КЛАССИФИКАЦИЯ АЛГОРИТСОВ СОРТИРОВКИ

http://dander.ru/gos/22.html#__RefHeading__933_53677018

Определение. Алгоритм сортировки — это алгоритм для упорядочения элементов в списке. В случае, когда элемент списка имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма.

Основания классификации алгоритмов сортировки

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

Естественность поведения — эффективность метода при обработке уже упорядоченных, или частично упорядоченных данных. Алгоритм ведёт себя естественно, если учитывает эту характеристику входной последовательности и работает лучше.

Использование операции сравнения. Алгоритмы, использующие для сортировки сравнение элементов между собой, называются основанными на сравнениях. Минимальная трудоемкость худшего случая для этих алгоритмов составляет , но они отличаются гибкостью применения. Для специальных случаев (типов данных) существуют более эффективные алгоритмы.

Ещё одним важным свойством алгоритма является его сфера применения. Здесь основных типов упорядочения два – внутренняя и внешняя сортировки.

1. Внутренняя сортировка 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 архитектурах персональных компьютеров широко применяется подкачка и кэширование памяти. Алгоритм сортировки должен хорошо сочетаться с применяемыми алгоритмами кэширования и подкачки.

2. Внешняя сортировка оперирует с запоминающими устройствами большого объёма, но с доступом не произвольным, а последовательным (упорядочение файлов), т. е. в данный момент мы 'видим' только один элемент, а затраты на перемотку по сравнению с памятью неоправданно велики. Это накладывает некоторые дополнительные ограничения на алгоритм и приводит к специальным методам упорядочения, обычно использующим дополнительное дисковое пространство. Кроме того, доступ к данным на носителе производится намного медленнее, чем операции с оперативной памятью. Доступ к носителю осуществляется последовательным образом: в каждый момент времени можно считать или записать только элемент, следующий за текущим. Объём данных не позволяет им разместиться в ОЗУ.

Также алгоритмы классифицируются по:
- потребности в дополнительной памяти или её отсутствии
- потребности в знаниях о структуре данных, выходящих за рамки операции сравнения, или отсутствии таковой

Алгоритмы устойчивой сортировки

1. Сортировка пузырьком (англ. Bubble sort) — сложность алгоритма: O(n2); для каждой пары индексов производится обмен, если элементы расположены не по порядку.

2. Сортировка перемешиванием (Шейкерная, Cocktail sort, bidirectional bubble sort) — Сложность алгоритма: O(n2)

3. Гномья сортировка — имеет общее с сортировкой пузырьком и сортировкой вставками. Сложность алгоритма — O(n2).

4. Сортировка вставками (Insertion sort) — Сложность алгоритма: O(n2); определяем где текущий элемент должен находиться в упорядоченном списке и вставляем его туда

5. Блочная сортировка (Корзинная сортировка, Bucket sort) — Сложность алгоритма: O(n); требуется O(k) дополнительной памяти и знание о природе сортируемых данных, выходящее за рамки функций "переставить" и «сравнить».

6. Сортировка подсчётом (Counting sort) — Сложность алгоритма: O(n+k); требуется O(n+k) дополнительной памяти, существует 3 варианта).

7. Сортировка слиянием (Merge sort) — Сложность алгоритма: O(n log n); требуется O(n) дополнительной памяти; выстраиваем первую и вторую половину списка отдельно, а затем — сливаем упорядоченные списки.

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

Алгоритмы неустойчивой сортировки

Сортировка выбором (Selection sort) — Сложность алгоритма: O(n2); поиск наименьшего или наибольшего элемента и помещения его в начало или конец упорядоченного списка

Сортировка Шелла (Shell sort) — Сложность алгоритма: O(n log2 n); попытка улучшить сортировку вставками

Сортировка расчёской (Comb sort) — Сложность алгоритма: O(n log n)

Пирамидальная сортировка (Сортировка кучи, Heapsort) — Сложность алгоритма: O(n log n); превращаем список в кучу, берём наибольший элемент и добавляем его в конец списка

Плавная сортировка (Smoothsort) — Сложность алгоритма: O(n log n)

Быстрая сортировка (Quicksort), в варианте с минимальными затратами памяти — Сложность алгоритма: O(n log n) — среднее время, O(n2) — худший случай; широко известен как быстрейший из известных для упорядочения больших случайных списков; с разбиением исходного набора данных на две половины так, что любой элемент первой половины упорядочен относительно любого элемента второй половины; затем алгоритм применяется рекурсивно к каждой половине. При использовании O(n) дополнительной памяти, можно сделать сортировку устойчивой.

Introsort — Сложность алгоритма: O(n log n), сочетание быстрой и пирамидальной сортировки. Пирамидальная сортировка применяется в случае, если глубина рекурсии превышает log(n).

Patience sorting — Сложность алгоритма: O(n log n + k) — наихудший случай, требует дополнительно O(n + k) памяти, также находит самую длинную увеличивающуюся подпоследовательность

Stooge sort — рекурсивный алгоритм сортировки с временной сложностью .

Поразрядная сортировка — Сложность алгоритма: O(n·k); требуется O(k) дополнительной памяти.

Цифровая сортировка — то же, что и Поразрядная сортировка.

Непрактичные алгоритмы сортировки

Bogosort — O(n·n!) в среднем. Произвольно перемешать массив, проверить порядок.

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

Глупая сортировка (Stupid sort) — O(n3); рекурсивная версия требует дополнительно O(n2) памяти

Bead Sort — O(n) or O(√n), требуется специализированное аппаратное обеспечение

Блинная сортировка (Pancake sorting) — O(n), требуется специализированное аппаратное обеспечение

Алгоритмы, не основанные на сравнениях:

Блочная сортировка (Корзинная сортировка, Bucket sort)

Лексикографическая или поразрядная сортировка (Radix sort)

Сортировка подсчётом (Counting sort)

Прочие алгоритмы сортировки
Топологическая сортировка

Внешняя сортировка.

СПИСОК АЛГОРИТМОВ СОРТИРОВКИ.

Сортировка простыми обменами, сортиро́вка пузырько́м (англ. bubble sort). Простой алгоритм сортировки. Для понимания и реализации этот алгоритм — простейший, но эффективен он лишь для небольших массивов. Сложность алгоритма: $O(n^2)$. Алгоритм считается учебным и практически не применяется вне учебной литературы, вместо него на практике применяются более эффективные алгоритмы сортировки. В то же время метод сортировки обменами лежит в основе некоторых более совершенных алгоритмов, таких как шейкерная сортировка, сортировка Шелла и быстрая сортировка.

Алгоритм состоит в повторяющихся проходах по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. При проходе алгоритма, элемент, стоящий не на своём месте, «всплывает» до нужной позиции как пузырёк в воде, отсюда и название алгоритма.

Псевдокод
Вход: массив A, состоящий из элементов A[1], A[2], ..., A[n-1], A[n]
t := истина
цикл пока t:
t := ложь
цикл для j = 1, 2, ..., n − 1:
если A[j] > A[j+1], то:
обменять местами элементы A[j] и A[j+1]
t := истина

Булевая переменная t используется для того, чтобы определить, был ли произведён хотя бы один обмен на очередной итерации внешнего цикла. Алгоритм останавливается, когда таких обменов не было.

Можно показать, что для сортировки требуется сделать не более n − 1 итераций внешнего цикла, поэтому в некоторых реализациях внешний цикл всегда выполняется ровно n − 1 или n раз, и не отслеживается, были ли обмены или нет на каждой итерации.

Алгоритм можно немного улучшить следующими способами.

Внутренний цикл можно выполнять для j = 1,2,...,n − i, где i — номер итерации внешнего цикла (нумерация с единицы), так как на i-й итерации последние i элементов массива уже будут правильно упорядочены.

Внутренний цикл можно модифицировать так, чтобы он поочерёдно просматривал массив то с начала, то с конца. Модифицированный таким образом алгоритм называется сортировкой перемешиванием или шейкерной сортировкой.

Сортировка перемешиванием (Шейкерная сортировка) (англ. Cocktail sort). Разновидность пузырьковой сортировки. Анализируя метод пузырьковой сортировки можно отметить два обстоятельства.

Во-первых, если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, ее можно исключить из рассмотрения.

Во-вторых, при движении от конца массива к началу минимальный элемент “всплывает” на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо.

Эти две идеи приводят к следующим модификациям в методе пузырьковой сортировки. Границы рабочей части массива (т.е. части массива, где происходит движение) устанавливаются в месте последнего обмена на каждой итерации. Массив просматривается поочередно справа налево и слева направо.

Лучший случай для этой сортировки — отсортированный массив ($O(n)$), худший — отсортированный в обратном порядке ($O(n^2)$).

Гномья сортировка (англ. Gnome sort). Алгоритм сортировки, похожий на сортировку вставками, но в отличие от последней перед вставкой на нужное место происходит серия обменов, как в сортировке пузырьком. Название происходит от предполагаемого поведения садовых гномов при сортировке линии садовых горшков, описанного на странице Дика Груна Гномья сортировка (https://dickgrune.com/Programs/gnomesort.html).

Гномья сортировка основана на технике, используемой обычным голландским садовым гномом (нидерл. tuinkabouter). Это метод, которым садовый гном сортирует линию цветочных горшков. По существу он смотрит на следующий и предыдущий садовые горшки: если они в правильном порядке, он шагает на один горшок вперёд, иначе он меняет их местами и шагает на один горшок назад. Граничные условия: если нет предыдущего горшка, он шагает вперёд; если нет следующего горшка, он закончил.

Алгоритм концептуально простой, не требует вложенных циклов. Время работы . На практике алгоритм может работать так же быстро, как и сортировка вставками.

Алгоритм находит первое место, где два соседних элемента стоят в неправильном порядке и меняет их местами. Он пользуется тем фактом, что обмен может породить новую пару, стоящую в неправильном порядке, только до или после переставленных элементов. Он не допускает, что элементы после текущей позиции отсортированы, таким образом, нужно только проверить позицию до переставленных элементов.

Ниже написан псевдокод сортировки. Это оптимизированная версия с использованием переменной j, чтобы разрешить прыжок вперёд туда, где он остановился до движения влево, избегая лишние итерации и сравнения:
```
gnomeSort(a[0..size - 1])
i = 1;
j = 2;
while i < size
if a[i - 1] <= a[i] //для сортировки по убыванию поменяйте знак сравнения на >=
i = j;
j = j + 1;
else
swap a[i - 1] and a[i]
i = i - 1;
if i == 0
i = j;
j = j + 1;
```

Пример. Если мы хотим отсортировать массив с элементами [4] [2] [7] [3] от большего к меньшему, то на итерациях цикла while будет происходить следующее:
[4] [2] [7] [3] (начальное состояние: i == 1, j == 2);
[4] [2] [7] [3] (ничего не произошло, но сейчас i == 2, j == 3);
[4] [7] [2] [3] (обмен a[2] и a[1], сейчас i == 1, a j == 3 по-прежнему);
[7] [4] [2] [3] (обмен a[1] и a[0], сейчас i == 3, j == 4);
[7] [4] [3] [2] (обмен a[3] и a[2], сейчас i == 2, j == 4);
[7] [4] [3] [2] (ничего не произошло, но сейчас i == 4, j == 5);
цикл закончился, т. к. i не < 4.

Сортировка вставками — простой алгоритм сортировки. Хотя этот алгоритм сортировки уступает в эффективности более сложным (таким как быстрая сортировка), у него есть ряд преимуществ:

- эффективен на небольших наборах данных, на наборах данных до десятков элементов может оказаться лучшим;
- эффективен на наборах данных, которые уже частично отсортированы;
- это устойчивый алгоритм сортировки (не меняет порядок элементов, которые уже отсортированы);
- может сортировать список по мере его получения;
- не требует временной памяти, даже под стек.

Минусом же является высокая сложность алгоритма: $O(n^2)$.

На каждом шаге алгоритма мы выбираем один из элементов входных данных и вставляем его на нужную позицию в уже отсортированном списке, до тех пор пока набор входных данных не будет исчерпан. Метод выбора очередного элемента из исходного массива произволен; может использоваться практически любой алгоритм выбора. Обычно (и с целью получения устойчивого алгоритма сортировки), элементы вставляются по порядку их появления во входном массиве. Приведенный ниже алгоритм использует именно эту стратегию выбора.

Время выполнения алгоритма зависит от входных данных: чем большее множество нужно отсортировать, тем большее время выполняется сортировка. Также на время выполнения влияет исходная упорядоченность массива. Так, лучшим случаем является отсортированный массив, а худшим — массив, отсортированный в порядке, обратном нужному. Временная сложность алгоритма при худшем варианте входных данных — $\theta(n^2)$.

Псевдокод
Вход: массив A, состоящий из элементов A[1], A[2], ..., A[n]
for i = 2, 3, ..., n:

```
key := A[i]
j := i - 1
while j > 0 and A[j] > key:
A[j + 1] := A[j]
j := j - 1
A[j + 1] := key
```

Реализация на C++

```
void insertionSort(int arr[], int length) {
int i, j, tmp;
for (i = 1; i < length; i++) {
j = i;
while (j > 0 && arr[j - 1] > arr[j]) {
tmp = arr[j];
arr[j] = arr[j - 1];
arr[j - 1] = tmp;
j--;
}
}
}
```

Блочная сортировка. (Карманная сортировка, корзинная сортировка, англ. Bucket sort) — алгоритм сортировки, в котором сортируемые элементы распределяются между конечным числом отдельных блоков (карманов, корзин) так, чтобы все элементы в каждом следующем по порядку блоке были всегда больше (или меньше), чем в предыдущем. Каждый блок затем сортируется отдельно, либо рекурсивно тем же методом, либо другим. Затем элементы помещаются обратно в массив. Этот тип сортировки может обладать линейным временем исполнения.

Данный алгоритм требует знаний о природе сортируемых данных, выходящих за рамки функций "сравнить" и «поменять местами», достаточных для сортировки слиянием, сортировки пирамидой, быстрой сортировки, сортировки Шелла, сортировки вставкой.

Преимущества: относится к классу быстрых алгоритмов с линейным временем исполнения O(N) (на удачных входных данных).

Недостатки: сильно деградирует при большом количестве мало отличных элементов, или же на неудачной функции получения номера корзины по содержимому элемента. В некоторых таких случаях для строк, возникающих в реализациях основанного на сортировке строк алгоритма сжатия BWT, оказывается, что быстрая сортировка строк в версии Седжвика значительно превосходит блочную сортировку скоростью.

Если входные элементы подчиняются равномерному закону распределения, то математическое ожидание времени работы алгоритма карманной сортировки является линейным. Это возможно благодаря определенным предположениям о входных данных. При карманной сортировке предполагается, что входные данные равномерно распределены на отрезке [0, 1).

Идея алгоритма заключается в том, чтобы разбить интервал [0, 1) на n одинаковых отрезков (карманов), и разделить по этим карманам n входных величин. Поскольку входные числа равномерно распределены, предполагается, что в каждый карман попадет небольшое количество чисел. Затем последовательно сортируются числа в карманах. Отсортированный массив получается путем последовательного перечисления элементов каждого кармана.

Псевдокод

```
function bucket-sort(A, n) is
buckets ← новый массив из n пустых элементов
for i = 0 to (length(A)-1) do
вставить A[i] в конец массива buckets[msbits(A[i], k)]
for i = 0 to n - 1 do
next-sort(buckets[i])
return {Конкатенация массивов buckets[0], ..., buckets[n-1]}
```

На вход функции bucket-sort подаются сортируемый массив (список, коллекция и т.п.) A и количество блоков - n. Массив buckets представляет собой массив массивов (массив списков, массив коллекций и т.п.), подходящих по природе к элементам A.

Функция msbits(x,k) тесно связана с количеством блоков - n (возвращает значение от 0 до n), и, в общем случае, возвращает k наиболее значимых битов из x (floor(x/2^(size(x)-k))). В качестве msbits(x,k) может быть использованы

разнообразные функции, подходящие к природе сортируемых данных и позволяющие разбить массив A на n блоков. Например, для символово A-Z это может быть сопоставление буквам чисел 0-25, или возврат кода первой буквы (0-255) для ASCII набора символов; для чисел [0, 1) это может быть функция floor(n*A[i]), а для произвольного набора чисел в интервале [a, b) - функция floor(n*(A[i]-a)/(b-a)).

Функция next-sort также реализует алгоритм сотрировки для каждого созданного на первом этапе блока. Рекурсивное использование bucket-sort в качестве next-sort превращает данный алгоритм в поразрядную сортировку. В случае n = 2 соответствует быстрой сортировке (хотя и с потенцильно плохим выбором опорного элемента).

Сортировка подсчётом

Алгоритм сортировки, в котором используется диапазон чисел сортируемого массива (списка) для подсчёта совпадающих элементов. Применение сортировки подсчётом целесообразно лишь тогда, когда сортируемые числа имеют (или их можно отобразить в) диапазон возможных значений, который достаточно мал по сравнению с сортируемым множеством, например, миллион натуральных чисел меньших 1000. Эффективность алгоритма падает, если при попадании нескольких различных элементов в одну ячейку, их надо дополнительно сортировать. Необходимость сортировки внутри ячеек лишает алгоритм смысла, так как каждый элемент придётся просматривать более одного раза.

Предположим, что входной массив состоит из n целых чисел в диапазоне от 0 до k − 1, где . Далее алгоритм будет обобщён для произвольного целочисленного диапазона. Существует несколько модификаций сортировки подсчётом, ниже рассмотрены три линейных и одна квадратичная, которая использует другой подход, но имеет то же название.

Это простейший вариант алгоритма. Создать вспомогательный массив C[0..k - 1], состоящий из нулей, затем последовательно прочитать элементы входного массива A, для каждого A[i] увеличить C[A[i]] на единицу. Теперь достаточно пройти по массиву C, для каждого в массив A последовательно записать число j C[j] раз.
SimpleCountingSort
for i = 0 to k - 1
C[i] = 0;
for i = 0 to n - 1
C[A[i]] = C[A[i]] + 1;
b = 0;
for j = 0 to k - 1
for i = 0 to C[j] - 1
A[b] = j;
b = b + 1;

Сортировка слиянием (англ. *merge sort*) — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». «Разделяй и властвуй» (англ. *divide and conquer*) в информатике — парадигма разработки алгоритмов, заключающаяся в рекурсивном разбиении решаемой задачи на две или более подзадачи того же типа, но меньшего размера, и комбинировании их решений для получения ответа к исходной задаче; разбиения выполняются до тех пор, пока все подзадачи не окажутся элементарными.Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

Для решения задачи сортировки эти три этапа выглядят так:

1. Сортируемый массив разбивается на две части примерно одинакового размера;
2. Каждая из получившихся частей сортируется отдельно, например — тем же самым алгоритмом;
3. Два упорядоченных массива половинного размера соединяются в один.

Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 можно считать упорядоченным).

Нетривиальным этапом является соединение двух упорядоченных массивов в один. Основную идею слияния двух отсортированных массивов можно объяснить на следующем примере. Пусть мы имеем две стопки карт, лежащих рубашками вниз так, что в любой момент мы видим верхнюю карту в каждой из этих стопок. Пусть также, карты в каждой из этих стопок идут сверху вниз в неубывающем порядке. Как сделать из этих стопок одну? На каждом шаге мы берём меньшую из двух верхних карт и кладём её (рубашкой вверх) в результирующую стопку. Когда одна из оставшихся стопок становится пустой, мы добавляем все оставшиеся карты второй стопки к результирующей стопке.

Псевдокод на C++-подобном языке:

```
L = *In1;
R = *In2;
if( L == R )
{
*Out++ = L;
```

```
In1++;
*Out++ = R;
In2++;
}
else if( L < R )
{
*Out++ = L;
In1++;
}
else
{
*Out++ = R;
In2++;
}
```

Сортировка с помощью двоичного дерева (сортировка двоичным деревом, сортировка деревом, древесная сортировка, сортировка с помощью бинарного дерева, англ. tree sort) — универсальный алгоритм сортировки, заключающийся в построении двоичного дерева поиска по ключам массива (списка), с последующей сборкой результирующего массива путём обхода узлов построенного дерева в необходимом порядке следования ключей. Данная сортировка является оптимальной при получении данных путём непосредственного чтения с потока (например с файла, сокета или консоли).
1. Построение двоичного дерева.
2. Сборка результирующего массива путём обхода узлов в необходимом порядке следования ключей.

Процедура добавления объекта в бинарное дерево имеет среднюю алгоритмическую сложность порядка O(log(n)). Соответственно, для n объектов сложность будет составлять O(n log(n)), что относит сортировку с помощью двоичного дерева к группе «быстрых сортировок». Однако, сложность добавления объекта в разбалансированное дерево может достигать O(n), что может привести к общей сложности порядка $O(n^2)$.

При физическом развёртывании древовидной структуры в памяти требуется не менее чем 4n ячеек дополнительной памяти (каждый узел должен содержать ссылки на элемент исходного массива, на родительский элемент, на левый и правый лист), однако, существуют способы уменьшить требуемую дополнительную память.

Итеративная реализация. Следующий алгоритм (немного модифицирован для избежания переполнения) из Стандарта Pascal:

```
min := 1;
max := N; {array size: var A : array [1..N] of integer}
repeat
mid := min + (max - min) div 2;
if x > A[mid] then
min := mid + 1
else
max := mid - 1;
until (A[mid] = x) or (min > max)
```

Сортировка выбором

Алгоритм сортировки, относящийся к неустойчивым алгоритмам сортировки. На массиве из n элементов имеет время выполнения в худшем, среднем и лучшем случае Θ(n2), предполагая что сравнения делаются за постоянное время.

Шаги алгоритма:

- находим минимальное значение в текущем списке
- производим обмен этого значения со значением на первой неотсортированной позиции
- теперь сортируем хвост списка, исключив из рассмотрения уже отсортированные элементы

Пирамидальная сортировка сильно улучшает базовый алгоритм, используя структуру данных «куча» для ускорения нахождения и удаления минимального элемента.

Существует также двунаправленный вариант сортировки методом выбора, в котором на каждом проходе отыскиваются и устанавливаются на свои места и минимальное, и максимальное значения.

Сортировка Шелла (англ. Shell sort). Алгоритм сортировки, являющийся усовершенствованным вариантом сортировки вставками. Идея метода Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом

расстоянии друг от друга. Иными словами — это сортировка вставками с предварительными «грубыми» проходами. Аналогичный метод усовершенствования пузырьковой сортировки называется сортировка расчёской.

При сортировке Шелла сначала сравниваются и сортируются между собой значения, отстоящие один от другого на некотором расстоянии d (о выборе значения d см. ниже). После этого процедура повторяется для некоторых меньших значений d, а завершается сортировка Шелла упорядочиванием элементов при d = 1 (то есть, обычной сортировкой вставками). Эффективность сортировки Шелла в определённых случаях обеспечивается тем, что элементы «быстрее» встают на свои места (в простых методах сортировки, например, пузырьковой, каждая перестановка двух элементов уменьшает количество инверсий в списке максимум на 1, а при сортировке Шелла это число может быть больше).

Невзирая на то, что сортировка Шелла во многих случаях медленнее, чем быстрая сортировка, она имеет ряд преимуществ:

- отсутствие потребности в памяти под стек;
- отсутствие деградации при неудачных наборах данных — быстрая сортировка легко деградирует до $O(n^2)$, что хуже, чем худшее гарантированное время для сортировки Шелла.

Пример. Пусть дан список A = (32,95,16,82,24,66,35,19,75,54,40,43,93,68) и выполняется его сортировка методом Шелла, а в качестве значений d выбраны 5,3,1.

На первом шаге сортируются подсписки A, составленные из всех элементов A, различающихся на 5 позиций, то есть подсписки A5,1 = (32,66,40), A5,2 = (95,35,43), A5,3 = (16,19,93), A5,4 = (82,75,68), A5,5 = (24,54).

В полученном списке на втором шаге вновь сортируются подсписки из отстоящих на 3 позиции элементов.

Процесс завершается обычной сортировкой вставками получившегося списка.

Пирамидальная сортировка (англ. Heapsort). Алгоритм сортировки, работающий в худшем, в среднем и в лучшем случае (то есть гарантированно) за $\Theta(n \log n)$ операций при сортировке n элементов.[1] Количество применяемой служебной памяти не зависит от размера массива (то есть, O(1)).

Может рассматриваться как усовершенствованная сортировка пузырьком, в которой элемент всплывает (min-heap) / тонет (max-heap) по многим путям.
Сортировка пирамидой использует сортирующее дерево. Сортирующее дерево — это такое двоичное дерево, у которого выполнены условия:
1. Каждый лист имеет глубину либо d, либо d ? 1, d — максимальная глубина дерева.
2. Значение в любой вершине больше, чем значения её потомков.

Удобная структура данных для сортирующего дерева — такой массив Array, что Array[1] — элемент в корне, а потомки элемента Array[i] —Array[2i] и Array[2i+1].

Алгоритм сортировки будет состоять из двух основных шагов:

1. Выстраиваем элементы массива в виде сортирующего дерева. Этот шаг требует O(n) операций.
2. Будем удалять элементы из корня по одному за раз и перестраивать дерево. То есть на первом шаге обмениваем Array[1] и Array[n], преобразовываем Array[1], Array[2], … , Array[n-1] в сортирующее дерево. Затем переставляем Array[1] и Array[n-1], преобразовываем Array[1], Array[2], … , Array[n-2] в сортирующее дерево. Процесс продолжается до тех пор, пока в сортирующем дереве не останется один элемент. Тогда Array[1], Array[2], … , Array[n] — упорядоченная последовательность. Этот шаг требует O(nlogn) операций.

Достоинства:
- имеет доказанную оценку худшего случая O(nlogn),
- требует всего O(1) дополнительной памяти (если дерево организовывать так, как показано выше).

Недостатки:
- сложен в реализации,
- неустойчив — для обеспечения устойчивости нужно расширять ключ,
- на почти отсортированных массивах работает столь же долго, как и на хаотических данных,
- на одном шаге выборку приходится делать хаотично по всей длине массива — поэтому алгоритм плохо сочетается с кэшированием и подкачкой памяти.

Сортировка слиянием при расходе памяти O(n) быстрее ( ) с меньшей константой) и не подвержена деградации на неудачных данных. Из-за сложности алгоритма выигрыш получается только на больших n. На небольших n (до нескольких тысяч) быстрее сортировка Шелла.

Быстрая сортировка (англ. quicksort), часто называемая qsort по имени реализации в стандартной библиотеке языка Си — широко известный алгоритм сортировки, разработанный английским информатиком Чарльзом Хоаром в 1960 году. Один из быстрых известных универсальных алгоритмов сортировки массивов (в среднем O(n log n) обменов при упорядочении n элементов), хотя и имеющий ряд недостатков.

Краткое описание алгоритма:

- выбрать элемент, называемый опорным,
- сравнить все остальные элементы с опорным, на основании сравнения разбить множество на три — «меньшие опорного», «равные» и «большие», расположить их в порядке меньшие-равные-большие,
- повторить рекурсивно для «меньших» и «больших».

Быстрая сортировка использует стратегию «разделяй и властвуй». Шаги алгоритма таковы.

Выбираем в массиве некоторый элемент, который будем называть опорным элементом. С точки зрения корректности алгоритма выбор опорного элемента безразличен. С точки зрения повышения эффективности алгоритма выбираться должна медиана, но без дополнительных сведений о сортируемых данных её обычно невозможно получить. Известные стратегии: выбирать постоянно один и тот же элемент, например, средний или последний по положению; выбирать элемент со случайно выбранным индексом.

Операция разделения массива: реорганизуем массив таким образом, чтобы все элементы, меньшие или равные опорному элементу, оказались слева от него, а все элементы, большие опорного — справа от него. Обычный алгоритм операции:

Два индекса — l и r, приравниваются к минимальному и максимальному индексу разделяемого массива соответственно.

Вычисляется индекс опорного элемента m.

Индекс l последовательно увеличивается до m до тех пор, пока l-й элемент не превысит опорный.

Индекс r последовательно уменьшается до m до тех пор, пока r-й элемент не окажется меньше либо равен опорному.

Если r = l — найдена середина массива — операция разделения закончена, оба индекса указывают на опорный элемент.

Если l < r — найденную пару элементов нужно обменять местами и продолжить операцию разделения с тех значений l и r, которые были достигнуты. Следует учесть, что если какая-либо граница (l или r) дошла до опорного элемента, то при обмене значение m изменяется на r-й или l-й элемент соответственно.

Рекурсивно упорядочиваем подмассивы, лежащие слева и справа от опорного элемента.

Базой рекурсии являются наборы, состоящие из одного или двух элементов. Первый возвращается в исходном виде, во втором, при необходимости, сортировка сводится к перестановке двух элементов. Все такие отрезки уже упорядочены в процессе разделения.

Поскольку в каждой итерации (на каждом следующем уровне рекурсии) длина обрабатываемого отрезка массива уменьшается, по меньшей мере, на единицу, терминальная ветвь рекурсии будет достигнута всегда и обработка гарантированно завершится.

Интересно, что Хоар разработал этот метод применительно к машинному переводу: дело в том, что в то время словарь хранился на магнитной ленте, и если упорядочить все слова в тексте, их переводы можно получить за один прогон ленты.

QuickSort является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена (его варианты известны как «Пузырьковая сортировка» и «Шейкерная сортировка»), известного, в том числе, своей низкой эффективностью. Принципиальное отличие состоит в том, что после каждого прохода элементы делятся на две независимые группы. Любопытный факт: улучшение самого неэффективного прямого метода сортировки дало в результате эффективный улучшенный метод.

Лучший случай. Для этого алгоритма самый лучший случай — если в каждой итерации каждый из подмассивов делился бы на два равных по величине массива. В результате количество сравнений, делаемых быстрой сортировкой, было бы равно значению рекурсивного выражения CN = 2CN/2+N, что в явном выражении дает примерно N lg N сравнений. Это дало бы наименьшее время сортировки.

Среднее. Даёт в среднем O(n lg n) обменов при упорядочении n элементов. В реальности именно такая ситуация обычно имеет место при случайном порядке элементов и выборе опорного элемента из середины массива либо случайно.

На практике (в случае, когда обмены являются более затратной операцией, чем сравнения) быстрая сортировка значительно быстрее, чем другие алгоритмы с оценкой O(n lg n), по причине того, что внутренний цикл алгоритма может быть эффективно реализован почти на любой архитектуре. 2CN/2 покрывает расходы по сортировке двух полученных подмассивов; N — это стоимость обработки каждого элемента, используя один или другой указатель. Известно также, что примерное значение этого выражения равно CN = N lg N.

Худший случай. Худшим случаем, очевидно, будет такой, при котором на каждом этапе массив будет разделяться на вырожденный подмассив из одного опорного элемента и на подмассив из всех остальных элементов. Такое может произойти, если в качестве опорного на каждом этапе будет выбран элемент либо наименьший, либо наибольший из всех обрабатываемых.

Худший случай даёт $O(n^2)$ обменов. Но количество обменов и, соответственно, время работы — это не самый большой его недостаток. Хуже то, что в таком случае глубина рекурсии при выполнении алгоритма достигнет n, что будет означать n-кратное сохранение адреса возврата и локальных переменных процедуры разделения массивов. Для больших значений n худший случай может привести к исчерпанию памяти во время работы алгоритма. Впрочем, на большинстве реальных данных можно найти решения, которые минимизируют вероятность того, что понадобится квадратичное время.

Последовательный поиск

Постановка задачи: пусть дан массив из N элементов. Необходимо определить номер элемента, который обладает определенными свойствами (чаще всего речь идет просто об элементе с определенным значением, равным k), или установить что такого элемента в массиве нет.

Если массив является неупорядоченным, то единственный алгоритм поиска, применимый в этом случае – последовательный. Суть метода – последовательно перебираются все элементы массива, если на каком-то шаге цикла обнаруживается, что массив закончился или обнаруживается искомый элемент, то цикл заканчивается.

Пример фрагмента программы

```
const N = 100;
Var a: array[1..N] of integer;
     i,k : integer;
begin {ввод массива, k}
i:=1;
while (i<=N)and(a[i]<>k) do inc(i); {после окончания цикла i равно номеру искомого элемента, а если элемент не найден, то i=N+1}
end.
```

Использование цикла while вместо for обусловлено сокращением количества сравнений, если нужный элемент находится близко к началу массива. В этом случае, как только элемент будет найден, цикл прекратит свою работу.

Возможна следующая оптимизация этого алгоритма. Добавим искомый элемент на N+1 место в массиве и перепишем программу следующим образом:

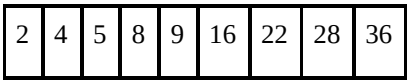
```
const N = 100;
Var a: array[1..N+1] of integer;
     i,k : integer;
begin {ввод массива, k}
a[N+1]:=k;
i:=1;
while (a[i]<>k) do inc(i); {после окончания цикла i равно номеру искомого элемента}
end.
```

Очевидно, что при такой модификации элемент будет найден либо в середине массива, либо на в позиции N+1, поэтому проверку на выход за границы массива можно не делать, потому что условие цикла (a[i]<>k) обязательно нарушится еще в границах массива. После отработки этого цикла i равно N+1, если элемента с заданным значением в массиве не найдено, или равно номеру этого элемента.

Бинарный поиск

Если массив упорядочен по возрастанию или убыванию, то можно применить более быстрые методы поиска, такие как, например, бинарный (методом деления пополам).

Рассмотрим пример. Пусть дан массив из 9 элементов, упорядоченный по возрастанию их значений.

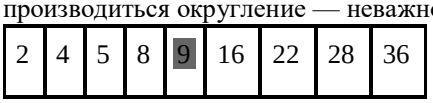


Пусть требуется найти номер элемента, значение которого равно -3. Сравним это значение с первым элементом массива (-3)<2. Из условия того, что массив упорядочен по возрастанию следует, что элементы с бо́льшими номерами будут обладать бо́льшими значениями, то есть остальные числа в массиве гарантировано больше чем 2. Отсюда можно сделать вывод, что числа (-3) в массиве нет.

Пусть требуется найти элемент со значением 44. Сравним данное число с первым элементом. 44>2. Следовательно, можно предположить, что в массиве число 44 встречается. Теперь сравним это число с последним элементом массива (из условия упорядоченности массива, его последний элемент является наибольшим, а все остальные элементы заведомо меньше его). 44>36. Следовательно, в массиве числа 44 быть не может.

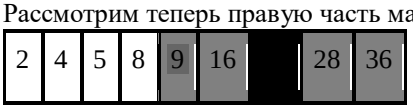
Пусть требуется найти элемент со значением 22. Сравним это число с крайними элементами массива, и убедимся, что такое число может присутствовать среди его элементов.

Возьмем элемент, находящийся в середине массива (если число элементов четно, то в какую сторону будет производиться округление — неважно).



Элемент, стоящий в середине массива не тот, который нам нужен. (9<>22). Тем не менее, этот элемент делит массив на две части: одну в которой искомого числа точно быть не может (в данном примере это левая половина массива; из условия упорядоченности все элементы, стоящие слева от 9 будут меньше этого числа) и вторую, в которой можно продолжать поиск.

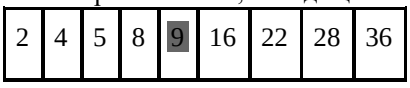
Рассмотрим теперь правую часть массива. Найдем элемент, стоящий в ее середине.



Это значение равно искомому, процесс поиска окончен.

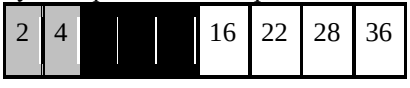
Пусть требуется найти элемент со значением 7. Сравним это число с крайними элементами массива и убедимся что поиск имеет смысл.

Рассмотрим элемент, находящийся в середине массива.

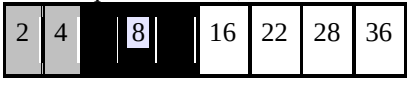


Этот элемент не равен искомому, следовательно, поиск необходимо продолжить. Так как 9>7, следовательно, правую половину массива можно исключить из рассмотрения, и продолжить поиск только в левой. Рассмотрим элемент, стоящий в середине левой половины:

Этот элемент снова не равен 7, следовательно поиск надо продолжить. Из условия упорядоченности массива, поиск будем продолжать в правой части нашего «укороченного» массива.



Рассмотрим элемент, находящийся в середине выделенной части массива.



Этот элемент не равен искомому. Однако, в результате постоянного уменьшения область поиска в массиве уменьшилась до трех элементов, и каждый из них был рассмотрен на каком-то шаге. Следовательно, процесс поиска окончен, результат — сообщение о том, что элемент с таким значением в массиве отсутствует.

На этих примерах видно, что с каждым шагом размер области поиска в массиве уменьшается в два раза, что приводит к сокращению числа операций. Например, за 10 шагов размер области поиска будет уменьшен в 1024 раза, а для массива из 1000 000 элементов понадобится всего 20 шагов.

Количество операций в данном методе равно O(log n).

Программная реализация данного метода может быть, например, такой.

```
const N = 100;
var a : array [1..N] of integer;
      i : integer;
      k : integer;
     c,left, right : integer;
begin {ввод данных}
left:=1;
right:=N;
if a[1]=k then write('Ответ=',1)
   else if a[N]=k then write('Ответ=',N) {сразу проверим, не является ли искомый}
      else {элемент крайним} if k<a[left] then write('Нет числа') {сравним с крайними, решим вопрос: }
         else {а стоит ли вообще искать?} if k>a[right] then write('Нет числа')
           else begin repeat {цикл поиска: найдем номер в середине}
                              c:=(left+right) div 2; {текущей области поиска}
                              if a[c]<k then left:=c;
                              if a[c]>k then right:=c;
                         until (a[c]=k)or(right-left<=1); {пока элемент не найден,
                               или пока область поиска не стала состоять всего
                               из двух соседних элементов}
if a[c]=k then write('Ответ=', c) else write('Нет числа');
end;
end.
```

Здесь left и rght — это номера элементов массива, которые являются границами текущей области поиска. Например, если left=3 и right=7 это значит что поиск ведется среди элементов с номерами от 3 до 7 и именно в этой области на данном шаге будет искаться середина.

Существует рекурсивная реализация данного метода, которая является более короткой и красивой.