Отчет по лабораторной работе №1

Алгоритмы сортировки.

Введение.

1. Counting sort. сложность: O(n^2)

Counting sort - это алгоритм сортировки, который сортирует элементы входной последовательности, основываясь на количестве элементов, которые меньше или равны данному элементу. Он использует два массива: массив входных данных и массив, называемый массивом подсчета. В отличие от других алгоритмов сортировки, которые сравнивают элементы входной последовательности между собой, counting sort считает количество элементов, которые меньше или равны текущему элементу, и использует эту информацию для размещения элемента на правильной позиции в отсортированной последовательности.

Counting sort хорошо подходит для сортировки целых чисел, когда известны границы входного диапазона. Например, если входная последовательность состоит из элементов, которые находятся в диапазоне от 0 до 999, counting sort может быть использован для сортировки этих элементов за линейное время.

Общая идея алгоритма counting sort заключается в следующих шагах:

1)Создать массив подсчета, который будет содержать количество элементов, которые меньше или равны каждому элементу входной последовательности.

2)Просканировать входную последовательность и инкрементировать значение в массиве подсчета для каждого элемента входной последовательности.

3)Пройти по массиву подсчета и вычислить, сколько элементов меньше или равны текущему элементу.

4)Создать выходной массив той же длины, что и входная последовательность.

5)Пройти по входной последовательности и поместить каждый элемент на правильную позицию в выходном массиве, используя информацию из массива подсчета.

Как и другие алгоритмы сортировки, counting sort имеет свои преимущества и недостатки. Преимуществами являются линейное время сортировки и стабильность (два элемента с равными значениями будут отсортированы в том же порядке, в котором они встречались во входной последовательности). Недостатком является требование наличия большого массива подсчета, который может занимать много памяти в случае большого диапазона входных данных.

2.Insertion sort. сложность: O(n^2)

Insertion sort - это алгоритм сортировки, который сортирует элементы входной последовательности один за другим, вставляя каждый элемент на правильную позицию в уже отсортированную часть последовательности. Он использует два массива: массив входных данных и выходной массив, который будет содержать отсортированную последовательность.

В отличие от алгоритмов сортировки, которые работают за время O(n*log n), insertion sort работает за время O(n^2). Однако, в случае, когда входная последовательность уже отсортирована или имеет почти отсортированный порядок, insertion sort может работать быстрее других алгоритмов.

Общая идея алгоритма insertion sort заключается в следующих шагах:

1)Начать со второго элемента входной последовательности.

2)Сравнить текущий элемент с элементами, находящимися перед ним, до тех пор, пока не будет найден элемент, который меньше текущего или до тех пор, пока не будет достигнут начало последовательности.

3)Вставить текущий элемент на правильную позицию в уже отсортированной части последовательности.

4)Повторить шаги 2-3 для всех элементов входной последовательности.

Insertion sort хорошо подходит для небольших последовательностей или для случаев, когда входная последовательность уже почти отсортирована. В случае больших последовательностей, время работы insertion sort может быть слишком высоким, поэтому в этом случае лучше использовать другие алгоритмы сортировки.

3.Quick sort.

Quick sort - это алгоритм сортировки, основанный на принципе "разделяй и властвуй". Он использует рекурсивный подход, чтобы разбить входную последовательность на меньшие подпоследовательности, затем сортирует каждую из них отдельно.

Идея алгоритма заключается в следующих шагах:

1)Выбрать элемент из последовательности, который называется опорным элементом (pivot). Этот элемент будет использоваться для разбиения последовательности на две части.

2)Разбить последовательность на две части: одна часть будет содержать элементы, которые меньше или равны опорному элементу, а другая часть будет содержать элементы, которые больше опорного элемента.

3)Рекурсивно применить алгоритм quick sort к обеим подпоследовательностям, пока не будет достигнута последовательность из одного элемента или пустая последовательность.

При правильном выборе опорного элемента, время работы алгоритма может достигать O(n*log n) в среднем и лучшем случае. Однако, в худшем случае, когда опорный элемент выбирается неудачно (например, когда он является минимальным или максимальным элементом в последовательности), время работы может достигать O(n^2).

Quick sort является одним из самых быстрых алгоритмов сортировки для больших наборов данных, поэтому он широко используется в практических приложениях.

Реализованные коды.

1.Counting sort.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdint.h>

#define MAX 1000000
#define RANGE 1000001

void countingSort(uint32_t arr[], uint32_t sorted[], int n) {
uint32_t *count = calloc(RANGE, sizeof(uint32_t)); // выделяем память для
массива для подсчета частоты вхождения каждого элемента в массиве
int i;

for (i = 0; i < n; i++) {
count[arr[i]]++; // увеличиваем значение соответствующего элемента в
массиве частот
}

for (i = 1; i < RANGE; i++) {
count[i] += count[i - 1]; // суммируем частоты до текущего элемента
}

int *used = calloc(RANGE, sizeof(int)); // массив, указывающий, было ли
число уже учтено при подсчете частот

for (i = n - 1; i >= 0; i--) {
if (!used[arr[i]]) {
sorted[count[arr[i]] - 1] = arr[i]; // помещаем каждый элемент на свою
позицию в отсортированном массиве
used[arr[i]] = 1; // помечаем число, как учтенное
}
count[arr[i]]--; // уменьшаем значение соответствующего элемента в
массиве частот
}
free(count); // освобождаем память, выделенную для массива count
free(used); // освобождаем память, выделенную для массива used
}

int main() {
uint32_t *arr = malloc(MAX * sizeof(uint32_t));
uint32_t *sorted = malloc(MAX * sizeof(uint32_t));
clock_t start, end;

srand(time(NULL));
for (int i = 0; i < MAX; i++) {
arr[i] = rand() % RANGE; // заполняем массив псевдослучайными числами из
интервала [0,100000]
}
```

```
start = clock();
countingSort(arr, sorted, MAX);
end = clock();

printf("Sorted array:\n");

printf("\n\nExecution time: %f seconds", (double)(end - start) /
CLOCKS_PER_SEC);

free(arr);
return 0;
}
```

2.Insertion sort.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdint.h>

#define MAX 100000

void insertionSort(uint32_t arr[], int n) {
    int i, j;
    uint32_t key;

    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

 // перемещаем элементы, которые больше ключевого,на одну позицию вперед
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = key; // помещаем ключевой элемент на свою позицию в
отсортированной последовательности
    }
}

int main() {
    uint32_t arr[MAX];
    clock_t start, end;

    srand(time(NULL));
    for (int i = 0; i < MAX; i++) {
        arr[i] = rand() % 100001; // заполняем массив псевдослучайными
числами из интервала [0,100000]
    }

    start = clock();
    insertionSort(arr, MAX);
    end = clock();
```

```
    printf("Sorted array:\n");


    printf("\n\nExecution time: %f seconds", (double)(end - start) /
CLOCKS_PER_SEC);
    return 0;
}
```

3.Quick sort.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdint.h>

#define MAX 100000

void quickSort(uint32_t arr[], int left, int right) {
    int i = left, j = right;
    uint32_t pivot = arr[(left + right) / 2];

    while (i <= j) {
        while (arr[i] < pivot) i++;
        while (arr[j] > pivot) j--;
        if (i <= j) {
            uint32_t temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            i++;
            j--;
        }
    }

    if (left < j) quickSort(arr, left, j);
    if (i < right) quickSort(arr, i, right);
}

int main() {
    uint32_t arr[MAX];
    clock_t start, end;

    srand(time(NULL));
    for (int i = 0; i < MAX; i++) {
        arr[i] = rand() % 100001; // заполняем массив псевдослучайными
числами из интервала [0,100000]
    }

    start = clock();
    quickSort(arr, 0, MAX - 1);
    end = clock();

    printf("Sorted array:\n");
```

```
    printf("\n\nExecution time: %f seconds", (double)(end - start) /
CLOCKS_PER_SEC);
    return 0;
}
```

Результаты экспериментов для counting sort.

#	Количество элементов в массиве	Время выполнения
1	50 000	0.001000 seconds
2	100 000	0.002000 seconds
3	150 000	0.002000 seconds
20	1 000 000	0.009000 seconds

Результаты экспериментов для Insertion sort.

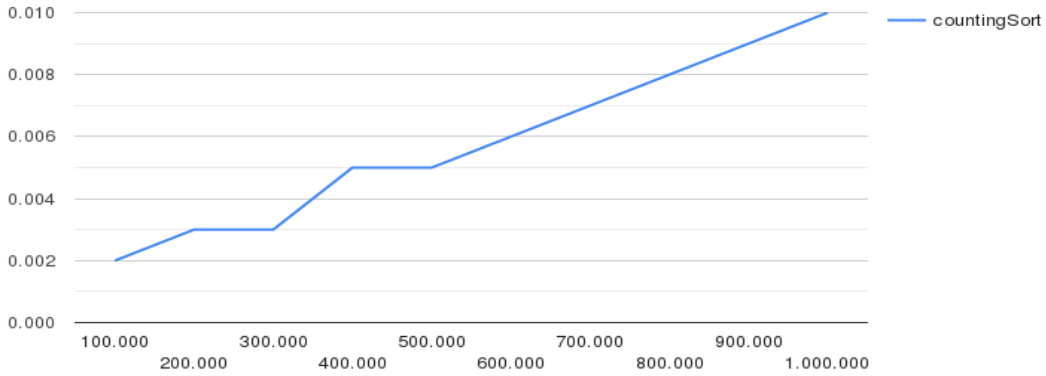
#	Количество элементов в массиве	Время выполнения
1	50 000	5.156250 seconds
2	100 000	8.171000 seconds
3	150 000	14.610000 seconds
20	1 000 000	521.406250 seconds

Результаты экспериментов для quick sort.

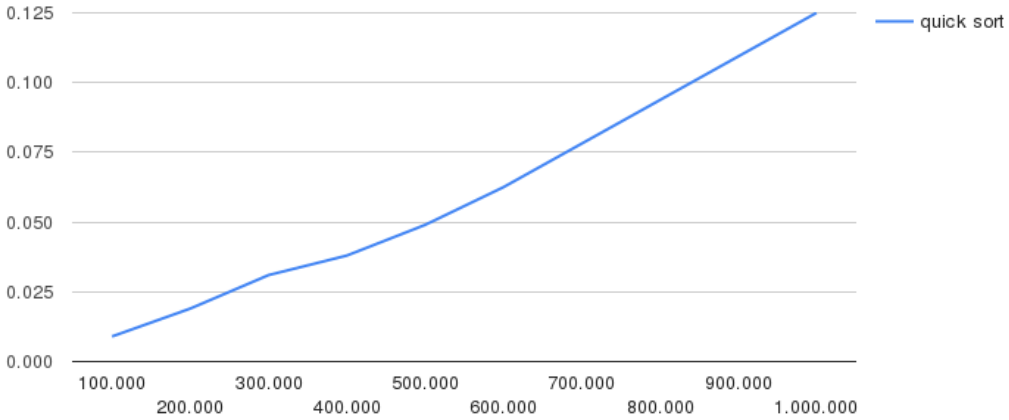
#	Количество элементов в массиве	Время выполнения
1	50 000	0.005000 seconds
2	100 000	
3	150 000	
20	1 000 000	

Графики зависимости времени времени выполнения алгоритмов.

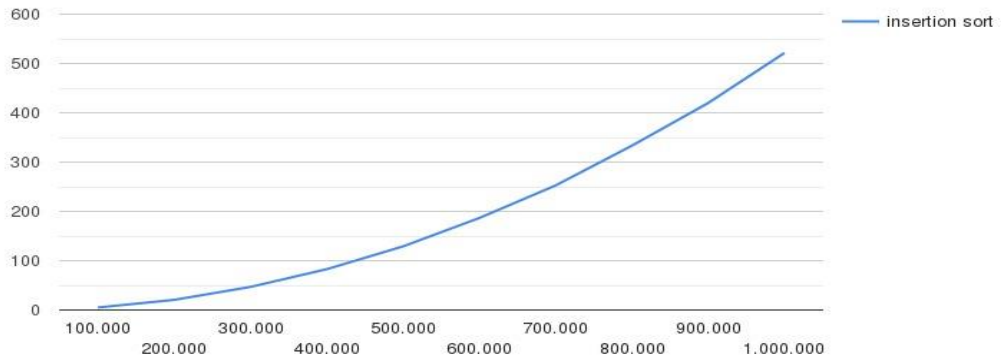
1.Counting sort.



2.Quick sort.

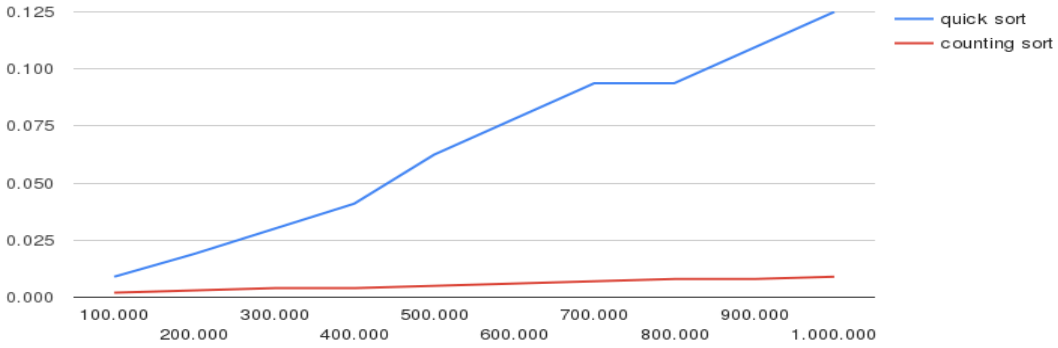


3. Insertion sort.



Сравнение сортировок Quick и Counting.

Мы сравниваем именно эти сортировки, потому что у них самые минимальные и похожие значения. Сортировка Insertion слишком затратна по времени и ресурсам компьютера.



Контрольные вопросы.

Вычислительная сложность алгоритма - это мера количества ресурсов (времени и/или памяти), необходимых для выполнения алгоритма в зависимости от размера входных данных.

f(n) = O(g(n)) означает, что f(n) растет не быстрее, чем g(n). f(n) = Θ(g(n)) означает, что f(n) растет так же быстро, как и g(n). f(n) = Ω(g(n)) означает, что f(n) растет не медленнее, чем g(n).

Устойчивый алгоритм сортировки - это алгоритм, который сохраняет порядок равных элементов при сортировке.

Сортировка "на месте" - это алгоритм сортировки, который сортирует элементы массива, используя только конечное количество дополнительной памяти, пропорциональное размеру входных данных.

Некоторые алгоритмы сортировки с вычислительной сложностью O(nlogn) для худшего случая включают быструю сортировку, сортировку слиянием и сортировку кучей.

Для худшего случая нет алгоритмов сортировки, работающих быстрее, чем O(nlogn). Однако, в среднем случае и на практике, некоторые алгоритмы сортировки могут работать быстрее O(nlogn), например, поразрядная сортировка.