ЛАБОРАТОРНАЯ РАБОТА № 20
СОРТИРОВКА

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Сортировка – изменение порядка следования элементов в массиве таким образом, что значения его элементов становятся упорядоченными по некоторому закону. Сортировать можно массивы с числовыми значениями (по возрастанию или по убыванию), массивы строк (по алфавиту), массивы дат, времен и других нестандартных типов.

Существует много методов сортировки, отличающихся по количеству операций сравнения, копирования и времени выполнения.

Пузырьковая сортировка

При пузырьковой сортировке сравниваются соседние элементы и, если следующий элемент меньше предыдущего, они меняются местами.

Требуется несколько проходов по массиву. Во время первого прохода сравниваются крайние два элемента в массиве. Если они стоят не в том в порядке, который нужен, они меняются местами. Затем сравнивается элементы в следующей паре и, если необходимо, тоже меняются местами и т.д.

Таким образом, сравнение происходит в каждом цикле, пока не будет достигнут конец массива.

```
int c;
for (int i = 0; i < N - 1; i++)
  for (int j = N - 2; j >= i; j--)
    if (Arr1[j] > Arr1[j + 1])
    {
      c = Arr1[j];
      Arr1[j] = Arr1[j + 1];
      Arr1[j + 1] = c;
    }
```

С использованием функций сравнения и перестановки, алгоритм будет понятнее, например, для сортировки массива целых чисел можно написать следующие функции:

```
void Swap(int* pa, int* pb)    // Перестановка
{
  int c = *pa;
  *pa = *pb;
  *pb = c;
}
void Order(int *a, int *b)    // Сравнение и перестановка
{
  if (*a > *b)
    Swap(a, b);
}
```

Сама функция сортировки будет выглядеть так:

```
void BubbleSort(int *A, int N) // Сортировка
{
  for (int i = 0; i < N - 1; i++)
    for (int j = N - 2; j >= i; j--)
      Order(&A[j], &A[j + 1]);
}
```

Пример применения функции сортировки:

```
. . . . . .
int Arr[10] = {25, 2, 3, 45, 5, 3, 7, 8, 9, 10};

for(int i = 0; i < 10; i++) // Отображаем массив до сортировки
  printf("%d   ", Arr1[i]);
BubbleSort(Arr, 10);          // сортировка
printf("\n============\n");
for (int i = 0; i < 10; i++) // Отображаем массив после сортировки
  printf("%d   ", Arr1[i]);
```

Сортировка переменных других типов можно проводить аналогично. Рассмотрим функцию сортировки массива структур “Ромб”, например по площади.

```
struct Rhombus  // Ромб
{
  int x;   // Координата x центра ромба
  int y;   // Координата y центра ромба
  int d1;  // Первая диагональ ромба
  int d2;  // Вторая диагональ ромба
};
```

Для этого нужно написать свои функции сравнения и перестановки. Используем в них функции, написанные в лабораторной работе 129.

```
float RArea(Rhombus a)  // Вычисление площади ромба
{
  return (float)a.d1 * a.d2 / 2;
}
bool CmpB(Rhombus a, Rhombus b) // Сравнение ромбов по площади
{
  return (RArea(a) > RArea(b));
}
void SwapR(Rhombus *pa, Rhombus *pb) // Перестановка ромбов
{
  Rhombus c = *pa;
  *pa = *pb;
  *pb = c;
}
void OrderR(Rhombus *pa, Rhombus *pb) // Сравнение и перестановка
{
  if (CmpB(*pa, *pb))
    SwapR(pa, pb);
}
```

В функцию сортировки в качестве параметров передаем указатель на массив ромбов и размер массива:

```
void BubbleSortR(Rhombus *A, int N)
{
  for (int i = 0; i < N - 1; i++)
    for (int j = N - 2; j >= i; j--)
      OrderR(&A[j], &A[j + 1]);
}
```

Для сортировки не по площади, а по другому параметру, например, периметру, нужно написать соответствующие функции.

В программе, показывающей работу алгоритма сортировки, используем функции заполнения и отображения массива, написанные в лабораторной работе 19.

```
Rhombus *ArrR;
int size;
// . . . . Здесь нужно задать размер массива size
ArrR = new Rhombus[size]; // Выделяем память под size структур
FillArrayRand(ArrR, size); // Заполняем случайными числами
ViewArr(ArrR, size);       // Отображаем массив до сортировки
BubbleSortR(ArrR, size);   // Сортируем
ViewArr(ArrR, size);       // Отображаем массив после сортировки
```

Метод пузырька работает медленно, особенно на больших массивах. При увеличении размера массива в 10 раз время выполнения программы увеличивается в 100 раз.

Еще один недостаток метода пузырька состоит в том, что приходится слишком часто переставлять местами соседние элементы.

Метод выбора минимального элемента

Избежать большого числа перестановок можно, если использовать метод выбора минимального элемента.

Алгоритм метода: из массива выбирается минимальный элемент (если сортировка по возрастанию) и переставляется с первым. Далее этот процесс повторяется для оставшейся части массива (без первого элемента), затем без первых двух и т. д.

В сравнении с методом пузырька, этот метод требует значительно меньше перестановок элементов (в худшем случае N-1). Он дает значительный выигрыш, если перестановки сложны и занимают много времени.

Функция сортировки методом выбора минимального элемента в массиве целых чисел

```
void SelectSort(int *A, int N)
{
  int imin;
  for(int i=0; i<N-1; i++)
  {
    imin = i;
    for(int j=i+1; j<N; j++)
      if (A[j] < A[imin])
        imin = j;
    Swap(&A[i], &A[imin]);
  }
}
```

Пример использования:

```
int *Arr;
int size;
// . . . . Здесь нужно задать размер массива size
Arr = new int[size]; // Выделяем память под size чисел
// . . . . Заполняем случайными числами
ViewArr(Arr, size);    // Отображаем массива до сортировки
SelectSort(Arr, size); // Сортируем
ViewArr(Arr, size);    // Отображаем массива после сортировки
```

Данную функцию сортировки также легко переделать и для нестандартных типов.

Например, для сортировки массива ромбов по площади нужно исправить заголовок функции (передавать в функцию указатель на массив ромбов), а вместо оператора сравнения:

```
      if (A[j] < A[imin])
```

использовать функцию сравнения ромбов:

```
      if (CmpB(A[j], A[imin]))
```

Функцию перестановки можно взять из предыдущего метода (SwapR).

Быстрая сортировка

Один из лучших известных методов сортировки массивов – **быстрая сортировка** Ч. Хоара – основана на применении рекурсии.

Пусть дан массив **A** из **n** элементов. Выберем сначала, так называемый, **опорный** элемент массива (назовем его **x**). Обычно выбирают средний элемент массива, хотя это не обязательно. На первом этапе переставляем элементы так, чтобы слева находились все числа, меньшие или равные **x**, а справа – большие или равные **x**. Элементы, равные **x**, могут находиться в обеих частях.

Теперь элементы расположены так, что ни один элемент из первой группы при сортировке не окажется во второй и наоборот. Поэтому далее достаточно отсортировать отдельно каждую часть массива. Размер обеих частей чаще всего не совпадает. Лучше всего выбирать **x** так, чтобы в обеих частях было равное количество элементов. Такое значение **x** называется *медианой* массива. Однако для того, чтобы найти медиану, надо сначала отсортировать массив, то есть заранее решить ту самую задачу, которую мы собираемся решить этим способом. Поэтому обычно в качестве **x** выбирают средний элемент массива.

Будем просматривать массив слева до тех пор, пока не обнаружим элемент, который больше **x** (и, следовательно, должен стоять справа от **x**), потом справа пока не обнаружим элемент меньше **x** (он должен стоять слева от **x**). Теперь поменяем местами эти два элемента и продолжим просмотр до тех пор, пока два «просмотра» не встретятся где-то в середине массива. В результате массив окажется разбитым на 2 части: левую со значениями меньшими или равными **x**, и правую со значениями большими или равными **x**. Затем такая же процедура применяется рекурсивно к обеим частям массива до тех пор, пока в каждой части не останется один элемент (и таким образом, массив будет отсортирован).

Так как функция будет вызываться рекурсивно для части массива, в нее кроме указателя на массив нужно передать индекс начального элемента (from), с которого нужно начинать сортировку этой части массива, и индекс конечного элемента (to) до которого нужно продолжать сортировку этой части массива.

```
void QuickSort (int *A, int from, int to)
{
  int x, i, j;
  if (from >= to)   // условие окончания рекурсии
    return;
  i = from; // рассматриваем элементы с A[from] до A[to]
  j = to;
  x = A[(from+to)/2]; // выбрали средний элемент
  while ( i <= j )
  {
    while (A[i] < x) i++; // ищем пару для перестановки
    while (A[j] > x) j--;
    if (i <= j)
      Swap(&A[i++], &A[j--]);
  }
  QuickSort (A, from, j); // сортируем левую часть
  QuickSort (A, i, to);   // сортируем правую часть
}
```

Пример использования:

```
int *Arr;
int size;
// . . . . Здесь нужно задать размер массива size
```

```
Arr = new int[size]; // Выделяем память под size чисел
// . . . . Заполняем случайными числами
ViewArr(Arr, size);    // Отображаем массива до сортировки
QuickSort (Arr, 0, size - 1); // Сортируем
ViewArr(Arr, size);     // Отображаем массива после сортировки
```

Данную функцию сортировки также легко переделать для нестандартных типов, по тому же принципу, что и в методе выбора минимального покрытия. Опорный элемент должен быть такого же типа, что и сортируемые элементы, кроме того, нужны две функции сравнения – на больше и на меньше.

Бинарный поиск

В предыдущих лабораторных работах рассматривался алгоритм линейного поиска заданного элемента в массиве. Но линейный поиск – неэффективен, так как в худшем случае для поиска нужного элемента придется пересмотреть все элементы массива.

Алгоритм двоичного (или бинарного) поиска гораздо эффективнее. Чем больше элементов в массиве, тем выгоднее использовать двоичный поиск, поскольку число операций при этом возрастает как логарифм от числа сравнений, то есть намного медленнее, чем увеличивается размер массива.

Бинарный поиск проводится в отсортированном массиве.

Для поиска элемента, равного x, в отсортированном массиве A размером N берем средний элемент массива и сравниваем его с x. Если x меньше этого значения, продолжаем поиск в первой половине массива, если больше – во второй.

Вновь берем средний элемент с выбранной половине и сравниваем с x. Процесс продолжается, пока не будет найден x или пока не станет пустым массив для поиска.

Функция поиска элемента x в массиве A, размером Size:

```
int BinarySearch(int* A, int Size, int x)
{
  int low = 0;            // нижняя граница поиска
  int high = Size - 1;    // верхняя граница поиска
  int m;

  while (low <= high)
  {
    m = (low + high) / 2; // середина
    if (A[m] == x)        // если нашли элемент массива, равный x,
      return m;           // возвращаем его индекс
    if (A[m] > x)     // если значение элемента массива > x,
      high = m - 1;   // сдвигаем верхнюю границу
    else              // если значение элемента массива < x,
      low = m + 1;    // сдвигаем нижнюю границу
  }
  return -1;  // если в массиве нет элемента со значением x
}
```

Аналогично можно проводить поиск заданной строки в массиве строк, нужной даты в массиве дат и т.д. Например, в функции поиска ромба с площадью s в массиве ромбов нужно сравнивать площадь текущего элемента массива с заданной площадью: RArea(A[m]) == s и RArea(A[m]) > s .

```
int BinarySearchR(Rhombus* A, int Size, int s)
{
  int low = 0;
  int high = Size - 1;
  int m;

  while (low <= high)
```

```
    {
      m = (low + high) / 2;
      if (RArea(A[m]) == s) // если площадь элемента A[m] равна s
        return m;
      if (RArea(A[m]) > s) // если площадь элемента A[m] > s
        high = m - 1;
      else               // если площадь элемента A[m] < s
        low = m + 1;
    }
    return -1;
  }
```

Аналогично можно написать функцию, которая ищет ромб не с конкретной площадью s, а с площадью, попадающей в некоторый диапазон от smin до smax. Отличие будет только в функциях сравнения:

`if (smin <= RArea(A[m]) && RArea(A[m]) <= smax)` – попали в диапазон

`if (RArea(A[m]) > smax)` – выше верхней границы

Заголовок функции будет таким:

```
int BinarySearchRect(Rect* A, int Size, int smin, int smax)
{ . . . }
```

Исследование эффективности алгоритмов

Для исследования эффективности алгоритмов сортировки нужно сравнить время выполнения алгоритма при достаточно больших размерах массивов.

Заполнять массивы нужно случайными числами. Для этого можно использовать генератор случайных чисел из стандартной библиотеки функций (как было описано в лабораторной работе 9):

Функцию rand() при каждом вызове генерирует случайное число (на самом деле псевдослучайное) в диапазоне от 0 до RAND_MAX (RAND_MAX = 32767).

При каждом запуске программы функция rand() начинает генерацию псевдослучайной последовательности с одного и того же числа. Чтобы избежать этого, нужно в начале программы один раз вызвать функцию srand (стартовая инициализация генератора псевдослучайной последовательности чисел), например так: srand(time(0)).

Чтобы изменить диапазон генерируемых чисел можно воспользоваться формулой:

```
(float)rand()/RAND_MAX * (NMAX - NMIN) + NMIN;
```

где NMAX и NMIN – константы, задающие максимальный и минимальный пределы диапазона генерации. Например, чтобы получить случайное число в диапазоне от 0 до 100 можно просто написать:

```
(float)rand()/RAND_MAX * 100;
```

а для диапазона от –1 до 1:

```
(float)rand()/RAND_MAX * 2 - 1;
```

Для работы с этими функциями нужно подключить заголовочные файлы

```
#include <cstdlib>
#include <time.h>
```

Случайными числами можно задавать и параметры структур, например, в массиве ромбов можно так задать расстояния и диагонали, чтобы они выбирались случайным образом из диапазона 5 - 105:

```
Rhombus *ArrR;
int size;
// . . . . Здесь нужно задать размер массива size
srand(time(0)); // стартовая инициализация генератора случ. чисел
```

```
ArrR = new Rhombus[size]; // Выделяем память под size структур
for (int i = 0; i < size; i++)
{
  RArr[i].x = (float)rand() / RAND_MAX * 100 + 5;
  RArr[i].y = (float)rand() / RAND_MAX * 100 + 5;
  RArr[i].d1 = (float)rand() / RAND_MAX * 100 + 5;
  RArr[i].d2 = (float)rand() / RAND_MAX * 100 + 5;
}
```

Для измерения времени выполнения операций можно использовать функцию `GetTickCount()`, которая возвращает число миллисекунд, которые истекли после того, как система была запущена. Функция считает время не по одной миллисекунде, а в соответствии с системным таймером, т.е. с шагом порядка 10-16 миллисекунд, поэтому маленькие времена с ее помощью измерить нельзя.

```
int told, tnew;
told = GetTickCount();
  // . . . Какие-то операции и т.д., время выполнения которых
  // . . . нужно измерить
tnew = GetTickCount();
printf("Время выполнения dt = %d ms\n", tnew - told);
```

Для использования этой функции нужно подключить заголовочный файл:

```
#include <Windows.h>
```

ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ

в начале программы ОБЯЗАТЕЛЬНО выводить:
ФИО, группа, номер лаб. работы, номер варианта.

Задание. Доработать программу для работы с массивами структур (лабораторная работа 19) функциями сортировки и двоичного поиска. **Программа должна содержать меню и позволять проводить повторные вычисления**.

Для получения максимального балла:

Написать функции сортировки для всех трех алгоритмов, описанных в работе, и сравнить их эффективность (время выполнения).

Варианты заданий

1. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:
- сортировку массива прямоугольников по периметру (по убыванию);
- двоичный поиск прямоугольника с периметром, попадающим в заданный диапазон, где диапазон задается с клавиатуры.

2. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:
- сортировку массива равносторонних треугольников по площади (по возрастанию);
- двоичный поиск треугольника с площадью, попадающей в заданный диапазон, где диапазон задается с клавиатуры.

3. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:
- сортировку массива квадратов по площади (по убыванию);
- двоичный поиск квадрата с площадью, попадающей в заданный диапазон, где диапазон задается с клавиатуры.

4. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:
- сортировку массива окружностей по длине окружности (по убыванию);
- двоичный поиск окружности с длиной, попадающей в заданный диапазон, где диапазон задается с клавиатуры.

5. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:
- сортировку массива прямоугольников по периметру (по возрастанию);
- двоичный поиск прямоугольника с периметром, попадающим в заданный диапазон, где диапазон задается с клавиатуры.

6. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:
- сортировку массива квадратов по площади (по возрастанию);
- двоичный поиск квадрата с площадью, попадающей в заданный диапазон, где диапазон задается с клавиатуры.

7. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:
- сортировку массива окружностей по площади (по убыванию);

- двоичный поиск окружности с площадью, попадающей в заданный диапазон, где диапазон задается с клавиатуры.

8. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:
- сортировку массива отрезков по длине (по убыванию);
- двоичный поиск отрезка с длиной, попадающей в заданный диапазон, где диапазон задается с клавиатуры.

9. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:
- сортировку массива отрезков по длине (по возрастанию);
- двоичный поиск отрезка с длиной, попадающей в заданный диапазон, где диапазон задается с клавиатуры.

10. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:
- сортировку массива равносторонних треугольников по периметру (по убыванию);
- двоичный поиск треугольника с периметром, попадающим в заданный диапазон, где диапазон задается с клавиатуры.

11. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:
- сортировку массива треугольников по периметру (по возрастанию);
- двоичный поиск треугольника с периметром, попадающим в заданный диапазон, где диапазон задается с клавиатуры.

12. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:
- сортировку массива прямоугольников по площади (по убыванию);
- двоичный поиск прямоугольника с площадью, попадающей в заданный диапазон, где диапазон задается с клавиатуры.

13. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:
- сортировку массива квадратов по периметру (по возрастанию);
- двоичный поиск квадрата с периметром, попадающим в заданный диапазон, где диапазон задается с клавиатуры.

14. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:
- сортировку массива окружностей по расстоянию их центров до начала координат (по возрастанию);
- двоичный поиск окружности с расстоянием от центра до начала координат, попадающим в заданный диапазон, где диапазон задается с клавиатуры.

15. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:
- сортировку массива ромбов по периметру (по возрастанию);
- двоичный поиск ромба с периметром, попадающим в заданный диапазон, где диапазон задается с клавиатуры.

16. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:

- сортировку массива окружностей по длине окружности (по возрастанию);
- двоичный поиск окружности с длиной, попадающей в заданный диапазон, где диапазон задается с клавиатуры.

17. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:

- сортировку массива квадратов по расстоянию их центров до начала координат (по возрастанию);
- двоичный поиск квадрата с расстоянием от центра до начала координат, попадающим в заданный диапазон, где диапазон задается с клавиатуры.

18. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:

- сортировку массива окружностей по расстоянию их центров до начала координат (по убыванию);
- двоичный поиск окружности с расстоянием от центра до начала координат, попадающим в заданный диапазон, где диапазон задается с клавиатуры.

19. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:

- сортировку массива точек по расстоянию до начала координат (по убыванию);
- двоичный поиск точки с расстоянием до начала координат, попадающим в заданный диапазон, где диапазон задается с клавиатуры.

20. Добавить в программу из лабораторной работы 19 следующие функции и соответствующие пункты меню:

- сортировку массива точек по расстоянию до начала координат (по возрастанию);
- двоичный поиск точки с расстоянием до начала координат, попадающим в заданный диапазон, где диапазон задается с клавиатуры.