**Лабораторная работа № 9**

**Выполнил: Лосев Данил ЕПИ-4-23**

|  |
| --- |
| #include <chrono> // Подключение библиотеки для работы со временем  #include <conio.h> // Библиотека для ввода-вывода в консоли  #include <cstddef> // Заголовочный файл для определения размера объектов  #include <cstdio> // Стандартный ввод-вывод на уровне C  #include <cstdlib> // Стандартные утилиты, включая случайные числа  #include <ctime> // Для генерации текущего времени  #include <iomanip> // Манипуляторы для форматирования вывода  #include <iostream> // Основная библиотека для ввода-вывода в C++  #include <synchapi.h> // Для работы с функциями синхронизации в Windows  #include <windows.h> // Основные функции Windows API  int PRINT\_MAX\_SIZE = 1000;  int TEST\_STEP\_SIZE = 50;  // Функция сортировки массива с использованием переданного алгоритма  // sort - указатель на функцию сортировки  // fArray - массив, который нужно отсортировать  // fSize - размер массива  // fTime - переменная для хранения времени выполнения сортировки  int \*ArraySort(int \*(\*sort)(int \*&, int), int \*&fArray, int fSize, long long &fTime);  // Функция для заполнения массива для лучшего случая (массив уже отсортирован)  void FillArrayBestCase(int \*&fArray, int fSize);  // Функция для заполнения массива для среднего случая (случайно перемешанный массив)  void FillArrayMiddleCase(int \*&fArray, int fSize);  // Функция для заполнения массива для худшего случая (массив отсортирован в обратном порядке)  void FillArrayBadCase(int \*&fArray, int fSize);  // Частичная реализация быстрой сортировки (рекурсивная сортировка подмассивов)  int \*PartQuickSort(int \*&fArray, int fSize, int fStart, int fEnd);  // Основная функция быстрой сортировки  int \*QuickSort(int \*&fArray, int fSize);  // Вставочная сортировка на подмассиве массива  int \*InsertionSort(int \*&fArray, int fSize, int fStart, int fEnd);  // Частичная реализация сортировки Шелла (рекурсивное разделение массива на интервалы)  int \*PartShellSort(int \*&fArray, int fSize, int fStart, int fEnd, int interval);  // Основная функция сортировки Шелла  int \*ShellSort(int \*&fArray, int fSize);  // Преобразование массива в кучу (используется в пирамидальной сортировке)  int \*ArrayToHeap(int \*&fArray, int fSize, int fIndex);  // Пирамидальная сортировка массива  int \*PyramidSort(int \*&fArray, int fSize);  // Шейкерная сортировка массива  int \*ShakerSort(int \*&fArray, int fSize);  // Функция для вывода массива на экран  void PrintArray(int \*fArray, int fSize);  // Автоматический тест сортировки массива с увеличивающимся размером  void arrayAutoTest(int fMaxSize);  // Автоматический тест для сортировки массива с заданным размером и алгоритмом заполнения  long long autoSortTestForArray(int \*(\*sort)(int \*&, int), void (\*fill)(int \*&, int), int fSize);  int main()  {  int programm = 1; // Переменная для контроля основного цикла программы  int choose = 1; // Выбор пользователем типа заполнения и сортировки  int size = 1; // Размер массива  long long sortTime = 0; // Переменная для хранения времени сортировки  while (programm != 0) // Основной цикл программы, завершается при вводе 0  {  std::system("cls"); // Очистка экрана перед вводом размера  std::cout << "Enter the size of array: ";  std::cin >> size; // Ввод размера массива  Sleep(500); // Пауза для плавности работы интерфейса  std::system("cls");  int \*array = new int[size]{}; // Создание массива с заданным размером  std::system("cls");  std::cout << "Enter the type of filling array\n";  std::cout << " 1) Best case\n"; // Лучший случай заполнения  std::cout << " 2) Middle case\n"; // Средний случай заполнения  std::cout << " 3) Bad case\n"; // Худший случай заполнения  std::cout << "\n> ";  std::cin >> choose;  switch (choose) // Заполнение массива на основе выбранного случая  {  case 1:  FillArrayBestCase(array, size);  break;  case 2:  FillArrayMiddleCase(array, size);  break;  case 3:  FillArrayBadCase(array, size);  break;  }  if (size <= PRINT\_MAX\_SIZE) // Проверка на размер для вывода массива  {  std::cout << "Array: ";  PrintArray(array, size); // Вывод элементов массива  }  getch(); // Ожидание нажатия клавиши для продолжения  std::system("cls");  std::cout << "Enter the sort for array:\n";  std::cout << " 1) Quick Sort\n";  std::cout << " 2) Shell Sort\n";  std::cout << " 3) Pyramid Sort\n";  std::cout << " 4) Shaker Sort\n> ";  std::cin >> choose;  switch (choose) // Сортировка массива выбранным алгоритмом  {  case 1:  array = ArraySort(QuickSort, array, size, sortTime);  break;  case 2:  array = ArraySort(ShellSort, array, size, sortTime);  break;  case 3:  array = ArraySort(PyramidSort, array, size, sortTime);  break;  case 4:  array = ArraySort(ShakerSort, array, size, sortTime);  break;  }  if (size <= PRINT\_MAX\_SIZE) // Вывод отсортированного массива  {  std::cout << "Sorted array: ";  PrintArray(array, size);  }  std::cout << "Sort time: " << sortTime << std::endl;  getch();  std::system("cls");  delete[] array; // Удаление динамического массива после использования  array = NULL;  int choose1 = 0; // Переменная для автотеста  std::cout << "Do you want to do auto test? (1=yes 0=no) : ";  std::cin >> choose1;  if (choose1 == 1)  {  std::system("cls");  int sizeOfTest; // Размер данных для автотеста  std::cout << "Enter the size of test: ";  std::cin >> sizeOfTest;  arrayAutoTest(sizeOfTest); // Запуск автотеста для массива  getch();  }  std::system("cls");  std::cout << "Go to start? (1=yes 0=no)\n> ";  std::cin >> programm; // Проверка на повтор программы  }  return 0;  }  int \*ArraySort(int \*(\*sort)(int \*&, int), int \*&fArray, int fSize, long long &fTime)  {  auto start = std::chrono::high\_resolution\_clock::now(); // Начало замера времени  int \*Temp = sort(fArray, fSize); // Вызов функции сортировки  auto end = std::chrono::high\_resolution\_clock::now(); // Конец замера времени  fTime = std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count(); // Вычисление времени выполнения  return Temp; // Возвращение отсортированного массива  }  void FillArrayBestCase(int \*&fArray, int fSize)  {  for (int i = 0; i < fSize; i++) // Заполнение массива отсортированными значениями  {  fArray[i] = i + 1; // Присвоение значения, увеличивающегося на 1  }  }  void FillArrayMiddleCase(int \*&fArray, int fSize)  {  for (int i = 0; i < fSize; i++) // Заполнение массива случайными значениями  {  fArray[i] = (rand() % fSize) + 1; // Присвоение случайного значения  }  }  void FillArrayBadCase(int \*&fArray, int fSize)  {  for (int i = 0; i < fSize; i++) // Заполнение массива значениями в порядке убывания  {  fArray[i] = fSize - i; // Присвоение значения, уменьшающегося на 1  }  }  int \*PartQuickSort(int \*&fArray, int fSize, int fStart, int fEnd)  {  if (fStart >= fEnd)  {  return fArray; // Условие выхода, если массив из одного элемента или пуст  }  int fSupportingElement = fArray[fEnd]; // Опорный элемент - последний элемент подмассива  int fIndexOfSupportingElement = fStart; // Индекс разделения  // Перестановка элементов  for (int i = fStart; i < fEnd; i++)  {  if (fArray[i] <= fSupportingElement)  {  std::swap(fArray[i], fArray[fIndexOfSupportingElement]);  fIndexOfSupportingElement++;  }  }  std::swap(fArray[fIndexOfSupportingElement], fArray[fEnd]); // Размещение опорного элемента  // Рекурсивные вызовы для левой и правой части массива  PartQuickSort(fArray, fSize, fStart, fIndexOfSupportingElement - 1);  PartQuickSort(fArray, fSize, fIndexOfSupportingElement + 1, fEnd);  return fArray;  }  int \*QuickSort(int \*&fArray, int fSize)  {  fArray = PartQuickSort(fArray, fSize, 0, fSize - 1);  return fArray;  }  int \*InsertionSort(int \*&fArray, int fSize, int fStart, int fEnd)  {  for (int i = fStart; i < fEnd;  i++) // Начинаем со второго элемента, так как первый элемент считается отсортированным  {  int curInt = fArray[i]; // Текущий элемент для вставки  int j; // Индекс для перемещения элементов  // Сдвигаем элементы, которые больше текущего элемента, вправо  for (j = i - 1; j >= 0 && fArray[j] > curInt; j--)  {  fArray[j + 1] = fArray[j]; // Сдвиг элемента вправо  }  fArray[j + 1] = curInt; // Вставка текущего элемента на его место  }  return fArray; // Возврат отсортированного массива  }  int \*PartShellSort(int \*&fArray, int fSize, int fStart, int fEnd, int interval)  {  interval = interval / 2; // Уменьшаем интервал на половину  if (interval <= 0)  {  return fArray; // Завершаем, если интервал стал меньше или равен нулю  }  // Сортировка подмассивов с текущим интервалом  fArray = InsertionSort(fArray, fSize, fStart, interval);  fArray = InsertionSort(fArray, fSize, interval, fEnd);  // Рекурсивные вызовы с меньшим интервалом для сортировки  fArray = PartShellSort(fArray, fSize, fStart, interval, interval);  fArray = PartShellSort(fArray, fSize, interval, fEnd, interval);  return fArray; // Возвращаем отсортированный массив  }  int \*ShellSort(int \*&fArray, int fSize)  {  // Вызов функции PartShellSort с начальной группой элементов  // Параметры: исходный массив, размер массива, начальный индекс (0), конечный индекс (fSize), и изначальный шаг,  // равный размеру массива  fArray = PartShellSort(fArray, fSize, 0, fSize, fSize);  return fArray; // Возвращаем отсортированный массив  }  int \*ArrayToHeap(int \*&fArray, int fSize, int fIndex)  {  int temp; // Временная переменная для обмена элементов  int fTreeRoot = fIndex; // Инициализация текущего корня поддерева  int fLeftTree = 2 \* fIndex + 1; // Левый потомок узла  int fRightTree = 2 \* fIndex + 2; // Правый потомок узла  // Проверка: является ли левый потомок больше текущего корня поддерева  if (fLeftTree < fSize && fArray[fLeftTree] > fArray[fTreeRoot])  {  fTreeRoot = fLeftTree; // Устанавливаем левый потомок в качестве нового корня поддерева  }  // Проверка: является ли правый потомок больше текущего корня поддерева  if (fRightTree < fSize && fArray[fRightTree] > fArray[fTreeRoot])  {  fTreeRoot = fRightTree; // Устанавливаем правый потомок в качестве нового корня поддерева  }  // Если корень изменился, выполняется обмен и рекурсивный вызов для перестройки поддерева  if (fTreeRoot != fIndex)  {  temp = fArray[fIndex]; // Сохраняем значение текущего узла  fArray[fIndex] = fArray[fTreeRoot]; // Перемещаем больший потомок в текущий корень  fArray[fTreeRoot] = temp; // Возвращаем сохраненное значение в потомок  // Рекурсивно перестраиваем измененное поддерево для поддержки свойств кучи  fArray = ArrayToHeap(fArray, fSize, fTreeRoot);  }  return fArray; // Возвращаем массив после преобразования  }  int \*PyramidSort(int \*&fArray, int fSize)  {  // Построение начальной кучи: формируем кучу из массива, начиная с первых непустых поддеревьев  for (int i = fSize / 2 - 1; i >= 0; i--)  {  ArrayToHeap(fArray, fSize, i); // Преобразование текущего поддерева в кучу  }  int temp; // Временная переменная для обмена корневого элемента  // Основной цикл сортировки: уменьшаем размер кучи и перемещаем наибольший элемент в конец  for (int i = fSize - 1; i >= 0; i--)  {  temp = fArray[0]; // Сохраняем корневой элемент (максимум в текущей куче)  fArray[0] = fArray[i]; // Перемещаем последний элемент в корень  fArray[i] = temp; // Сохраняем максимальный элемент в правильной позиции  // Преобразуем оставшуюся часть массива в кучу после удаления текущего корня  ArrayToHeap(fArray, i, 0);  }  return fArray; // Возвращаем отсортированный массив  }  int \*ShakerSort(int \*&fArray, int fSize)  {  int LeftSide = 1, RightSide = fSize - 1, temp; // Инициализация границ сортировки и временной переменной для обмена  bool sorted = true; // Флаг для отслеживания отсортированности массива  do  {  sorted = true; // Предполагается, что массив отсортирован, пока не будет найдено несоответствие  // Проход слева направо  for (int i = LeftSide; i <= RightSide; i++)  {  if (fArray[i - 1] > fArray[i]) // Если текущий элемент больше следующего, требуется обмен  {  temp = fArray[i - 1]; // Сохраняем значение текущего элемента  fArray[i - 1] = fArray[i]; // Перемещаем следующий элемент в текущее место  fArray[i] = temp; // Восстанавливаем сохраненное значение в следующем элементе  sorted = false; // Если обмен произведён, массив ещё не отсортирован  }  }  RightSide--; // Сокращаем правую границу, так как крайний правый элемент отсортирован  // Проход справа налево  for (int i = RightSide; i >= LeftSide; i--)  {  if (fArray[i] < fArray[i - 1]) // Если текущий элемент меньше предыдущего, выполняется обмен  {  temp = fArray[i]; // Сохраняем значение текущего элемента  fArray[i] = fArray[i - 1]; // Перемещаем предыдущий элемент на текущую позицию  fArray[i - 1] = temp; // Восстанавливаем сохраненное значение на предыдущей позиции  sorted = false; // Обмен показывает, что массив ещё не отсортирован  }  }  LeftSide++; // Увеличиваем левую границу, так как крайний левый элемент отсортирован  } while (sorted == false); // Повторяем, пока не будет прохода без обменов  return fArray; // Возвращаем отсортированный массив  }  void PrintArray(int \*fArray, int fSize)  {  if (fArray == 0) // Проверка на пустой массив  {  return; // Завершение функции, если массив пуст  }  for (int i = 0; i < fSize; i++) // Цикл для вывода элементов массива  {  std::cout << fArray[i] << ' '; // Вывод значения элемента массива  }  std::cout << std::endl; // Переход на новую строку после вывода массива  }  void arrayAutoTest(int fMaxSize)  {  std::cout << "\n Array Time Auto Test\n"; // Заголовок теста для массивов  const int width = 12; // Ширина столбцов вывода  std::cout << std::string(width \* 15 + 8, '-') << std::endl; // Отделитель  std::cout << std::setw(width / 2) << ' ' << " | " << std::setw(width) << ' ' << " " << std::setw(width \* 2)  << "Best case " << " " << std::setw(width) << ' ' << " | " << std::setw(width) << ' ' << " "  << std::setw(width \* 2) << "Average case " << " " << std::setw(width) << ' ' << " | "  << std::setw(width) << ' ' << " " << std::setw(width \* 2) << "Worst case " << " "  << std::setw(width) << ' ' << " | " << std::endl;  // Заголовки типов сортировки  std::cout << std::setw(width / 2) << "Count" << " | " << std::setw(width) << "Quick" << " | " << std::setw(width)  << "Shell" << " | " << std::setw(width) << "Pyramid" << " | " << std::setw(width) << "Shaker" << " | "  << std::setw(width) << "Quick" << " | " << std::setw(width) << "Shell" << " | " << std::setw(width)  << "Pyramid" << " | " << std::setw(width) << "Shaker" << " | " << std::setw(width) << "Quick" << " | "  << std::setw(width) << "Shell" << " | " << std::setw(width) << "Pyramid" << " | " << std::setw(width)  << "Shaker" << " | " << std::endl;  // Отделитель  std::cout << std::string(width \* 15 + 8, '-') << std::endl;  for (int i = 100; i <= fMaxSize; i += TEST\_STEP\_SIZE) // Цикл по размерам массива  {  int size = i; // Установка размера массива  std::cout << std::setw(width / 2) << i << " | " << std::setw(width)  << autoSortTestForArray(QuickSort, FillArrayBestCase, size) << " | " << std::setw(width)  << autoSortTestForArray(ShellSort, FillArrayBestCase, size) << " | " << std::setw(width)  << autoSortTestForArray(PyramidSort, FillArrayBestCase, size) << " | " << std::setw(width)  << autoSortTestForArray(ShakerSort, FillArrayBestCase, size) << " | " << std::setw(width)  << autoSortTestForArray(QuickSort, FillArrayMiddleCase, size) << " | " << std::setw(width)  << autoSortTestForArray(ShellSort, FillArrayMiddleCase, size) << " | " << std::setw(width)  << autoSortTestForArray(PyramidSort, FillArrayMiddleCase, size) << " | " << std::setw(width)  << autoSortTestForArray(ShakerSort, FillArrayMiddleCase, size) << " | " << std::setw(width)  << autoSortTestForArray(QuickSort, FillArrayBadCase, size) << " | " << std::setw(width)  << autoSortTestForArray(ShellSort, FillArrayBadCase, size) << " | " << std::setw(width)  << autoSortTestForArray(PyramidSort, FillArrayBadCase, size) << " | " << std::setw(width)  << autoSortTestForArray(ShakerSort, FillArrayBadCase, size) << " | " << std::endl;  }  std::cout << std::string(width \* 15 + 8, '-') << std::endl; // Завершение теста  }  long long autoSortTestForArray(int \*(\*sort)(int \*&, int), void (\*fill)(int \*&, int), int fSize)  {  int \*testArray = new int[fSize]{}; // Создание массива заданного размера  fill(testArray, fSize); // Заполнение массива  long long sortTime = 0; // Переменная для времени сортировки  testArray = ArraySort(sort, testArray, fSize, sortTime); // Сортировка массива  delete[] testArray; // Освобождение памяти  testArray = NULL; // Обнуление указателя  return sortTime;  } |

**Таблицы и графики**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Кол-во | Хороший случай | | | | Средний случай | | | | Плохой случай | | | |
| Быстрая | Шелла | Пиромидальная | Шейкерная | Быстрая | Шелла | Пиромидальная | Шейкерная | Быстрая | Шелла | Пиромидальная | Шейкерная |
| 100 | 45 | 7 | 11 | 0 | 8 | 14 | 12 | 28 | 28 | 20 | 10 | 25 |
| 150 | 99 | 12 | 17 | 0 | 14 | 29 | 18 | 59 | 61 | 46 | 15 | 56 |
| 200 | 188 | 15 | 24 | 1 | 20 | 47 | 27 | 93 | 108 | 70 | 22 | 99 |
| 250 | 288 | 19 | 31 | 1 | 26 | 69 | 34 | 163 | 264 | 138 | 29 | 188 |
| 300 | 404 | 24 | 56 | 1 | 31 | 97 | 42 | 205 | 243 | 258 | 36 | 335 |
| 350 | 539 | 32 | 45 | 1 | 35 | 130 | 51 | 275 | 327 | 248 | 42 | 385 |
| 400 | 726 | 38 | 52 | 2 | 44 | 160 | 58 | 361 | 427 | 315 | 48 | 426 |
| 450 | 895 | 42 | 59 | 2 | 50 | 192 | 67 | 515 | 563 | 450 | 57 | 531 |
| 500 | 1323 | 45 | 66 | 2 | 57 | 230 | 77 | 535 | 662 | 503 | 63 | 714 |
| 550 | 1347 | 65 | 74 | 3 | 63 | 284 | 83 | 644 | 801 | 548 | 70 | 774 |
| 600 | 1592 | 69 | 82 | 3 | 71 | 337 | 94 | 861 | 1101 | 644 | 78 | 917 |
| 650 | 1884 | 75 | 89 | 3 | 76 | 361 | 103 | 834 | 1117 | 733 | 86 | 1081 |
| 700 | 2476 | 80 | 100 | 4 | 80 | 421 | 112 | 978 | 1295 | 834 | 92 | 1233 |
| 750 | 2701 | 84 | 105 | 4 | 94 | 465 | 124 | 1349 | 1598 | 952 | 100 | 1431 |
| 800 | 2794 | 91 | 113 | 4 | 98 | 526 | 130 | 1349 | 1768 | 1060 | 103 | 1663 |
| 850 | 3347 | 92 | 120 | 4 | 99 | 614 | 137 | 1371 | 1964 | 1182 | 115 | 1823 |
| 900 | 4996 | 153 | 128 | 4 | 106 | 663 | 149 | 1603 | 2134 | 1310 | 123 | 2190 |
| 950 | 5363 | 166 | 135 | 6 | 140 | 786 | 161 | 1768 | 2379 | 1512 | 131 | 2240 |
| 1000 | 5645 | 168 | 171 | 7 | 162 | 812 | 169 | 1828 | 2710 | 1586 | 138 | 2477 |

**Блок схемы**

|  |  |
| --- | --- |
| int \*ArraySort(int \*(\*sort)(int \*&, int), int \*&fArray, int fSize, long long &fTime)  {  auto start = chrono::high\_resolution\_clock::now();  int \*Temp = sort(fArray, fSize);  auto end = chrono::high\_resolution\_clock::now();  fTime = chrono::duration\_cast<std::chrono::microseconds>(end - start).count();  return Temp;  } |  |
| int \*ArrayToHeap(int \*&fArray, int fSize, int fIndex)  {  int temp;  int fTreeRoot = fIndex;  int fLeftTree = 2 \* fIndex + 1;  int fRightTree = 2 \* fIndex + 2;  if (fLeftTree < fSize && fArray[fLeftTree] > fArray[fTreeRoot])  {  fTreeRoot = fLeftTree;  }  if (fRightTree < fSize && fArray[fRightTree] > fArray[fTreeRoot])  {  fTreeRoot = fRightTree;  }  if (fTreeRoot != fIndex)  {  temp = fArray[fIndex];  fArray[fIndex] = fArray[fTreeRoot];  fArray[fTreeRoot] = temp;  fArray = ArrayToHeap(fArray, fSize, fTreeRoot);  }  return fArray;  } |  |
| void FillArrayBadCase(int \*&fArray, int fSize)  {  for (int i = 0; i < fSize; i++)  {  fArray[i] = fSize - i;  }  } |  |
| void FillArrayBestCase(int \*&fArray, int fSize)  {  for (int i = 0; i < fSize; i++)  {  fArray[i] = i + 1;  }  } |  |
| void FillArrayMiddleCase(int \*&fArray, int fSize)  {  for (int i = 0; i < fSize; i++)  {  fArray[i] = (rand() % fSize) + 1;  }  } |  |
| int \*InsertionSort(int \*&fArray, int fSize, int fStart, int fEnd)  {  for (int i = fStart; i < fEnd; i++)  {  int curInt = fArray[i];  int j;  for (j = i - 1; j >= 0 && fArray[j] > curInt; j--)  {  fArray[j + 1] = fArray[j];  }  fArray[j + 1] = curInt;  }  return fArray;  } |  |
| int \*PartQuickSort(int \*&fArray, int fSize, int fStart, int fEnd)  {  if (fStart >= fEnd)  {  return fArray;  }  int fSupportingElement = fArray[fEnd];  int fIndexOfSupportingElement = fStart;    for (int i = fStart; i < fEnd; i++)  {  if (fArray[i] <= fSupportingElement)  {  swap(fArray[i], fArray[fIndexOfSupportingElement]);  fIndexOfSupportingElement++;  }  }  swap(fArray[fIndexOfSupportingElement], fArray[fEnd]);  PartQuickSort(fArray, fSize, fStart, fIndexOfSupportingElement - 1);  PartQuickSort(fArray, fSize, fIndexOfSupportingElement + 1, fEnd);  return fArray;  } |  |
| int \*PartShellSort(int \*&fArray, int fSize, int fStart, int fEnd, int interval)  {  interval = interval / 2;  if (interval <= 0)  {  return fArray;  }  fArray = InsertionSort(fArray, fSize, fStart, interval);  fArray = InsertionSort(fArray, fSize, interval, fEnd);  fArray = PartShellSort(fArray, fSize, fStart, interval, interval);  fArray = PartShellSort(fArray, fSize, interval, fEnd, interval);  return fArray;  } |  |
| void PrintArray(int \*fArray, int fSize)  {  if (fArray == 0)  {  return;  }  for (int i = 0; i < fSize; i++)  {  cout << fArray[i] << ' ';  }  cout << std::endl;  } |  |
| int \*PyramidSort(int \*&fArray, int fSize)  {  for (int i = fSize / 2 - 1; i >= 0; i--)  {  ArrayToHeap(fArray, fSize, i);  }  int temp;  for (int i = fSize - 1; i >= 0; i--)  {  temp = fArray[0];  fArray[0] = fArray[i];  fArray[i] = temp;  ArrayToHeap(fArray, i, 0);  }  return fArray;  } |  |
| int \*QuickSort(int \*&fArray, int fSize)  {  fArray = PartQuickSort(fArray, fSize, 0, fSize - 1);  return fArray;  } |  |
| int \*ShakerSort(int \*&fArray, int fSize)  {  int LeftSide = 1, RightSide = fSize - 1, temp;  bool sorted = true;  do  {  sorted = true;  for (int i = LeftSide; i <= RightSide; i++)  {  if (fArray[i - 1] > fArray[i])  {  temp = fArray[i - 1];  fArray[i - 1] = fArray[i];  fArray[i] = temp;  sorted = false;  }  }  RightSide--;  for (int i = RightSide; i >= LeftSide; i--)  {  if (fArray[i] < fArray[i - 1])  {  temp = fArray[i];  fArray[i] = fArray[i - 1];  fArray[i - 1] = temp;  sorted = false;  }  }  LeftSide++;  } while (sorted == false);  return fArray;  } |  |
| int \*ShellSort(int \*&fArray, int fSize)  {  fArray = PartShellSort(fArray, fSize, 0, fSize, fSize);  return fArray;  } |  |
| void arrayAutoTest(int fMaxSize)  {  cout << "\n Array Time Auto Test\n";  const int width = 12;  cout << string(width \* 15 + 8, '-') << endl;  cout << setw(width / 2) << ' ' << " | " << setw(width) << ' ' << " " << setw(width \* 2)  << "Best case " << " " << setw(width) << ' ' << " | " << setw(width) << ' ' << " "  << setw(width \* 2) << "Average case " << " " << setw(width) << ' ' << " | "  << setw(width) << ' ' << " " << setw(width \* 2) << "Worst case " << " "  << setw(width) << ' ' << " | " << endl;  cout << setw(width / 2) << "Count" << " | " << setw(width) << "Quick" << " | " << setw(width)  << "Shell" << " | " << setw(width) << "Pyramid" << " | " << setw(width) << "Shaker" << " | "  << setw(width) << "Quick" << " | " << setw(width) << "Shell" << " | " << setw(width)  << "Pyramid" << " | " << setw(width) << "Shaker" << " | " << setw(width) << "Quick" << " | "  << setw(width) << "Shell" << " | " << setw(width) << "Pyramid" << " | " << setw(width)  << "Shaker" << " | " << endl;  cout << string(width \* 15 + 8, '-') << endl;  for (int i = 100; i <= fMaxSize; i += TEST\_STEP\_SIZE)  {  int size = i;  cout << setw(width / 2) << i << " | " << setw(width)  << autoSortTestForArray(QuickSort, FillArrayBestCase, size) << " | " << setw(width)  << autoSortTestForArray(ShellSort, FillArrayBestCase, size) << " | " << setw(width)  << autoSortTestForArray(PyramidSort, FillArrayBestCase, size) << " | " << setw(width)  << autoSortTestForArray(ShakerSort, FillArrayBestCase, size) << " | " << setw(width)  << autoSortTestForArray(QuickSort, FillArrayMiddleCase, size) << " | " << setw(width)  << autoSortTestForArray(ShellSort, FillArrayMiddleCase, size) << " | " << setw(width)  << autoSortTestForArray(PyramidSort, FillArrayMiddleCase, size) << " | " << setw(width)  << autoSortTestForArray(ShakerSort, FillArrayMiddleCase, size) << " | " << setw(width)  << autoSortTestForArray(QuickSort, FillArrayBadCase, size) << " | " << setw(width)  << autoSortTestForArray(ShellSort, FillArrayBadCase, size) << " | " << setw(width)  << autoSortTestForArray(PyramidSort, FillArrayBadCase, size) << " | " << setw(width)  << autoSortTestForArray(ShakerSort, FillArrayBadCase, size) << " | " << endl;  }  cout << string(width \* 15 + 8, '-') << endl;  } |  |
| long long autoSortTestForArray(int \*(\*sort)(int \*&, int), void (\*fill)(int \*&, int), int fSize)  {  int \*testArray = new int[fSize]{};  fill(testArray, fSize);  long long sortTime = 0;  testArray = ArraySort(sort, testArray, fSize, sortTime);  delete[] testArray;  testArray = NULL;  return sortTime;  } |  |

**Ответы на вопросы**

1. **Назовите алгоритмы улучшенных сортировок?**

 **Быстрая сортировка (Quick Sort)**

 **Сортировка Шелла (Shell Sort)**

 **Пирамидальная сортировка (сортировка кучей, Heap Sort)**

 **Шейкерная сортировка (Shaker Sort)**

1. **Опишите алгоритмы их работы.**

 **Быстрая сортировка (Quick Sort)**: Быстрая сортировка — это алгоритм «разделяй и властвуй». Он выбирает опорный элемент (pivot) и разделяет массив на две части: элементы, меньшие опорного, и элементы, большие опорного. Эти части рекурсивно сортируются. Время работы в худшем случае O(n²), но в среднем — O(n log n).

 **Сортировка Шелла (Shell Sort)**: Это улучшенная версия сортировки вставками. Идея заключается в том, что элементы массива сравниваются не только соседними, а с некоторым интервалом, который постепенно уменьшается. Это позволяет более эффективно упорядочить массив на первых этапах. Время работы зависит от выбора интервалов, но в худшем случае может быть O(n²), хотя для некоторых последовательностей интервалов это время может быть значительно лучше.

 **Пирамидальная сортировка (Heap Sort)**: Этот алгоритм использует структуру данных "кучу", которая представляет собой частично упорядоченное дерево. Он сначала строит кучу из исходного массива, а затем поочередно извлекает наибольший (или наименьший) элемент и восстанавливает кучу. Время работы алгоритма — O(n log n) в худшем случае.

 **Шейкерная сортировка (Shaker Sort)**: Это двунаправленная версия сортировки пузырьком. Она проходит по массиву сначала слева направо, затем справа налево, попеременно. Это улучшение позволяет быстрее перемещать элементы на их конечные позиции. Время работы аналогично пузырьковой сортировке и составляет O(n²) в худшем случае.

1. **Сравните их работу на массивах со случайно сгенерированными числами и на частично упорядоченных.**

 **Быстрая сортировка**:

* На случайных массивах быстрая сортировка работает быстро, но в худшем случае (если опорный элемент всегда наименьший или наибольший) её сложность может быть O(n²).
* На частично упорядоченных массивах алгоритм будет работать быстрее, если выбран хороший опорный элемент (например, медиана).

 **Сортировка Шелла**:

* На случайных массивах сортировка Шелла может быть эффективной в зависимости от выбранной последовательности интервалов, хотя её производительность может варьироваться.
* На частично упорядоченных массивах сортировка Шелла показывает хорошие результаты, так как она работает быстрее, чем сортировка вставками.

 **Пирамидальная сортировка**:

* На случайных массивах пирамидальная сортировка работает с постоянной сложностью O(n log n) независимо от порядка элементов.
* На частично упорядоченных массивах её производительность также остаётся стабильной, так как она не зависит от исходного порядка элементов.

 **Шейкерная сортировка**:

* На случайных массивах она будет работать почти так же, как и пузырьковая сортировка, с той разницей, что процесс двунаправленный.
* На частично упорядоченных массивах шейкерная сортировка может быть немного быстрее, так как элементы уже находятся в частичном порядке и могут быстрее попасть на свои места.

1. **Сравните их работу на массивах с разной размерностью.**

 **Быстрая сортировка**:

* Для маленьких массивов её производительность схожа с другими алгоритмами с O(n log n), но для очень больших массивов она может занять больше времени в случае худшего выбора опорных элементов.
* В среднем работает быстро, особенно если массив очень большой, благодаря хорошему разделению массива.

 **Сортировка Шелла**:

* Для малых массивов может быть быстрее сортировки вставками. Для больших массивов её эффективность сильно зависит от выбора последовательности интервалов, что может привести к различной производительности.

 **Пирамидальная сортировка**:

* Для всех массивов, независимо от их размера, алгоритм работает за O(n log n), что делает его стабильным для больших данных.
* Может быть менее эффективной по сравнению с быстрой сортировкой, если сортировка на кучах строится неэффективно.

 **Шейкерная сортировка**:

* Для малых массивов она может быть быстрой, но для больших массивов её сложность O(n²) делает её неэффективной по сравнению с другими алгоритмами.

1. **Сделайте выводы об особенностях работы сортировок: какие сортировки лучшие и при каких условиях?**

 **Быстрая сортировка**: Лучше всего работает на случайных или сбалансированных массивах, когда можно эффективно выбрать опорный элемент. Однако для массивов с большим количеством повторяющихся элементов или в худших случаях её сложность может возрасти до O(n²).

 **Сортировка Шелла**: Это хорошая сортировка для средних массивов с небольшим размером и для частично упорядоченных массивов. Однако её эффективность сильно зависит от выбранных интервалов, и в худшем случае она может работать как сортировка вставками.

 **Пирамидальная сортировка**: Идеальна для случаев, когда важно избежать худших случаев и получить гарантированное время O(n log n). Она не зависит от исходного порядка данных и всегда будет работать с этим временем.

 **Шейкерная сортировка**: В большинстве случаев уступает другим алгоритмам. Она может быть полезна для маленьких или частично упорядоченных массивов, но её эффективность для больших массивов крайне низкая.