Отчет по выполнению 1 Лабораторной работы

Алгоритмы и 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. Сортировка** пузырьком (**Bubble** **Sort**) — это один из наиболее известных алгоритмов, суть которого состоит в последовательном сравнении двух соседних элементов. В том случае, если предыдущий элемент больше последующего, они меняются местами. Сложность **в** **худшем случае** составляет - **O(n2).**

Преимущества:

1.Не требует доп. Массивов.

2.Самый простой в реализации.

Недостатки:

1.Время выполнения пропорциональна квадрату числа элементов.

2.Эффективен только для небольших массивов.

**3.** Пирамидальная сортировка (или сортировка кучей, HeapSort) — это метод сортировки сравнением, основанный на такой структуре данных как двоичная куча. Она похожа на сортировку выбором, где мы сначала ищем максимальный элемент и помещаем его в конец. Далее мы повторяем ту же операцию для оставшихся элементов. Что такое двоичная куча?Двоичная куча — это законченное двоичное дерево, в котором элементы хранятся в особом порядке: значение в родительском узле больше (или меньше) значений в его двух дочерних узлах. Первый вариант называется max-heap, а второй — min-heap. Куча может быть представлена двоичным деревом или массивом.

Алгоритм пирамидальной сортировки в порядке по возрастанию:

1.Постройте max-heap из входных данных.

2.На данном этапе самый большой элемент хранится в корне кучи. Замените его на последний элемент кучи, а затем уменьшите ее размер на 1. Наконец, преобразуйте полученное дерево в max-heap с новым корнем.

3.Повторяйте вышеуказанные шаги, пока размер кучи больше 1.

Сложность пирамидальной сортировки — O(nLogn).

Преимущества:

1. Имеет доказанную оценку худшего случая O(n⋅log⁡n).

2.Сортирует на месте, то есть требует всего O(1) дополнительной памяти (если дерево организовывать так, как показано выше).

Недостатки:

1. Неустойчив — для обеспечения устойчивости нужно расширять ключ.

2.На почти отсортированных массивах работает столь же долго, как и на хаотических данных.

3.На одном шаге выборку приходится делать хаотично по всей длине массива — поэтому алгоритм плохо сочетается с кэшированием и подкачкой памяти.

4.Методу требуется доступ к произвольному элементу структуры; не работает на связанных списках и других структурах памяти последовательного доступа.

**Реализованные коды.**

1.Counting sort.

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <stdint.h>

#define M 1000000

#define R 1000001

void countSort(uint32\_t arr[], uint32\_t sorted[], int n)

{

uint32\_t \*count = calloc(R, sizeof(uint32\_t));

int i;

for (i = 0; i < n; i++)

{

count[arr[i]]++;

}

for (i = 1; i < R; i++)

{

count[i] += count[i - 1];

}

int \*used = calloc(R, sizeof(int));

for (i = n - 1; i >= 0; i--)

{

if (!used[arr[i]])

{

sorted[count[arr[i]] - 1] = arr[i];

used[arr[i]] = 1;

}

count[arr[i]]--;

}

free(count);

free(used);

}

int main()

{

uint32\_t \*arr = malloc(M \* sizeof(uint32\_t));

uint32\_t \*sorted = malloc(M \* sizeof(uint32\_t));

clock\_t start, end;

srand(time(NULL));

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

{

arr[i] = rand() % R;

}

start = clock();

countSort(arr, sorted, M);

end = clock();

printf("Отсортированный массив элементов:\n");

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

{

printf("%d ", sorted[i]);

}

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

free(arr);

return 0;

}

2.Bubble Sort

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <time.h>

#include <stdint.h>

#include <stdbool.h>

#define R 1000001

#define M 1000000

void BubbleSort( uint32\_t mass[] )

{

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

{

mass[i] = rand() % R;

}

mass = calloc(R, sizeof(uint32\_t));

uint32\_t tmp;

bool noSwap;

for (int i = R - 1; i >= 0; i--)

{

noSwap = 1;

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

{

if (mass[j] > mass[j + 1])

{

tmp = mass[j];

mass[j] = mass[j + 1];

mass[j + 1] = tmp;

noSwap = 0;

}

}

if (noSwap == 1)

break;

}

}

int main()

{

clock\_t start, end;

uint32\_t \*mass= malloc(M \* sizeof(uint32\_t));

start = clock();

BubbleSort( mass );

end = clock();

printf("Отсортированный массив элементов:\n");

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

{

printf("%d ", mass[i]);

}

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

free(mass);

return 0;

}

3.Heap Sort

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define left\_child(node) ( (node) \* 2 + 1 )

#define right\_child(node) ( (node) \* 2 + 2 )

#define M 1000000

#define R 1000001

void swap(int \* array, int i, int j) {

int tmp = array[i];

array[i] = array[j];

array[j] = tmp;

}

void heap\_it(int \* array, int length, int root)

{

int leftChild = left\_child(root);

int rightChild = right\_child(root);

int biggest = root;

if ( leftChild < length && array[root] < array[leftChild] )

biggest = leftChild;

if ( rightChild < length && array[biggest] < array[rightChild] )

biggest = rightChild;

if ( biggest != root )

{

swap(array, biggest, root);

heap\_it(array, length, biggest);

}

}

void make\_heap(int \* array, int length)

{

int i = length / 2;

for ( ; i >= 0; --i )

heap\_it(array, length, i);

}

void heap\_sort(int \* array, int count)

{

int last;

make\_heap(array, count);

for ( last = count - 1; last > 0; --last )

{

swap(array, 0, last);

heap\_it(array, last, 0);

}

}

int main(void)

{

clock\_t start, end;

int array[M], i;

for ( i = 0; i < M; ++i )

array[i] = rand() % R;

printf("Unsorted:\n");

for ( i = 0; i < M; ++i )

printf("%02d ", array[i]);

start = clock();

heap\_sort(array, M);

end = clock();

printf("\nSorted:\n");

for ( i = 0; i < M; ++i )

printf("%02d ", array[i]);

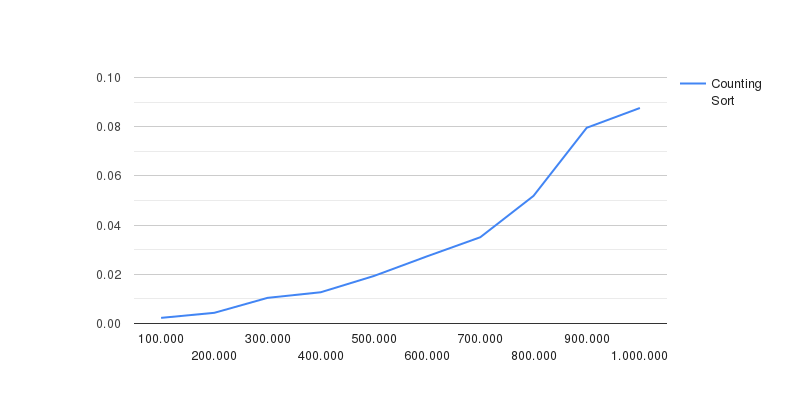
printf("\n\nExecution time: %f seconds", (double)(end - start) / CLOCKS\_PER\_SEC);

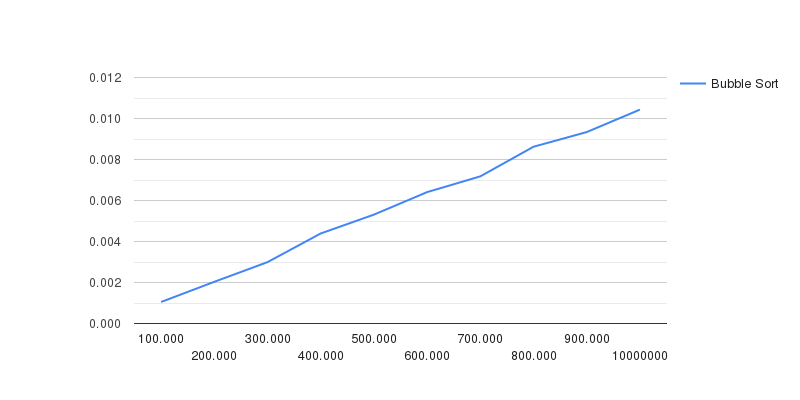
printf("\n");

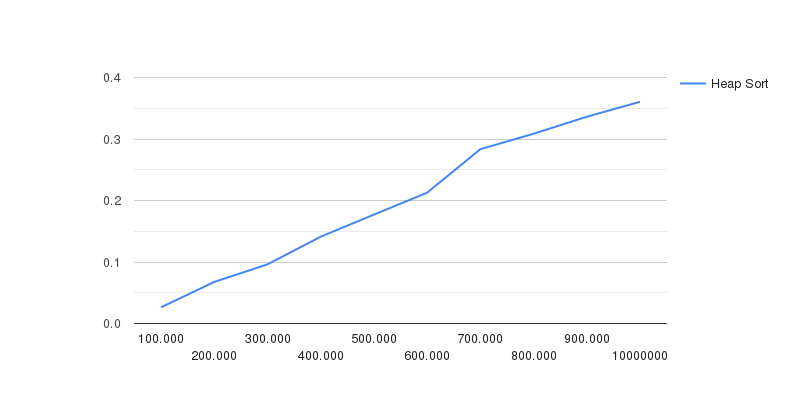
return 0;

}

**Графики зависимости времени выполнения алгоритмов от количества элементов.**

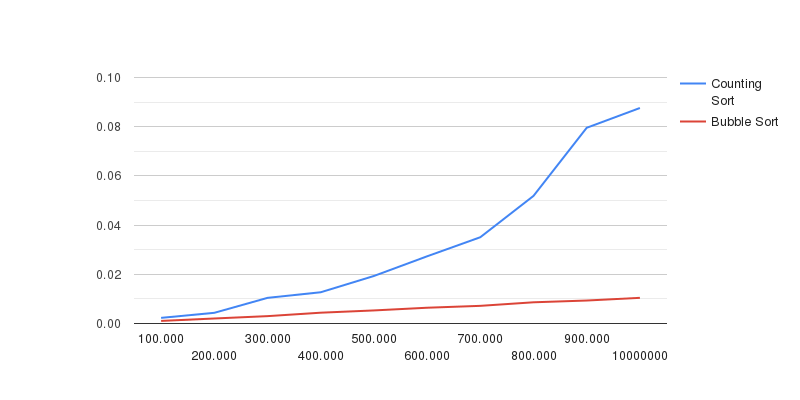






**Сравнение Counting и Bubble сортировок.**

Мы будем сравнивать Counting sort и Bubble Sort, так как их значения больше схожи друг с другом.



**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.