Балтийский государственный технический университет  
«ВОЕНМЕХ» им. Д. Ф. Устинова

Кафедра О7 «Информационные системы и программная инженерия»

**Практическая работа №3**по дисциплине «Структуры и организация данных»  
на тему «Оценка эффективности алгоритмов»  
часть 1 «Алгоритмы сортировки»  
  
вариант 12

Выполнил:  
Студент Костров Г. Ю  
Группа О712Б  
  
Преподаватель:  
Палехова О.А.

Санкт-Петербург  
2022 г.

Уровень сложности – повышенный. Провести сравнение указанных алгоритмов сортировки массивов, содержащих *N1*, *N2*, *N3* и *N4* элементов, по указанному в вариативной части критерию и объему требуемой дополнительно памяти.

Дополнительно провести анализ того, как наличие повторяющихся ключей во входной последовательности влияет на трудоемкость каждого из рассматриваемых алгоритмов сортировки.

Вариант № 12

Порядок: по возрастанию элементов. Методы: простых вставок, Алгоритм Шелла (шаг сортировки hk-1=2hk+1, ht=1, t=log2n-1), Timsort, сортировка естественным слиянием. N1=10000, N2=30000, N3=70000, N4=100000. Критерий – количество присваиваний.

1. Алгоритм сортировки методом простых вставок:

Для каждого элемента массива производится поиск подходящего места в уже упорядоченной части.

Трудоемкость сортировки методом простых вставокпо количеству присваиваний:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Случай** | **Ситуация, соответст-вующая случаю** | **Обоснование** | **Ожидаемое число присваиваний элементов массива** | **Асимптотическая оценка сложности по количеству присваиваний** | **Ожидаемое число вспомогат. присваиваний** |
| наилучший | массив упорядочен | перемещение элементов не требуется на каждый элемент приходится 2 присваивания – вспомогательной переменной и обратно | 2(n-1)  N1: 19998  N2: 59998  N3: 139998  N4: 199998 | Ω(n) | 2n – 1  *N1:* 19999  N2: 59999  N3: 139999  N4: 199999 |
| наихудший | массив упорядоченного в обратном порядке | требуется переместить все элементы, для вставки каждого элемента нужно сдвинуть все предшествующие | (n2 + 3n)/2 - 2  N1: 50014998  N2: 450044998  N3: 2450104998  N4: 5000149998 | O(n2) | (n2 + 3n)/2 - 1  *N1:*  50014999  *N2:* 450044999  *N3:* 2450104999  N4: 5000149999 |
| средний | Неупорядо-ченный массив | требуется переместить все элементы, для вставки каждого элемента нужно сдвинуть половину предшествующих | (n2 + 7n)/4  N1: 25017500  N2: 225052500  N3: 1225122500  N4: 2500175000 | O(n2) | (n2 + 7n)/4 + 1  *N1:* 2501750*1*  *N2:* 22505250*1*  *N3:* 122512250*1*  *N4:* 250017500*1* |

наихуд

Пространственная сложность – *O(1) (две переменных цикла и вспомогательная переменная)*.

Сортировка простыми вставками является устойчивой.

При повторениях ключей количество присваиваний должно уменьшаться, так как элементы с одинаковыми значениями менять местами не надо. Следовательно, с увеличением числа повторов ключей сортировка массива, упорядоченного в обратном направлении, должна стать не более трудоемкой, чем сортировка неупорядоченного массива.

1. Алгоритм Шелла (шаг сортировки hk-1=2hk+1, ht=1, t=log2n - 1):

Использование сортировки вставками для элементов, находящихся на некотором расстоянии друг от друга.

Трудоемкость алгоритма Шелла по количеству сравнений:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Случай** | **Ситуация, соответствующая случаю** | **Обоснование** | **Ожидаемое число присваиваний элементов массива** | **Асимптотическая оценка сложности по количеству присваиваний** | **Ожидаемое число вспомогат. присваиваний** |
| наилучший | Массив упорядочен | Алгоритм проходит те же действия, что и в сортировке простыми вставками, но с заданным по условию шагом | (log2n-1)\*2(n1)  N1: 245729  N2: 832332  N3: 2113280  N4: 3121900 | Ω(n) | (log2n-1)\*((2n1)+1)  N1: 245754  N2: 832360  N3: 2113331 N4: 3121930 |
| наихудший | Неупорядо-ченный массив | Количество перестановок зависит от того, насколько упорядочена последовательность | (log2n-1)\*5(n-1)  N1: 614324  N2: 2080830  N3: 5283200  N4: 7804740 | O(n2) | (log2n-1)\*(5(n1)+1)  N1: 614336  N2: 2080850 N3: 5283210 N4: 7804760 |
| средний | массив упорядочен в обратном порядке | Требуется переместить все элементы, для вставки каждого элемента нужно сдвигать предшествующие с заданным по условию шагом | (log2n1)\*3\*(n-1)  N1: 368594  N2: 1248500  N3: 3169920  N4: 4682850 | O(n\*log(n)) | (log2n1)\*(3\*(n-1)+1)  N1: 368606  N2: 1248510 N3: 3169930 N4: 4682860 |

Пространственная сложность – O(n), 3 переменных цикла и 3 вспомогательных.

Сортировка Шелла является неустойчивой.

Повторяющиеся значения ключей уменьшают количество присваиваний.

1. Алгоритм сортировки Tim sort:

Разделить списки на одинаковые части и слить обратно в нужном порядке

Трудоемкость сортировки прямым слиянием по количеству сравнений:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Случай** | **Ситуация, соответст-вующая случаю** | **Обоснование** | **Ожидаемое число присваиваний элементов массива** | **Асимптотическая оценка сложности по количеству присваиваний** | **Ожидаемое число вспомогат. присваиваний** |
| наилучший | Массив упорядочен | Одна серия | 0  N1: 0  N2:0  N3:0  N4:0 | Ω (n) | n  *N1: 10000*  *N2: 30000*  *N3: 70000*  *N4: 100000* |
| средний | массив упорядочен в обратном порядке | Отсутствие отсортированных участков | 1,5n  *N1: 15000*  *N2: 45000*  *N3: 105000*  *N4: 150000* | O(n\*log(n)) | 2n  *N1: 20000*  *N2: 60000*  *N3: 140000*  *N4: 200000* |
| наихудший | Неупорядо-ченный массив | Небольшое количество отсортированных участков | 2n\*log(n)  *N1: 265754*  *N2:892360*  *N3:2253310*  *N4:3321930* | O(n\*log(n)) | 2n\*log(n)  *N1: 265754*  *N2:892360*  *N3:2253310*  *N4:3321930* |

Пространственная сложность – О(n)

Tim sort является устойчивой сортировкой.

Повторяющиеся значения ключей уменьшают количество присваиваний, так как становится больше отсортированных участков.

1. Алгоритм сортировки естественным слиянием:

Разделить списки на упорядоченные фрагменты максимальной длины и слить обратно в нужном порядке

Трудоемкость сортировки естественным слиянием по количеству сравнений:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Случай** | **Ситуация, соответст-вующая случаю** | **Обоснование** | **Ожидаемое число присваиваний элементов массива** | **Асимптотическая оценка сложности по количеству присваиваний** | **Ожидаемое число вспомогат. присваиваний** |
| наилучший | массив упорядочен | Одна серия | 0  N1: 0  N2: 0  N3: 0  N4: 0 | Ω(n) | n + 10  N1: 10010  N2: 30010  N3: 70010  N4: 100010 |
| наихудший | массив  обратно упорядочен | Отсутствие отсортированных участков | log2 n \* n  N1: 132880  N2: 446180  N3: 1126650  N4: 1660960 | O(n\*log(n)) | log2 n \* 3n  N1: 398640  N2: 1338540  N3: 3379950  N4: 4982880 |
| средний | случайное расположение элементов в массиве | Небольшое количество отсортированных участков | (log2 n -1) \* n  N1: 122880  N2: 416180  N3: 1056650  N4: 1560960 | O(n\*log(n)) | (log2 n -1) \* 3n  N1: 368640  N2: 1248540  N3: 3169950  N4: 4682880 |

Пространственная сложность – О(n)

Сортировка естественным слиянием является неустойчивой.

Повторяющиеся значения ключей уменьшают количество присваиваний, так как становится больше отсортированных участков.

Текст программы:

#include <iostream>

#include <random>

#include <fstream>

#include <math.h>

#include <time.h>

#include <vector>

#include <algorithm>

#define N 100000

using namespace std;

long long int counter =0, counterHelp = 0;

// Количество памяти

int memory\_StraightInsertion=0,

memory\_ShellSort=0,

memory\_tim\_sort=0,

memory\_NaturalMergeSort=0;

void StraightInsertion (int \*a, int n);

void ShellSort (int \*a, int n) ;

void tim\_sort(int \* a, int n);

void NaturalMergeSort(int \*a, int n);

void writeFromFile(string file,int n);

void writeDown(string file,int n);

int p[N];

clock\_t start, finish;

string TEST\_NUMBERS = "test\_numbers.txt";

string KEYS\_10 = "10key.txt";

string KEYS\_100 = "100key.txt";

string KEYS\_500 = "500key.txt";

string KEYS\_1000 = "1000key.txt";

int main() {

void (\*f[4])(int \*, int) = {

StraightInsertion,

ShellSort,

tim\_sort,

NaturalMergeSort

};

//system("pause");

///Основная часть

cout << "\t\t\t\t\t\t\t\t\tMAIN" << endl;

cout << endl;

for (int j=0; j<4; j++) //Для каждого алгоритма

{

if(j==0) cout << "\t\t\t\t\t\t\t\t\tStraightInsertion" <<endl;

else if (j==1) cout << "\t\t\t\t\t\t\t\t\Shell" <<endl;

else if (j==2) cout << "\t\t\t\t\t\t\t\t\Timsort" <<endl;

else if (j==3) cout << "\t\t\t\t\t\t\t\t\tNaturalMergeSort" <<endl;

cout << "----------------------------------" << endl;

int ii = 10000;

while(ii <= 100000) { //Разные N

cout << "\t\t\t\t\t\t\t\t\t|" << ii << "|" <<endl;

for (int i=0;i<3;i++) //Разной упорядочности

{

switch(i)

{

case 0: writeFromFile(TEST\_NUMBERS,ii); cout << "Random: " <<endl; break; //Рандомный порядок из файла

case 1: cout << "Up: " <<endl; break; //Значения из файла отсортированные по возрастанию

case 2: writeDown(TEST\_NUMBERS,ii); cout << "Down: " <<endl; //Значения из файла отсортированные по убыванию

}

counterHelp = counter = 0;

f[j](p,ii);

cout <<"counterHelp: "<< counterHelp<<" counterMain: "<< counter << " time: " << (double)(finish-start) / CLOCKS\_PER\_SEC << "s" << endl ;

cout << endl;

}

cout << "----------------------------------" << endl;

switch (ii) {

case 10000: ii = 30000; break;

case 30000: ii = 70000; break;

case 70000: ii = 100000; break;

case 100000: ii = 200000; break;

}

}

}

cout << "memory\_StraightInsertion: " << memory\_StraightInsertion << endl << "memory\_ShellSort: " << memory\_ShellSort << endl;

cout << "memory\_tim\_sort: " << memory\_tim\_sort << endl << "memory\_NaturalMergeSort: " << memory\_NaturalMergeSort << endl;

system("pause");

cout << endl;

cout << "\t\t\t\t\t\t\t\t\tADVANCED" << endl;

cout << endl;

///На повышенный

{

//Все аналогично базовой части за исключением набора значений(кол-во повторяющихся элементов)

for (int j=0; j<4; j++)

{

if(j==0) cout << "\t\t\t\t\t\t\t\t\tStraightInsertion" <<endl;

else if (j==1) cout << "\t\t\t\t\t\t\t\t\Shell" <<endl;

else if (j==2) cout << "\t\t\t\t\t\t\t\t\Timsort" <<endl;

else if (j==3) cout << "\t\t\t\t\t\t\t\t\tNaturalMergeSort" <<endl;

cout << "----------------------------------" << endl;

int ii = 10000;

while(ii <= 100000) {

cout << "\t\t\t\t\t\t\t\t\t|" << ii << "|" <<endl;

for(int t=0;t<4;t++)

{

string file;

switch(t){

case 0: file = KEYS\_10; break;

case 1: file = KEYS\_100; break;

case 2: file = KEYS\_500; break;

case 3: file = KEYS\_1000;

}

cout << "\t" << file << endl;

for (int i=0;i<3;i++)

{

switch(i){

case 0: writeFromFile(file,ii); cout << "Random: " <<endl; break;

case 1: cout << "Up: " <<endl; break;

case 2: writeDown(file,ii); cout << "Down: " <<endl;

}

counterHelp= counter = 0;

f[j](p,ii);

cout <<"per: "<< counterHelp<<" main: "<< counter << " time: " << (double)(finish-start) / CLOCKS\_PER\_SEC <<endl;

cout << endl;

}

cout << "----------------------------------" << endl;

}

switch (ii) {

case 10000: ii = 30000; break;

case 30000: ii = 70000; break;

case 70000: ii = 100000; break;

case 100000: ii = 200000; break;

}

}

}

}

return 0;

}

// прямые вставки

void StraightInsertion (int \*arr, int sizeOfArray) {

// cout << memory\_StraightInsertion << endl;

start = clock();

int i, j, currentKey;

if(sizeOfArray==100000)memory\_StraightInsertion += (3\*sizeof(int));

for ( counterHelp++, i = 1; i < sizeOfArray; i++, counterHelp++) {

currentKey = arr[i];

counter++;

for (j = i, counterHelp++; j && currentKey < arr[j-1]; j--, counterHelp++) { /\* находим место вставки, попутно сдвигая элементы \*/

arr[j] = arr[j-1];

counter++;

}

arr[j] = currentKey;

counter++;

}

finish = clock();

}

// shell

void ShellSort(int\* massive, int n) {

/\*шаг задается формулой h[t]=1, h[m]=2h[m+1]+1; t=log2(n)-1\*/

start = clock();

const int t = (int)(log(n) / log(2) - 1);

int i, j, k, m, x;

int\* h = (int\*)malloc(t \* sizeof(int)); h[t - 1] = 1; counterHelp++;

for (m = t - 2, counterHelp++; m >= 0; m--, counterHelp++) {

h[m] = h[m + 1] \* 2 + 1; counterHelp++;

}

if(n==100000)memory\_ShellSort += (6\*sizeof(int) + t \* sizeof(int));

for (counterHelp++, m = 0; m < t; m++, counterHelp++) {/\*последовательно перебираем все расстояния\*/

k = h[m]; counterHelp++;

for (counterHelp++, i = k; i < n; i++, counterHelp++) {/\*до конца цикла метод вставки с учетом шага h[m]\*/

x = massive[i]; j = i - k; counterHelp++; counter++;

while (j >= 0 && x < massive[j]) {

massive[j + k] = massive[j]; j -= k;

counterHelp++; counter++;

}

massive[j + k] = x;counter++;

}

}

free(h);

finish = clock();

}

// естественное слияние

void NaturalMergeSort(int \*a, int n)

{

int split; /\* индекс, по которому делим массив \*/

int last, end, i, \*p=a, \*tmp; counterHelp++;

char flag = 0, sorted = 0; counterHelp+=2;

int pos1, pos2, pos3;

if(n==100000)memory\_NaturalMergeSort += (n\*sizeof(int) + sizeof(last)\*7 + sizeof(p));

tmp = (int\*) malloc (n\*sizeof(int)); counterHelp++;

do { /\* если есть более 1 элемента \*/

end = n; pos2 = pos3 = 0; counterHelp+=3;

do {

p += pos2; end = n - pos3; counterHelp+=2;

for (split=1, counterHelp++; split < end && p[split-1] <= p[split]; split++, counterHelp++); /\*первая серия\*/

if (split == n) {

sorted = 1 ; counterHelp++; break;

}

pos1 = 0; pos2 = split; counterHelp+=2;

while ( pos1 < split && pos2 < end ) { /\*идет слияние, пока есть хоть один элемент в каждой серии\*/

if (p[pos1] < p[pos2]) {

tmp[pos3++] = p[pos1++]; counter++; counterHelp+=2;

} else {

tmp[pos3++] = p[pos2++]; counter++; counterHelp+=2;

if (p[pos2] < p[pos2-1]) break;

}

}

/\* одна последовательность закончилась - копировать остаток другой в конец буфера \*/

while ( pos2 < end && tmp[pos3-1]<=p[pos2] ) {

tmp[pos3++] = p[pos2++]; counter++; counterHelp+=2;

} /\* пока вторая последовательность не пуста \*/

while ( pos1 < split ) {

/\* пока первая последовательность не пуста \*/

tmp[pos3++] = p[pos1++]; counter++; counterHelp+=2;

}

} while (pos3 < n );

if (sorted) break;

p = tmp; counterHelp++;

tmp = a; counterHelp++;

a = p; counterHelp++;

flag = !flag; counterHelp++;

} while (split < n);

if (flag) {

for (pos1 = 0, counterHelp++; pos1 < n; pos1++, counterHelp++) {

tmp[pos1] = a[pos1]; counter++;

}

free (a);

}

else

free (tmp);

}

// Timsort

typedef struct {

int \*beg; /\* адрес начала последовательности \*/

int len; /\* длина последовательности \*/

}segment;

/\* вычисление длины последовательности в зависимости от N\*/

/\* ответ должен быть в диапазоне (32,64]\*/

int get\_min\_size (int n) {

int r = 0; counterHelp++;

if (n==100000) memory\_tim\_sort += sizeof(r);

while (n >= 64) {

n >>= 1; counterHelp++;

r |= n & 1; counterHelp++;

}

return n + r;

}

/\*разворот массива\*/

void reverse (int \*a, int n) {

int i, j, tmp;

for (i=0, j=n-1, counterHelp+=2; i<j; i++, j--, counterHelp+=2) {

if (a[i]!=a[j]) {

tmp = a[i]; counter++;

a[i] = a[j]; counter++;

a[j] = tmp; counter++;

}

}

}

/\* сортировкa вставками \*/

void insertion\_sort(int \*a, int n, int i) {/\*адрес начала массива, его размер, размер уже упорядоченной части\*/

int j, x;

for (; i<n; ++i, counterHelp++) {

x = a[i]; counter++;

for (j=i, counterHelp++; j && a[j-1]>x; --j, counterHelp++) {

a[j] = a[j-1]; counter++;

}

a[j]=x; counter++;

}

}

/\*слияние с предварительным копированием во вспомогательный массив более короткой последовательности\*/

void merge\_copy\_less (segment \* seg) {

int \*a = seg[0].beg; /\*адрес начала первой из сливаемых последовательностей\*/

int split = seg[0].len; /\*длина первой последовательности\*/

int n = split + seg[1].len; /\*длина результирующей последовательности\*/

int pos1=0; /\* текущая позиция чтения из первой последовательности a[0]...a[split] \*/

int pos2=split; /\*текущая позиция чтения из второй последовательности a[split]...a[n]\*/

int pos3=0; /\*текущая позиция записи в результирующей последовательности\*/

int \*temp;

counterHelp+=6;

if (seg[0].len < seg[1].len) {

temp = (int\*)malloc(sizeof(int)\*split);

/\*копируем первую последовательность во вспомогательный массив\*/

for (pos1 = 0, counterHelp++; pos1 < split; pos1++, counterHelp++) {

temp[pos1] = a[pos1]; counter++;

}

pos1 = 0;

while (pos1 < split && pos2 < n) { /\*идет слияние, пока есть хоть один элемент в каждой последовательности\*/

if (temp[pos1] <= a[pos2]) {

a[pos3++] = temp[pos1++]; counterHelp+=2; counter++;

} else {

a[pos3++] = a[pos2++]; counterHelp+=2; counter++;

}

}

/\* одна последовательность закончилась - копировать остаток другой в конец буфера \*/

/\* даже если вторая последовательность не пуста, она уже на месте \*/

while (pos1 < split) {

/\* пока первая последовательность не пуста \*/

a[pos3++] = temp[pos1++]; counter++; counterHelp+=2;

}

} else {

temp = (int\*)malloc(sizeof(int)\*seg[1].len);

/\*копируем вторую последовательность во вспомогательный массив\*/

for (pos1 = 0, pos2 = split, counterHelp+=2; pos2 < n; ) {

temp[pos1++] = a[pos2++]; counter++; counterHelp+=2;

}

/\*в этом случае слияние производится справа налево\*/

pos1 = split-1; pos2 = seg[1].len-1; pos3 = n-1; counterHelp+=3;

while (pos1 >= 0 && pos2 >= 0) { /\*идет слияние, пока есть хоть один элемент в каждой последовательности\*/

if (temp[pos2] > a[pos1]) {

a[pos3--] = temp[pos2--]; counter++; counterHelp+=2;

} else {

a[pos3--] = a[pos1--]; counter++; counterHelp+=2;

}

}

/\* одна последовательность закончилась - копировать остаток другой в конец буфера \*/

/\* даже если первая последовательность не пуста, она уже на месте \*/

while (pos2 >= 0) { /\* пока вторая последовательность не пуста \*/

a[pos3--] = temp[pos2--]; counter++; counterHelp+=2;

}

}

free (temp);

seg[0].len = n; counterHelp++;/\*фиксируем длину полученной последовательности в стеке\*/

}

/\*слияние близких по размеру последовательностей\*/

int try\_merge(segment \*seg, int top) {

int x, y, z;

while (top>0) {/\*пока в стеке больше одного элемента\*/

x = seg[top].len; counterHelp++;

y = seg[top-1].len; counterHelp++;

z = top>1 ? seg[top-2].len : 0; counterHelp++;

if (top>1 && z <= x+y) {/\*если в стеке не меньше трех элементов и третий сверху маловат\*/

if (z < x) {

merge\_copy\_less(&seg[top-2]); /\*сливаем ZY\*/

seg[top-1] = seg[top]; /\*корректируем стек\*/ counterHelp++;

} else {

merge\_copy\_less(&seg[top-1]); /\*сливаем YX\*/

}

top--; /\*стало на одну последовательность меньше\*/ counterHelp++;

}

else

if (y <= x) {/\*если предпоследний не больше последнего\*/

merge\_copy\_less(&seg[top-1]); /\*сливаем YX\*/

top--; counterHelp++;

} else {

break;

}

}

return top;

}

void tim\_sort(int \* a, int n) {

start = clock();

int min\_size = get\_min\_size(n); counterHelp++;

int size;

int i, j=0; counterHelp++;

/\*стек координат последовательностей\*/

segment \* seg = (segment\*) malloc (((n-1)/min\_size + 1) \* sizeof(segment));counterHelp++;

int t=-1; /\*вершина стека\*/ counterHelp++;

if (n==100000) memory\_tim\_sort += (sizeof(min\_size) + sizeof(size) + sizeof(i) + sizeof(j) + sizeof(t) + ((n-1)/min\_size + 1)\*sizeof(segment) + sizeof(seg));

/\*формирование упорядоченных последовательностей\*/

for (i=0, counterHelp++; i<n; i+=size, counterHelp++ ) {

for (j=i+1, counterHelp++; j<n && a[j-1]>=a[j]; j++, counterHelp++); /\*поиск обратно упорядоченной последовательности\*/

if (j!=i+1) {

reverse (a+i, j-i);

if (n==100000) memory\_tim\_sort += sizeof(int)\*3;

}

/\*переворачиваем найденную обратную последовательность\*/

if (j!=n && j-i<min\_size) {

size = n-i < min\_size ? n-i : min\_size; counterHelp++;

if (n==100000) memory\_tim\_sort += sizeof(int)\*2;

insertion\_sort (a+i, size, j-i ); /\*адрес начала фрагмента, его размер, размер упорядоченной части\*/

j = i + size; counterHelp++;

}

for (; j<n && a[j-1]<=a[j]; j++, counterHelp++); /\*ищем конец последовательности, если она не закончилась\*/

/\*запоминаем адрес начала и длину последовательности в стеке\*/

seg[++t].beg = a+i; counterHelp++;

size = seg[t].len = j-i; counterHelp+=2;

if (n==100000) memory\_tim\_sort += sizeof(int)\*3;

t = try\_merge(seg, t); counterHelp++;

}

/\*слияние всех оставшихся последовательностей\*/

while (t>0)

{

if (n==100000) memory\_tim\_sort += sizeof(int\*)\*2 + sizeof(int)\*5;

merge\_copy\_less(&seg[t-1]); /\*сливаем две последние последовательности\*/

t--; /\*количество последовательностей уменьшилось\*/counterHelp++;

}

free(seg);

finish = clock();

}

void writeFromFile(string fileName,int n)

{

ifstream file;

file.open(fileName);

if(file)

{

int t;

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

file >> t; //в t значения очередного числа из второй строчки

p[i] = t;

}

file.close();

}else

cout<<"nope";

}

void writeDown(string fileName,int n) {

for(int i=0;i<n/2;i++) {

int temp = p[i];

p[i] = p[n-i-1];

p[n-i-1] = temp;

}

}

Результаты работы программы:

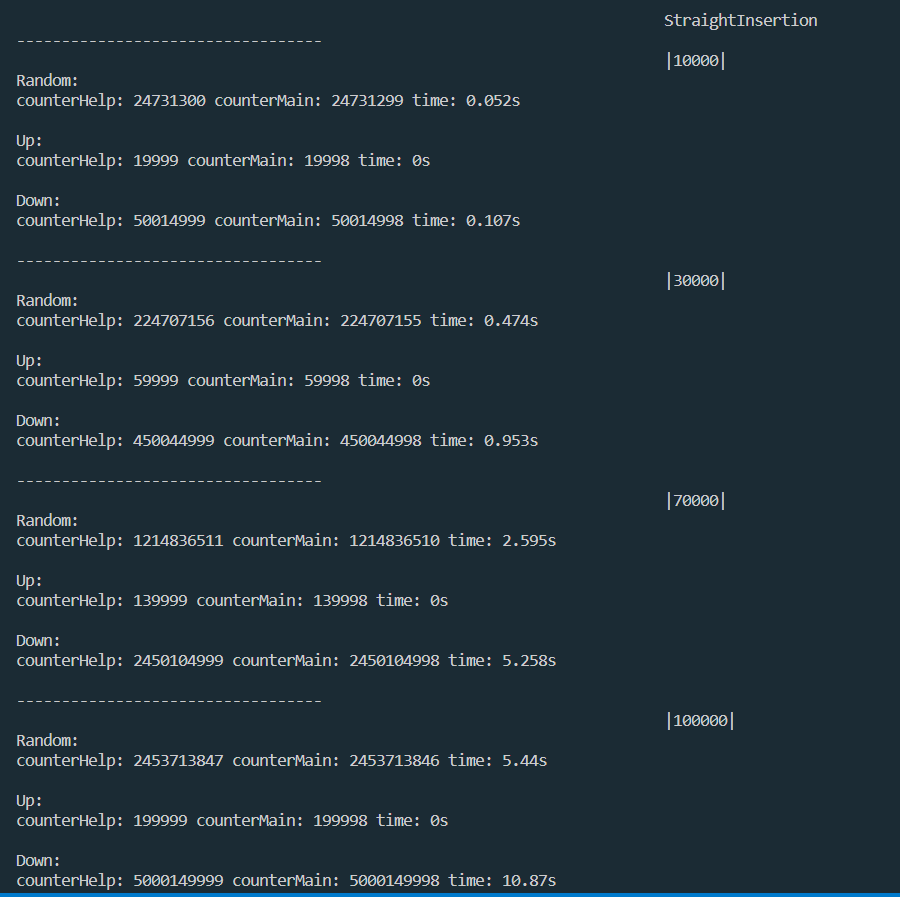


Рисунок 1 - сортировка методом простых вставок.

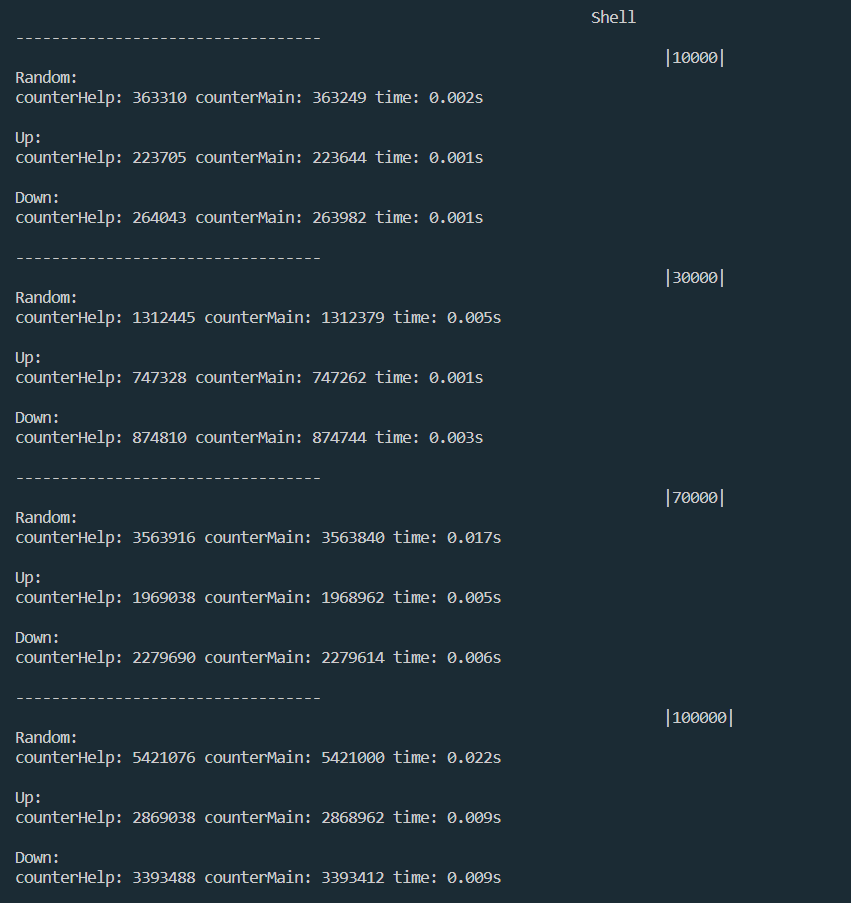


Рисунок 2 – сортировка с помощью алгоритма Шелла.

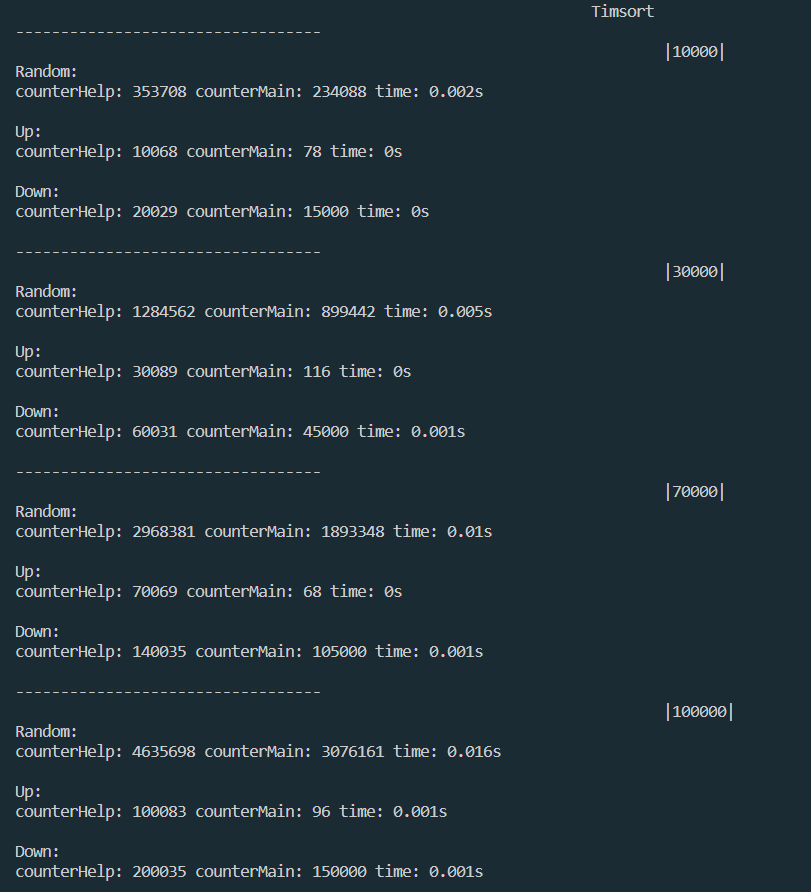


Рисунок 3 - сортировка «Timsort».

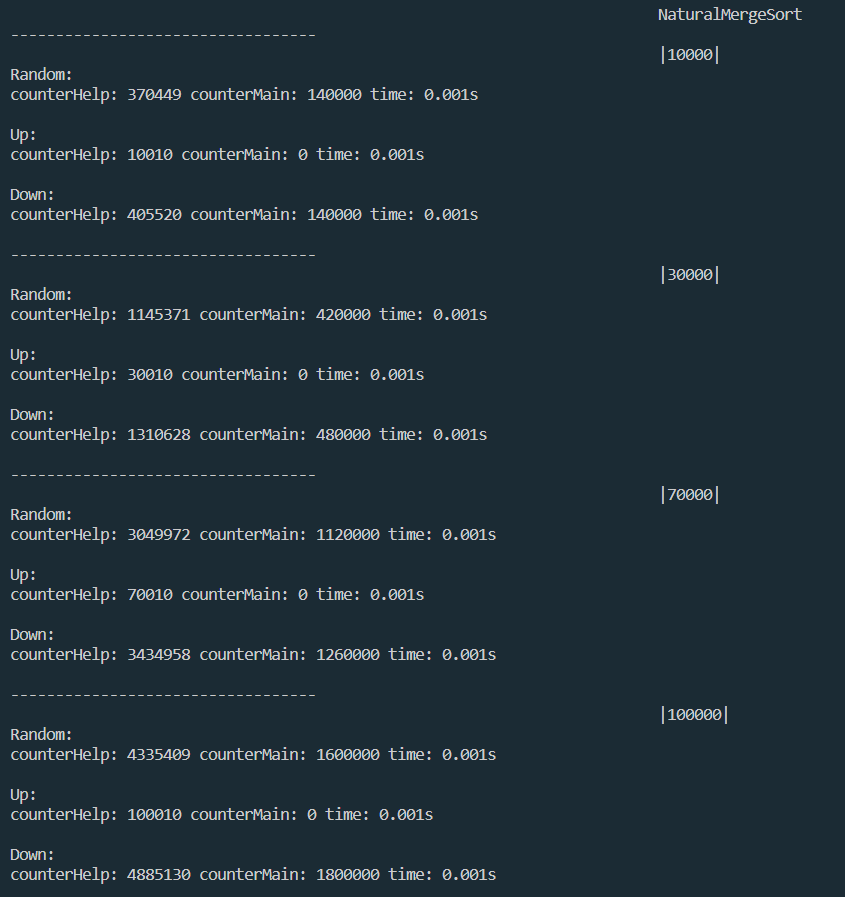


Рисунок 4 - сортировка естественным слиянием + кол-во памяти.

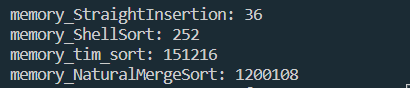


Рисунок 5 - кол-во памяти.

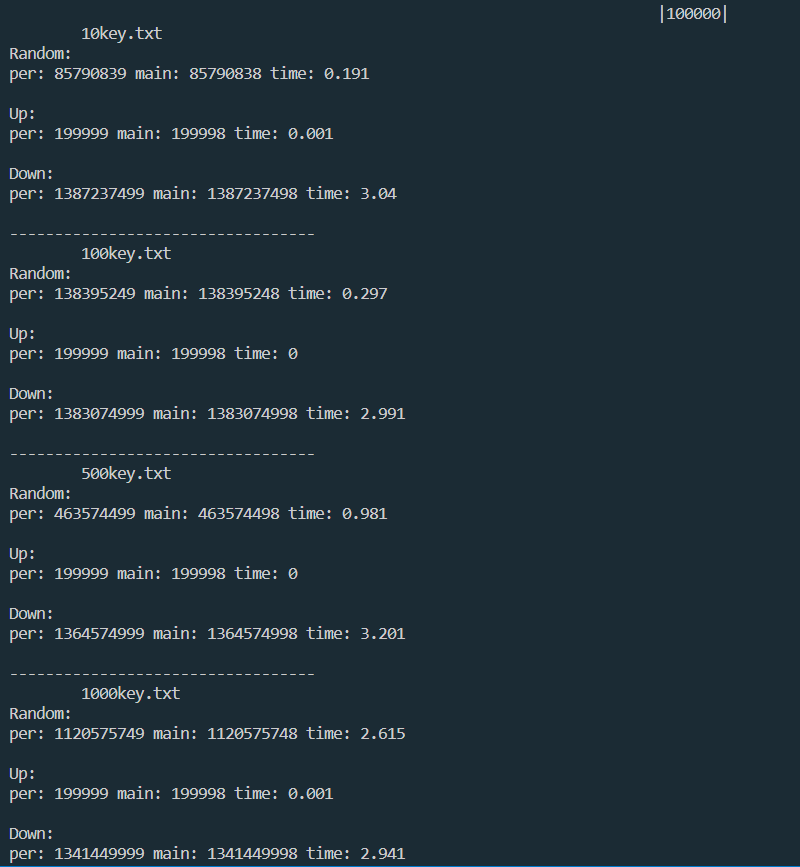


Рисунок 6 - сортировка методом простых вставок для N = 100.000.

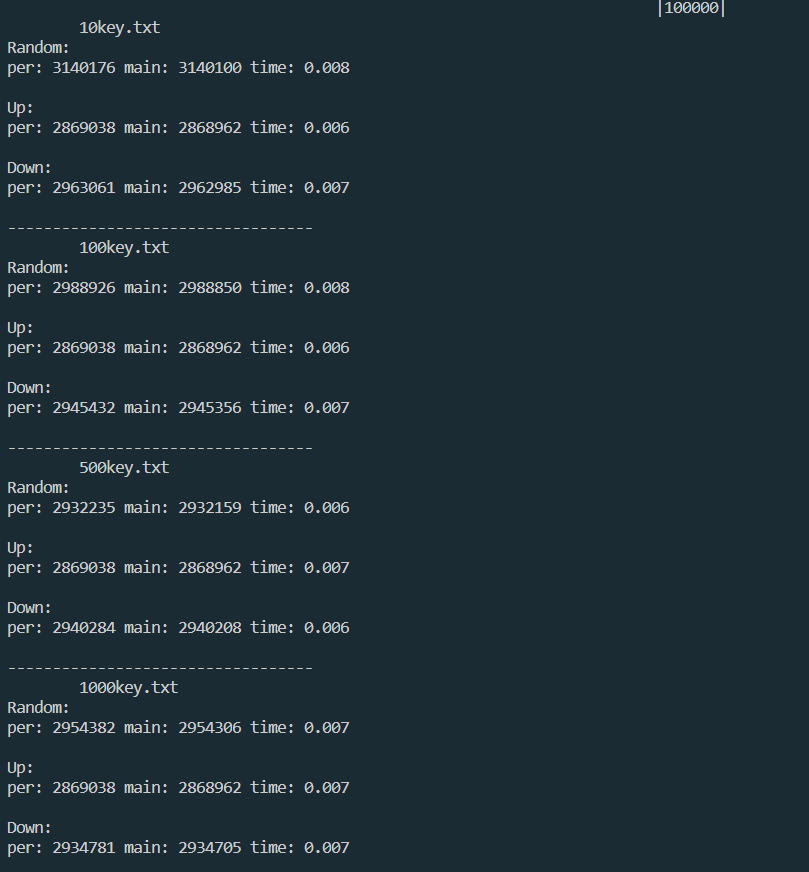


Рисунок 7 - сортировка с помощью алгоритма Шелла для N = 100.000.

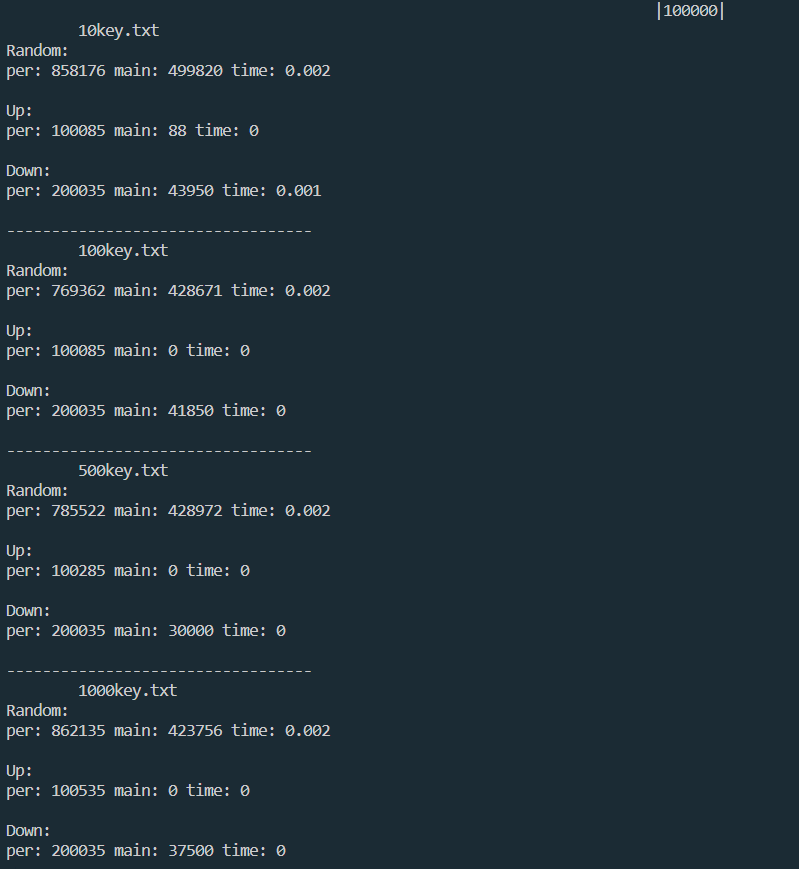


Рисунок 8 - сортировка «Timsort» для N = 100.000

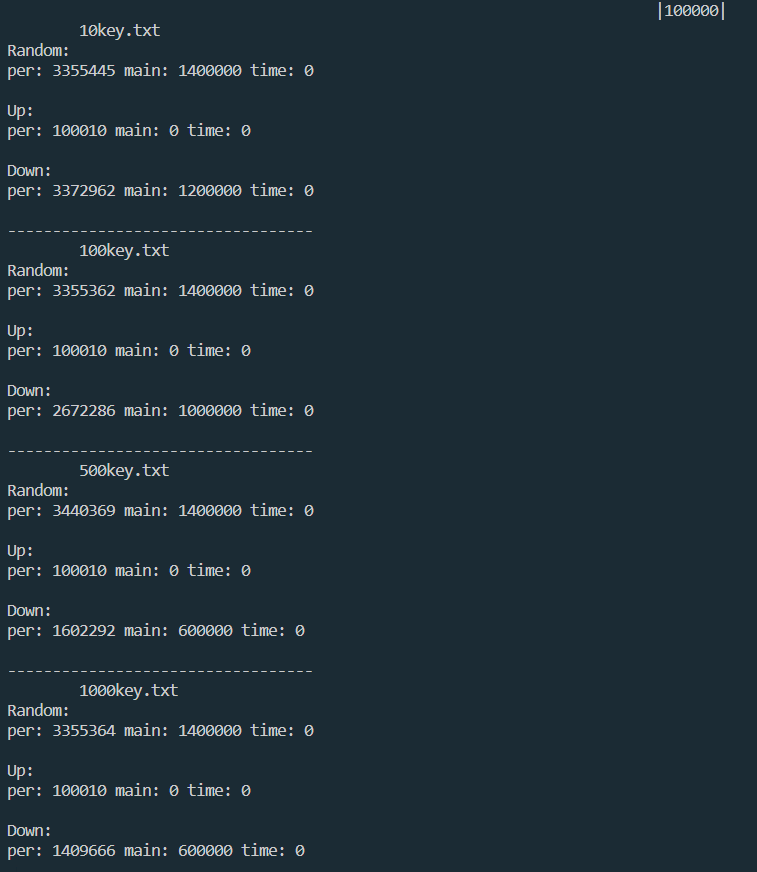


Рисунок 9 - сортировка естественным слиянием для N = 100.000

Анализ полученных данных:

1. Сравнение количества операций:

Сортировка Шелла - Количество вспомогательных присваиваний приблизительно равно присваиваниям элементов массива.

Cортировка методом простых вставок – Количество вспомогательных присваиваний равно(+-1) присваиваниям элементов массива.

Сортировка «Timsort» - Количество вспомогательных присваиваний в 1.5 раза больше присваиваний элементов массива.

Сортировка естественным слиянием – Количество вспомогательных присваиваний в 2.5 больше основных присваиваний.

Вывод: лидеры по кол-ву сравнений среди 4-х данных нам алгоритмов – сортировка естественным слиянием и «Timsort». Данные алгоритмы конкурируют между собой, т.к. результат зависит от набора данных. «Timsort» проигрывает сортировке естественным слиянием, если данных не упорядочены, но выигрывает если данные упорядочены в обратном порядке.

1. Сравнение по асимптотике:

У всех алгоритмов(кроме метода простых вставок средняя асимптотическая сложность - O(n2)) - одинаковая средняя асимптотическая сложность - O(n\*log(n)), следовательно, стоит обратиться к частным случаям.

Сортировка Шелла в отличие от сортировок «Timsort», сортировки естественным слиянием питает слабость к массивам упорядоченным в обратном порядк - асимптотическая сложность O(n2).

Опираясь на факты из реальной жизни сортировка «Timsort», будет более эффективна по сравнению с сортировка естественным слиянием, т.к. чаще в боевых условиях мы вынуждены обрабатывать данные, которых уже частично упорядочены.

На основе полученных данных из программы можно расставить алгоритмы от лучшего к худшему по признаку (сумма операций сравнения)/ (затраченное время), что покажет какой алгоритм решает задачу эффективнее:

Средние значения:

1)Сортировка «Timsort»

2)Сортировка естественным слиянием

3)Сортировка Шелла

4)Сортировка методом простых вставок

Повышенный:

Для всех алгоритмов сортировки зависят от повторяющихся ключей в той или иной степени. Безусловным лидером в данной области является Timsort, т.к. он изначально базируется на теории об использования на уже упорядоченных элементах, которые существуют в большинстве реальных наборов данных

Вывод: по результатам проведенного анализа самым эффективным из рассмотренных алгоритмов по соотношению время-память является алгоритм Timsort