Лабораторная работа

Сортировки

**Цель работы:**

1. Знакомство с основными алгоритмами сортировки;
2. Получение практических навыков использования алгоритмов сортировки.

**Теоретические сведения:**

Сортировкой называют упорядочение информации по какому-либо признаку - **ключу** сортировки.

Сортировки делятся на **внешние** и **внутренние**.

1. Внешние сортировки используются, когда вся сортируемая информация не может быть одновременно размещена в памяти ЭВМ.
2. Внутренняя сортировка применяется, когда вся сортируемая информация помещается в память ЭВМ.

Внешняя сортировка использует внутреннюю как подпрограмму.

**Алгоритмы внутренней сортировки:**

1. Сортировка вставками (*insertion sort*);
2. Сортировка Шелла (*Shell sort*);
3. Сортировка пузырьком (*bubble sort*);
4. Сортировка посредством выбора (*selection sort*);
5. Сортировка подсчетом;
6. Сортировка слиянием (*merge sort*);
7. Распределяющая, или поразрядная сортировка (*radix sort*);
8. Быстрая сортировка (*quick sort*).

**Сортировка вставками**

Элементы просматриваются по одному, и каждый новый элемент вставляется в подходящее место среди ранее упорядоченных элементов.

void insertionsort(int\* left, int\* right)

{

for (int\* i = left + 1; i <= right; i++)

for (int\* j = i; j > left && \*(j - 1) > \*j; j--)

swap(\*(j - 1), \*j);

}

**Сортировка Шелла**

Сортировка Шелла является модификацией сортировки вставками.  
Идея метода заключается в сравнении разделенных на группы элементов последовательности, находящихся друг от друга на некотором расстоянии **delta**.   
Изначально это расстояние равно **size / 2**, где **size** — общее число элементов.

1. На первом шаге каждая группа включает в себя два элемента расположенных друг от друга на расстоянии **size / 2**; они сравниваются между собой и, в случае необходимости, меняются местами;
2. На последующих шагах также происходят проверка и обмен, но расстояние **delta** сокращается   
   **delta / 2**;
3. Постепенно расстояние между элементами уменьшается, и при **delta = 1** проход по массиву происходит в последний раз.

Единственной характеристикой сортировки Шелла является *приращение* - расстояние между сортируемыми элементами, в зависимости от прохода. В конце приращение всегда равно единице - метод завершается обычной сортировкой вставками, но именно последовательность приращений определяет рост эффективности.

void ShellSort(int \*arr, int size)

{  
 int delta = size / 2; // шаг

while (delta > 0)   
 {  
 for (int i = 0; i + delta < size; i++)   
 {  
 for (int j = i; j >= 0; j -= delta)   
 {  
 if (arr[j] > arr[j + delta]) { swap(arr[j], arr[j + delta]); }  
 else break;  
 }  
 }  
 delta = delta / 2; // шаг уменьшается в два раза  
 }  
}

**Сортировка пузырьком**

Если два элемента расположены не по порядку, то они меняются местами. Этот процесс повторяется до тех пор, пока элементы не будут упорядочены.

void bubblesort(int\* left, int\* right)

{

int size = right - left;

if (size < 1) return;

bool f = true;

while (f)

{

f = false;

for(int\* i = left; i < right; i++)

if (\*i > \*(i + 1))

{

swap(\*i, \*(i + 1));

f = true;

}

right--;

}

}

**Сортировка выбором**

Сначала выделяется наименьший (наибольший) элемент и отделяется от остальных, затем выбирается наименьший (наибольший) их оставшихся и т.д.

void selectionsort(int\* left, int\* right)

{

for (int\* i = left; i <= right; i++)

{

int min = \*i, \*num = i;

for (int\* j = i + 1; j <= right; j++)

if (\*j < min)

{

min = \*j;

num = j;

}

swap(\*i, \*num);

}

}

**Сортировка подсчетом**

Сортировка подсчетом подразумевает создание корзин (*buckets*), в каждой из которых хранится *количество элементов* исходного массива, значение которых *совпадает с индексом корзины*.

Соответственно, нужно иметь корзины, индексы которых будут от *минимального* значения массива до *максимального*.

При использовании массива для хранения корзин индексы будут от 0 до X.

Сортируемый массив:



Вспомогательный массив (корзина):



Удобный вариант:



Чтобы при этом можно было работать с отрицательными числами, а также чтобы не хранить ненужные значения от 0 до минимального значения из массива, имеет смысл перед сортировкой (или непосредственно перед добавлением в корзину) вычесть из всех элементов массива минимум, а после сортировки (при извлечении из корзины) - добавить его обратно.

void countsort(int\* arr, int size)

{

int min = \*arr;

int max = \*arr;

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

{

if (max < arr[i]) max = arr[i];

if (min > arr[i]) min = arr[i];

}

int cnt\_sz = max - min + 1; // резервируется память под массив - "корзину"

int\* counter = new int[cnt\_sz];

for (int i = 0; i < cnt\_sz; i++) // заполнение массива нулями

counter[i] = 0;

for (int i = 0; i < size; i++) // подсчёт кол-во повторяющихся элементов

counter[arr[i] - min]++;

int i = 0, j = 0;

while (i < size) { // сортировка в первоначальный массив arr

while (counter[j] != 0)

{

arr[i] = j + min;

counter[j]--;

i++;

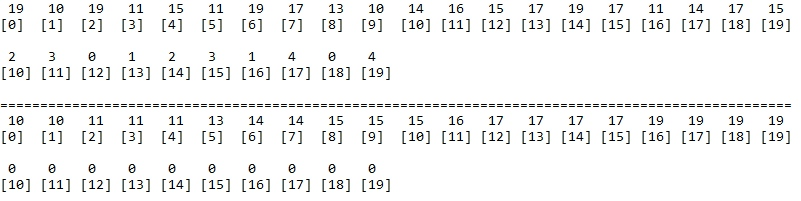
}

j++;

}

}

Результат работы программы:



**Сортировка слиянием**

Сортированные подмножества объединяются в более крупные подмножества без нарушения их *отсортированности*.

*Сортировка слиянием* – алгоритм, который сортирует такие структуры данных, где доступ к элементам осуществляется последовательно.

**Алгоритм:**

1) Последовательность А разбивается на две части В и С.

2) Последовательность В и С сливаются при помощи объединения отдельных элементов в упорядоченные пары.

3) Полученной последовательности присваивается имя А, и повторяются шаги 1) и 2); на этот раз упорядоченные пары сливаются в упорядоченные четверки.

4) Предыдущие шаги повторяются: четвёрки сливаются в восьмёрки, и весь процесс повторяется до тех пор, пока не будет упорядочена вся последовательность.

struct edges   
{  
 int\* l; // указатель на левую границу массива  
 int\* r // указатель на правую границу массива  
};

edges MergeSort(int \*l, int \*r) // функция деления массива пополам  
{  
 if (l **==** r) return { l, r }; // условие выхода из рекурсии  
 int size = r - l + 1;  
 int\* l1 = l + size / 2;  
 int\* r1 = l1 - 1;

return Merge(MergeSort(l, r1), MergeSort(l1, r)); // вызов самой сортировки через рекурсивный вызов

}

edges Merge(edges first, edges last)   
{  
 int\* l1 = first.l; // для удобства запишем короткие названия переменных  
 int\* r1 = first.r;  
 int\* l2 = last.l;  
 int\* r2 = last.r;

int size = r2 - l1 + 1;  
 int\* arr = new int[size]; // промежуточный массив, в который будут записываться результаты сортировки

int i = 0;  
 while (l1 <= r1 || l2 <= r2) // пока не закончились подмассивы  
 {   
 if (l1 > r1) { arr[i] = \*l2; l2++; i++; continue; } // если закончился первый массив, то записываем оставшиеся элементы в конец промежуточного массива  
 if (l2 > r2) { arr[i] = \*l1; l1++; i++; continue; } // если закончился второй массив  
 if (\*l1 < \*l2) // если элемент меньше  
 {  
 arr[i] = \*l1;  
 l1++;   
 i++;  
 }  
 else   
 {  
 arr[i] = \*l2;  
 l2++;  
 i++;  
 }  
 }  
 l1 = first.l; // заново присваиваем начало первого подмассива  
 for (int i = 0; i < size; i++) // переписываем из промежуточного массива в основной  
 {  
 \*l1 = arr[i];  
 l1++;  
 }  
 delete[] arr;  
 return { first.l,last.r };  
}

**Распределяющая сортировка**

Метод, прямо противоположный методу слияния:

1. Элементы вначале разбиваются на группы по одному признаку;
2. Каждая группа вновь разбивается на подгруппы.

Распределяющая, или *поразрядная сортировка* — сортировка по разрядам.

Существует две разновидности: **LSD** и **MSD**. В первом случае происходит сортировка элементов по младшим разрядам (все оканчивающиеся на 0, затем на 1 и так до 9). После этого они группируются по следующему с конца разряду, пока они не закончатся. В MSD сортировка происходит по старшему разряду.

**Быстрая сортировка**

*Быстрая сортировка* — в целом это один из самых быстрых алгоритмов сортировки массивов, однако на практике он чаще всего применяется с разного рода модификациями. Является примером принципа *«разделяй и властвуй»*.

Идея алгоритма заключается в том, что выбирается *опорный элемент*, относительно которого будет происходит сортировка. Равные и большие элементы помещаются справа, меньшие – слева. Затем к полученным подмассивам рекурсивно применяются два первых пункта.

*Рандомная* *быстрая* *сортировка* – то же самое, что и быстрая сортировка, лишь с тем отличием, что опорный элемент выбирается случайно.

Рассмотрим сортировку на примере массива:

**10, 4, 2, 14, 67, 2, 11, 33, 1, 15.**

Пусть крайний левый элемент — разрешающий *pivot*. Установим указатель *left* на следующий за ним элемент; *right* — на последний. Алгоритм должен определить правильное положение элемента 10 и по ходу дела поменять местами неправильно расположенные элементы.



Рисунок 1 – Быстрая сортировка

Указатель *left* перемещается до тех пор, пока не покажет элемент больше 10; *right* движется, пока не покажет элемент меньше 10.

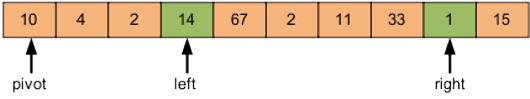


Рисунок 2 – Быстрая сортировка

Эти элементы меняются местами и движение указателей возобновляется.

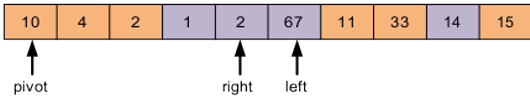


Рисунок 3 – Быстрая сортировка

Процесс продолжается до тех пор, пока *right* не окажется слева от *left*.  
Тем самым будет определено правильное место разрешающего элемента.

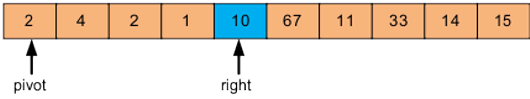


Рисунок 4 – Быстрая сортировка

Осуществляется перестановка разрешающего элемента с элементом, на который указывает *right*.

void QuickSort(int \*left, int \*right)

{  
 if (left == right) return; // Если массив/подмассив состоит из одного эл-та  
 int\* l = left; // Индексы для обхода  
 int\* r = right;  
 int \*pivot = left; // Опорный эл-т

while (l <= r) // Пока индексы не пересеклись  
 {

if (\*l > \*pivot && \*r <= \*pivot) { swap(\*l, \*r); continue; }// Если оба эл-та под индексами не удов. условию  
 if (\*l <= \*pivot)    { l++; continue; } // Если эл-т под l меньше или равен опорному  
 if (\*r > \*pivot)   { r--; continue; } // Если эл-т под r строго больше опорного   
 }  
 if (l > right) { l--; r--; swap(\*l, \*pivot); } // Если "левый индекс" вышел за правую границу массива

QuickSort(left, r); // Рекурсивные вызовы для обоих получившихся подмассивов  
 QuickSort(l, right);  
}

**ЗАДАЧА НЕ ДЛЯ ВЫПОЛНЕНИЯ!**

**Постановка задачи:**

1. Реализовать сортировку Шелла;
2. Реализовать на выбор: быстрая сортировка, сортировка слиянием;
3. Сравнить на различных входных данных (массивы разной длинны) скорость работы сортировки Шелла и выбранной сортировки, сделать выводы.