Лабораторная работа №3
по курсу «Структуры и алгоритмы обработки данных»

АЛГОРИТМЫ СОРТИРОВКИ МАССИВОВ

1. Цель работы

В данной работе рассматриваются наиболее известные алгоритмы внутренней сортировки данных в применении к числовым массивам. Выполнение работы преследует следующие цели:

- изучение основных идей, алгоритмов, приемов программирования, связанных с классической задачей сортировки массивов;
- получение навыков сравнения алгоритмов по эффективности.

2. Основные сведения

2.1. Задача внутренней сортировки

Сортировкой называется перестановка записей таблицы (или в частном случае – элементов массива) в соответствии с некоторым заданным отношением порядка (по алфавиту, по возрастанию числового значения поля и т.п.). Сортировка, во-первых, позволяет просматривать, обрабатывать, выдавать записи таблицы в нужной последовательности, и, во-вторых, дает возможность выполнять быстрый (бинарный) поиск данных в таблице.

Различают алгоритмы внутренней сортировки (когда вся сортируемая таблица может быть полностью размещена в оперативной памяти) и алгоритмы внешней сортировки (когда таблица велика и должна считываться в память из файла по частям).

Ввиду того, что сортировка – одна из наиболее часто используемых процедур обработки данных, к эффективности алгоритмов сортировки предъявляются очень высокие требования.

Производительность алгоритмов может оцениваться как с помощью теоретически полученных оценок, так и эмпирически, путем экспериментального сравнения алгоритмов.

Теоретические оценки производительности чаще всего выражаются в виде оценок скорости роста времени сортировки в зависимости от числа записей сортируемой таблицы «с точностью до **O**-большого». Говорят, например: «Данный алгоритм имеет оценку $O(N^2)$». Это означает, что при увеличении числа записей **N**, к примеру, в 10 раз, время работы алгоритма возрастает примерно в $10^2 = 100$ раз. При этом следует различать оценки максимального времени (т.е. для случая самых неудачных для этого алгоритма данных) и среднего времени (т.е. математического ожидания времени сортировки для случайной таблицы).

Оценки «с точностью до **O**-большого» имеют смысл для сравнения поведения алгоритмов при больших значениях **N**. Практически для больших **N** считаются приемлемыми алгоритмы с оценкой $O(N \cdot \log(N))$ или близкой к этому значению, в то время как алгоритмы сортировки с оценкой $O(N^2)$ считаются неприемлемо медленными. В то же время при небольших размерах таблицы (скажем, несколько десятков записей) простые алгоритмы с оценкой $O(N^2)$ могут работать даже быстрее, чем усложненные алгоритмы с лучшей оценкой.

Экспериментальная оценка алгоритмов дает более конкретный результат, однако время работы оцениваемой программы зависит, кроме качества алгоритма, еще от многих факторов, в частности, от производительности ЭВМ и от конкретных сортируемых данных, которые при одном и том же размере могут быть более или менее удачными для испытываемого алгоритма. Чтобы уменьшить зависимость экспериментальной оценки от производительности ЭВМ, желательно, кроме времени работы, фиксировать также количество выполненных операций, характерных для данного алгоритма. Для сортировки такими операциями являются сравнение ключей элементов таблицы и перестановка (присваивание) записей. Кроме того, полезно сравнить на одних и тех же исходных данных исследуемый алгоритм с каким-либо из хорошо

известных алгоритмов. Для устранения влияния конкретных данных на оценку алгоритма следует многократно повторять испытания со случайными исходными данными, однако такие эксперименты требуют много машинного времени, а также грамотной оценки результатов с помощью методов математической статистики.

При сортировке таблиц различают ключ записи (т.е. ту часть записи, которая используется для сравнения записей) и саму запись, которая, кроме ключа, может содержать дополнительные данные. Алгоритмы сортировки состоят, главным образом, из сравнения ключей и перестановки записей и отличаются друг от друга тем, что и в каком порядке сравнивается и переставляется. В задаче сортировки массивов ключами являются сами элементы массива, и они же переставляются. Несущественность этого различия позволяет изучать алгоритмы сортировки для массивов, при необходимости без труда получая модификации этих алгоритмов для таблиц общего вида.

Ниже приводится сжатый обзор наиболее популярных алгоритмов внутренней сортировки.

2.2. Простые алгоритмы сортировки массивов

К простым алгоритмам принято относить следующие три очевидных и несложных в реализации алгоритма, не отличающихся, к сожалению, высокой эффективностью.

2.2.1. Алгоритм пузырька (BubbleSort)

Идея алгоритма заключается в следующем. Сравним элементы массива с индексами 1 и 2. Если первый больше второго, то поменяем эти элементы местами. Затем таким же образом сравним (и, если нужно, переставим) элементы с индексами 2 и 3, потом 3 и 4 и т.д. После сравнения элементов (**`N-1`**) и **`N`** первый проход алгоритма завершается. Можно гарантировать, что после этого прохода максимальный элемент массива находится на последнем месте (т.е. имеет индекс **`N`**). На втором проходе сравниваем пары 1 и 2, 2 и 3, ... (**`N-2`**) и (**`N-1`**). Далее аналогично. После **`N-1`** прохода все элементы займут свои законные места.

Можно попытаться сократить число проходов, отмечая с помощью специального флажка, были ли сделаны какие-либо перестановки на очередном проходе. Если ни одной перестановки за весь проход не было, то массив, очевидно, уже отсортирован и работу алгоритма можно завершить досрочно. Однако такое усовершенствование редко дает заметный выигрыш. Нетрудно показать, что для массива, заполненного случайным образом, с вероятностью 75% все равно будут выполнены все проходы алгоритма.

Ненамного лучшие результаты дает и другая модификация алгоритма пузырька, называемая «шейкер-сортировкой». Она отличается тем, что на нечетных проходах пузырек проходит слева направо (индекс в цикле возрастает), а на четных – справо налево (индекс убывает). Таким образом, после первых двух проходов на свои места станут максимальный и минимальный элементы массива.

2.2.2. Алгоритм простого выбора (SelectionSort)

Идея этого алгоритма еще проще. Найдем минимальный элемент массива и поменяем его местами с первым элементом. Затем повторим ту же процедуру, начиная со второго элемента массива, затем начиная с третьего и т.д. После **`N-1`** проходов все элементы станут на места.

2.2.3. Алгоритм простых вставок (InsertionSort)

Идея заключается в следующем. Пусть к некоторому моменту работы алгоритма первые **`k`** элементов массива уже отсортированы, т.е. расположены по возрастанию. На очередном проходе постараемся добиться, чтобы стали отсортированными (**`k+1`**) элементов. Для этого запомним значение элемента (**`k+1`**) в рабочей переменной **`R`** и будем сравнивать **`R`** со значениями элементов **`k`**, (**`k-1`**), (**`k-2`**) и т.д. Если значение сравниваемого элемента больше **`R`**, то этот элемент перемещается на одну позицию правее. Сравнения продолжаются, пока не будет найдено место, куда должен быть помещен элемент **`R`** (это случится либо когда очередной сравниваемый элемент меньше или равен **`R`**, либо когда мы дойдем до начала массива).

Таким образом, на очередном проходе отсортированная часть массива удлиняется на 1 элемент. Начав со значения **k=1**, можно за **N-1** проход отсортировать весь массив.

Алгоритм простых вставок можно улучшить, если выбирать место для вставки **k+1**-го элемента не последовательным просмотром элементов от **k** до **1**, а бинарным поиском (т.е. сравнить **R** с элементом **j := (k+1) div 2**, затем продолжить поиск на одном из интервалов **[1..j-1]** или **[j+1..k]** и т.д.). Этот подход, называемый алгоритмом бинарных вставок, позволяет существенно сократить число сравнений, но, к сожалению, не влияет на число перестановок.

2.2.4. Сравнение простых алгоритмов

Все три вышеописанных алгоритма и все их модификации имеют как максимальную, так и среднюю оценку $O(N^2)$, а потому используются только в случае небольших массивов или отсутствия жестких требований к времени сортировки.

Эксперименты показывают, что алгоритм пузырька является обычно наиболее медленным из трех, а алгоритмы выбора и вставок дают примерно одинаковое время сортировки.

В некоторых случаях можно ожидать, что исходный массив окажется «почти отсортированным», т.е. лишь небольшое число элементов будет нарушать порядок. Такое бывает, например, если массив ранее уже был отсортирован, но потом данные в нем подверглись небольшим модификациям. В этом случае вне конкуренции оказываются алгоритмы вставок, которые показывают очень высокую скорость на почти отсортированных массивах. Эта особенность алгоритмов вставок используется в описанном ниже алгоритме Шелла (п. 2.3.1).

2.3. Усовершенствованные алгоритмы сортировки массивов

Усложнение алгоритмов позволяет значительно повысить эффективность сортировки больших массивов. Перечислим наиболее известные алгоритмы этого класса.

2.3.1. Алгоритм Шелла

Разобьем элементы сортируемого массива на **h** цепочек, каждая из которых состоит из элементов, отстоящих друг от друга на расстояние **h** (здесь **h** – произвольное натуральное число). Первая цепочка будет содержать элементы с индексами **1, h+1, 2h+1, 3h+1** и т.д., вторая – **2, h+2, 2h+2** и т.д., последняя цепочка – **h, 2h, 3h** и т.д. Отсортируем каждую цепочку как отдельный массив, используя для этого метод простых вставок. Затем выполним все вышеописанное для ряда убывающих значений **h**, причем последний раз – для **h=1.**

Очевидно, массив после этого окажется отсортированным. Неочевидно, что все проходы при **h>1** не были пустой тратой времени. Тем не менее, оказывается, что дальние переносы элементов при больших **h** настолько приближают массив к отсортированному состоянию, что на последний проход остается очень мало работы. Эксперименты показывают, что для больших значений **N** оценка среднего времени работы алгоритма примерно $O(N^{1.26})$. Это значительно лучше, чем $O(N^2)$ для простых алгоритмов.

Большое значение для эффективности алгоритма Шелла имеет удачный выбор убывающей последовательности значений **h**. Желательно, чтобы при соседних значениях **k** значения h_k не были кратны друг другу. В литературе обычно рекомендуется использовать одну из двух последовательностей: $h_{k+1} = 3h_k+1$ или $h_{k+1} = 2h_k+1$. В обоих случаях в качестве начального h_k выбирается такое значение из последовательности, при котором все сортируемые цепочки имеют длину не меньше 2. Чтобы воспользоваться, например, первой из этих формул, надо сначала положить h_1**:=1**, а затем в цикле увеличивать значение **h** по формуле $h_{k+1}:=3*h_k+1$, пока для очередного h_k не будет выполнено неравенство h_k **>= (N-1) div 3**. Это значение h_k следует использовать на первом проходе алгоритма, а затем можно получать следующие значения по обратной формуле: h_{k-1} **:= (**h_k**-1) div 3**, вплоть до h_1=1.

Более сложными формулами определяется последовательность Седжвика:

$$h_k = \begin{cases} 9 \cdot 2^k - 9 \cdot 2^{k/2} + 1, & \textit{если } k \textit{ четно;} \\ 8 \cdot 2^k - 6 \cdot 2^{(k+1)/2} + 1, & \textit{если } k \textit{ нечетно.} \end{cases}$$

Доказано, что при выборе h_k по Седжвику среднее время работы алгоритма есть $O(N^{7/6})$, а максимальное – $O(N^{4/3})$.

На практике удобно раз и навсегда вычислить достаточное количество членов последовательности Седжвика (вот они: 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, 8929, 16001, 36289, 64769, 146305, 260609, 587521, 1045505, ...), затем по заданному **N** выбрать такое **k**, при котором **hk >= (N-1) div 3**, а далее в цикле выбирать значения последовательности **hk** по убыванию **k**.

2.3.2. Алгоритм быстрой сортировки (QuickSort)

В основе этого алгоритма – операция ***разделения массива***. Пусть **x** – некоторый произвольно выбранный элемент сортируемого массива (разделяющий элемент). Операция разделения имеет целью переставить элементы массива таким образом, чтобы сначала шли элементы, меньшие или равные **x** (не обязательно в порядке возрастания), а затем элементы, большие или равные **x**. При этом совершенно неважно, где именно окажется после разделения сам элемент **x**.

Чтобы выполнить разделение, начнем просматривать элементы массива слева направо и остановимся на первом из элементов, большем или равном **x**. Пусть индекс этого элемента – **i**. Аналогичным образом, начав с правого конца массива, остановимся на самом правом элементе, меньшем или равном **x** (индекс элемента – **j**). Поменяем местами элементы **i** и **j**, а затем продолжим идти вправо от **i** и влево от **j**, меняя местами пары элементов, стоящих не на месте. Разделение окончится, когда будет **i>j**.

Алгоритм быстрой сортировки можно теперь описать таким образом. Возьмем произвольный элемент массива **x** и выполним для него операцию разделения. Если левый и/или правый отрезки массива содержат более одного элемента, то выполним для каждого из этих отрезков ту же операцию, что и для всего массива (т.е. выберем произвольный элемент, выполним разделение и т.д.).

Организовать вышеописанное многократное выполнение операции разделения проще всего с помощью рекурсивного определения процедуры сортировки. Нерекурсивное описание алгоритма оказывается сложнее, т.к. при необходимости сортировать два отрезка массива, образовавшиеся после разделения, приходится явно сохранять в стеке один из отрезков (точнее, два граничных значения его индексов), пока сортируется другой. В то же время нерекурсивный вариант обычно оказывается более эффективным и, что немаловажно, он позволяет уменьшить используемую глубину стека. Для этого следует всегда из двух отрезков массива, подлежащих сортировке, заносить в стек более длинный. При этом можно гарантировать, что максимальное количество отрезков, одновременно находящихся в стеке, не может превысить $\log_2(N)$.

На эффективность алгоритма быстрой сортировки влияет также выбор разделяющего элемента **x**. Хотя теоретически **x** может выбираться произвольным образом, легко видеть, что выбор в качестве **x** первого или последнего элемента приводит к очень плохим результатам, если исходный массив оказывается уже сортированным или близким к сортированному. В связи с этим принято выбирать в качестве **x** либо элемент из середины массива, либо элемент со случайно выбранным индексом. Иногда используют чуть более сложное правило: взять первый элемент массива, последний элемент, элемент из середины массива и выбрать в качестве **x** средний по величине из этих трех.

Алгоритм **QuickSort** имеет оценку среднего времени $O(N \cdot \log(N))$ и на практике оказывается быстрее всех прочих, однако в худшем случае он может потребовать времени порядка $O(N^2)$.

В некоторых случаях может оказаться разумным скомбинировать **QuickSort** с алгоритмом пузырька. Дело в том, что достаточно громоздкая процедура разделения выполняет очень мало полезной работы в конце, когда разделяемые отрезки массива становятся короткими. Можно не

разделять отрезки, длина которых не больше некоторого **L**. После этого можно «досортировать» массив, выполнив **L-1** проход «пузырька». При этом нет необходимости организовывать отдельные проходы «пузырька» для множества коротких массивов, проще пропустить «пузырек» по всему сортируемому массиву. Практика показывает, что разумные значения **L** не превышают 3 – 4.

2.3.3. Алгоритм пирамидальной сортировки (HeapSort)

В основе этого алгоритма лежит понятие ***пирамиды***.

Массив $A_1, A_2, \ldots A_N$ называется пирамидой, если:

$A_k \geq A_{2k}$ для всех **k** таких, что $2k \leq N$;

$A_k \geq A_{2k+1}$ для всех **k** таких, что $2k+1 \leq N$.

Пирамиду можно рассматривать как представленное в виде массива двоичное дерево, у которого значение, связанное с вершиной-родителем, не меньше, чем значения, связанные с сыновьями. При этом A_1 – корень дерева, A_2 и A_3 – его сыновья, A_4, … A_7 – внуки и т.д.

Алгоритм сортировки состоит из двух фаз: построения пирамиды и преобразования пирамиды в отсортированный массив. В основе каждой из фаз лежит операция ***просеивания*** элементов через пирамиду.

Суть операции просеивания заключается в следующем. Предположим, имеется «почти правильная» пирамида, лишь для одного из элементов которой A_k, возможно, нарушены приведенные выше неравенства. Чтобы восстановить правильность пирамиды, следует сначала сравнить между собой сыновей A_k, т.е. элементы A_{2k} и A_{2k+1}. Обозначим номер большего из этих элементов буквой **r**. Теперь следует сравнить A_r с отцом (элементом A_k) и, если неравенство $A_k \geq A_r$ нарушено, то поменять местами A_k и A_r. Теперь оба неравенства для A_k будут выполняться, однако могут оказаться нарушенными аналогичные неравенства для A_r. Поэтому следует положить **k:=r** и повторять циклически те же действия, пока не окажется, что на очередной итерации элемент A_k либо уже не имеет сыновей, либо для него выполнены требуемые неравенства. На этом просеивание элемента A_k заканчивается.

Если представить пирамиду как двоичное дерево, то просеивание будет выглядеть как «падение» элемента A_k вдоль одной из ветвей дерева, пока он не займет подобающее ему место.

Построение пирамиды начинается с просеивания элемента, индекс которого **k := N div 2** (это самый правый из элементов, имеющих сыновей в дереве). После просеивания этого элемента отрезок, начиная с индекса **k** и до конца массива, будет соответствовать требованиям к пирамиде (потому что для элементов с номерами, большими **k**, никаких неравенств проверять не надо, у них нет сыновей!). Далее уменьшаем значение **k** на единицу и опять выполняем просеивание. Построение пирамиды будет завершено, когда будет просеян элемент A_1. Отметим, что первый элемент построенного массива-пирамиды – это его наибольший элемент.

Фаза преобразования пирамиды в отсортированный массив начинается с обмена значениями первого (наибольшего) элемента пирамиды с последним элементом. При этом наибольший элемент становится на свое окончательное место и не считается далее частью пирамиды (т.е. **N := N-1**). Сама пирамида может оказаться испорченной за счет замены корня. Чтобы восстановить пирамиду, следует таким же образом, как при построении пирамиды, еще раз просеять только новый корневой элемент A_1. Когда пирамида (уменьшившаяся на один элемент) восстановлена, ее корневой элемент – наибольший среди элементов пирамиды. Он опять обменивается с последним ее элементом, пирамида уменьшается еще на один элемент и т.д., пока пирамида не выродится в один элемент, за которым будут следовать отсортированные элементы массива.

Для алгоритма **HeapSort** имеются гарантированные оценки как максимального, так и среднего времени работы порядка **O(N·log(N))**, однако эксперименты показывают, что этот алгоритм обычно работает несколько медленнее, чем **QuickSort**, а в некоторых случаях уступает и алгоритму Шелла.

2.3.4. Алгоритмы слияния

Алгоритмы этой группы раньше широко использовались для сортировки последовательных файлов на магнитной ленте, однако и в задачах внутренней сортировки алгоритмы слияния показывают хорошую производительность. Их существенным недостатком является потребность во вспомогательном массиве того же размера, что и сортируемый массив.

Разнообразные алгоритмы слияния основываются на процедуре слияния двух уже отсортированных массивов в один отсортированный массив. Такая процедура реализуется легко и эффективно, за один проход по массивам. Рассмотрим два наиболее известных алгоритма сортировки путем многократных слияний.

Алгоритм простого слияния основан на следующем рассуждении. Пусть дан массив из **N** элементов. Каждый элемент исходного массива можно формально рассматривать как подмассив длины 1, причем, конечно, отсортированный. Сгруппируем попарно подмассивы из элементов с индексами 1 и 2, 3 и 4 и т.д. Будем сливать эти пары, записывая результат слияния во вспомогательный массив. (Возможен также другой вариант слияния: элементы 1 и **(N div 2)+1**, 2 и **(N div 2)+2** и т.п. Иногда это удобнее с точки зрения программирования.) В результате во вспомогательном массиве образуется **(N div 2)** сортированных подмассивов длины 2, а при нечетном **N** – еще один подмассив длины 1. Повторим операцию слияния, считая теперь источником вспомогательный массив, а приемником – исходный, и сливая подмассивы длины 2 в подмассивы длины 4 (последний подмассив опять может получиться неполным). Повторяем слияния до тех пор, пока не получится один массив длины **N**. Если для этого потребуется нечетное число проходов, то результат окажется во вспомогательном массиве и его нужно будет еще перенести в исходный массив.

Алгоритм естественного слияния исходит из того, что в исходном массиве почти наверняка уже есть отрезки, где элементы идут по возрастанию. Назовем такие отрезки ***сериями***, допуская в том числе «серии» длиной 1. Проход алгоритма состоит из двух фаз. На фазе распределения из массива выделяются серии, причем нечетные по счету серии записываются во вспомогательный массив от его начала, а четные – от конца по убыванию индекса. На фазе слияния сливаются пары серий, одна от начала вспомогательного массива, другая от конца, и результат записывается в исходный массив от его начала. Проходы повторяются, пока не останется одна серия. При реализации этого алгоритма следует учитывать, что число сливаемых серий на фазе слияния может оказаться меньше, чем число серий, записанных на фазе распределения. Такое возможно, если две серии (например, первая и третья) сами сольются в одну, т.е. если последний элемент первой серии не больше, чем первый элемент третьей серии. Из-за этого при слиянии количество нечетных и четных серий может оказаться различным. Лишние серии, оставшиеся без пары, просто переписываются из вспомогательного массива в исходный.

Оба описанных выше алгоритма слияния имеют оценку **O(N·log(N))** как для среднего, так и для максимального времени выполнения.

2.4. Экспериментальная оценка производительности алгоритма

Оценка производительности алгоритма имеет первостепенное значение при определении пригодности этого алгоритма для решения конкретных типов задач. Важнейшей частью данной работы является сбор статистики о времени работы алгоритмов при сортировке различных массивов данных.

Трудоемкость алгоритма может измеряться либо в единицах времени, либо в единицах числа операций, наиболее характерных для алгоритмов данного типа. Для алгоритмов сортировки обычно измеряют число операций сравнения элементов массива и отдельно – число операций присваивания элементов. Остальными операциями – инициализацией и изменением счетчиков циклов, проверками окончания и т.п. – можно пренебречь, поскольку их число либо значительно меньше, чем числа сравнений и присваиваний, либо, в крайнем случае, прямо пропорционально им.

Поскольку для большей части алгоритмов трудоемкость может сильно зависеть от более удачных или менее удачных для данного алгоритма данных, для экспериментальной оценки трудоемкости чаще всего используют случайные данные. Результаты одного прогона алгоритма на случайном массиве данных не несут надежной информации, однако при увеличении числа прогонов с усреднением результатов обычно удается получить достаточно объективную оценку алгоритма. Эти вопросы подробно исследуются в теории математической статистики, здесь же можно лишь отметить, что случайная ошибка определения трудоемкости убывает обратно пропорционально квадратному корню из числа экспериментов.

Оценка трудоемкости непосредственно в единицах времени тоже является достаточно полезной, поскольку показывает, чего можно ждать от использования конкретной программы сортировки.

ОС Windows предоставляет несколько различных API-функций для измерения системного времени. Однако наиболее простая из этих функций **`GetTickCount`** дает достаточно низкую точность (10 мс для версий от Windows XP и выше). При этом для получения достаточно малой относительной ошибки измерений требуется, чтобы измеряемые интервалы были, по крайней мере, не меньше секунды. Таким образом, доверять можно только таким оценкам времени сортировки, которые получены для больших массивов данных, сортировка которых требует нескольких секунд. Учитывая высокую эффективность усовершенствованных алгоритмов сортировки и большую производительность современных процессоров, размеры массивов должны при этом измеряться, по крайней мере, миллионами элементов, что не всегда удобно. Можно использовать функцию **`QueryPerformanceCounter`**, однако она, обеспечивая высокую точность измерения времени, не позволяет отделить время данного процесса от затрат времени системой и другими приложениями.

Наиболее подходящим средством измерения времени работы алгоритма представляются API-функции **`GetProcessTimes`** и **`GetThreadTimes`**. Эти функции позволяют получить чистое процессорное время, затраченное, соответственно, всем процессом или только одной из его нитей. При этом имеется возможность отделить время работы в режиме задачи (т.е. время выполнения прикладного кода) от времени работы в режиме системы (выполнения API-функций и т.п.). Номинальная точность измерения времени – 100 нс.

3. Выполнение задания

В данной лабораторной работе требуется запрограммировать и протестировать алгоритмы внутренней сортировки, указанные в варианте задания.

В каждом варианте задания требуется запрограммировать один или два (в зависимости от сложности) алгоритма сортировки и проверить их работу на ряде тестовых примеров.

В качестве тестовых массивов следует использовать:

- массив из 1000 первых натуральных чисел в порядке возрастания (т.е. пример уже отсортированного массива);
- массив из 1000 первых натуральных чисел в порядке убывания (т.е. пример массива, отсортированного «наоборот»);
- не менее 4 сгенерированных массивов псевдослучайных чисел разного размера, от 100 до 100000 элементов (максимальный размер массива может быть изменен в зависимости от производительности процессора).

Таблица результатов должна для каждого алгоритма и для каждого тестового массива включать время работы алгоритма, число выполненных сравнений и присваиваний элементов массива.

По крайней мере, для одного не очень большого примера следует для визуальной проверки правильности работы алгоритма выдать на экран весь сортируемый массив до и после сортировки.

Отчет о работе должен включать:

- постановку задачи, вариант задания, список бригады;

- исходные тексты разработанных процедур сортировки (не обязательно полный текст всей программы);
- итоговую таблицу результатов тестирования;
- выводы, которые можно сделать по этим результатам.

Отчет и программа представляются также в электронном виде.

4. Пример выполнения задания

Дан вариант задания: запрограммировать алгоритм BubbleSort и рекурсивный вариант алгоритма QuickSort, получить статистику работы алгоритмов на ряде массивов.

Программа была реализована в системе программирования Borland Delphi как консольное приложение Windows. Исходные тексты программы прилагаются в проекте **`Laba3`**.

Ниже приводится таблица результатов работы программы для ряда тестовых массивов. Использовался процессор Intel E2180 @2.00 ГГц.

		BubbleSort			**QuickSort**		
Данные в массиве	N	Время (мс)	Сравнения	Присваивания	Время (мс)	Сравнения	Присваивания
Возрастающие	1000	15.625	499500	0	0.000	12724	4812
Убывающие	1000	0.000	499500	1498500	0.000	12724	6309
Случайные	100	0.000	4950	8193	0.000	1115	874
Случайные	1000	15.625	499500	757047	0.000	17149	10598
Случайные	10000	703.125	49995000	75153522	0.000	210353	132601
Случайные	100000	74578.125	4999950000	7509426354	31.250	2544779	1605720

Как видно из таблицы, алгоритм BubbleSort показал приемлемые результаты только на массиве из 100 элементов, а для больших массивов продемонстрировал квадратичный рост числа выполненных операций. При этом число выполненных сравнений не зависит от конкретных данных, оставаясь одним и тем же даже для заранее отсортированного массива.

Алгоритм QuickSort показал значительно лучшие результаты уже для массива из 1000 элементов, а для N=100000 он проработал быстрее, чем BubbleSort, более чем в 2000 раз.

Таким образом, эксперименты подтвердили непригодность алгоритма BubbleSort для сортировки больших массивов и высокую эффективность, достигаемую в этом случае алгоритмом QuickSort.

5. Варианты заданий

Запрограммировать и протестировать следующие алгоритмы, описанные в разделе 2:

1. Нерекурсивный вариант алгоритма QuickSort со случайным выбором разделяющего элемента и с выбором среднего по величине из трех элементов;
2. Алгоритм простого выбора и алгоритм HeapSort;
3. Комбинация QuickSort и пузырька для 2-3 различных значений L;
4. Алгоритм бинарных вставок и алгоритм Шелла;
5. Алгоритм простых вставок и алгоритм естественного слияния;
6. Алгоритм шейкер-сортировки и алгоритм простого слияния.

Контрольные вопросы

1. Зачем нужна сортировка?
2. В чем существенное различие алгоритмов внутренней и внешней сортировки?

3. Как изменится время работы алгоритма пузырька, если размер сортируемого массива увеличится в 1000 раз?
4. Как в аналогичном случае изменится время работы алгоритма HeapSort?
5. Какой алгоритм лучше выбрать для сортировки массива из 50 элементов?
6. Почему в алгоритме Шелла используются именно вставки, а не пузырек и не простой выбор?
7. Как скажется на эффективности алгоритма Шелла замена в нем простых вставок на бинарные?
8. Какая глубина стека может потребоваться в худшем случае при работе рекурсивной версии алгоритма QuickSort?
9. Как зависит время выполнения одной операции просеивания в алгоритме HeapSort от размера массива?
10. Предположим, на вычислительном центре должны постоянно решаться задачи сортировки больших массивов, и успешность работы центра оценивается по числу решенных за месяц задач. Какой алгоритм сортировки лучше использовать в этом случае?
11. Изменится ли выбор алгоритма сортировки, если в качестве важнейшего требования задано, что ни один массив не должен обрабатываться дольше определенного интервала времени?