, а также найти плюсы и минусы по сравнению с другим алгоритмом.

1.2. Разработка технологии проведения

Исходя из задачи и поставленной цели был составлен следующий план проведения работы:

1. Поэтапная реализация алгоритма плавной сортировки:
   a) Создание из массива целых чисел дерева, построенного на числах Леонардо.
   b) "Разрушение" ранее созданного дерева и сортировка имеющегося массива.
2. Генерирование случайных входных данных.
3. Выполнение алгоритма на сгенерированных данных.
4. Интерпретация полученных результатов

2. РЕАЛИЗАЦИЯ АЛГОРИТМА ПЛАВНОЙ СОРТИРОВКИ

2.1. Теоретические сведения

Леонардовы кучи - кучи, построенные на числах леонардо. Данные кучи используются для проведения плавной сортировки.

Изначально существовала сортировка с помощью двоичных куч. Метод изобрёл Эдсгер Дейкстра. Сортировка кучей сама по себе очень хороша, так как её сложность по времени O(n log n) вне зависимости от данных. Чтобы из себя не представлял массив, сложность heapsort никогда не деградирует до O(n2), что может приключиться с быстрой сортировкой. Обратной стороной медали является то, что сортировку бинарной кучей нельзя и ускорить, сложности O(n) ожидать также не приходится (а вот та же быстрая сортировка, при определённых условиях может достигать таких показателей).

Тогда появился вопрос: можно ли сделать так, чтобы временная сложность сортировка кучей, с одной стороны, была не ниже чем O(n log n), но при благоприятном раскладе (в частности, если обрабатывается почти отсортированный массив) повышалась до O(n)? Эдсгер Дейкстра доказал, что это возможно.

Очевидном минусом сортировки бинарной кучей была сортировка почти упорядоченного массива.

Первое: при просейке максимумы постоянно выталкиваются в корень кучи, который соответствует первому элементу массива. Если массив на входе будет почти упорядочен, то для алгоритма это только немного добавит работы. Меньшие элементы всё равно сначала будут опускаться вниз по дереву, т.е. перемещаться ближе к концу массива, а не к его началу.

Второе: стандартная двоичная куча сама по себе всегда является сбалансированным деревом. В случае изначально упорядоченных данных это играет в минус. Если в первоначальном массиве рандомные данные — то они равномерно распределяются в сбалансированном дереве и многократная просейка проходится по всем веткам примерно одинаковое число раз. Для почти упо-

рядоченных данных, более желательно несбалансированное дерево — в этом случае данные на той части массива, которые соответствуют более длинным веткам дерева просейка будет обрабатывать реже, чем другие.

Чтобы решить обе проблемы, Дейкстра предложил использовать специальные двоичные кучи, построенные на числах Леонардо.

Ряд чисел Леонардо задаётся рекурретно:

L0 = 1

L1 = 1

Ln = Ln-1 + Ln-2 + 1

Первые 20 чисел Леонардо:

1, 1, 3, 5, 9, 15, 25, 41, 67, 109, 177, 287, 465, 753, 1219, 1973, 3193, 5167, 8361, 13529

Очевидно, что абсолютно любое целое число можно представить в виде суммы чисел Леонардо, имеющих разные порядковые номера. Но, массив из n элементов не всегда можно представить в виде одной кучи Леонардо (если n не является числом Леонардо). Но, зато, любой массив всегда можно разделить на несколько подмассивов, которые будут соответствовать разным числам Леонардо, т.е. представлять из себя кучи разного порядка.

Можно выделить три особенности леонардовых куч:

1. Каждая леонардова куча представляет собой несбалансированное бинарное дерево.

2. Корень каждой кучи — это последний (а не первый, как в обычной бинарной куче) элемент соответствующего подмассива.

3. Любой узел со всеми своими потомками также представляет из себя леонардову кучу меньшего порядка.

Общая идея алгоритма состоит в следующем:

1. Построить кучу, основанную на числах леонардо.

2. Для каждого поддерева проверять свойство леонардовой кучи: значение в любой вершине не меньше, чем значения её потомков

2.2. Реализация дерева

Для реализации дерева была написана следующая структура:

```
typedef struct Elem{
        int size;
        int x;
   struct Elem* left;
   struct Elem* rigth;
}Elem;
```

Данная структура хранит в себе число массива, количество элементов в этом дереве, включая самого себя, а также указатели на левое и правое поддерево.

Чтобы реализовать дерево был создан массив из структур типа Elem, который хранит самые верхние элементы каждого созданного дерева. При добавлении элемента происходит проверка: могут ли два крайних правых элемента образовать леонарову кучу. Если да, то эта куча создаётся, иначе в массив добавляется новый элемент. При каждом добавлении элемента происходит просеивание, чтобы в каждый момент времени сохранялись свойства леонардовых куч.

2.3 Разрушение дерева

После создания дерева уже начинается сортировка. В каждый момент времени сравниваются значения верхних элементов. Среди них находится максимум. Далее этот максимум меняется с последним элементом, максимум удаляется из массива, его дерево разрушается на два поддерева, а для элемента, с которым максимум менялся, происходит просеивание получившегося дерева. Так происходит до того момента, пока не закончатся элементы.

3. АНАЛИЗ РЕЗУЛЬТАТОВ

3.1. Получение результатов

Для каждого нового случая в файл заносятся сведения о проделанных операциях, а именно: количество чисел в массиве, среднее время сортировки алгоритмом быстрой сортировки, среднее время сортировки алгоритмом плавной сортировки, количество входных наборов.

3.2. Сравнение полученных данных с теоретическими

Теоретическая сложность для среднего случая обоих сортировок составляет O(nlogn).

Были проведены тестирования для 100 наборов при 50, 100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600 числах. Полученные данные предоставлены в таблице ниже.

Количество чисел в наборе	Среднее время быстрой сортировки, с	Среднее время плавной сортировки, с	Количество наборов
50	$5.73*10^{-6}$	$5.213*10^{-5}$	100
100	$1.224*10^{-5}$	$8.327*10^{-5}$	100
200	$2.93*10^{-5}$	0.00015317	100
400	$6.607*10^{-5}$	0.00028848	100
800	0.00015328	0.00086419	100
1600	0.00034381	0.00158008	100
3200	0.00078284	0.00308934	100
6400	0.00165632	0.00617762	100
12800	0.00363666	0.0133494	100
25600	0.00982597	0.0291817	100

Проверим выполнение теоретической сложности при n = 2, так как в каждом наборе количество данных увеличивалось в два раза.

nlogn при n = 2 ~= 2.1. Проанализировав полученные данные было замечено, что при увеличении данных в два раза время увеличивается в 2.2-2.5 раза в среднем для обоих алгоритмов. Эти данные показывают, что точность выполнения алгоритмов достаточно высокая. Также было проведено исследование для двух худших случаев: массив чисел от n до 1.

В данном эксперименте были взяты n = 800 и n = 1000. Полученные результаты приведены в таблице ниже.

Количество чисел в массиве	Время быстрой сортировки, с	Время плавной сортировки, с
1000	0.011365	0.000834
800	0.005595	0.000571

Теоретическая сложность для среднего случая плавной сортировки составляет O(nlogn), а для быстрой сортировки $O(n^2)$.

Исходя из полученных данных, количество исходных данных увеличилось в 1.25 раза.

Из полученных практических результатов видим, что время быстрой сортировки возросло более чем в 1.5625 раза, что говорит о том, что быстрая сортировка для полностью неупорядоченного массива является не самым быстрым алгоритмом, а плавная сортировка для худшего случая сохранила сложность O(nlogn).

3.3 Сравнение двух алгоритмов

Исходя из всех полученных данных можно выделить следующие плюсы и минусы: в средних случаях алгоритм быстрой сортировки работает быстрее алгоритма плавной сортировки. Это обусловлено тем, что в алгоритме плавной сортировки создаётся дерево, что является затратным процессом, в то время как алгоритм быстрой сортировки вызывается рекурсивно.

Но с худшим случаем ситуация другая: алгоритм быстрой сортировки не эффективен, так как на каждом шаге количество элементов в массиве уменьшается всего на 1, что обуславливает большое число сравнений. В тоже время, на алгоритм плавной сортировки, это влияет не сильно, так как сравнений будет меньше, которые будут происходить только при просеивании дерева.

ЗАКЛЮЧЕНИЕ

Были поставлены цели и задачи: реализовать алгоритм плавной сортировки, сравнить его с алгоритмом быстрой сортировки.

Исходя из этих целей было сделано:

1. Изучены теоретические сведения о плавной сортировке
2. Написан код, реализующий алгоритм быстрой сортировки
3. Проведено исследование на случайных данных для среднего случая, и на заданных для худшего.

Исходный код программы представлен в приложении А, результаты тестирования - в приложении Б.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#INCLUDE <IOSTREAM>
#include <fstream>
#include <set>
#include <ctime>

typedef struct Elem{
    int size;
    int x;
    struct Elem* left;
    struct Elem* rigth;
}Elem;

void my_qsort(int* arr, int start, int end, int& counter){
    if(end - start > 0){
        counter++;
        int num_base = end;
        int cur = start;
        int quanity_base = 1;
        int base = arr[num_base];
        int size = num_base - cur;
        for(int i = 0; i < size; i++){
            counter++;
            if (base > arr[cur]){
                cur += 1;
                counter++;
            }
            else if (base == arr[cur]){
                counter++;
                for(int j = 0; j < num_base - cur - 1; j++){
                    counter++;
                    int t = arr[cur + j];
                    arr[cur + j] = arr[cur + j + 1];
                    arr[cur + j + 1] = t;
```

```
                }
                num_base -= 1;
                counter++;
                quanity_base += 1;
                counter++;
            }
            else{
                int t = arr[cur];
                arr[cur] = arr[num_base - 1];
                arr[num_base - 1] = t;
                for(int k = 0; k < quanity_base; k++){
                    counter++;
                    t = arr[num_base - 1 + k];
                    arr[num_base - 1 + k] = arr[num_base + k];
                    arr[num_base + k] = t;
                }
                num_base -= 1;
                counter++;
            }
            counter++;
        }
        my_qsort(arr, start, num_base - 1, counter);
        my_qsort(arr, num_base + quanity_base, end, counter);
    }
}

template <typename T>
void swap(T& a, T& b){
    T tmp = a;
    a = b;
    b = tmp;
}

int leo(int x, int& counter){
    if(x == 1)
```

```
        return 1;
    else{
        if(x == 2)
            return 1;
        else{
            counter++;
            return leo(x - 2, counter) + leo(x - 1, counter) + 1;
        }
    }
}

void clearing(Elem* tree, int& counter){
    if(tree->left && tree->rigth){
        counter++;
            if(tree->x < tree->left->x && tree->left->x >= tree-
>rigth->x){
            counter++;
            swap(tree->x, tree->left->x);
            clearing(tree->left, counter);
        }
         else if(tree->x < tree->rigth->x && tree->left->x < tree-
>rigth->x){
            counter++;
            swap(tree->x, tree->rigth->x);
            clearing(tree->rigth, counter);
        }
    }
}

Elem* make_new(int x){
    Elem* new_elem = new Elem;
    new_elem->left = nullptr;
    new_elem->rigth = nullptr;
    new_elem->x = x;
    new_elem->size = 1;
```

```
    return new_elem;
}

Elem* union_elem(int x, Elem* left, Elem* rigth, int& counter){
    Elem* new_elem = new Elem;
    new_elem->left = left;
    new_elem->rigth = rigth;
    new_elem->x = x;
    new_elem->size = left->size + rigth->size + 1;
    counter++;
    return new_elem;
}

int make_and_sort(Elem** p, int* x, int n, int& num, std::set
<int>& s){
    int counter = 0;
    Elem* new_elem = make_new(x[0]);
    p[0] = new_elem;
    for(int i = 1; i < n; i++){
        counter++;
        if(num == 1){
            counter++;
            num += 1;
            counter++;
            p = (Elem**)realloc(p, num * sizeof(Elem*));
            p[1] = make_new(x[i]);
        }
        else{
             if(s.find(p[num - 1]->size + p[num - 2]->size + 1) !=
s.end()){
                counter++;
                  Elem* tmp = union_elem(x[i], p[num - 2], p[num -
1], counter);
                num -= 1;
                counter++;
```

```
            p[num] = nullptr;
            p = (Elem**)realloc(p, num * sizeof(Elem*));
            p[num - 1] = tmp;
            clearing(p[num - 1], counter);
        }
        else{
            num += 1;
            counter++;
            p = (Elem**)realloc(p, num * sizeof(Elem*));
            p[num - 1] = make_new(x[i]);
        }
    }
}

for(int i = 0; i < n; i++){
    int max = p[0]->x;
    int index = 0;
    if(n > 1 || p[0]->size != 1){
        int* tmp = new int[num];
        for(int j = 1; j < num; j++){
            counter++;
            tmp[j] = p[j]->x;
            if(tmp[j] > max){
                max = tmp[j];
                index = j;
            }
            counter++;
        }
        if(num != 1 && index != num - 1){
            swap(p[index]->x, p[num - 1]->x);
            clearing(p[index], counter);
        }
        counter += 2;
    }
    counter += 2;
```

```
        x[n - 1 - i] = max;

        if(p[num - 1]->size != 1){
            counter++;
            num += 1;
            counter++;
            p = (Elem**)realloc(p, num * sizeof(Elem*));
            p[num - 1] = p[num - 2]->rigth;
            p[num - 2] = p[num - 2]->left;
        }
        else{
            counter++;
            num -= 1;
            counter++;
            p = (Elem**)realloc(p, num * sizeof(Elem*));
        }
    }
    return counter;
}


void leo_chain(std::set<int>& s, int x, int& counter){
    int k = 1;
    int max = leo(1, counter);
    while(x >= max){
        counter++;
        s.insert(max);
        k += 1;
        counter++;
        max = leo(k, counter);
    }
}

void generate(int* arr, int* copy, int size, int a, int b){
    for(int i = 0; i < size; i++){
```

```
        arr[i] = rand() % (b - a + 1) + a;
        copy[i] = arr[i];
    }
}

int main() {
    srand(time(0));
    float t_q = 0, t_s = 0;
    int counter = 0;
    int num;
    std::cout << "Enter the number of sets: ";
    std::cin >> num;

    if (num < 1) {
        std::cout << "Error! Sets < 1!!\n";
        return 0;
    }

    int size;
    std::cout << "Enter the number of numbers in the set: ";
    std::cin >> size;

    if (size < 1) {
        std::cout << "Error! Size < 1!!\n";
        return 0;
    }
    int a, b;
    std::cout << "Enter the boundaries of random numbers: ";
    std::cin >> a >> b;

    if (b - a <= 0) {
        std::cout << "Error! Wrong boundaries!!\n";
        return 0;
    }
    std::cout << '\n';
```

```
    int* worst1 = new int[1000];
    int* worst1_copy = new int[1000];
    int* worst2 = new int[800];
    int* worst2_copy = new int[800];

    Elem** p = new Elem*[1];
    int n = 1;
    std::set <int> chain;
    clock_t time;

    std::fstream file;
     file.open("stats.txt", std::fstream::in | std::fstream::out |
std::fstream::app);

    for(int i = 1000; i > 0; i--){
        worst1[1000 - i - 1] = i;
        worst1_copy[1000 - i - 1] = i;
    }

    for(int i = 800; i > 0; i--){
        worst2[800 - i - 1] = i;
        worst2_copy[800 - i - 1] = i;
    }

    std::cout << "Generated array: ";
    for (int i = 0; i < 1000; i++)
        std::cout << worst1[i] << ' ';
    std::cout << '\n';

    time = clock();
    my_qsort(worst1, 0, 1000 - 1, counter);
    time = clock() - time;
     std::cout << "Sorting time with my_qsort: " << (float)time/
CLOCKS_PER_SEC << "s, base operation = " << counter << '\n';
```

```
    std::cout << "Array after my_qsort: ";
    for (int i = 0; i < 1000; i++)
        std::cout << worst1[i] << ' ';
    std::cout << '\n';

    counter = 0;
    time = clock();
    leo_chain(chain, 1000, counter);
    counter += make_and_sort(p, worst1_copy, 1000, n, chain);
    time = clock() - time;
     std::cout << "Sorting time with smoothsort: " << (float)time/
CLOCKS_PER_SEC << "s, base operation = " << counter << '\n';
    std::cout << "Array after smoothsort: ";
    for (int i = 0; i < 1000; i++)
        std::cout << worst1_copy[i] << ' ';
    std::cout << '\n';
    std::cout << '\n';

    std::cout << "Generated array: ";
    for (int i = 0; i < 800; i++)
        std::cout << worst2[i] << ' ';
    std::cout << '\n';

    n = 1;
    counter = 0;
    time = clock();
    my_qsort(worst2, 0, 800 - 1, counter);
    time = clock() - time;
     std::cout << "Sorting time with my_qsort: " << (float)time/
CLOCKS_PER_SEC << "s, base operation = " << counter << '\n';
    std::cout << "Array after my_qsort: ";
    for (int i = 0; i < 800; i++)
        std::cout << worst2[i] << ' ';
    std::cout << '\n';
```

```
    counter = 0;
    time = clock();
    leo_chain(chain, 800, counter);
    counter += make_and_sort(p, worst2_copy, 800, n, chain);
    time = clock() - time;
     std::cout << "Sorting time with smoothsort: " << (float)time/
CLOCKS_PER_SEC << "s, base operation = " << counter << '\n';
    std::cout << "Array after smoothsort: ";
    for (int i = 0; i < 800; i++)
        std::cout << worst2_copy[i] << ' ';
    std::cout << '\n';
    std::cout << '\n';

    for(int i = 0; i < num; i++){
        n = 1;
        counter = 0;
        int *arr = new int[size];
        int *copy = new int[size];
        generate(arr, copy, size, a, b);
        std::cout << "Generated array: ";
        for (int j = 0; j < size; j++)
            std::cout << arr[j] << ' ';
        std::cout << '\n';

        time = clock();
        my_qsort(arr, 0, size - 1, counter);
        time = clock() - time;
        t_q += (float)time/CLOCKS_PER_SEC;
               std::cout << "Sorting time with my_qsort: " <<
(float)time/CLOCKS_PER_SEC << "s, base operation = " << counter <<
'\n';
        std::cout << "Array after my_qsort: ";
        for (int j = 0; j < size; j++)
            std::cout << arr[j] << ' ';
        std::cout << '\n';
```

```
        counter = 0;
        time = clock();
        leo_chain(chain, b, counter);
        counter += make_and_sort(p, copy, size, n, chain);
        time = clock() - time;
        t_s += (float)time/CLOCKS_PER_SEC;
              std::cout << "Sorting time with smoothsort: " <<
(float)time/CLOCKS_PER_SEC << "s, base operation = " << counter <<
'\n';
        std::cout << "Array after smoothsort: ";
        for (int j = 0; j < size; j++)
            std::cout << copy[j] << ' ';
        std::cout << '\n';
        std::cout << '\n';
        delete[] arr;
        delete[] copy;
    }
    file << "New data !!\n";
    file << "size\ttime qsort\ttime smooth sorting\t\tsets\n";
      file << size << "\t\t" << t_q / (float)(num) << "\t\t" <<
t_s / (float)(num) << "\t\t\t" << num << '\n';
    file.close();
    return 0;
}
```

ПРИЛОЖЕНИЕ Б

ТЕСТИРОВАНИЕ ПРОГРАММЫ

NEW DATA !!

size	time qsort	time smooth sorting	sets
50	5.73e-06	5.213e-05	100

New data !!

size	time qsort	time smooth sorting	sets
100	1.224e-05	8.327e-05	100

New data !!

size	time qsort	time smooth sorting	sets
200	2.93e-05	0.00015317	100

New data !!

size	time qsort	time smooth sorting	sets
400	6.607e-05	0.00028848	100

New data !!

size	time qsort	time smooth sorting	sets
800	0.00015328	0.00086419	100

New data !!

size	time qsort	time smooth sorting	sets
1600	0.00034381	0.00158008	100

New data !!

size	time qsort	time smooth sorting	sets
3200	0.00078284	0.00308934	100

New data !!

size	time qsort	time smooth sorting	sets
6400	0.00165632	0.00617762	100

New data !!

size	time qsort	time smooth sorting	sets
12800	0.00363666	0.0133494	100

New data !!

size	time qsort	time smooth sorting	sets
25600	0.00982597	0.0291817	100