МІНІСТЕРСТВО ОСВІТИ І НАУКИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ “ХПІ”

Кафедра “Обчислювальна техніка та програмування”

Розрахункове завдання з дисципліни

«Алгоритми та структури даних»

Пояснювальна записка

КІТ.120А.17-01 81 01-1 -ЛЗ

Розробники

Виконав:

студент групи КІТ-120А

Макаренко Владислав Олександрович

Перевірила:

Бречко Вероніка Олександрівна

Харків 2021

ЗАТВЕРДЖЕНО

КІТ.120А.17-01 80 01-1 -ЛЗ

Розрахункове завдання з дисципліни

«Алгоритми та структури даних»

Пояснювальна записка

КІТ.120А.17-01 81 01-1

Листів 15

Харків 2021

**1. Вимоги**

**1.1 Розробник**

Макаренко Владислав Олександрович;

студент групи КІТ-120а;

21-травня-2021

**1.2 Загальне завдання**

Виконати завдання згідно варіанту.

Варіант – 17.

**1.3 Індивідуальне завдання**

**Алгоритми сортувань.** Реалізувати алгоритми сортувань c порядком O(N\*log2N). Протестувати їх на випадкових наборах, упорядкованих за зростанням і зменшенням. Порівняти результати, побудувати таблиці і графіки.

**2. Опис програми**

**2.1 Турнірне сортування**

Цей метод сортування отримав свою назву через схожість з кубкової системою проведення спортивних змагань: елементи набору розбиваються на пари, менші елементи пар створюють пари сусіднього верхнього рівнів і т.д. Алгоритм сортування складається з двох етапів. На першому етапі будується дерево: аналогічне схемі розіграшу кубка.

Наступний етап полягає у вибірці значень з дерева і формування з них впорядкованої послідовності. У кожній ітерації циклу вибирається значення з вершини дерева - це найбільша з наявних значень ключа. Вузол-вершина при цьому звільняється, звільняються також і всі вузли, займані вибраним значенням на більш низьких рівнях турнірного дерева. За вузли що звільнилися влаштовується (від низу до верху) змагання між їхніми нащадками.

Побудова дерева вимагає **N-1** порівнянь, вибірка - ***N\*log2(N)*** порівнянь. Порядок алгоритму, таким чином, ***O(N\*log2(N)).*** Складність операцій над зв'язковими структурами даних, значно вище, ніж над статичними структурами. Алгоритм неекономічний щодо пам'яті: дублювання даних на різних рівнях піраміди призводить до того, що робоча область пам'яті містить приблизно **3\*N** вузлів.

**2.2 Пірамідальне сортування**

Через зазначеного недоліку (необхідний потрійний обсяг пам'яті) більш економічним є алгоритм так званої пірамідального сортування, в якому масив ***а*** представляється у вигляді піраміди.

У 1964 р Флойд запропонував метод побудови піраміди у вигляді масиву без явного побудови дерева. Відповідність між геометричною структурою піраміди як дерева і масивом встановлюється за такою схемою:

– в **a[0]** зберігається корінь дерева

– лівий і правий сини елемента **a[i]** зберігаються, відповідно, в **a[2i+1]** і **a[2i+2]**

Таким чином, для масиву, що зберігає в собі піраміду, виконується наступна характеристична властивість: **a[i]>=a[2i+1]** і **a[i]>=a[2i+2].** Причому елементи з індексами меншими, кількість яких дорівнює **(N/2)+1**, утворюють як би нижній рівень дерева.

Для побудови піраміди запропонований алгоритм, при якому в вузол верхнього рівня переміщається елемент з великим ключем.

Сортування відбувається наступним чином. На першому етапі будується піраміда на основі масиву. Як видно з властивостей піраміди, в корені завжди знаходиться максимальний елемент. Звідси випливає алгоритм другого етапу:

1). верхній елемент піраміди **a[0]...a[n]** (перший в масиві) міняється місцями з останнім;

2). після цього розглядається масив **a[0]...a[n-1].** Для перетворення його в піраміду досить просіяти лише новий перший елемент.

**2.3 Сортування частково впорядкованим деревом**

У двійковому дереві, яке будується в цьому методі сортування для кожного вузла справедливо наступне твердження: значення ключа, записане в вузлі, менше, ніж ключі його нащадків. Для повністю впорядкованого дерева є вимоги і до співвідношення між ключами нащадків. Для даного дерева таких вимог немає, тому таке дерево і називається частково упорядкованим. Крім того, дерево має бути абсолютно збалансованим. Це означає не тільки те, що довжини шляхів до будь-яких двох листів відрізняються не більш, ніж на 1, але і те, що при додаванні нового елемента в дерево перевага завжди віддається лівій гілці/подветви, поки це не порушує збалансованість.

Для частково упорядкованого дерева вводяться поняття «початку» і «кінця». Початком вважається вершина піраміди, а кінцем - крайній правий елемент в самому нижньому ряду.

Для сортування цим методом повинні бути визначені дві операції:

– вставка в дерево нового елемента

– вибірка з дерева мінімального елемента.

Причому виконання будь-якої з цих операцій не повинно порушувати ні сформульованої вище часткової впорядкованості дерева, ні його збалансованості.

При сортуванні частково впорядкованим деревом ефективним буде алгоритм, при якому двійкові дерева представляються в статичної пам'яті тобто у вигляді в одновимірного масиву.

У такому поданні відпадає необхідність зберігати в складі вузла дерева покажчики, так як адреси нащадків можуть бути обчислені. Так для вузла, представленого елементом масиву з індексом **i**, індекси його лівого і правого нащадків будуть **2\*i** та **2\*i+1**, відповідно. Для вузла з індексом i індекс його предка буде **i/2.**

**2.4 Швидке сортування Хоара**

Алгоритм сортування розроблений англійським інформатиком Ч. Хоаром в 1960 році. Алгоритм відноситься до розподільних і забезпечує показники ефективності O(N\*log(N)) навіть при найгіршому вихідному розподілі.

Загальна ідея алгоритму полягає в наступному:

1). Вибрати з масиву елемент, званий опорним. Це може бути будь-який з елементів масиву. Від вибору опорного елемента не залежить коректність алгоритму, але в окремих випадках може сильно залежати його ефективність;

2). Порівняти всі інші елементи з опорним і переставити їх у масиві так, щоб розбити масив на три безперервних відрізка, наступні один за одним: «менші опорного», «рівні» і «великі»;

3). Для відрізків «менших» і «великих» значень виконати рекурсивно ту ж послідовність операцій, якщо довжина відрізка більше одиниці.

На практиці масив зазвичай ділять не на три, а на дві частини: наприклад, «менші опорного» та «рівні і великі»; такий підхід в загальному випадку ефективніше, так як спрощує алгоритм поділу.

**2.5 Сортування злиттям**

Алгоритми сортування злиттям, як правило, мають порядок ***O(N\*log(N)),*** але відрізняються від інших алгоритмів більшою складністю і вимагають великого числа пересилань. Алгоритми злиття застосовуються в основному, як складова частина зовнішнього сортування.

Вхідна множина розглядається, як послідовність підмножин, кожне з яких складається з єдиного елемента і, отже, є вже впорядкованим. На першому проході кожні два сусідніх одно елементні множини зливаються в одну двоелементну впорядковану множину. На другому проході двоелементні множини зливаються в 4-елементні впорядковані множини і т.д. Зрештою виходить одна велика впорядкована множина.

**3. Програмний код**

**3.1 Турнірне сортування**

// Пеpеупоpядочивание пpедков узла tree[i].  
void Readjust (int *tree*[], unsigned short &*i*){  
 unsigned short j;  
 if ((*i* % 2)!=0) *tree*[*i* / 2] = *i* - 1;  
 else *tree*[*i* / 2] = *i* + 1;  
 for ( *i* /= 2; *i*>1; *i* /= 2) // Пpодвижение к коpню.  
 { if ((*i* % 2)!=0) j = *i* - 1; //j - бpат i.  
 else j = *i* + 1;  
 if (*tree*[*tree*[*i*]]>*tree*[*tree*[j]]) *tree*[*i* / 2] = *tree*[*i*];  
 else *tree*[*i* / 2] = *tree*[j];  
 }  
}  
  
void TournSort (int *a*[], int *sizeArr*){   
 const int size = 64; ; // Число листьев, необходимых в п о л н о м бинаpном деpеве.  
 // Значение пеpеменной size есть наименьшая степень 2, большая или == N.  
 int tree[128];  
 int k;  
 unsigned short i;  
 Initialize(*a*, tree, size);  
 // После того, как деpево постpоено, повтоpяем опеpацию пеpемещения элемента,  
 // пpедставленного коpнем, в следующуюпозицию с меньшим индексом в массиве x и пеpеупоpядочивание деpева.  
 for(k=*sizeArr*-1;k>=0;k--){   
 i = tree[1]; // i - индекс узла с листом, соответствующим коpню.  
 *a*[k] = tree[i]; // Поместить элемент, на котоpый ссылается коpень в позицию k.  
 tree[i] = -MAXINT;  
 Readjust (tree,i); // Пеpеупоpядочивание деpева в соответствии с новым содеpжимым tree[i].  
 }  
}

**3.2 Пірамідальне сортування**

// Процедура просеивания следующего элемента  
void downHeap(int *a*[], int *k*, int *n*){  
 // До процедуры: a[k+1]...a[n] - пирамида  
 // После: a[k]...a[n] - пирамида  
 int new\_elem;  
 int child;  
 new\_elem = *a*[*k*];  
  
 while(*k* <= *n*/2) // пока у a[k] есть дети  
 { child = 2\**k*;  
 // выбираем большего сына  
 if( child < *n* && *a*[child] < *a*[child+1] )  
 child++;  
 if( new\_elem >= *a*[child] ) break;  
 // иначе  
 *a*[*k*] = *a*[child]; // переносим сына наверх  
 *k* = child;  
 }  
 *a*[*k*] = new\_elem;  
}  
  
// Пирамидальная сортировка  
void HeapSort(int *a*[], int *size*){  
 int i;  
 // строим пирамиду  
 for(i=*size*/2-1; i >= 0; i--) {  
 downHeap(*a*, i, *size*-1);  
 }  
 // теперь a[0]...a[size-1] пирамида  
 for(i=*size*-1; i > 0; i--)  
 { // меняем первый с последним  
 swap(*a*[i], *a*[0]);  
 // восстанавливаем пирамидальность a[0]...a[i-1]  
 downHeap(*a*, 0, i-1);  
 }  
}

**3.3 Сортування частково впорядкованим деревом**

void Surface(int *a*[], int *i*, int *k*){  
 int j, m = 2\**i*, temp = *a*[*i*];  
 while(m <= *k*)  
 { if( m == *k*) j = m;  
 else if (*a*[m] > *a*[m+1]) j = m;  
 else j = m+1;  
 if(*a*[j] > temp)  
 { *a*[*i*] = *a*[j];  
 *i* = j;  
 m = 2\**i*;  
 }  
 else break;  
 }  
 *a*[*i*]=temp;  
}  
  
void TreeSort(int *a*[], int *size*)  
{ int i, k, w;  
 for(i = *size*/2; i >=0; i--)  
 Surface (*a*, i, *size*);  
 for(k = *size*-1; k >=0; k--)  
 { Surface (*a*, 0, k);  
 swap(*a*[k],*a*[0]);  
 }  
}

**3.4 Швидке сортування Хоара**

void HoaraSort(int *a*[], int *b*, int *e*)  
{  
 int i = *b*;  
 int j = *e*;  
 int tmp = 0;  
 int x = *a*[(i+j)/2];  
 do {  
 while (*a*[i] < x) i++;  
 while (*a*[j] > x) j--;  
 if (i <= j)  
 {  
 swap(*a*[i], *a*[j]);  
 i++;  
 j--;  
 }  
 } while (i <= j);  
 if (i<*e*) HoaraSort(*a*, i, *e*);  
 if (*b*<j) HoaraSort(*a*, *b*, j);  
}

**3.5 Сортування злиттям**

void PairMergeSort(int *a*[], int *size*){   
 int i0,j0,i,j,si,sj,k,ke,t,m ;  
 si=1; // начальный размер одного множества  
 while (si<*size*) //цикл пока одно множество не составит весь массив}  
 { i0 =0; // нач. индекс 1-го множества пары  
 while (i0<*size*) // цикл пока не пересмотрим весь массив  
 { j0 = i0+si; // нач. индекс 2-го множества пары  
 i=i0; j=j0;  
//размер 2-го множества пары может ограничиваться концом массива  
 if (si>=*size*-j0) sj=*size*-j0;  
 else sj=si;  
 if (sj>0)  
 { k = i0; // нач. индекс слитого множества  
 while ( (i < i0+si+sj) && (j < j0+sj)) // цикл пока не исчерпаются оба входные множества  
 { if (*a*[i]>*a*[j])  
// если эл-т 1-го <= элемента 2-го, он остается на своем месте, но вых.множество расширяется  
// иначе - освобождается место в вых.множестве и туда заносится эл-т из 2-го множества  
 { t=*a*[j];  
 for (m =j-1; m>=k; m--) *a*[m+1] =*a*[m];  
 *a*[k] =t; j++; //к след. эл-ту во 2-м множестве  
 }  
 k++; // вых. множество увеличилось  
 i++; // если был перенос - за счет сдвига, если не было - за счет перехода эл-та в вых.  
 }  
 }  
 i0 += si\*2; // начало следующей пары  
 }  
 si\*=2; // размер эл-тов пары увеличивается вдвое  
 }  
}

**4. Тести сортувань**

**4.1 Турнірне сортування**

* Тест сортування для 100 елементів(рис.1)

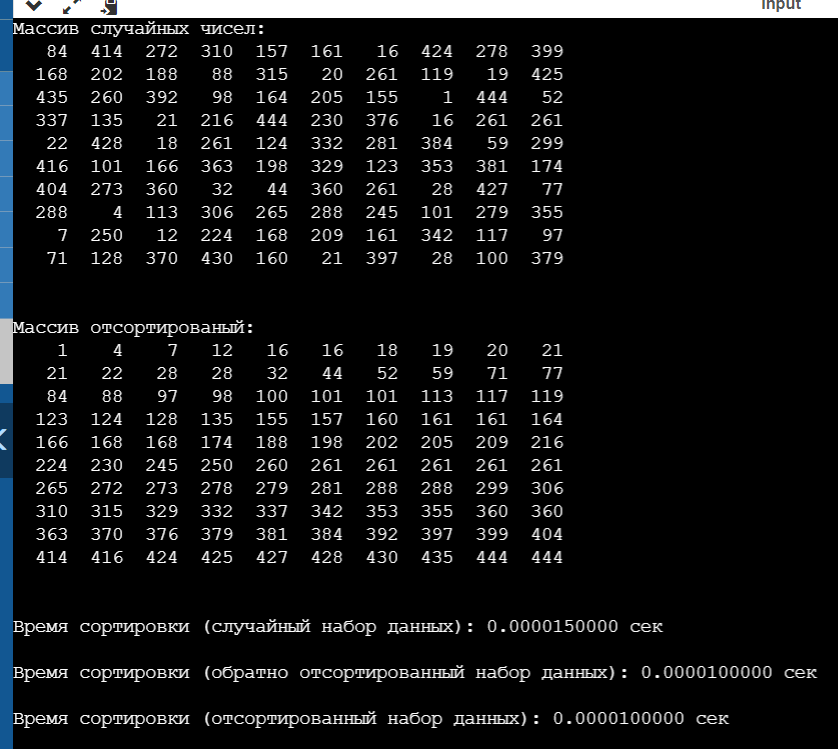


Рисунок 1 – результат тесту

* Тест сортування для 1000 елементів(рис.2)

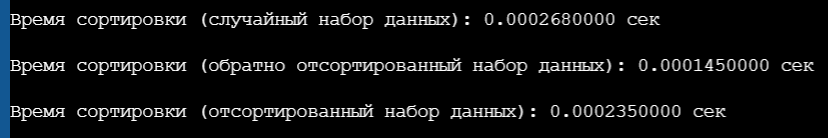


Рисунок 2 – результат тесту

* Тест сортування для 10000 елементів(рис.3)

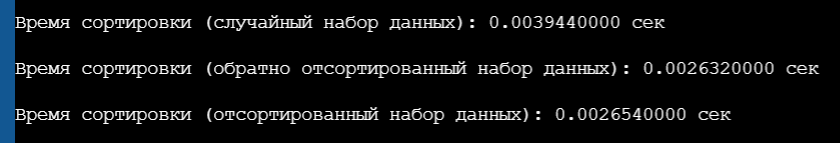


Рисунок 3 – результат тесту

**4.2 Сортування частково впорядкованим**

* Тест сортування для 100 елементів(рис.4)

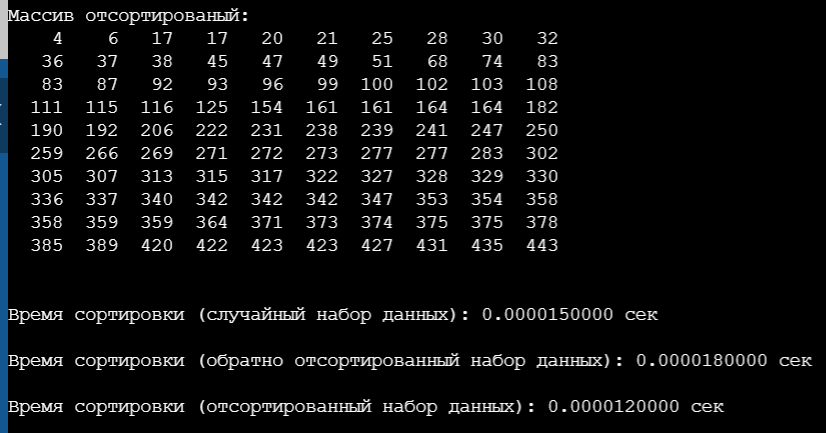


Рисунок 4 – результат тесту

* Тест сортування для 1000 елементів(рис.5)

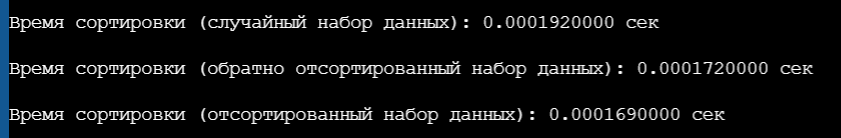


Рисунок 5 – результат тесту

* Тест сортування для 10000 елементів(рис.6)

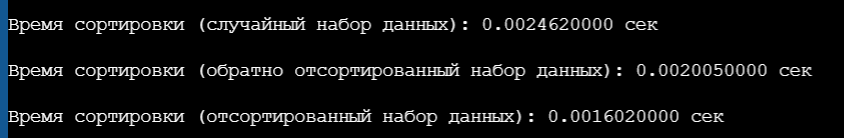


Рисунок 6 – результат тесту

**4.3 Пірамідальне сортування**

* Тест сортування для 100 елементів(рис.7)

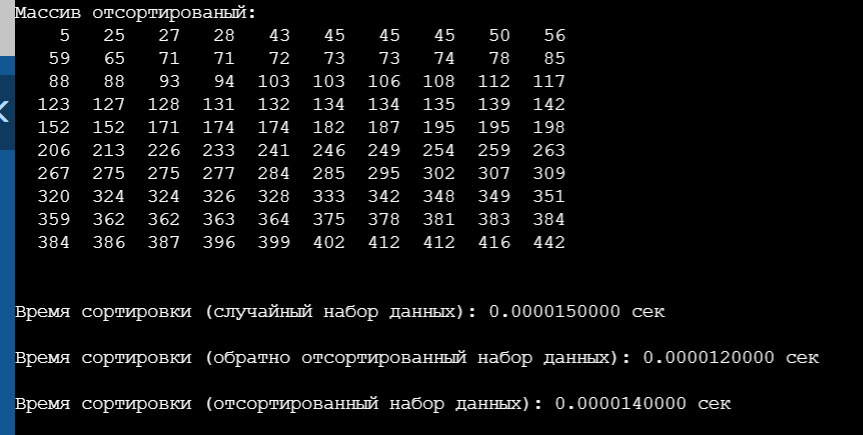


Рисунок 7 – результат тесту

* Тест сортування для 1000 елементів(рис.8)

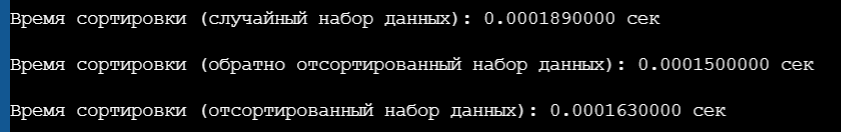


Рисунок 8 – результат тесту

* Тест сортування для 10000 елементів(рис.9)

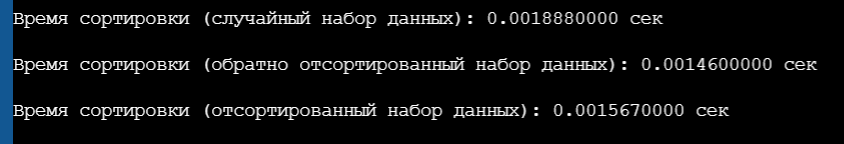


Рисунок 9 – результат тесту

**4.4 Швидке сортування Хоара**

* Тест сортування для 100 елементів(рис.10)

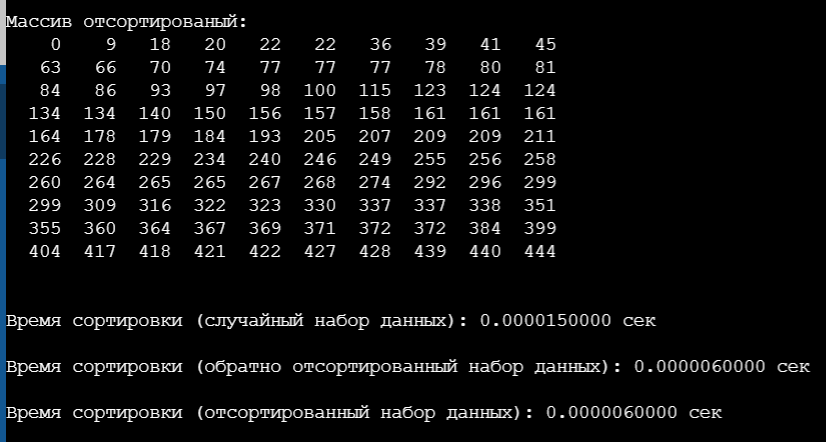


Рисунок 10 – результат тесту

* Тест сортування для 1000 елементів(рис.11)

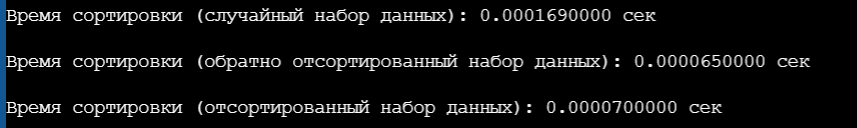


Рисунок 11 – результат тесту

* Тест сортування для 10000 елементів(рис.12)

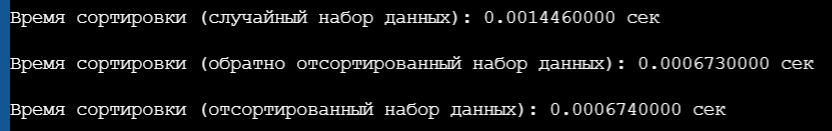


Рисунок 12 – результат тесту

**4.5 Сортування злиттям**

* Тест сортування для 100 елементів(рис.13)

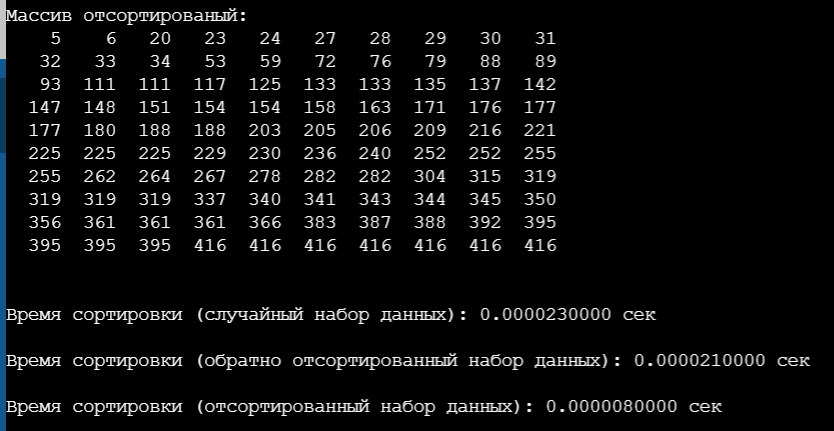


Рисунок 13 – результат тесту

* Тест сортування для 1000 елементів(рис.14)

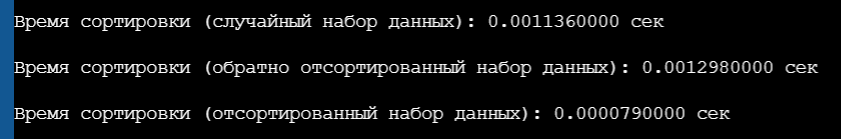


Рисунок 14 – результат тесту

* Тест сортування для 10000 елементів(рис.15)

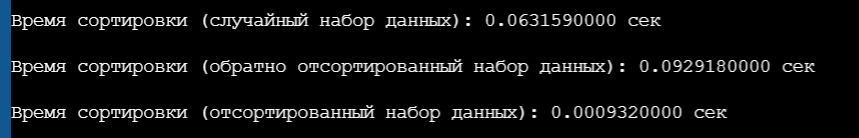


Рисунок 15 – результат тесту

**4. Результати**

У загальному випадку QuickSort є найшвидшим алгоритмом. Завдяки своїй ефективності, що дорівнює **O(n\*log n),** він явно перевершує будь-який алгоритм порядку **O(n2).** Судячи з результатів випробувань, він також швидше будь-якого з сортувань порядку **O(n\*log n).** Ефективність «швидкого сортування» становить **O(n log n)** навіть в екстремальних випадках.

Таблиця 2. – Сортування порядку **O(n log2 n)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Турнірне сортування | Сортування за допомогою дерева | Пірамідальне сортування | "Швидке сортування" | Сортування злиттям |
| n = 100 (випадковий набір даних) | 0.000015 | 0.000015 | 0.000015 | 0.000015 | 0.000023 |
| n = 100 (впорядкований за зростанням) | 0.00001 | 0.000012 | 0.000014 | 0.000006 | 0.000008 |
| n = 100 (впорядкований за спаданням) | 0.00001 | 0.000018 | 0.000012 | 0.000006 | 0.0000021 |
| n = 1,000 (випадковий набір даних) | 0.000268 | 0.000192 | 0.000189 | 0.000169 | 0.001136 |
| n = 1,000 (впорядкований за зростанням) | 0.000235 | 0.000169 | 0.000163 | 0.00007 | 0.000079 |
| n = 1,000 (впорядкований за спаданням) | 0.000145 | 0.000172 | 0.00015 | 0.000065 | 0.001298 |
| n = 10,000 (випадковий набір даних) | 0.003944 | 0.002462 | 0.001888 | 0.001446 | 0.063159 |
| n = 10,000 (впорядкований за зростанням) | 0.002654 | 0.001602 | 0.001567 | 0.00674 | 0.000932 |
| n = 10,000 (впорядкований за спаданням) | 0.002632 | 0.002005 | 0.00146 | 0.00673 | 0.092918 |

**Висновки**

* При сортуванні великих масивів вихідних даних краще використовувати швидке сортування.
* Якщо ж додається вимога гарантувати прийнятний час роботи методу, то треба застосовувати або сортування деревом, або сортування злиттям. Як видно з таблиць, сортування злиттям працює швидше, але слід пам'ятати, що вона вимагає додаткову пам'ять розміром порядку **N**.
* При аналізі алгоритмів сортувань говорять про кількість порівнянь (воно визначає порядок алгоритму) і кількість перестановок. Для машин з конвеєрним виконанням команд значення має в основному кількість порівнянь, тому що порівняння пов'язано з припиненням «конвеєра» процесора до прийняття рішення за результатами команди порівняння. Перестановки цей процес не порушують і тому виконуються швидко. Для повільних машин кількість перестановок істотно впливає на час сортування.