Лабораторная работа №9
ИССЛЕДОВАНИЕ И ОЦЕНКА АЛГОРИТМОВ СОРТИРОВКИ
Краткая теория

Цель работы. Разработка программ, реализующих различные алгоритмы сортировки, и оценка их временной и пространственной сложности.

Сортировка. Основные определения и классы алгоритмов

Сортировка — это упорядочение элементов в списке. В случае, когда элемент имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся любые данные, не влияющие на работу алгоритма.

Мы рассмотрим наиболее распространенные алгоритмы сортировки для чисел, хотя они могут быть распространены на строки и списки элементов, о которых говорилось в общем определении.

Классификация алгоритмов сортировки

При классификации учитывают следующие основные характеристики алгоритмов.

1) *Устойчивость* (stability) — устойчивая сортировка не меняет взаимного расположения равных элементов.

1) *Естественность поведения* — эффективность метода при обработке уже упорядоченных или частично упорядоченных данных. Алгоритм ведѐт себя естественно, если учитывает эту особенность входной последовательности и работает лучше.

2) *Использование операции сравнения*. Алгоритмы, применяющие для сортировки сравнение элементов между собой, называются основанными на сравнениях. Минимальная трудоемкость

худшего случая для этих алгоритмов составляет $O(n^2)$, но они отличаются гибкостью применения. Для специальных случаев (типов данных) существуют более эффективные алгоритмы.

Ещѐ одним важным свойством алгоритма является сфера его применения. При этом различают два основных типа упорядочения.

1. *Внутренняя сортировка,* которая оперирует с массивами, целиком помещающимися в оперативной памяти с произвольным доступом к любой ячейке. Данные обычно упорядочиваются на том же месте, без дополнительных затрат. В современных персональных компьютерах, как уже отмечалось, широко применяется подкачка и кэширование памяти. Алгоритм сортировки должен хорошо сочетаться с этими операциями.

2. *Внешняя сортировка,* которая применяется для запоминающих устройств большого объѐма, с последовательным доступом (упорядочение файлов). При этом в каждый момент доступен только один элемент, а затраты на перемотку по сравнению с памятью неоправданно велики. Это накладывает дополнительные ограничения на алгоритм и приводит к методам упорядочения, обычно использующим дополнительное дисковое пространство. Кроме того, доступ к данным на носителе производится намного медленнее, чем операции с оперативной памятью. Их объѐм настолько велик, что не позволяет разместить информацию в ОЗУ.

К алгоритмам ***устойчивой сортировки*** относят следующие.

1. Сортировка *пузырьком* (англ. Bubble sort) — сложность алгоритма: $O(n^2)$; для каждой пары индексов производится обмен, если элементы расположены не по порядку.

2. Сортировка *перемешиванием* (Шейкерная, Cocktail sort, bidirectional bubble sort) — сложность алгоритма: $O(n^2)$.

3. Сортировка *вставками* (Insertion sort) — сложность алгоритма: $O(n^2)$; определяют место элемента в упорядоченном списке и вставляют его туда.

4. *Гномья* сортировка — сложность алгоритма: O(n^2); сочетает методы пузырьковой сортировки и вставками.

5. *Блочная* сортировка (Корзинная, Bucket sort) — сложность алгоритма: O(n); требуется O(k) дополнительной памяти и знание о природе сортируемых данных.

6. Сортировка *подсчѐтом* (Counting sort) — сложность алгоритма: O($n+k$); требуется O($n+k$) дополнительной памяти (существует три варианта).

7. Сортировка *слиянием* (Merge sort) — сложность алгоритма: O($n \log n$); требуется O(n) дополнительной памяти; упорядочивают две половины списка отдельно (первую и вторую), а затем — сливают их воедино.

8. Сортировка *с помощью двоичного дерева* (англ. Tree sort) — сложность алгоритма: O($n \log n$); требуется O(n) дополнительной памяти.

Алгоритмами ***неустойчивой сортировки*** являются следующие методы.

1. Сортировка *выбором* (Selection sort) — сложность алгоритма: O(n^2); выполняется поиск наименьшего или наибольшего элемента и помещение его в начало или конец упорядоченного списка.

2. Сортировка *Шелла* (Shell sort) — сложность алгоритма: O($n \log_2 n$); попытка улучшить сортировку вставками.

3. Сортировка *расчѐской* (Comb sort) — сложность алгоритма: O($n \log n$)

4. *Пирамидальная* сортировка (Сортировка кучи, Heapsort) — сложность алгоритма: O($n \log n$); список превращается в кучу, берется наибольший элемент и добавляется в конец списка.

5. *Плавная* сортировка (Smoothsort) — сложность алгоритма: O($n \log n$).

6. *Быстрая* сортировка (Quicksort), в варианте с минимальными затратами памяти сложность алгоритма: O($n \log n$) — среднее время, O(n^2) — худший случай; широко известен как быстрейший из

известных для упорядочения больших случайных списков; исходный набор данных разбивается на две половины так, что любой элемент первой половины упорядочен относительно любого элемента второй; затем алгоритм применяется рекурсивно к каждой половине. При использовании O(n) дополнительной памяти сортировка становится устойчивой.

7. *Поразрядная* сортировка — сложность алгоритма: O($n \cdot k$); требуется O(k) дополнительной памяти.

8. *Сортировка перестановкой* — O($n \cdot n!$) — худшее время. Для каждой пары осуществляется проверка верного порядка и генерируются всевозможные перестановки исходного массива.

Описание алгоритмов сортировки

Рассмотрим наиболее распространенные алгоритмы.

Алгоритм 1. Пузырьковая сортировка

Этот метод - наиболее распространенный и простой. Он требует минимального объема памяти для данных, но затраты времени на его реализацию велики. При упорядочении выполняются следующие операции:

1) элементы массива сравниваются *попарно*: первое со вторым; второе с третьим; i-тое – с (i+1) - вым;

2) если они стоят неправильно (при *упорядочении* по возрастанию первый должен быть меньше второго или равен ему), то элементы меняются местами.

За один такой просмотр массива при сортировке по возрастанию минимальный элемент «вытолкнется», по крайней мере, на одно место вверх (вперед), а максимальный – переместится в самый конец (вниз). Таким образом, минимальный элемент как легкий пузырек воздуха в жидкости постепенно

«всплывает» вверх (в начало последовательности). Отсюда – название метода.

Поскольку последний элемент после первого просмотра массива окажется на своем месте, следующий просмотр можно закончить раньше, т.е. в конце сравнивать (n-2)-рой и (n-1)-вый элементы. Вообще, при k-том просмотре достаточно сравнить элементы от первого до (n-k) –того.

За n-1 просмотр произойдет полное упорядочение массива при любом исходном расположении элементов в нем.

Алгоритм сортировки пузырьком

1. Задать массив из n чисел.
2. Для номера_просмотра (k) от 1 до n-1 выполнить
   2.1. Для номера_элемента (i) от 1 до n - k выполнить
      Если элементы i-тый и (i+1)-вый стоят неправильно, то
         Поменять их местами;
3. Вывести отсортированный массив.
4. Закончить.

Сокращение количества просмотров улучшает временные характеристики метода. Из алгоритма можно понять, что если на одном из просмотров не было перестановок, то их не будет и потом – данные уже отсортированы, а процесс следует закончить. Такой подход дает значительную экономию времени при работе с большими «почти отсортированными» массивами.

В ускоренном алгоритме будем использовать булевскую переменную «Перестановка», которой сначала присваивается значение «ложь», так как перестановок элементов не было. Затем, если в процессе просмотра массива некоторые элементы менялись метами, то признак получает значение «истина». Это позволит прервать выполнение внешнего цикла при условии, что на очередном просмотре перестановки не выполнялись.

Алгоритм ускоренной сортировки пузырьком

1. Задать массив из *n* чисел.

2.1 Номер_просмотра (k) = 1.

2.2. Повторять

2.2.1. Перестановка = *ложь*.

2.2.2. Для i от 1 до n-k выполнить

Если элементы i-тый и (i+1)-вый стоят неправильно, то

a) Поменять местами i-тый и (i+1)-вый элементы;

b) Перестановка = *истина*.

2.2.3. $k = k + 1$

Пока Перестановка.

3. Вывести отсортированный массив.

4. Закончить.

Алгоритм 2. Сортировка вставками

Это достаточно простой, но эффективный в некоторых случаях метод. В начале считается, что первый элемент находится на своем месте. Далее, начиная со второго, каждый элемент сравнивается со всеми, стоящими перед ним, и если они стоят неправильно, то меняются местами. Таким образом, новый элемент сравнивается и меняется местами не со всем массивом, а только до тех пор, пока в начале не найдется элемент, меньший его. Поэтому рассматриваемый алгоритм примерно в два раза быстрее сортировки пузырьком. Для уже частично отсортированных массивов такой метод является наилучшим.

Словесное описание алгоритма можно представить так.

Алгоритм сортировки вставками

1. Задать исходный массив A из n элементов, например, с помощью генератора случайных чисел.
2. Для i от 0 до n

2.1. Индекс элемента в конце массива $j = i$

2.2. Пока ($j > 0$) И ($A_{j-1} > A_i$)

2.2.1. *Сдвиг большего элемента, чтобы было место для вставки*

$A_j = A_{j-1}$

2.2.2. $j = j - 1$

2.3. Вставка j - го элемента на его место

$A_j = A_i$

3. Вывести массив *A*.

Время сортировки можно сократить, если учесть, что начальная часть массива (до индекса *j*) упорядочена. При этом точку вставки можно найти методом деления пополам диапазона индексов этой части (от 0 до *j*).

Алгоритм сортировки двоичными вставками

1. Задать исходный массив *A* из *n* элементов.
2. Для *i* от 0 до *n*

2.1. *Вставляемый элемент* $x = A_i$

2.2. Левая граница диапазона $L = 0$.

2.3. Правая граница диапазона $R = i$.

2.4. Пока ($L < R$)

2.4.1. Искомый индекс $m = (L + R)/2$

2.4.2. Если $A_m \leq x$

$L = m + 1$

Иначе

$R = m$.

2.5. Перепись элементов на одну позицию в конец массива (освобождение места для вставки)

2.5.1. Для *j* от *i* до $R + 1$ с шагом -1

$A_j = A_{j-1}$

2.6. Вставка последнего элемента

$A_R = x$

3. Вывести массив *A*.

Число сравнений и пересылок рассмотренным методом в худшем случае пропорционально *n*log(n)*, т.е. сложность алгоритма равна O(*n*log(n)*).

Алгоритм 3. Сортировка выбором

Этот метод – один из наиболее простых. Он требует выполнения *n*-1 просмотра массива. Во время *k*-го просмотра выявляется наименьший элемент, который затем меняется местами с *k*-м.

Алгоритм сортировки выбором

1. Задать массив *A* из *n* чисел.
2. Для *k* от 0 до *n*-1
    2.1. $j = k$
    2.2. Для i от $k + 1$ до n-1
        2.2.1.Если $A_i < A_j$,
            $j = i$.
    2.3. Если $k \neq j$, то
        Поменять местами A_k и A_j.
3. Вывести массив *A*.
4. Закончить.

Алгоритм 4. Сортировка Шейкером

Если данные сортируются не в оперативной памяти, а на жестком диске, то количество перемещений элементов существенно влияет на время работы. Этот алгоритм уменьшает количество таких перемещений. За один проход из всех элементов выбираются минимальный и максимальный. Потом минимальный элемент помещается в начало массива, а максимальный - в конец. Таким образом, за каждый проход два элемента оказываются на своих местах, поэтому понадобится $n/2$ проходов, где n — количество элементов.

На каждом проходе массив просматривается слева направо до середины, а потом – наоборот (справа налево). В результате, как при сортировке «пузырьком», при просмотре слева направо максимальный элемент попадает на свое место, а при обратном – минимальный.

Алгоритм сортировки Шейкером

1. Задать массив из *n* чисел.

2.1 Левая_граница = 0.

2.2. Правая_граница = *n* - 1.

2.3. Повторять

2.3.1. *Движение слева направо:*

1) Для *i* = Левая_граница; *i* < Правая_граница; *i*++

Если элементы *i*-тый и (*i*+1)-вый стоят неправильно, то

a) Поменять их местами;

b)Номер = *i*.

2.3.2. *Движение справа налево:*

1) Правая_граница = Номер.

2) Для *i* = Правая_граница; *i* > Левая_граница; *i*--

Если элементы *i*-тый и (*i*+1)-вый стоят неправильно, то

a) Поменять их местами;

b)Номер = *i*.

2.3.3. Левая_граница = Номер.

Пока Левая_граница < Правая_граница.

3. Вывести отсортированный массив.

4. Закончить.

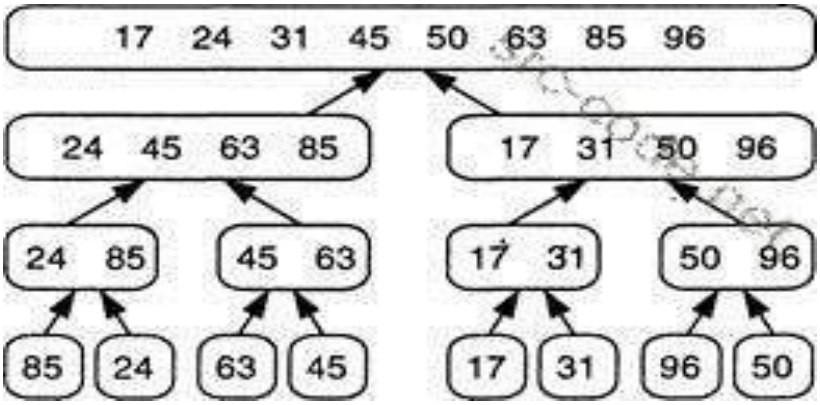
Алгоритм 5. Сортировка слиянием

Этот метод является одним из самых простых и быстрых. Особенностью его является то, что он работает с элементами массива последовательно, благодаря чему может использоваться, например,

для данных на жѐстком диске. Кроме того, алгоритм, может быть эффективен для сортировки таких структур, как связанные списки. Он был предложен Джоном фон Нейманом в 1945 году.

Алгоритм использует методику «разделяй и властвуй», которая может быть представлена в виде трех стадий:

1) разделение — данные разбиваются на две, три или более составляющих, обработка которых осуществляется простыми методами;



2) рекурсия — применяется для обработки полученных составляющих;

3) слияние — результаты действий над составляющими «сливаются» в единое целое.

Основой алгоритма является *подзадача слияния* двух отсортированных массивов в один, тоже отсортированный. Пусть исходные массивы A и B имеют размерность n_a и n_b. Результат слияния – массив С размерностью $n_a + n_b$. Он формируется так. Из первых элементов массивов A и B в C записывается сначала меньший, а потом – больший. Как показано на рисунке 4.1. Считая, что индексы элементов начинаются с 0 (как это принято в системах C++ и java), алгоритм можно описать так.

Рис. 4.1. Сортировка слиянием

Алгоритм слияния двух отсортированных массивов

1. Номер A (i_a) =0.
2. Номер B (i_b) =0.
3. Пока ($i_a < n_a$) *и* ($i_b < n_b$)

    3.1. Если $A\,[i_a] \le B\,[i_b]$, то

        а) $C[i_a + i_b] = A\,[i_a]$;

b) $i_a = i_a + 1$.

Иначе

a) C[$i_a + i_b$] = B [i_b];

b) $i_b = i_b + 1$.

4. Если $i_a < n_a$, то

Переписать оставшийся массив A в C.

5. Если $i_b < n_b$, то

Переписать оставшийся массив B в C.

Собственно сортировка слиянием

Процедура слияния использует два отсортированных массива. Очевидно, что массив из одного элемента по определению является отсортированным. Тогда сортировку можно осуществить в соответствии с рисунком 2.1 следующим образом.

1. Разбить элементы массива на пары и осуществить слияние элементов каждой пары, получив отсортированные цепочки длины 2 (кроме может быть, одного элемента, для которого не нашлось пары);

2. Разбить отсортированные цепочки на пары, и осуществить слияние цепочек каждой пары;

3. Если число отсортированных цепочек больше единицы, перейти к шагу 2.

Этот алгоритм можно реализовать рекурсивным способом или с помощью обычного цикла. Рекурсивный алгоритм будет таким.

Рекурсивный алгоритм сортировки слиянием

1. Если Правая_граница = 0, то

Результат = 0

Иначе

1.1. Если Правая_граница = 1, то

Результат = A_0

Иначе

1.1.1. Если Правая_граница = 2, то

Если $A_1 < A_0$

Результат = A_1, A_0

Иначе

Результат = A_0, A_1

1.1.2. Иначе

a) Упорядочить первую половину массива A:

Вызвать Сортировку(*A*; Левая_граница;
Правая_граница/2);

b) Упорядочить вторую половину массива A:

Вызвать Сортировку (*A*; Левая_гр +Правая_гр/2+1;
Правая_гр);

c) *Слияние Первой и Второй половин массива A:*

Слияние *A*[0… Левая_гр + Правая_гр/2] и
A[Левая_гр + Правая_гр/2 + 1; Правая_гр]

2. Вывести массив *A*.
3. Закончить.

Алгоритм 6. Сортировка Боуза- Нельсона

Этот метод реализует один из способов слияния. Он удобен для сортировки больших массивов, находящихся на устройствах последовательного доступа (например, винчестерах). Массив *A* разбивается на две половины: *B* и *C*. Эти половины сливаются в упорядоченные пары, объединяя первые элементы из *B* и *C* в первую пару, вторые – во вторую и т.д. Полученному массиву присваивается имя *A*, после чего операция повторяется. При этом пары сливаются в упорядоченные четверки. Предыдущие шаги повторяются: четверки сливаются в восьмерки и т.д., пока не будет упорядочен весь массив, т.к. длины частей каждый раз удваиваются. Если размер массива нечетный, или на некотором шаге получатся неполные части, то выполняют отдельно слияние начал, концов и центральной частей. Описанный алгоритм можно проиллюстрировать следующим примером.

Пример. Исходная последовательность

A = 44 55 12 42 94 18 06 67

Шаг 1

Первая половина *B* = 44 55 12 42

Вторая половина *C* = 94 18 06 67

A = 44 94 18 55 06 12 42 67 – для наглядности нечетные пары выделены.

Шаг 2

Первая половина *B* = 44 94 18 55

Вторая половина *C* = 06 12 42 67

A = 06 12 44 94 18 42 55 67 – выделены упорядоченные четверки.

Шаг 3

Первая половина *B* = 06 12 44 94

Вторая половина *C* = 18 42 55 67

A = 06 12 18 42 44 55 67 94 – отсортированный массив.

Судя по описанию, алгоритм проще всего реализовать с помощью рекурсии, причем, в отличие от традиционной сортировки слиянием он не требует дополнительной памяти. Обозначим *r* - расстояние между началами сливаемых частей, *m* - их размер и *j* - наименьший номер элемента. Будем считать, что выполняется сортировка по возрастанию.

Рекурсивный алгоритм сортировки Боуза- Нельсона

1. *Слияние*

1.1. *Начало рекурсии.* Если $j+r \le n$, то

1.1.1. Если $m = 1$, то

а) Если $A[j] > A[j+r]$, то

Поменять местами $A[i]$ и $A[j+r]$.

1.2. Иначе

1.2.1. $m = m / 2$.

1.2.2. Выполнить Слияние «начал» от *j* до *m* с расстоянием *r*.

1.2.3. Если $j+r+m \le n$, то

Выполнить Слияние «концов» от $j+m$ до m с расстоянием r.

1.2.4. Выполнить Слияние «центральной части» от $j+m$ до m с расстоянием $m - r$.

2. *Основная часть рекурсии*

2.1.$m = 1$.

2.2.Повторять

2.3.*Цикл слияния списков равного размера*

2.3.1. $j = 1$.

2.3.2. Пока $j+m \leq n$

1) Выполнить Слияние (j,m,m);

2) $j = j + 2m$

2.3.3. Удвоение размера списка перед началом нового прохода

$m = 2m$.

Конец цикла 2.2, реализующего все слияния

Пока $m < n$.

3. Вывести массив A.

4. Закончить.

Число сравнений и пересылок при реализации сортировки методом Боуза - Нельсона в самом худшем случае пропорционально $n \log n$, т.е. сложность алгоритма O($n \log n$). Дополнительная память не требуется.

Алгоритм 7. Быстрая сортировка

Метод использует тот факт, что число перестановок в массиве может резко сократиться, если менять местами элементы, находящиеся на большом расстоянии. Он реализует метод «разделяй и властвуй». Для этого обычно в середине массива выбирается один элемент (опорный). Далее слева от опорного располагают все элементы, меньше его, а справа – больше. Затем тот же прием

применяют к половинкам массива. Процесс заканчивается, когда в частях массива останется по одному элементу.

В алгоритме используются два разнонаправленных процесса. Первый выполняется от начала массива и ищет элемент, больший опорного. Второй - работает с конца и ищет элемент, меньший опорного. Как только такие элементы найдены, производится их обмен местами. Далее поиск продолжается с того места, где процессы остановились.

Таким образом, когда процессы встречаются, любой элемент в первой части меньше любого во второй. Это значит, что их уже сравнивать друг с другом не придется. Остается только провести такую же операцию по отношению к полученным половинам, и так далее, пока в очередной части массива не останется один элемент. Это будет *означать*, что массив отсортирован. Очевидно, что наиболее удобный способ реализации рассмотренного метода – рекурсивный.

Общий алгоритм можно представить так.

1. Из массива выбирается некоторый опорный элемент, Опорный = $a[i]$, $i=n/2$.
2. Запускается процедура разделения массива, которая перемещает все элементы, меньшие, либо равные Опорному, влево от него, а все, большие, равные Опорному,
   - вправо.
3. Теперь массив состоит из двух подмассивов, причем длина левого меньше, либо равна длине правого.
4. Для обоих подмассивов, если в них больше двух элементов, рекурсивно заполняется та же процедура.

   В конце получится полностью отсортированная последовательность.

Уточненный рекурсивный алгоритм быстрой сортировки.

Исходные данные: Массив A; Левая_граница (0); Правая_граница (n-1)

1. Индекс в начале, *i* = Левая_граница.
2. Индекс в конце, *j* =Правая_граница.
3. Опорный = *A*[(Левая_граница + Правая_граница) / 2].
4. Повторять
    4.1. *Движение слева направо:*
        Пока *A[i*] < Опорный
            i = *i* + 1
    4.2. *Движение справа налево:*
        Пока *A[j*] > Опорный
            j = *j* - 1.
    4.3. Поменять местами *A[i*] и *A[j*].
    4.4. *i* = *i* + 1.
    4.5. *j* = *j* - 1.

Пока Левая_граница < Правая_граница.

5. Если Левая_граница < *j,* то отсортировать *A* от Левая_граница до *j,*
6. Если *i* > Правая_граница, то отсортировать *A* от *i* до Правая_граница.
7. Вывести массив *A*.

Алгоритм 8. Сортировка Шелла

Метод является ускоренным вариантом сортировки вставками. При этом ускорение достигается за счет увеличения расстояния, на которое перемещаются элементы. Исходный массив делится на *d* частей, содержащих *n* / *d* элементов каждый (последний подмассив может быть короче). Подмассивы содержат элементы с номерами [0, *d*, *2 d* и т.д.], [1, *d* +1, *2 d* +1 и т.д.], [2, *d* + 2, 2 *d* +2 и т.д.] и т.д.

Вначале сравниваются и упорядочиваются с помощью алгоритма вставок элементы, отстоящие один от другого на расстоянии *d*, т.е. имеющие номера 0 и 1, *d* и *d* + 1, 2 *d* и 2 *d* + 1 и т.д. Затем процедура повторяется при меньших значениях *d*, например, *d* / 2. Завершается

алгоритм упорядочением элементов при $d = 1$, то есть обычной

сортировкой вставками. Мы рассмотрим сортировку Шелла для начального значения d, равного n /2, и будем последовательно уменьшать его вдвое.

Алгоритм сортировки Шелла

1. Задать массив A из n чисел.
2. Выполнять
   2.1. $d = n / 2$.
   2.2. Для i от 0 до $n - d$ с шагом d
      2.2.1. $j = i$.
      2.2.2. $x = A_i$
      2.2.3. Пока ($j \geq d$) $(A[j] > A[j + d])$
         a) $A[j] = A[j + d]$
         b) $j = j - d$.
   2.3. $d = d / 2$.
   Пока $d > 0$.
3. Вывести массив A.
1. Закончить.

Порядок выполнения лабораторной работы

Работа предполагает выполнение следующих этапов.

1. Знакомство со всеми разделами руководства.

2. Получение у преподавателя задания на разработку программы для алгоритмов сортировки (см. Приложение 4).

3. Разработка и отладка заданных программ.

4. Получение верхней и экспериментальной оценки времени выполнения заданных алгоритмов и программ.

5. Нахождение предельной оценки емкости памяти, необходимой для выполнения разработанных программ.

Содержание отчета о выполненной работе

Отчет о выполненной работе должен содержать.

1. Название и цель работы.
2. Словесное описание заданных алгоритмов сортировки.
3. Тексты программ.
4. Формулы верхней оценки временной и емкостной сложности заданных алгоритмов.
5. Результаты экспериментальной оценки временной и емкостной сложности заданных алгоритмов.

Контрольные вопросы

1. Что такое сортировка и для чего она нужна?
2. По каким признакам выполняется классификация алгоритмов сортировки?
3. Как оценивается временная сложность алгоритмов упорядочения?
4. Как оценивается емкостная сложность алгоритмов сортировки?
5. Какой метод упорядочения самый простой?
6. Какой алгоритм сортировки самый быстрый?
7. Какие алгоритмы пригодны для упорядочения файлов?
8. Чем отличается сортировка чисел от строк?
9. Как Вы определили время выполнения Ваших алгоритмов?
10. Как Вы определили объем памяти, необходимой для выполнения Ваших алгоритмов?
11. Каковы основные отличия сортировки вставками от «пузырьковой»?
12. Каковы основные отличия упорядочения слиянием от «пузырьковой»?
13. Каковы основные отличия сортировки слиянием от метода Боуза-Нельсона?
14. Каковы основные отличия упорядочения слиянием и вставками?
15. Каковы отличительные особенности быстрой сортировки?
16. Как выполняется упорядочение Шейкером?
17. Каковы особенности сортировки Шелла и для каких данных она предпочтительна?
18. У каких известных Вам методов сортировки временная сложность зависит от объема используемой памяти?

Индивидуальные задания

1. Составить две программы, которые реализуют алгоритмы простой сортировки «пузырьком» и вставками. Исходные данные задавать с помощью датчика случайных чисел.

2. Составить две программы, которые реализуют алгоритмы простой сортировки «пузырьком» и выбором. Исходные данные задавать с помощью датчика случайных чисел.

3. Составить две программы, которые реализуют алгоритмы простой сортировки «пузырьком» и шейкером. Исходные данные задавать с помощью датчика случайных чисел.

4. Составить две программы, которые реализуют алгоритмы простой сортировки «пузырьком» и слиянием. Исходные данные задавать с помощью датчика случайных чисел.

5. Составить две программы, которые реализуют алгоритмы простой сортировки «пузырьком» и быстрой сортировки. Исходные данные задавать с помощью датчика случайных чисел.

6. Составить две программы, которые реализуют алгоритмы простой сортировки «пузырьком» и сортировки Шелла. Исходные данные задавать с помощью датчика случайных чисел.

7. Составить две программы, которые реализуют алгоритмы простой сортировки «пузырьком» и методом Боуза- Нельсона. Исходные данные задавать с помощью датчика случайных чисел.

8. Составить две программы, которые реализуют алгоритмы ускоренной сортировки «пузырьком» и вставками. Исходные данные задавать с помощью датчика случайных чисел.

9. Составить две программы, которые реализуют алгоритмы ускоренной сортировки «пузырьком» и выбором. Исходные данные задавать с помощью датчика случайных чисел.

10. Составить две программы, которые реализуют алгоритмы ускоренной сортировки «пузырьком» и шейкером. Исходные данные задавать с помощью датчика случайных чисел.
11. Составить две программы, которые реализуют алгоритмы ускоренной сортировки «пузырьком» и слиянием. Исходные данные задавать с помощью датчика случайных чисел.
12. Составить две программы, которые реализуют алгоритмы ускоренной сортировки «пузырьком» и быстрой сортировки. Исходные данные задавать с помощью датчика случайных чисел.
13. Составить две программы, которые реализуют алгоритмы ускоренной сортировки «пузырьком» и сортировки Шелла. Исходные данные задавать с помощью датчика случайных чисел.
14. Составить две программы, которые реализуют алгоритмы ускоренной сортировки «пузырьком» и методом Боуза- Нельсона. Исходные данные задавать с помощью датчика случайных чисел.