**Содержание**

[ОБЩИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ 2](#_Toc124363595)

[ВНУТРЕННИЕ МЕТОДЫ СОРТИРОВКИ 2](#_Toc124363596)

[1. Сортировка прямым (простым) выбором 2](#_Toc124363597)

[2. Сортировка простым обменом 3](#_Toc124363598)

[3. Сортировка простыми вставками 5](#_Toc124363599)

[4. Сортировка Шелла 7](#_Toc124363600)

[5. Быстрая сортировка (сортировка Хоара) 9](#_Toc124363601)

[МЕТОДЫ ВНЕШНЕЙ СОРТИРОВКИ 11](#_Toc124363602)

[1. Простое слияние 11](#_Toc124363603)

[2. Естественное слияние 12](#_Toc124363604)

[3. Многопутевое сбалансированное слияние 13](#_Toc124363605)

[ИССЛЕДОВАНИЯ ЭФФЕКТИВНОСТИ МЕТОДОВ СОРТИРОВКИ 14](#_Toc124363606)

[Рост функций 14](#_Toc124363607)

[Время выполнения программы 15](#_Toc124363608)

[Рекомендации для выбора алгоритма сортировки 19](#_Toc124363609)

[ЛАБОРАТОРНАЯ РАБОТА 1 АЛГОРИТМЫ ВНУТРЕННЕЙ СОРТИРОВКИ 20](#_Toc124363610)

## ОБЩИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

**Сортировка -** это процесс расстановки элементов в некотором порядке. Сортировка двух записей состоит из сравнения их ключевых полей и определения, которое из них меньше; далее записи переставляются так, что запись с меньшим ключом ставится перед записью с большим ключом.

При рассмотрении алгоритмов сортировки будем рассматривать одномерный массив A целых чисел *(это не ограничивает область применения алгоритмов).*

Элементы массива можно сортировать:

• по возрастанию — каждый следующий элемент больше предыдущего:

А [1] < А [2] < ... < A[N];

• по неубыванию — каждый следующий элемент не меньше предыдущего:

А[1] <= А [2] <= ... <= A[N];

• по убыванию — каждый следующий элемент меньше предыдущего:

А[1] > А [2] > ... > A[N];

• по невозрастанию — каждый следующий элемент не больше предыдущего:

А [1] >= А [2] >= ... >= A [N].

Методы сортировки делятся на *внутренние* и *внешние*:

* Внутренние методы предполагают, что сортируемые данные целиком располагаются в оперативной памяти.
* Внешние методы используются для сортировки файлов, которые слишком велики, чтобы полностью поместиться в оперативной памяти.

### ВНУТРЕННИЕ МЕТОДЫ СОРТИРОВКИ

### Сортировка прямым (простым) выбором

Данный алгоритм обычно изучается первым среди множества алгоритмов сортировки, так как основан на естественном способе поиска максимального/минимального элемента в массиве: среди элементов массива A[1], ..., A[n] выбрать элемент с наибольшим значением и поменять местами с элементом A[n]. Повторять поиск для оставшихся n-1 элементов, n-2 элементов и т.д. до тех пор, пока не останется один минимальный элемент. Применение алгоритма естественно выглядит при работе со списками.

**Алгоритм сортировки прямым выбором**:

В приведенном алгоритме переменные имеют следующие назначения:

**j** - определяет количество сортируемых в данном проходе элементов;

**M** - максимальный элемент;

**k** - индекс максимального элемента.

1**. j=n.**

**2. M=A[1]; k=1; i=2.**

**3. Если M<A[i], то M=A[i]; k=i.**

**4. i=i+1.**

5. Если **i<=j**, то на шаг 3.

6. Иначе, поменять местами **A[i] и A[k].**

7. **j=j-1.**

8. Если j**>1,** то на шаг 2.

9. Иначе, "Массив отсортирован".

**Анализ**

Алгоритм сортировки прямым выбором "не учитывает" отсортированность массива, поэтому количество просмотров всегда постоянно. Оценим эффективность алгоритма через количество сравнений. При первом просмотре сравнений N — 1, при втором — N — 2, при последнем — 1. Общее количество С = N — 1+N — 2 + ...+ 1= N• (N- l)/2, или C=O(N2). Полные сведения об эффективности алгоритма приведены в таблице 1.

Таблица 1 - Показатели эффективности алгоритма сортировки прямым выбором

|  |  |  |
| --- | --- | --- |
|  | **Число  сравнений** | **Число  обменов** |
| **Лучший случай (минимальное значение)** | (n-1) | 3\*(n - 1) |
| **Средний случай (среднее значение)** | (n2 - n)/2 | (n - 1) |
| **Худший случай (максимальное значение)** | (n2 - n)/2 | n2/4+3\*(n – 1) |

### Сортировка простым обменом

**Сортировка** (сортировка стандартным обменом, сортировка прямым обменом сортировка методом пузырька) перемещает один элемент массива S в соответствующую позицию при каждом просмотре. При первом просмотре каждый элемент A[i] сравнивается с элементом A[i+1]  (1 <= i <= n-1) и при необходимости, если A[i]>A[i+1], меняется с ним местами. В результате наибольший элемент помещается в последнюю позицию. При втором просмотре выполняются те же действия, но уже не для n, а для (n-1) первых элементов массива. В результате следующий по величине элемент перемещается в предпоследнюю позицию. При третьем просмотре рассматриваются уже первые (n-2) элемента массива. Таким образом, для выполнения сортировки требуется максимально (n-1) просмотров. Во время каждого просмотра необходимо фиксировать наличие обменов. Если при очередном просмотре обменов не было, то массив уже упорядочен, сортировка заканчивается.

**Алгоритм сортировки прямым обменом:**

В приведенном алгоритме переменные имеют следующие назначения:

t - признак отсортированности массива (признак окончания сортировки);

j - количество (j+1) сортируемых в данном проходе элементов;

i - сравниваемые элементы (A[i] и A[i+1]).

1. t='Истина'; j=n-1.

2. Если t - 'Ложь', то 'Массив отсортирован'. Закончить.

3. t='Ложь'.

4. i=1.

5. Если A[i]>A[i+1], то t='Истина' и поменять местами A[i] и A[i+1]

6. i=i+1.

7. Если i<=j, то на шаг 5.

8. j=j-1. На шаг 2.

**Анализ**

Рассматриваемый алгоритм отличается от большинства алгоритмов сортировки тем, что "учитывает" отсортированность массива. Для этого используется логическая переменная (в приведенном алгоритме - переменная t), которая принимает значение "Истина", если при очередном проходе была хотя бы одна перестановка. Если же перестановок не было, то массив уже отсортирован, и переменная t будет иметь значение "Ложь".

Оценим эффективность алгоритма через количество сравнений. При сортировке методом "пузырька" выполняется N — 1 просмотров, на каждом i-м просмотре производится N — i сравнений. Общее количество С равно N\* (N—1)/2,или O(N2). Полные сведения об эффективности алгоритма приведены в таблице 2.

Таблица 2 - Показатели эффективности алгоритма сортировки простым обменом

|  |  |  |
| --- | --- | --- |
|  | **Число  сравнений** | **Число обменов** |
| **Лучший случай (минимальное значение)** | (n-1) | 0 |
| **Средний случай (среднее значение)** | ((n2)/2) - (3n/4) | (n2/4) |
| **Худший случай (максимальное значение)** | n\*(n-1)/2 | (n2/2) |

Если сравнивать быстродействие алгоритмов простого обмена и простого выбора, то последний в среднем работает быстрее. Это объясняется тем, что количество сравнений в обоих алгоритмах одинаково: среднее количество перестановок в алгоритме простого обмена - (n2)/4. Количество перестановок в алгоритме прямого выбора равно (n - 1), что значительно меньше.

На основе этого метода создан алгоритм с использованием рекурсии — "шейкер-сортировка". В нем учитывается тот факт, что от последней перестановки до конца массива будут находиться уже упорядоченные данные. Тогда просмотр имеет смысл делать не до конца массива, а до последней перестановки на предыдущем просмотре. Просмотр выполняется попеременно в двух направлениях с фиксированием нижней и верхней границ неупорядоченной части данных.

### Сортировка простыми вставками

Сортировка простыми вставками (прямого включения) основана на последовательной вставке элементов в уже упорядоченную последовательность.

Алгоритм сортировки заключается в следующем: на первом этапе упорядоченным считается один, первый элемент. Второй элемент либо меняется местами с первым, либо остается на своем месте. Далее 3-й элемент включается в нужное место уже упорядоченной последовательности из 2-х элементов, за ним 4-й и т.д. до n-го.

Пусть необходимо вставить i-й (i>=2) элемент в уже упорядоченную последовательность из i-1 элементов. Элемент A[i] последовательно сравнивается с каждым элементом A[k] (0<=k<=i-1) и, либо вставляется на свободное место, если A[i]>=A[k], либо элемент A[k] сдвигается на одну позицию вправо и процесс выполняется для элемента A[k-1]. Весь процесс вставки элемента заканчивается либо когда найден первый элемент A[k] такой, что A[k]<=A[i], либо достигнута левая граница упорядоченной последовательности (элемент A[i] меньше всех элементов упорядоченной последовательности).

Очевидным улучшением описанного выше процесса является установка барьера в нулевом элементе массива А (A[0]=A[i]), что позволит избежать проверки на выход за левую границу массива.

**Алгоритм сортировки простой** **вставки**

Назначения переменных:

A[i] - вставляемый элемент; A[0] - "барьер".

1. i=2.

2. A[0]=A[i].

3. j=i-1.

4. Если A[0]>=A[j], то на шаг 7.

5. A[j+1]=A[j].

6. j=j-1. На шаг 4.

7. A[j+1]=A[0].

8. i=i+1. Если i<=n, то на шаг 2.

9. Иначе, "Массив отсортирован".

**Анализ**

Алгоритм сортировки простыми вставками "не замечает" отсортированности массива, поэтому количество просмотров всегда постоянно и равно (n - 1). Если сравнивать быстродействие алгоритмов простыми вставками и прямого выбора, то первый в среднем работает быстрее. Это объясняется тем, что при примерно равном количестве перестановок количество сравнений в алгоритме сортировки прямым включением в среднем в два раза меньше. Алгоритм сортировки методом прямого выбора имеет сложность O(n2). Полные сведения об эффективности алгоритма приведены в таблице 3.

Таблица 3 - Показатели эффективности алгоритма сортировки простыми вставками

|  |  |  |
| --- | --- | --- |
|  | **Число  сравнений** | **Число обменов** |
| **Лучший случай (минимальное значение)** | (n-1) | 3\*(n-1) |
| **Средний случай (среднее значение)** | (n2  + n -2)/4 | (n2 +9n - 10)/4 |
| **Худший случай (максимальное значение)** | (n2 +n -4)/4 | (n2 +3n -4)/20 |

Рассмотренный алгоритм сортировки не относится к категории быстродействующих, поскольку единственный вид операции обмена, который она использует, выполняется над двумя соседними элементами, в связи с чем элемент может передвигаться вдоль массив; лишь на одно место за один раз. Например, если элемент с наименьшим значением ключа оказывается в конце массива, потребуется сделать *N* шагов, чтобы поместить его в надлежащее место.

Разработано несколько модификаций данного метода: сортировка бинарными вставками, сортировка Шелла.

Сортировка бинарными вставками. На момент вставки элемента с номером i элементы массива с номерами от 1 до i — 1 отсортированы. Выберем из них средний элемент (или один из двух средних) и сравним его с элементом А [i]. Если А [i] меньше этого элемента, то поиск места вставки следует продолжать в левой половине, иначе в правой половине отсортированного массива.

### Сортировка Шелла

Идея сортировки Шелла (сортировки с убывающим шагом) состоит в следующем. Исходный массив разбивается на n частей, в каждую из которых попадают элементы с шагом m, начиная от 0,1,...,m-1 соответственно. Каждая часть сортируется отдельно с использованием алгоритма вставок. Затем выбирается меньший шаг, и алгоритм повторяется. Шаг может быть выбран равным степеням 2, например 64,32,16,8,4,2,1. Последняя сортировка выполняется с шагом 1.

Сортировка методом Шелла представляет собой расширение метода вставок, быстродействие которого выше за счет обеспечения возможности обмена местами элементов, которые находятся далеко один от другого.

Возникает вопрос: какую последовательность шагов следует использовать? В общем случае на этот вопрос трудно найти правильный ответ. В литературе опубликованы результаты исследований различных последовательностей шагов; некоторые из них хорошо зарекомендовали себя на практике, однако наилучшую последовательность, по-видимому, отыскать не удалось. В общем случае на практике используются убывающие последовательности шагов, близкие к геометрической прогрессии в результате чего число шагов находится в логарифмической зависимости от размеров файлов. Например, если размер следующего шага равен примерно половине предыдущего, то для сортировки файла, состоящего из 1 миллиона элементов, потребуется примерно 20 шагов, если же такое соотношение примерно равно одной четвертой, то достаточно будет 10 шагов. Использование как можно меньшего числа шагов — это крайне важное требование.

Практический результат от обнаружения хорошей последовательности шагов, по-видимому, ограничен повышением быстродействия алгоритма на 25%, в то время сама проблема представляет собой довольно таки увлекательную головоломку.

Последовательность шагов **1 4 13 40 121 364 093 3280 9841 ... .**Она просто вычисляется (начав с 1, получить значение следующего шага, множив предыдущее значение на 3 и добавив 1) и обеспечивает реализацию сравнительно эффективной сортировки даже в случае относительно больших файлов.

Многие другие последовательности шагов позволяет получить еще более эффективную сортировку, однако довольно трудно превзойти эффективность более чем на 20% даже в случае сравнительно больших значений *N.* Одной из таких последовательностей является **1 8 23 77 281 1073 193 16577 ...,** т.е. последовательность 4i+1 +3\*2i+ для *i* > 0. Можно доказать, что приведенная последовательность обеспечивает повышенное быстродействие для самых трудных случаев сортировки.

С другой стороны, существуют и плохие последовательности шагов: например,

**1 2 4 8 16 32 64 128 256 512 1024 2048 ...** (первая последовательность шагов, предложенная Шеллом еще в 1959 г. скорее всего, служит причиной низкой эффективности сортировки, поскольку элементы на нечетных позициях не сравниваются с элементами на четных позициях вплоть до последнего прохода. Этот эффект заметен на файлах с произвольной организацией, и он становится катастро­фическим в наихудших случаях: эффективность метода резко снижается и время вы­полнения сортировки становится пропорциональным квадрату *N,* если, например, по­ловина элементов файла с меньшими значениями находится в четных позициях, а другая половина элементов (с большими значениями) — в нечетных позициях.

**Свойство 1.** *Сортировка методом Шелла выполняет менее N(h — l)(k — l)/g операций сравнения при g-сортировке h- и k-упорядоченного файла при условии, что h и k взаим­но просты.*

**Свойство 2.** *Сортировка методом Шелла выполняет менее О (N* 3/2) *операций сравне­ния для последовательности шагов* **1 4 13 40 121 364 1093 3280 9841...**

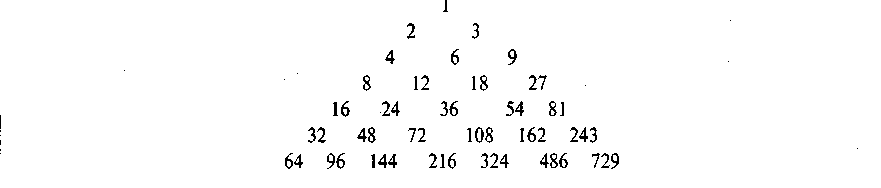
Для больших шагов, когда имеются *h* подфайлов размером *N/h,* в наихудшем слу­чае расходы составляют примерно *N2/h.* При малых шагах из свойства 1 следует, что стоимость составляет приблизительно *Nh.* Все зависит от того, насколько ус­пешно удается вписаться в эти границы на каждом шаге. Это справедливо для каждой относительно простой последовательности, возрастающей экспонен­циально.

**Свойство 3.** *Сортировка методом Шелла выполняет менее О (N 4/3) операций срав­нения для последовательности шагов* **1 8 23 77 281 1073 4193 16577...**

Последовательности шагов, которые рассматривались до сих пор, эффективны в силу того, что следующие один за другим элементы последовательности взаимно просты. Другое семейство последовательностей шагов эффективно именно благодаря тому, что такие элементы *не являются* взаимно простыми.

**Свойство 4.** *Сортировка методом Шелла выполняет менее 0(N(* logN)2) *операций сравнения для последовательности* шагов **1 2 3 4 6 9 8 12 18 27 16 24 36 54 81...**

Рассмотрим треугольник, составленный из шагов, в котором каждое число в два раза больше, чем число выше и правее, и в три раза больше, чем число выше и

Если мы используем эти числа снизу вверх и справа налево как последователь­ность шагов в рамках сортировки методом Шелла, то каждому шагу *х* в нижнем

ряду предшествуют значения *2х* и *Зх,* так что каждый подфайл оказывается 2-упорядочен и 3-упорядочен, при этом ни один элемент не передвигается больше, чем на одну позицию в процессе всей сортировки!

Число шагов из треугольника, которое меньше N по величине, и подавно будет меньше (Log2 N)2.

**Эмпирические исследования последовательностей шагов сортировки методом Шелла**

Сортировка методом Шелла выполняется в несколько раз быстрее по сравнению с другими элементарными методами сортировки даже в тех случаях, когда шаги являются степенями 2, в то же время некоторые специальные виды последовательностей шагов позволяют увеличить ее быстродействие в 5 и более раз. Три лучших последовательности, приведенные в данной таблице, существенно различаются по положенным в их основу принципам. Сортировка методом Шелла вполне пригодна для практических приложений даже в случае файлов больших размеров. По эффективности она *намного* превосходит методы выбора и вставок, равно как и пузырьковую сортировку.

### Быстрая сортировка (сортировка Хоара)

Опишем идею метода. В исходном массиве А выбирается некоторый элемент X (его называют "барьерным"). Нашей целью является запись Х "на свое место" в массиве, пусть это будет место k, такое, чтобы слева от Х были элементы массива, меньшие или равные X, а справа — элементы массива, большие X, т.е. массив А будет иметь вид:

(A[1], A[2], ..., A[*k* - 1] ), A[*k*] = (X), (A[*k* + 1], ..., A[n] ).

В результате элемент A [k] находится на своем месте и исходный массив А разделен на две неупорядоченные части, барьером между которыми является элемент А [*k*]. Далее требуется отсортировать полученные части тем же методом до тех пор, пока в каждой из частей массива не останется по одному элементу, то есть пока не будет отсортирован весь массив.

Алгоритм реализуется при помощи рекурсии, оформим его в виде процедуры **Сортировка** (iLo, iHi).

1. Lo=iLo, Hi=iHi, Mid=A[(Lo+Hi) div 2];

2. Если A[Lo] < Mid, то Lo=Lo+1, на шаг 2

3. Если A[Hi] > Mid, то Hi=Hi-1, на шаг 3

4. Если Lo > Hi, на шаг 8

5. Поменять местами A[Lo] и A[Hi]

6. Lo=Lo+1, Hi=Hi-1

7. Если Lo <= Hi, на шаг 3

8. Если Hi > iLo, Вызов **Сортировка**(A,iLo,Hi)

9. Если Lo < iHi, Вызов **Сортировка**(A,Lo,iHi)

10. Возврат (на предыдущий уровень рекурсии)

**Анализ**

Если количество элементов в массиве не многим меньше максимального их значения, то в данном случае наиболее эффективным и по быстродействию, и по простоте. В среднем требует только около N log N операций для того, чтобы отсортировать N элементов, в худшем случае он требует N2 операций. Эффективность метода:

С = N + 2 • (N/2) + 4 • (N/4) + ... + N • (N/N) = О (N • logN).

И преимущества и недостатки метода объясняются рекурсивной организацией:

+ Основные преимущества состоят в том, что он точечный (использует лишь небольшой дополнительный стек), имеет *экстремально* короткий внутренний цикл.

- Недостатки состоят в том, что реализация очень затруднена, когда рекурсия недоступна. Рекурсия предъявляет повышенные требования к квалификации программиста, небольшая ошибка в реализации, которая легко может пройти незамеченной, может привести к тому, что алгоритм будет работать очень плохо на некоторых файлах.

## МЕТОДЫ ВНЕШНЕЙ СОРТИРОВКИ

Внешние сортировки, как правило, применяются для упорядочения большого количества элементов, расположенных на внешних носителях. При этом доступ к элементам осуществляется последовательно.

Большинство алгоритмов внешних сортировок используют *слияние*.

Слияние - объединение двух или более последовательностей в одну, упорядоченную некоторым образом, с помощью повторяющегося выбора из доступных в данный момент элементов.

### Простое слияние

Рассмотрим *двухфазную сортировку простым слиянием*.

Пусть дана некоторая последовательность **A**. Для работы алгоритма потребуются 2 вспомогательные последовательности **B** и **C**.

Фаза 1, проход 1 – по 1 элементу

А: 7 15 2 3 10 4 9 12 11 5 1

В: 7 2 10 9 11 1

С: 15 3 4 12 5

Фаза 2, проход 1

А: 7 15 2 3 4 10 9 12 5 11 1

Фаза 1, проход 2– по 2 элемента

А: 7 15 2 3 4 10 9 12 5 11 1

В: 7 15 4 10 5 11

С: 2 3 9 12 1

Фаза 2, проход 2

А: 2 3 7 15 4 9 10 12 1 5 11

Суть алгоритма заключается в чередующемся выполнении двух его фаз: фазы распределения и фазы слияния элементов. В первой фазе элементы "распределяются" определённым образом из последовательности **A** в последовательности **B** и **C**. Во второй фазе элементы "сливаются" из последовательностей **B** и **C** в последовательность **A**.

В результате каждого такого прохода, состоящего из двух фаз, упорядоченность последовательности **A** будет возрастать: в результате первого прохода будет упорядочена каждая пара элементов, в результате второго прохода будет упорядочена каждая четвёрка элементов и т.д. Работа алгоритма заканчивается тогда, когда последовательность **A** будет упорядочена полностью.

Число проходов, необходимых для упорядочения последовательности, логарифмически зависит от количества элементов. Например, для упорядочения 1.000 элементов потребуется 10 проходов, для упорядочения 1.000.000 элементов потребуется около 20 проходов, для 1.000.000.000 элементов - около 30 проходов и т.д.

**Анализ сортировки слияния**. Поскольку на каждом проходе р удваивается и сортировка заканчивается при р >= n,то всего требуется [log *n*]проходов. На каждом проходе по определению копируются по одному разу все *п* элементов. Поэтому общее число пересылок:

**M=n\*[log n ]**

Число сравнений ключей С даже меньше *М,* поскольку при копировании остатков никаких сравнений не производится.

Данный метод сортировки обычно употребляются в ситуациях, где используются внешние запоминающие устройства, то затраты на операции пересылки на несколько порядков превышают затраты на сравнения. Поэтому детальный анализ числа сравнений особого практического интереса не представляет.

Алгоритм сортировки слиянием выдерживает сравнение с усовершенствованными методами внутренней сортировки, разбиравшимися ранее. Однако, хотя здесь относительно высоки затраты на работу с индексами, самым существенным недостатком является необходимость работать с памятью размером *2N.* Поэтому сортировка слиянием для массивов, т. е. для данных, размещаемых в оперативной памяти, используется редко.

### Естественное слияние

*Сортировка естественным слиянием* является другим усовершенствованием сортировки простым слиянием. В ней вводится понятие *серии элементов* как последовательности элементов, в которой каждый последующий элемент больше (меньше) предыдущего.

Фаза 1, проход 1 – распределение серий

А: 7 15 2 3 10 4 9 12 11 5 1

В: 7 15 4 9 12 5

С: 2 3 10 11 1

Фаза 2, проход 1 – слияние

А: 2 3 7 10 11 15 1 4 5 12 5

Фаза 1, проход 2 – распределение серий

В: 2 3 7 10 11 15 5

С: 1 4 9 12

Фаза 2, проход 2– слияние

А: 1 2 3 4 7 9 10 11 12 15 5

В сортировке простым слиянием серии составлялись без учёта первоначального расположения элементов в исходной последовательности. Сначала длина серий была равна 1, затем 2, 4 и т.д. Естественное слияние учитывает первоначальное расположение элементов в последовательности. В результате этого удаётся уменьшить число проходов, необходимых для упорядочения последовательности. В остальном же сортировка естественным слиянием аналогична простому слиянию: серии элементов из последовательности **A** распределяются поочерёдно в последовательности **B** и **C**. Затем элементы "сливаются" обратно в последовательность **A**, образуя при этом серии большей длины, чем первоначально, и т.д.

### Многопутевое сбалансированное слияние

В предыдущих алгоритмах внешних сортировок распределение элементов происходило в две последовательности. Во *многопутевом слиянии* число таких последовательностей (путей) может быть любым целым числом *n* ³ 2. Чем больше путей используется в сортировке, тем меньше проходов понадобится для полного упорядочения последовательности. Многопутевое *сбалансированное* слияние использует серии естественной длины, и каждый проход состоит из одной фазы.

Рассмотрим работу алгоритма на примере. Число путей = 3. Предварительный проход 1 – распределение серий

S1: 7 15 2 3 10 4 9 12 11 5 1

S2:

S3:

S4: 7 15 11

S5: 2 3 10 5

S6: 4 9 12 1

Проход 2

S1: 1 2 3 4 7 9 10 12 15

S2: 1 5 11

Проход 3

S4: 1 2 3 4 5 7 9 10 11 12 15

### ИССЛЕДОВАНИЯ ЭФФЕКТИВНОСТИ МЕТОДОВ СОРТИРОВКИ

### Рост функций

Для большинства алгоритмов *главным параметром (primary parameter)* является N, который оказывает существенное влияние на время их выполнения. Параметр N может быть степенью полинома, размером обрабатываемого файла, количеством символов в строке или некоторой другой абстрактной мерой размера рассматриваемой задачи: чаще всего, он прямо пропорционален величине обрабатываемого набора данных. Когда таких параметров несколько (> 1), анализ сводиncz к одному параметру, задавая его как функцию от других параметров или рассматривая одновременно только один параметр (считая остальные постоянными). Таким образом, ограничиваем рассмотрение только одним параметром *N* без потери общности.

Целью является выражение требований к ресурсам, предъявляемых разрабатываемыми программами (как правило, это время выполнения) в зависимости от *N* с использованием максимально простых математических формул, которые обеспечивают точность расчетов для больших значений параметров. Изучаемые алгоритмы обычно имеют время выполнения, пропорциональное одной из следующих функций:

* **1** Большинство инструкций большинства программ выполняется один или максимум несколько раз. Если все инструкции программы обладают этим свой­ством, мы говорим, что время выполнения программы *постоянно (constant).*
* **log N** Когда время выполнения программы описывается *логарифмической (logarithmic)* зависимостью, программа немного утрачивает быстродействие с ростом *N.* Такое время выполнения обычно характерно для программ, которые сводят крупную задачу к некоторой последовательности задач меньшего размера, уменьшая на каждом шаге размер задачи на некоторую небольшую часть. В интересующем диапазоне будем рассматривать время выполнения как величину, не превосходящую некоторое большое постоянное значение. Основание логарифма изменяет это значение, но ненамного: когда *N —* тысяча, log *N* равно 3, если основание равно 10, либо примерно 10, если основание равно 2; когда *N* равно миллиону, значения log N всего лишь удвоится. При удвоении N значение *logN* возрастет на постоянную величину, а удваивается лишь, когда *N* увеличится до *N2.*
* **N**Когда время выполнения программы *линейно (linear),* это обычно означает, что каждый элемент ввода подвергается небольшой обработке. Когда *N* равно миллиону, такого же порядка и время выполнения алгоритма. Когда *N*  удваивается, то же происходит и со временем выполнения. Эта ситуация оптимальна для алгоритма, который должен обработать *N* вводов (или произве­сти *N* выводов).
* ***NlogN*** Время выполнения, пропорциональное .N log N имеет место, когда алгоритм решает задачу, разбивая ее на подзадачи меньших размеров, решая их неза­висимо и затем объединяя решения. Из-за отсутствия подходящего прилага­тельного *("линерифмический" "linerithmic")* мы просто говорим, что время выполнения такого алгоритма равно *N* log N. Когда *N* равно 1 миллион, N log N возрастает примерно до 20 миллионов. Когда *N* удваивается, то вре­мя выполнения возрастает более чем вдвое (но не намного более).
* **N 2** Когда время выполнения алгоритма является *квадратичным (quadratic),* он полезен для практического использования применительно к небольшим зада­чам. Квадратичное время выполнения обычно характерно для алгоритмов, которые обрабатывают все элементы данных парами (возможно, в цикле двойного уровня вложения). Когда N равно одной тысяче, время выполне­ния равно одному миллиону. Когда *N* удваивается, время выполнения увели­чивается в четыре раза.
* **N3**  Аналогичный алгоритм, обрабатывающий элементы данных тройками (возможно, в цикле тройного уровня вложения), имеет *кубическое (cubic)* время выполнения и практически применим лишь для решения малых задач. Когда *N* равно 100, время выполнения равно 1 миллиону. Когда *N* удваивается, время выполнения увеличивается в восемь раз.
* ***2N*** Лишь немногие алгоритмы с *экспоненциальным (exponential)* временем выполнения имеют практическое применение, хотя такие алгоритмы возникают ес­тественным образом при попытках решения задачи "в лоб". Когда *N* равно 20, время выполнения равно 1 миллиону. Когда *N* удваивается, время выполнения увеличивается в четыре раза!

### Время выполнения программы

Время выполнения конкретной программы, скорее всего, будет некоторой константой, умноженной на одно из перечисленных выше выражений *(главный член — leading term)* плюс некоторые слагаемые меньшего порядка. Значения постоянного коэффициента и остальных слагаемых зависят от результатов анализа и деталей реализации. В грубом приближении коэффициент при главном члене связан с количеством инструкций во внутреннем цикле: на любом уровне разработки алгоритма разумно со­кратить количество таких инструкций. Для больших *N* доминирует главный член, для малых N или в случае тщательно разработанных алгоритмов свой вклад вносят и другие слагаемые, поэтому сравнение алгоритмов становятся более сложным. В большинстве случаев мы будем называть время выполнения программ просто "линейным", "кубическим" и т.д.

В итоге, чтобы уменьшить общее время выполнения программы, минимизируем количество инструкций во внутреннем цикле. Каждую инструкцию, необходимо подвергнуть исследованию: нужна ли она вообще? Существует ли более эффективный способ выполнить ту же задачу? Некоторые программисты считают, что автоматические инструменты, содержащиеся в современных компиляторах, могут создавать наилучший машинный код; другие утверждают, что наилучшим способом является написание внутренних циклов вручную на машинном языке, или ассемблере.

**Перевод секунд**

Огромная разница между такими числами, как 104 и 108, становится более очевидной, когда их применяют для измерения промежутков времени, а затем переводят в привычные единицы измерения времени;(210 ~= 103 то таблицей перевода можно воспользоваться и для перевода степеней 2 в привычные единицы времени; например, 232 секунд составляет примерно 124 года).

Таблица 4 - Перевод секунд в крупные единицы измерения времени

|  |  |  |
| --- | --- | --- |
| **Степень** | **Секунды** | **Единицы измерения времени** |
| 2 | 102 | 1,7 минуты |
| 4 | 104 | 2,8 часа |
| 5 | 105 | 1,1 дня |
| 6 | 106 | 1,6 недели |
| 7 | 107 | 3,8 месяца |
| 8 | 108 | 3,1 года |
| 9 | 109 | 3,1 десятилетия |
| 10 | 1010 | 3,1 столетия |
| 11 | 1011 | никогда |

Во многих приложениям “единственным шансом” решить задачу обработки большого объема данных остается использование эффективного алгоритма. В таблице 5 приведено минимальное количество времени, необходимое для решения задач размером 1 миллион и 1 миллиард с использованием линейных алгоритмов (N), алгоритмов с зависимостью N log N и квадратичных алгоритмов (N2) на компьютерах с быстродействием 1 миллион, 1 миллиард и 1 триллион операций в секунду. Быстрый алгоритм помогает существенно ускорить решение задачи на медленной машине, однако “быстрая” машина не сможет выручить, когда используется медленный алгоритм.

Таблица 5 - Минимальное количество времени

| **Количество операций  в секунду** | **Размер задач 1 миллион** | | | **Размер задачи 1 миллиард** | | |
| --- | --- | --- | --- | --- | --- | --- |
| **NN** | **N log N** | **N2** | **N** | **N log N** | **N2 N2** |
| 106 | секунд | секунд | недель | часов | часов | никогда |
| 109 | мгновенно | мгновенно | часов | секунд | секунд | десятилетий |
| 1012 | мгновенно | мгновенно | секунд | мгновенно | мгновенно | недель |

В таблице 6 сравниваются значения, принимаемые рядом функций, с которыми придется часто сталкиваться при анализе алгоритмов. Очевидно, доминирующей является квадратичная функция, особенно на больших значениях N, а на малых значениях N различие между функциями оказываются не такими, как можно было бы ожидать. Например, N3/2 больше, чем N lg2 N, на очень больших значениях N, однако на небольших N наблюдается обратная картина. Точное время выполнения алгоритма может быть выражено в виде линейной комбинации этих функций. Можно легко отделить быстрые алгоритмы от медленных из-за огромной разницы, например, междуlg N и N или N и N2, тем не менее, различия между двумя быстрыми алгоритмами может потребовать более тщательных исследований.

Таблица 6 - Значения часто встречающихся функций

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **lg N** | **√N** | **N** | **N lg N** | **N (lg N)2** | **N3/2** | **N2** |
| 3 | 3 | 10 | 33 | 110 | 32 | 100 |
| 7 | 10 | 100 | 664 | 4414 | 1000 | 10000 |
| 10 | 32 | 1000 | 9966 | 99317 | 31623 | 1000000 |
| 13 | 100 | 10000 | 132877 | 1765633 | 100000 | 100000000 |
| 17 | 316 | 100000 | 1990964 | 27588016 | 31622777 | 10000000000 |
| 20 | 1000 | 1000000 | 19931569 | 397267426 | 1000000000 | 1000000000000 |

При анализе алгоритмов можно воспользоваться еще несколькими функциями. Например, алгоритм с N2 входными данными, имеющий время выполнения N 3, лучше рассматривать как алгоритм с зависимостью N3/2. Кроме того, алгоритмы, допускающие разбиение на две подзадачи, имеют время выполнения, пропорциональное N log2 N. Из таблицы 6 видно, что обе эти функции ближе к N log N, чем к N 2.

Логарифмическая функция играет особую роль при разработке и анализе алгоритмов, поэтому ее стоит рассмотреть подробнее. Поскольку часто приходится давать оценку аналитическим результатам, в которых опущен постоянный множитель, мы будем пользоваться записью "logN", опуская основание. Изменение основания логарифма с одной константы на другую меняет значение лишь на постоянный множитель, однако в определенных контекстах используем конкретные значения оснований логарифмов. В математике настолько важным является понятие натуральный логарифм (natural logarithm) с основанием е = 2.71828..., что широкое распространение получило следующее сокращение: log e N = In N. В вычислительной технике очень важен двоичный логарифм (binary logarithm) (т.е. по основанию 2), поэтому используется сокращение log 2 N = Ig N.

Иногда приходится вычислять логарифмы, особенно в отношении больших чи­сел. Например, lg lg 2256 = lg 256 = 8. Из этого примера должно быть понятно, что в большинстве практических случаев lg IgN рассматривается как константа, поскольку значение этого выражения достаточно мало даже для очень больших N.

### Рекомендации для выбора алгоритма сортировки

* Если сортируется небольшой объем данных (N < 100), то рекомендуется выбирать простой метод сортировки, так как реализация сложных алгоритмов занимают больше кода и они менее эффективны при малом количестве сортируемых элементов.
* Если сортируются наборы большого размера, то рекомендуется использовать специальные таблицы с помощью которых осуществляется доступ к самим элементам (например, это могут быть указатели на элементы или их порядковые номера), причем ключ, по которому сортируются данные может входить или не входить в данные такой таблицы. После сортировки таблицы можно или переставить исходные данные по таблице, или оставить все как есть и осуществлять дальнейший доступ к данным по уже отсортированной таблице.
* Если сортируются данные в файле, то необходимо учитывать, что большая часть времени будет тратиться на чтение, запись элемента и перемещение по файлу. В такой ситуации методы вставок являются не эффективными, так как требуют большое число перестановок. (Кстати, при внутренней сортировке время сравнения и перестановки двух элементов практически одинаковое.) Так же возможно считывать небольшие участки данных в память, там их сортировать и записывать обратно в память, после чего файл будет уже частично отсортирован и останется довести дело до конца каким-либо методом, предпочитающим частично отсортированные данные.
* Если сортируются списковые структуры данных, то необходимо выбирать метод сортировки наиболее подходящий для данной структуры. Например, если сортируется связный список, то для него будет эффективен метод вставок. Надо преобразовать соответствующий алгоритм таким образом, чтобы вначале находилось место для вставки очередного элемента, а затем уже вставлять его.
* Рекомендуется выбирать алгоритм сортировки с учетом предполагаемого входного состояния данных (является таблица частично отсортированной, отсортированной в обратном порядке и т.д.) и рекомендаций приведенных почти к каждому алгоритму.
* Если после сортировки надо получить новую таблицу содержащую отсортированные данные (имеется ввиду, что в памяти хранится две таблицы: исходная и отсортированная), то неплохие результаты могут дать алгоритмы основанные на выборках элементов.

# ЛАБОРАТОРНАЯ РАБОТА 1 АЛГОРИТМЫ ВНУТРЕННЕЙ СОРТИРОВКИ

ЦЕЛЬ РАБОТЫ: освоение основных методов упорядочивания данных, расположенных в оперативной памяти.

ЗАДАНИЕ: Составить программу, состоящую из следующих пунктов:

1. Сортировка простыми вставками;
2. Сортировка простым обменом;
3. Сортировка простым выбором;
4. Улучшенная сортировка **\***;
5. Быстрая сортировка.
6. Характеристики сортировок;
7. Выход.

**\***п. 4 меню Улучшенная сортировка определяется в зависимости от номера варианта = *остаток от целочисленного деления на 7*.

Улучшенные / Специальные сортировки:

Сортировка простым обменом (пузырька):

1. Шейкерная (перемешиванием)
2. Гномья
3. Расческа

Сортировка простым выбором

1. Пирамидальная

Сортировка простыми вставками

1. Шелла (с убывающим шагом);
2. Сортировка бинарными вставками

Специальная сортировка

1. Карманная (блочная, корзинная) сортировка

В пп. 1- 5 продемонстрировать работу соответствующей сортировки с различными типами данных.

В п.6 для заданного типа данных и количества подсчитать время сортировки, число сравнений и перестановок (сдвигов ) для каждого из алгоритмов.

ВХОДНЫЕ ДАННЫЕ: чтение данных осуществляется из файла;

Выбор типа данных:

1. массив записей; (количество 10, 25)
2. массив чисел (количество 500, 1000, 5000).

Структура массива записей и диапазон чисел приведены в Приложении 1.

ВЫХОДНЫЕ ДАННЫЕ: В пп. 1- 5 результат выводится на экран;

В п. 6 результат записывается в файл и считывается из него на экран в виде таблицы:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **вид сортировки** | **количество элементов** | **количество сравнений** | **количество перестановок** | **время выполнения** |

**Примеры организации выходных данных**

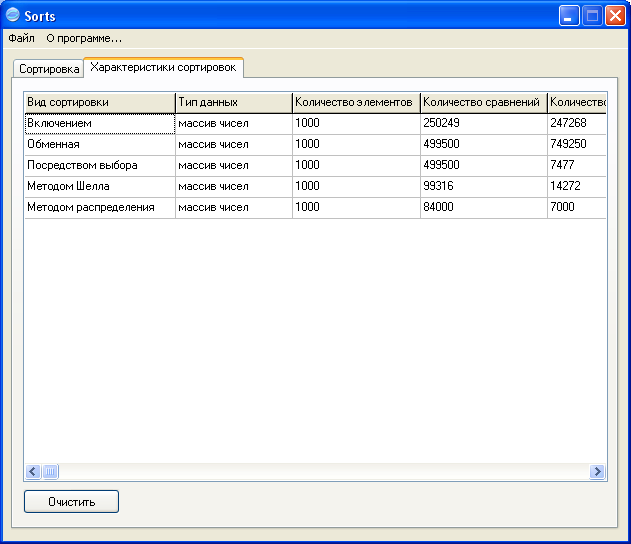


Рисунок 1 - Таблица характеристик сортировки в консоли

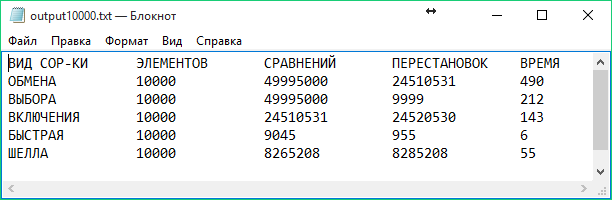


Рисунок 2 - Таблица характеристик сортировки в выходном файле