БАЗОВЫЕ АЛГОРИТМЫ ВНУТРЕННЕЙ СОРТИРОВКИ

Процесс сортировки данных может быть осуществлен различными алгоритмами. Если объем входных данных позволяет обходиться исключительно основной (оперативной) памятью, то говорят об алгоритмах внутренней сортировки, в противном случае – об алгоритмах внешней сортировки. В данном разделе будут рассмотрены основные алгоритмы внутренней сортировки. Поскольку каждый из алгоритмов имеет свои преимущества и недостатки, то выбор алгоритма должен осуществляться исходя из конкретной постановки задачи. Большое разнообразие алгоритмов сортировки приводит к необходимости анализа этих алгоритмов (их производительности). На примере алгоритмов внутренней сортировки легко проиллюстрировать, как путем усложнения алгоритмов (несмотря на существование простых и очевидных методов) добиться большей эффективности.

Во всех алгоритмах сортировки на вход поступает последовательность из n чисел, поэтому размерность задачи $\Theta(n)$.

Определение. Пусть дана последовательность из n элементов:

$$a_1, a_2, \ldots, a_n$$

(назовем их записями), выбранных из множества, на котором задан линейный порядок. Каждая запись a_j имеет ключ k_j, который и управляет процессом сортировки (помимо ключа запись может иметь дополнительную информацию, не влияющую на процесс сортировки, но всегда присутствует в этой записи). Задача сортировки заключается в поиске перестановки $\pi = \pi(1), \pi(2), \ldots, \pi(n)$ этих n записей, после которой ключи расположились бы в неубывающем порядке:

$$k_{\pi(1)} \le k_{\pi(2)} \le \ldots \le k_{\pi(n)}.$$

Определение. Алгоритм сортировки будем называть *устойчивым*, если в процессе сортировки относительное расположение элементов с одинаковыми ключами не изменяется (предполагается, что элементы уже были ранее отсортированы по некоторому вторичному ключу):

$$\pi(i) < \pi(j), \text{ если } k_{\pi(i)} \le k_{\pi(j)} \text{ и } i < j.$$

Хороший алгоритм внутренней сортировки затрачивает на сортировку n записей время порядка $n \cdot \log n$. К таким алгоритмам относятся алгоритмы групповой сортировки, которые работают с упорядоченными группами элементов:

- сортировка слиянием (merge sort);
- сортировка с помощью разделения (quick sort).

Существуют более простые методы упорядочивания, которые затрачивают на сортировку время порядка n^2. К таким алгоритмам относятся следующие простые алгоритмы парной сортировки:

- сортировка с помощью включения (by insertion);
- сортировка выбором (by selection);
- сортировка с помощью обменов (by exchange).

Сортировка с помощью включения (вставками)

Пусть элементы $a_1, a_2, \ldots, a_{i-1}, 1 < i \le n$, уже упорядочены на предыдущих этапах данным алгоритмом (первоначально в качестве упорядоченной части можно взять первый элемент массива a_1). На очередном этапе необходимо взять a_i (первый элемент из неупорядоченной части) и включить в нужное место упорядоченной последовательности $a_1, a_2, \ldots, a_{i-1}$ так, чтобы первые i элементов массива стали упорядочены. Поскольку элемент a_i как бы «проникает» на положенный для него уровень, то процесс поиска требуемой позиции часто называют «просеиванием». В зависимости от того, как происходит процесс включения элемента, различают *прямое* и *двоичное включение*.

Приведем алгоритм сортировки с помощью прямого включения. Для того чтобы вместо двух условий окончания цикла (конец интервала либо найдено место вставки для элемента) использовать одно, введем барьер $a[0]$.

Алгоритм сортировки с помощью прямого включения

```
для i от 2 до n
{ a[0] = a[i]
  j = i
  пока a[0] < a[j – 1]
    {a[j] = a[j – 1]
      j = j–1}
  a[j] = a[0] }
```

Для оценки времени работы алгоритма составим рекуррентное уравнение:

$$\begin{cases} T(n) = T(n-1) + C \cdot n, \; n > 1, \\ T(1) = 0. \end{cases}$$

Время работы алгоритма есть $O(n^2)$.

Пример. Упорядочить, используя сортировку с помощью прямого включения, последовательность элементов: 2, –7, 0, 100, 13, –10, 3, 11.

Решение

1 итерация: (2), –7, 0, 100, 13, –10, 3, 11;

2 итерация: (–7, 2), 0, 100, 13, –10, 3, 11;

3 итерация: (–7, 0, 2), 100, 13, –10, 3, 11;

4 итерация: (–7, 0, 2, 100), 13, –10, 3, 11;

5 итерация: (–7, 0, 2, 13, 100), –10, 3, 11;

6 итерация: (–10, –7, 0, 2, 13, 100), 3, 11;

7 итерация: (–10, –7, 0, 2, 3, 13, 100), 11;

8 итерация: (–10, –7, 0, 2, 3, 11, 13, 100).

Заметим, что поскольку готовая последовательность уже упорядочена, то для поиска места включения текущего элемента можно использовать метод дихотомии (поиск делением пополам, или двоичный поиск), для которого каждый раз исключается из рассмотрения половина элементов. Если линейный поиск в среднем требовал $n/2$ сравнений на n элементах, то двоичный поиск – $\log_2 n$ сравнений (в худшем случае).

Алгоритм сортировки
с помощью двоичного включения

```
для i от 2 до n
    { x = a[i], l = 1, r = i
      пока l < r
        {m = (l + r) / 2
          если a[m] < x : l = m + 1 иначе r = m
         }
      для j = i до (r + 1) a[j] = a[j – 1]
      a[r] = x
    }
```

Следует отметить, что в правой части интервала [*l, r*] поиск осуществляется быстрее, чем в левой.

Следует также отметить, что алгоритм двоичного включения не позволяет улучшить оценку числа пересылок элементов, которая остается в худшем случае $O(n)$ (хотя получено улучшение оценки для числа сравнений элементов $O(\log n)$).

Сортировка выбором

Идея сортировки заключается в следующем:

1. Среди n элементов массива выбрать элемент с наименьшим ключом и поменять его с первым элементом. Теперь первый элемент стоит на своем месте.

При поиске минимального элемента нельзя выполнять постоянне обмены элементов массива в памяти компьютера, т.е. нужно найти индекс минимального элемента и выполнить единственный обмен двух элементов массива. Именно поэтому данный алгоритм не относится к серии обменных алгоритмок, как, например, пузурьковая сортировка.

2. Повторить описанные действия с оставшимися $n-1$ элементами.

3. Процесс заканчивается, когда $n-1$ элементов будут помещены на свои места.

Как следует из описания алгоритма, сортировка выбором асимметрична прямому включению: просматриваем элементы оставшейся исходной последовательности и получаем один элемент упорядоченной последовательности.

Алгоритм сортировки выбором

```
для i=1 до n – 1
      { k = i
        для  j  = i + 1 до n  если a[j] < a[k] : k  = j
        x = a[k]
        a[k] = a[i]
        a[i] = x
    end;
end;
```

Оценим время работы сортировки выбором.

Поиск минимального элемента из n элементов и добавление его в уже готовую упорядоченную последовательность требует в худшем случае $C \cdot n$ операций, после чего повторяем описанный процесс для оставшихся $n-1$ элементов.

Следует отметить, что количество сравнений ключей не зависит от начального расположения элементов, а количество пересылок минимально, когда массив упорядочен, и максимально, когда элементы расположены в обратном порядке.

Тогда рекуррентное уравнение для оценки времени работы алгоритма будет иметь следующий вид:

$$\begin{cases} T(n) = C \cdot n + T(n-1), n > 1, \\ T(1) = 0. \end{cases}$$

Алгоритм сортировки выбором работает за время $\mathrm{O}(n^2)$.

Пример. Упорядочить, используя сортировку выбором, последовательность элементов:

2, –7, 0, 100, 13, –10, 3, 11.

Решение

1 итерация: (–10), –7, 0, 100, 13, 2, 3, 11;
2 итерация: (–10, –7), 0, 100, 13, 2, 3, 11;
3 итерация: (–10, –7, 0), 100, 13, 2, 3, 11;
4 итерация: (–10, –7, 0, 2), 13, 100, 3, 11;
5 итерация: (–10, –7, 0, 2, 3), 100, 13, 11;
6 итерация: (–10, –7, 0, 2, 3, 11), 13, 100;
7 итерация: (–10, –7, 0, 2, 3, 11, 13), 100.

На практике, как правило, алгоритм выбором предпочтительнее алгоритма с прямым включением. Однако если ключи вначале упорядочены или почти упорядочены, то прямое включение будет оставаться несколько более быстрым.

Сортировки с помощью обменов

Сортировки с помощью обменов основаны на сравнении двух элементов. Если порядок элементов не соответствует упорядоченности, то происходит их обмен. Процесс повторяется до тех пор, пока элементы не будут упорядочены.

Следует отметить, что и для рассмотренных ранее алгоритмов также имел место обмен элементов, однако в рассматриваемом типе алгоритмов обмен местами двух элементов представляет собой наиболее характерную особенность процесса.

Пузырьковая сортировка

Алгоритм заключается в просмотре исходной последовательности справа налево, и при каждом шаге меньший из двух соседних элементов перемещается к левой позиции.

В результате первого просмотра самый маленький элемент будет находиться в крайней левой позиции.

После этого повторяем описанный выше процесс, рассматривая в качестве исходной последовательности массив, начиная со 2-й позиции и т. д.

Алгоритм пузырьковой сортировки

для i = 2 до n
  для $j = n$ до i
    если $a[j-1] > a[j]$:
      { $x = a[j-1]$, $a[j-1] = a[j]$, $a[j] = x$}

Оценим время работы алгоритма пузырьковой сортировки.

Перемещение минимального из n элементов в крайнюю левую позицию требует в худшем случае $C \cdot (n-1)$ операций (число сравнений ключей не зависит от порядка элементов; число присваиваний минимально, если элементы упорядочены, и максимально, если они расположены в обратном порядке).

Рекуррентное уравнение для числа операций алгоритма пузырьковой сортировки будет иметь следующий вид:

$$\begin{cases} T(n) = C \cdot (n-1) + T(n-1), n > 1, \\ T(1) = 0. \end{cases}$$

Время работы алгоритм пузырьковой сортировки $O(n^2)$.

Пример. Упорядочить, используя сортировку пузырьком, последовательность элементов: 2, –7, 0, 100, 13, –10, 3, 11.

Решение

1 итерация: (–10), 2, –7, 0, 100, 13, 3, 11;
2 итерация: (–10, –7), 2, 0, 3, 100, 13, 11;
3 итерация: (–10, –7, 0), 2, 3, 11, 100, 13;
4 итерация: (–10, –7, 0, 2), 13, 100, 3, 11;
5 итерация: (–10, –7, 0, 2, 3), 13, 100, 11;
6 итерация: (–10, –7, 0, 2, 3, 11), 13, 100;
7 итерация: (–10, –7, 0, 2, 3, 11, 13), 100.

Шейкерная сортировка

Анализ алгоритма пузырьковой сортировки приводит к следующим наблюдениям:

1. Если при некотором проходе нет перестановок элементов, то алгоритм можно завершить досрочно.

2. Если зафиксировать индекс k последнего обмена (все пары левее этого индекса уже упорядочены), то просмотр можно завершить на этом индексе, а не идти до нижнего предела для индекса i.

3. Чередование направлений для просмотра, при котором «всплывает» самый легкий, а «тонет» самый тяжелый элемент.

Алгоритм шейкерной сортировки

$l = 2,\ r = n,\ k = n$
повторять
  для $j = r$ до l
    если $a[j-1] > a[j]$:
      { $x = a[j-1],\ a[j-1] = a[j],\ a[j] = x$
        $k = j$
      }
  $l = k + 1$
  для $j = l$ до r
    если $a[j-1] > a[j]$:
      { $x = a[j-1],\ a[j-1] = a[j],\ a[j] = x$
        $k = j$
  $r = k - 1;$
пока $L > R$

Рекуррентное уравнение для числа операций алгоритма шейкерной сортировки будет иметь следующий вид:

$$\begin{cases} T(n) = C_1 \cdot (n-1) + C_2 \cdot (n-2) + T(n-2), & n \geq 3, \\ T(2) = C_3, \\ T(1) = 0. \end{cases}$$

Алгоритм шейкерной сортировки работает за время $O(n^2)$. Шейкерная сортировка с успехом используется в тех случаях, когда известно, что элементы почти упорядочены.

Сортировка слиянием

Сортировка слиянием заключается в следующем.

1. Делим последовательность элементов на две части (если сортируемая последовательность состояла из n элементов, то первая часть может содержать первые $\lfloor n/2 \rfloor$ элементов, а вторая часть – оставшиеся, порядок следования элементов в каждой из полученных частей совпадает с их порядком следования в исходной последовательности).
2. Сортируем отдельно каждую из полученных частей этим же алгоритмом.
3. Производим слияние отсортированных частей последовательности (при слиянии сравниваем наименьшие элементы каждой из отсортированных частей и меньший из них отправляем в список вывода; повторяем описанные действия до тех пор, пока не исчерпается одна из частей; все оставшиеся элементы другой части пересылаем в список вывода).

Алгоритм сортировки слиянием

```
sort_merge (l, r)
  если l ≠ r :
     { z  = (l + r) / 2
       sort_merge (l, z)
       sort_merge (z + 1, r)
       merge (a, l, z, r) }
```

В выше приведенной программе использовалась процедура *merge* (*a*, *l*, *k*, *r*). Эта процедура выполняет слияние двух отсортированных частей последовательности *a* (первая часть – элементы последовательности *a* с индексами от *l* до *z*, вторая часть – элементы последовательности *a* с индексами от $z+1$ до *r*) в последовательность *a* (индексы элементов от *l* до *r*) таким образом, чтобы сохранилась упорядоченность элементов.

```
merge (a , l, z, r )
   i  = l,  k  = 1,  j  = z + 1
   пока (i ≤ z) и (j ≤ r)
      если a[i] < a[j] :  { c[k] = a[i], i = i + 1,  k  = k + 1}
      иначе  { c[k] = a[j], j= j + 1,  k  = k + 1}
   пока (i ≤ z)   c[k] = a[i], i= i + 1,  k  = k + 1}
   пока (j ≤ r)   c[k] = a[j], j= j+ 1,  k  = k + 1}
   k  = 0;
   для i  = l до r { k : = k + 1,  a[i] = c[k]}
```

Рекуррентное уравнение для числа операций алгоритма сортировки слиянием будет иметь следующий вид:

$$\begin{cases} T(n) = 2 \cdot T\left(\frac{n}{2}\right) + C \cdot n, \quad n > 1, \\ T(1) = 0. \end{cases}$$

Время работы алгоритм сортировки слиянием $O(n \cdot \log n)$.В алгоритме сортировки слиянием можно уменьшить дополнительную память, сделав её не более, чем *n*/2. Существуют алгоритмы, которые выполяют сортировку слиянием, используя дополнительную память, которая не зависит от n.

Сортировка с помощью разделения

Данный алгоритм сортировки был разработан Ч. Хоаром. Хотя трудоемкость его работы в худшем случае для *n* элементов равна $\Omega(n^2)$, на практике этот алгоритм является одним из самых быстрых методов внутренней сортировки.

Данный алгоритм часто называют *быстрой сортировкой* (Quicksort). Мы покажем, что при определенном правиле выбора сепаратора время работы алгоритма быстрой сортировки равно $O(n \cdot \log n)$, причем логарифмический множитель довольно мал.

Еще одно достоинство данного метода в том, что быстрая сортировка не требует дополнительной памяти.

Суть алгоритма состоит в следующем:

1. Выбрать некоторый элемент *x* (сепаратор) для сравнения (это может быть центральный, первый или последний элемент сортируемой части массива). Ч. Хоар предполагает, что элемент *x* надо выбирать случайно, а для небольших выборок останавливаться на медиане (медиана – это элемент, стоящий в позиции $\lceil n/2 \rceil$ в отсортированном массиве).

2. Используя обмены, выполнить процедуру разделения, суть которой заключается в том, чтобы разбить массив на две части: левую с ключами $\le x$ и правую с ключами $\ge x$. Данные действия могут быть выполнены, например, таким алгоритмом:

- просматриваем массив слева направо, пока не встретим некоторый элемент $a[i] > x$;
- просматриваем массив справа налево, пока не встретим некоторый элемент $a[j] < x$;
- меняем местами эти два элемента;
- продолжаем просмотр и обмен до тех пор, пока не будут просмотрены все элементы массива $(i > j)$.

3. Затем необходимо применить процедуру разделения к получившимся двум частям, затем к частям частей и так далее, пока каждая из частей не будет состоять из одного единственного элемента.

Заметим, что если в алгоритме сортировки слиянием процесс разделения массива на части был достаточно простым (находили индекс центарльного элемента), то здесь этот процесс более трудоемкий. В то же время процесс слияния отсортированных частей уже более прост в алгоритме быстрой сортировки.

Приведем программную реализацию алгоритма быстрой сортировки Хоара. В качестве сепаратора выбираем центральный элемент сортируемой части массива.

```
Quicksort (l, r)
    x  = a[(l + r)  / 2
    i  = l ,   j  = r
    повторять
       пока  a[i] < x   i  = i + 1
       пока a[j] > x    j = j – 1
       если  i ≤ j  :
          { x = a[i], a[i] = a[j], a[j]  = x
            i = i + 1,  j  = j – 1 }
    пока (i  > j)
    если l < j Quicksort (l, j)
    если i < r  Quicksort (i, r);
```

Оценим время работы алгоритма.

Процедура разделения n элементов требует $C \cdot n$ операций, так как каждый элемент последовательности достаточно сравнить с выбранным элементом.

Если в качестве сепаратора выбирается элемент, который всегда делит сортируемую область пропорционально, то рекуррентное уравнение для числа арифметических операций имеет вид

$$\begin{cases} T(n) = C \cdot n + T(\alpha \cdot n) + T\big((1-\alpha)\cdot n\big), \quad n > 1, \\ T(1) = 0, \end{cases}$$

$$\text{const} \le \alpha \le \frac{1}{\text{const}},$$

где $\text{const} > 0$ – некоторая константа.

Решая данное рекуррентное уравнение, получим, что

$$T(n) \le C \cdot n \cdot \log_{\frac{1}{\beta}} n,$$

$$\beta = \min(\alpha, 1-\alpha).$$

В общем случае такой выбор сепаратора не гарантирован, т. е. существуют такие наборы входных данных, когда сепаратор не всегда делит массив в требуемой пропорции, поэтому

$$T(n_1) = C \cdot n_1 + T(n_1 - k) + T(k), \quad n_1 > 1.$$

Если на всех этапах не происходит деление сортируемой области в пропорции, то рекуррентное соотношение для числа арифметических операций имеет вид

$$\begin{cases} T(n) = C \cdot n + T(n-k) + T(k), \quad n > 1, \quad k - \text{константа}, \\ T(1) = 0, \end{cases}$$

и $T(n) \geq C' \cdot n^2$.

Чтобы избежать худшего случая, на каждой итерации алгоритма быстрой сортировки сепаратор должен выбираться таким образом, чтобы делить сортируемую область в пропорции. Для этого при выборе сепаратора можно использовать, например, алгоритм нахождения *k*-го наименьшего элемента, который работает за время $O(n)$ и будет рассмотрен в следующем разделе.

Выбор в качестве сепаратора *k*-ого наименьшего элемента, где $k = n/c$, $c > 0$ – целочисленая константа, гарантирует разделение массива в пропорции.

Например, если $c = 2$, то $k = n/2$, т. е. в качестве сепаратора выбирается медиана (средний по значению элемент массива), то рекуррентное соотношение для числа арифметических операций сортировки Хоара имеет вид

$$\begin{cases} T(n) = C_1 \cdot n + C_2 \cdot n + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right), \quad n > 1, \\ T(1) = 0. \end{cases}$$

Решая данное рекуррентное соотношение, получаем, что $T(n) = O(n \cdot \log n)$.

АЛГОРИТМЫ ВЫБОРКИ

Рассмотрим несколько алгоритмов поиска *k*-го наименьшего элемента массива. Напомним, что *k*-м минимальным элементом называется элемент, который стоит на *k*-м месте в отсортированном массиве.

Определение. Медианой массива из *n* элементов называется такой его элемент, который стоит на месте $\lceil n/2 \rceil$ в отсортированном массиве.

Для последовательности элементов

18 24 12 27 19

медиана равна 19.

Очевидно, что для нахождения медианы можно сначала отсортировать массив, а после этого выбрать центральный элемент. Тогда это потребует $O(n \cdot \log n)$ операций.

Поиск медианы является частным случаем более общей задачи: нахождение *k*-го наименьшего элемента из *n* элементов массива.

Рассмотрим сначала алгоритм для нахождения *k*-го наименьшего элемента из *n* элементов массива, предложенный Ч. Хоаром.

Алгоритм 1
нахождения k-го наименьшего элемента

1. Выполняется операция разделения на отрезке $[L, R]$, где первоначально $L = 1, R = n,$ а в качестве сепаратора берется $x = a[k]$ (см. алгоритм быстрой сортировки).

В результате разделения получаются индексы $i,\ j$ такие, что

$$a[h] \le x, \quad L \le h \le i,$$
$$a[h] \ge x, \quad j \le h \le R,$$
$$i > j.$$

а) Если $j \le k \le i$, то элемент $a[k]$ разделяет массив на две части в нужной пропорции; алгоритм заканчивает свою работу.

б) Если $i < k$, то выбранное значение *x* было слишком мало, поэтому процесс разделения необходимо выполнить на отрезке $[i, R]$.

в) Если $k < j$, то значение *x* было слишком велико, поэтому процесс разделения необходимо выполнить на отрезке $[L, j]$.

2. Процесс разделения повторять до тех пор, пока не возникнет ситуация а). Значение *x* соответствует значению *k*-го наименьшего элемента.

Схематично алгоритм может быть записан следующим образом:

```
L = 1, R = n
пока (L < R)
   { x = a[k];
     разделение a[L], …, a[R]
     если i < k :  L = i
     если  k < j : R = j
     если  j ≤ k ≤ i  : x  –  k-й минимальный элемент, завершение работы
   }
```

Если предположить, что сепаратор *x* всегда разделяет последовательность из n_1 элементов в пропорции: в одной части $\alpha \cdot n_1$ элементов, в другой – $(1-\alpha) \cdot n_1$ элементов, где $c \le \alpha / 1-\alpha \le 1/c$, c – константа, то рекуррентное уравнение будет иметь вид

$$\begin{cases} T(n) = C_1 \cdot n + T(\alpha \cdot n), \quad n > \dfrac{1}{\alpha}, \\ T(1) = C_2. \end{cases}$$

В этом случае время работы приведенного алгоритма нахождения *k*-го наименьшего элемента $O(n)$.

Таким образом, получаем явное преимущество по сравнению с прямыми методами, где сначала сортируется все множество, а затем выбирается *k*-й элемент.

Следует заметить, что в худшем случае, когда на всех этапах не происходит деление области в пропорции, рекуррентное уравнение будет иметь вид

$$\begin{cases} T(n) = C_1 \cdot n + T(n-k), \quad n > 1, \quad k - \text{константа}, \\ T(1) = C_2, \end{cases}$$

и трудоемкость алгоритма 1 нахождения k-го наименьшего элемента есть $\Omega(n^2)$.

Рассмотрим теперь еще один алгоритм нахождения *k*-го наименьшего элемента, время работы которого есть $O(n)$ в худшем случае.

Алгоритм 2
нахождения k - го наименьшего элемента

1. Разбиваем исходную последовательность *S* на $\lfloor n/5 \rfloor$ подпоследовательностей по пять элементов в каждой. В каждой такой подпоследовательности находим медиану. Это потребует $C_1 \cdot n$ операций.

2. Из найденных на первом шаге медиан строим последовательность *M* и рекурсивно находим ее медиану *x*. Поскольку длина рассматриваемой последовательности *M* равна $\lfloor n/5 \rfloor$, то трудоемкость нахождения медианы для этой последовательности равна $T(n/5)$.

3. Для полученного элемента *x* выполним процесс разделения, который потребует $C_2 \cdot n$ операций. В результате вся рассматриваемая последовательность S будет разбита на части: S_1, где элементы не больше *x*; S_2, где элементы не меньше *x*. Одна из частей может быть отброшена, причем несложно показать, что количество элементов в каждой из частей не меньше $\lceil n/4 \rceil$.

Решаем задачу нахождения *k*-го наименьшего элемента оставшихся $\lfloor 3n/4 \rfloor$ элементов, что потребует времени $T(3n/4)$.

Таким образом, рекуррентное уравнение для количества арифметических операций приведенного алгоритма имеет вид:

$$T(n) = C_1 \cdot n + T\left(\frac{n}{5}\right) + C_2 \cdot n + T\left(\frac{3 \cdot n}{4}\right)$$

или

$$T(n) = C \cdot n + T\left(\frac{n}{5}\right) + T\left(\frac{3 \cdot n}{4}\right).$$

Решая данное уравнение методом подстановки ($g(n) = 20 \cdot C \cdot n$) или методом рекурсивных деревьев, получаем, что время работы алгоритма $\mathrm{O}(n)$.

Следует отметить, что если исходную последовательность разбивать на семерки, то рекуррентное уравнение будет иметь вид

$$T(n) = C_3 \cdot n + T\left(\frac{n}{7}\right) + T\left(\frac{3 \cdot n}{4}\right) \le \frac{28}{3} \cdot C_3 \cdot n\,.$$

Упражнение. Выписать рекуррентное уравнение, если элементы разбиваются на группы по *m* элементов. Провести анализ трудоемкости алгоритма в зависимости от выбранного значения *m*.

Реализация функций сортировки в высокоуровневы языках языках программированиях

Во многих высокоуровневых языках программирования реализованы функции, позволяющие выполнить упорядочивание элементов.

C++ *std::sort*()

Основой служит алгоритм быстрой сортировки – модифицированный *QuickSort*, он же *IntroSort*, разработанный специально для *stl*. Отличие этого алгоритма от *QuickSort* состоит в том, что количество рекурсивных вызовов осуществляется не до самого конца. Если количество итераций (процедур разделения массива) превысило величину $1{,}5 \log_2 n$, где n – количество элементов массива, то рекурсивные операции прекращаются. Затем поступают следующим образом:

- ✓ если количество элементов в рассматриваемой области меньше 32-х, то их упорядочивание выполняется алгоритмом вставки *InsertionSort* ;
- ✓ если количество элементов в рассматриваемой области болеше, чем 32, то для их упорядочивания применяется пирамидальный метод *HeapSort*.

К недостатком данной сортирровки следует отнести то, что она в худшем случае работает за квадратичное время и не устойчива (порядок следования одинаковых элементов в исходном массиве и упорядоченном не сохраняется).

Java *java.util.Collctions.sort*()

Сортировка реализована на базе сортировки слиянием. Данная сортировка была выбрана разработчиками из-за её устойчивости и показывает лучшую производительность по сравнению с другими устойчивыми алгоритмами сортировками, как, например, обменный алгоритм сортировки пузырьком.

Python *sort*() и *sorted*()

Функции в сортировки в *Python* реализуют алгоритм *Tim Sort*, основанный на сортировке слиянием и сортировке вставкой.

Основная идея алгоритма состоит в следующем.

По специальному алгоритму входной массив разделяется на подмассивы. Для этого сначала выбирают минимальную длину подотрезка, на которые будет делиться исходный отрезок. Так как эта длина должна быть не сильно большой, чтобы сортировка вставками работала быстро, а также количество отрезков $^{n}/_{minrun}$ должно быть примерно степенью двойки, чтобы слияние отрезков также работало быстро. Экспериментально Тим Петерсон получил, что лучше всего подходят числа из диапазона [32, 64]. Таким образом, выбирая *minrun*, берут первые 6 бит числа *n* и прибавляют 1, если в оставшихся битах есть хотя бы одна единица. Определение числа $^{n}/_{minrun}$ занимает время $O(\log n)$.

Каждый подмассив сортируется сортировкой вставками, а затем отсортированные подмассивы собираются в единый массив с помощью модифицированной сортировки слиянием.

Оценим сначала время работы сортировки вставками. На случайных данных каждый подотрезок имеет длину не более *minrun*. А значит совершенное количество сдвигов равно количеству инверсий в подотрезке и не превышает

$$\frac{minrun*(minrun-1)}{n}.$$

Так как всего подотрезков не более $^{n}/_{minrun}$, то общее время работы не превысит $n * minrun$, что примерно равно $n \log n$.

Теперь рассмотрим процедуру слияния подмассивов. На случайных данных, выделенные подмассивы будут иметь примерно одинаковую длину, не превышающую *minrun*. При слиянии длина полученного массива примерно в 2 раза превышает длину исходных, а значит каждый массив будет участвовать примерно в $\log n$ операциях слияния, а каждый элемент примерно в $\log n$ сравнениях. Таким образом общее количество сравнений будет примерно равно $O(n \log n)$.

См. ссылку *https://neerc.ifmo.ru/ wiki/index.php?title=Timsort*