**РАЗДЕЛ 6. БАЗОВЫЕ АЛГОРИТМЫ ПСОРТИРОВКИ**

6.1. Общая характеристика алгоритмов сортировки

Операции сортировки находят свое применение практически во всех сферах человеческой деятельности. Алгоритмы сортировок являются одними из наиболее распространенных в IT-технологиях обработки данных. В общем случае задачу сортировки можно сформулировать следующим образом: существует последовательность однотипных записей, одно поле из которых выбрано в качестве ключевого (*ключ сортировки*). Требуется преобразовать исходную последовательность в последовательность, содержащую те же записи, но в порядке возрастания (или убывания) значений ключа. Цель сортировки – облегчение последующего поиска элементов в отсортированном множестве.

В настоящее время разработано достаточное количество алгоритмов сортировки, обладающих различными возможностями применения в зависимости от числа элементов в наборе. Одни алгоритмы просты в реализации и подходят для небольших наборов входных данных, однако в случае больших наборов они требуют бóльшего времени. Другие алгоритмы эффективны для сортировки больших наборов данных, но их использование для малых наборов просто не эффективно. Кроме того, алгоритмы различаются по сложности реализации. Понимание некоторых из них требуют визуального сопровождения, в частности, с помощью дракон-диаграмм для представления логики алгоритма и поясняющих иллюстраций. Наборы данных в виде среза практически охватывают и массивы, поэтому в данном разделе рассматривается сортировка среза.

6.2. Сортировка «пузырьком»

Cортировка «пузырьком» - простейший алгоритм, легко реализуемый для наборов с небольшим количеством элементов. Свое название алгоритм получил вследствие того, что бóльшие значения постепенно «всплывают» в конце набора. Дракон-диаграмма алгоритма представлена на рис. 6.1. Реализация алгоритма сортировки методом «пузырька» состоит из двух модулей: *main()*  и  *bubblesort (ar [] int).*

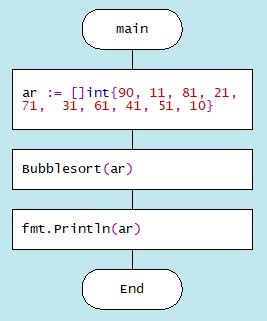
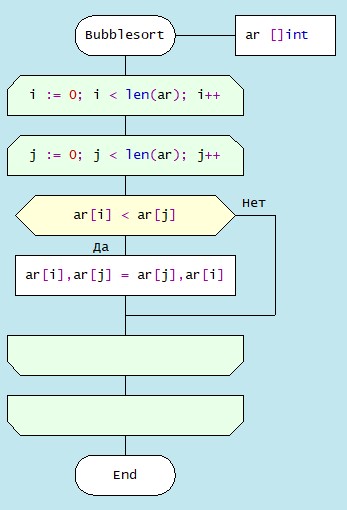
 

Рис. 6.1. Дракон-диаграмма алгоритма «пузырьковой» сортировки

В этом алгоритме проход по набору осуществляется по индексу (j) для каждого индекса (i). При этом сравниваются каждые пары значений ar[i] и ar[j]. Если ставится задача отсортировать значения в порядке возрастания, тогда два элемента меняются местами, если значение элемента ar[j] меньше значения элемента ar[i]. В противном случае происходит переход к следующему проходу по индексу (i). Таким образом, самые большие значения появляются в конце набора. На рис.6.2. представлены фрагмент алгоритма при i = 8, а на рис.6.3. отображены только строки, в которых произошел обмен элементами.

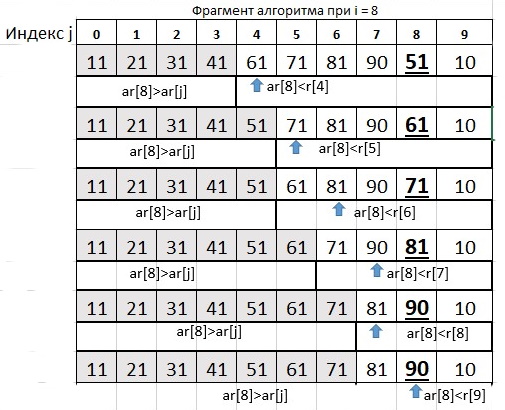


Рис.6,2. Фрагмент процесса сортировки «пузырьком»

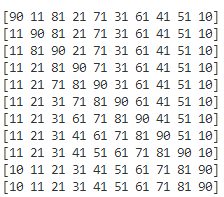


Рис.6.3. Строки набора, в которых состоялся обмен элементами

Очевидно, что временнáя сложность этого алгоритма достаточно высока О(n2), поскольку определяется числом проверок условия ar[i]<ar[j] и числом обменов ar[i] ar[j]. В то же время пространственная сложность «пузырькового» алгоритма составляет О(1), поскольку не требует дополнительной памяти для организации вычислительного процесса. Алгоритм обладает высоким уровнем стабильности.

Благодаря своей простоте, Bubble-сортировка часто используется, например, в компьютерной графике, где он популярен за свою способность обнаруживать незначительную ошибку в почти отсортированных массивах и исправлять её с линейной сложностью (2n) . Однако для сортировки больших наборов данных «пузырьковый» алгоритм крайне неэффективен.

|  |  |
| --- | --- |
| Пути доступа к программным файлам (Bubble sort) | |
| Дракон-диаграмма | <https://github.com/ISA-victory/dsa-dg.git>, |
| Сгенерированный код | https://github.com/ISA-victory/dsa-dg.git |

6.3. Сортировка выбором (Selection Sort)

Алгоритм сортировки выбором основан на операциях сравнения, в котором набор данных делится на две части: отсортированную - в левой части и неотсортированную - в правой. Дракон-диаграмма алгоритма сортировки выбором представлена на рис. 6.4.

|  |  |
| --- | --- |
|  |  |

Рис. 6.4. Дракон-диаграммы алгоритма сортировки методом выбора

Сортировка выбором выполняется путём получения наименьшего значения в каждой итерации, а затем его замены на текущий индекс. Иyначально отсортированная часть пуста, а несортированная часть — весь набор. Наименьший элемент выбирается из несортированного массива и заменяется самым левым элементом, и этот элемент становится частью отсортированного массива. Этот процесс продолжает перемещение неотсортированной границы массива на один элемент вправо. Например, дан набор целых чисел [90, 12, 83, 24, 75, 38, 62, 41, 59, 10].  В первой позиции, где в настоящее время хранится 90, алгоритм проходит весь набор и находит наименьшее значение – 10, после чего эти два значения меняются местами. Далее такой процесс применяется к остальным элементам в срезе (рис. 6.5.):

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 90 | 75 | 83 | 24 | 12 | 38 | 62 | 41 | 59 | 10 |
| 10 | 75 | 83 | 24 | 12 | 38 | 62 | 41 | 59 | 90 |
| 10 | 12 | 83 | 24 | 75 | 38 | 62 | 41 | 59 | 90 |
| 10 | 12 | 24 | 83 | 75 | 38 | 62 | 41 | 59 | 90 |
| 10 | 12 | 24 | 38 | 75 | 83 | 62 | 41 | 59 | 90 |
| 10 | 12 | 24 | 38 | 41 | 83 | 62 | 75 | 59 | 90 |
| 10 | 12 | 24 | 38 | 41 | 59 | 62 | 75 | 83 | 90 |
| 10 | 12 | 24 | 38 | 41 | 59 | 62 | 75 | 83 | 90 |
| 10 | 12 | 24 | 38 | 41 | 59 | 62 | 75 | 83 | 90 |
| 10 | 12 | 24 | 38 | 41 | 59 | 62 | 75 | 83 | 90 |

Рис. 6.5. Процесс выбора и замены элементов среза

Оценка сложности алгоритма сортировки выбором представлена в твблице.

|  |  |
| --- | --- |
| Временнáя сложность: |  |
| Худший случай | O(n2) |
| Средний случай | O(n2) |
| Лучший случай | O(n2) |
| Пространственная сложность: | О(1) |
| Сортировка выбором - нестабильна |  |

|  |  |
| --- | --- |
| Пути доступа к программным файлам (Selection Sort) | |
| Дракон-диаграмма | <https://github.com/ISA-victory/dsa-dg.git>, |
| Сгенерированный код | https://github.com/ISA-victory/dsa-dg.git |

6.4. Сортировка методом вставки (Insertion Sort)

Сортировка вставки происходит путем многократного извлечения элемента из несортированной части набора и последующей вставки в отсортированную часть набора до тех пор, пока не будут вставлены все элементы. Этот алгоритм обычно используется людьми при сортировке стопок бумаг. Дракон-диаграмма алгоритма сортировки методом вставки представлена на рис. 6.6.

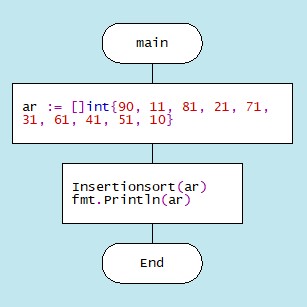
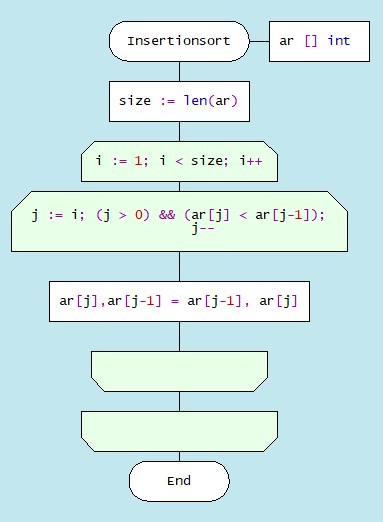
 

Рис. 6.6. Дракон-диаграмма алгоритма сортировки методом вставки

В данном примере элементы с максимальным значением «продвигаются» вправо, после чего выполняется цикл, в котором сравниваются соседние элементы и при необходимости меняются местами (отмечено подчеркиванием “\_”).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i = 1 | i = 4 | i = 8 |  |  |  |  |  |
| [11 **90** 81 21 71 31 61 41 51 10] | [11 21 71 81 **90** 31 61 41 51 10] | [11 21 31 41 51 61 71 81 90 **10**] |
| [11 81 **90** 21 71 31 61 41 51 10] | [11 21 71 81 31 **90** 61 41 51 10] | [11 21 31 41 51 61 71 81 **10** 90] |
| i = 2 | [11 21 *71* 31 81 **90** 61 41 51 10] | [11 21 31 41 51 61 71 **10** 81 90] |
| [11 81 **90** 21 71 31 61 41 51 10] | [11 21 31 *71* 81 **90** 61 41 51 10] | [11 21 31 41 51 61 **10** 71 81 90] |
| [11 81 21 **90** 71 31 61 41 51 10] | i = 5 | [11 21 31 41 51 **10** 61 71 81 90] |  |  |  |  |
| [11 21 81 **90** 71 31 61 41 51 10] | [11 21 31 71 81 **90** 61 41 51 10] | [11 21 31 41 **10** 51 61 71 81 90] |
| i = 3 | i = 7 | [11 21 31 **10** 41 51 61 71 81 90] |  |  |  |  |
| [11 21 81 **90** 71 31 61 41 51 10] | [11 21 31 41 61 71 81 **90** 51 10] | [11 21 **10** 31 41 51 61 71 81 90] |
| [11 21 81 71 **90** 31 61 41 51 10] | [11 21 31 41 61 71 81 51 **90** 10] | [11 **10** 21 31 41 51 61 71 81 90] |
| [11 21 71 81 **90** 31 61 41 51 10] | [11 21 31 41 61 71 51 81 90 10] | [**10** 11 21 31 41 51 61 71 81 90] |
|  | [11 21 31 41 61 51 71 81 90 10] | i = 9 |  |  |  |  |  |
|  | [11 21 31 41 51 61 71 81 90 10] | [10 11 21 31 41 51 61 71 81 90] |

Рис. 6.7. Вставка и замена элементов срезов

Сложность алгоритма сортировки вставкой определяется количеством сравнений и перемещений элементов, которые необходимо выполнить для упорядочивания массива. Это зависит от того, как изначально отсортирован массив.

Наихудший сценарий — когда массив сортируется в обратном порядке. В этом случае каждый элемент следует сравнить со всеми предыдущими элементами и переместить в начало массива. Количество сравнений и ходов равно

(n(n-1)/2,

что ведет к O(n2).

Средний случай: когда массив частично отсортирован. При этом каждый элемент следует сравнить в среднем с половиной предыдущих элементов и переместить на соответствующую позицию. Количество сравнений и движений равно

n2/4,

что ведет к O(n2).

Лучше всего, когда массив уже отсортирован. При этом каждый элемент следует сравнить только с одним предыдущим элементом и оставить на своем месте. Количество сравнений и ходов равно

(n – 1),

что ведет к O(n).

Оценка сложности алгоритма сортировки вставкой представлена в таблице.

|  |  |
| --- | --- |
| Временнáя сложность: |  |
| Худший случай | O(n2) |
| Средний случай | O(n2) |
| Лучший случай | O(n) |
| Пространственная сложность: | О(1) |
| Сортировка вставками - стабильна |  |

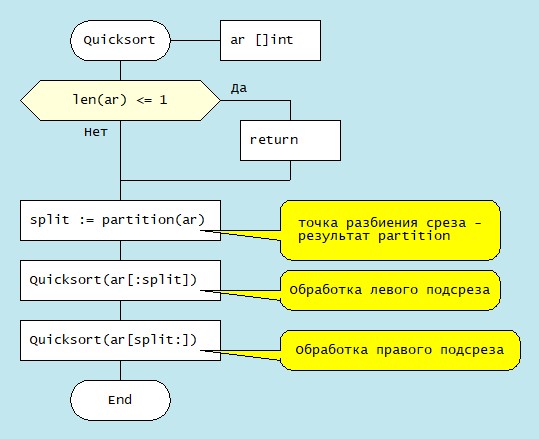
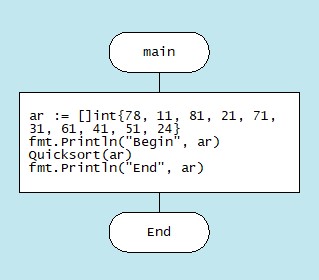
|  |  |
| --- | --- |
| Пути доступа к программным файлам (Insetion Sort) | |
| Дракон-диаграмма | <https://github.com/ISA-victory/dsa-dg.git>, |
| Сгенерированный код | https://github.com/ISA-victory/dsa-dg.git |

6.5. Быстрая сортировка выбором (Quick Sorting)

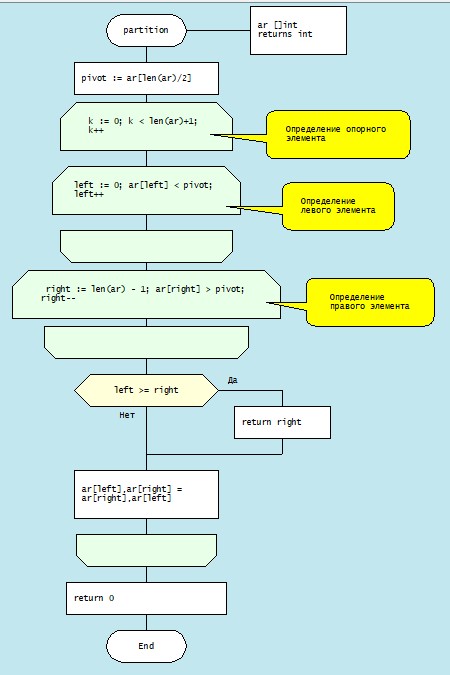
Быстрая сортировка является высокоэффективным алгоритмом сортировки, общая схема которого состоит их следующих этапов:

1. Выбор опорного элемент из среза.
2. Перераспределение элементов в срезе таким образом, что элементы, меньшие опорного, помещаются перед ним, а большие или равные - после.
3. Рекурсивное применение первых предыдущих шагов к фрагментам среза слева и справа от опорного элемента.
4. В результате формируется полностью отсортированный массив.

Дракон-диаграммы алгоритма быстрой сортировки представлена на рис.6.6.



а) функция main б) функция Quicksort



в) функция Partition

Рис. 6.8. Дракон-диаграмма алгоритма быстрой сортировки

Рассмотрим этот алгоритм подробнее. В функции main представлена коллекция (набор) целочисленных данных. Опорным элементом pivot выбран элемент среза со значением 31.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 78 | 11 | 81 | 21 | 71 | 31 | 61 | 41 | 51 | 28 |

Введем два указателя: left и right. В начале алгоритма они указывают, соответственно, на левый и правый конец набора. В представленном алгоритме указатель left сдвигается с шагом в 1 элемент по направлению к концу среза, пока значение текущего элемента будет меньше значения опорного элемента. Индекс первого элемента, значение которого будет больше pivot (ar[left] > pivot), фиксируется в переменной left. Затем аналогичным образом в алгоритме указатель right перемещается от конца среза к началу, пока не будет найден элемент, для которого выполняется условие ar[right] <= pivot. Этот фрагмент алгоритма выполняется до тех пор, пока слева не будет обнаружен элемент, значение которого превысит значение правого элемента справа. В этом случае эти элементы меняются местами. В приведенном примере первый же (левый) элемент больше опорного (78 >31) и последний (правый) элемент меньше опорного (24 < 31). Во второй строчке таблицы они поменялись местами. Далее процесс продолжается (рис.6.7.):

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Индекс | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |  |
| Левый |  |  |  |  |  |  |  |  |  |  | Правый |
|  | 78 | 11 | 81 | 21 | 71 | 31 | 61 | 41 | 51 | 24 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  | 24 | 11 | 81 | 21 | 71 | 31 | 61 | 41 | 51 | 78 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  | 24 | 11 | 31 | 21 | 71 | 81 | 61 | 41 | 51 | 78 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  | 24 | 11 | 21 | 31 | 71 | 81 | 61 | 41 | 51 | 78 |  |

Рис. 6.9. Фрагмент выполнения алгоритма QuickSort

Затем формируется новый фрагмент исходного набора, в котором переопределяется опорный элемент. В этом фрагменте pivot = 11.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |  |
| 24 | 11 | 21 |
|  | |  | | --- | |  | |  |
| 11 | 24 | 21 |
|  |  |  |
| 11 | 21 | 24 |

После обработки левой части набора таким же образом обрабатывается правая часть. В конечном счете, срез становится отсортированным:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 11 | 21 | 24 | 31 | 41 | 51 | 61 | 71 | 78 | 81 |

Этот алгоритм довольно эффективен для больших наборов данных, так как его средняя и наихудшая сложность O(n)2), соответственно.

Анализ сложности:

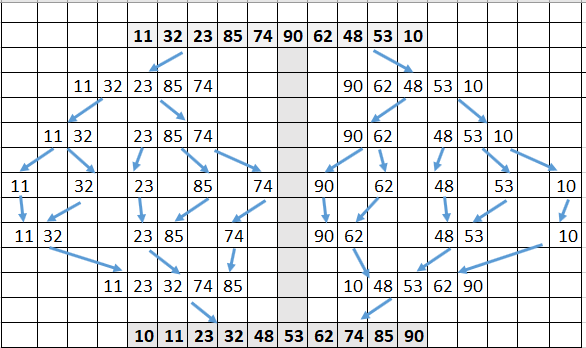
|  |  |
| --- | --- |
| Временнáя сложность: |  |
| Худший случай | O(n2) |
| Средний случай | O(nlogn) |
| Лучший случай | O(nlogn) |
| Пространственная сложность: | O(nlogn) |
| Быстрая сортировка - нестабильна |  |

Worst Case Time Complexity O(n 2 ) Best Case Time Complexity O(nlogn) Average Time Complexity O(nlogn) Space Complexity O(nlogn) Stable Sorting No

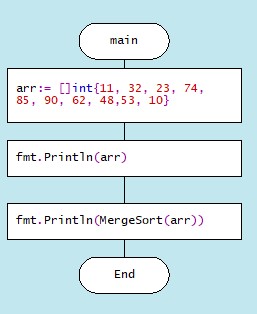
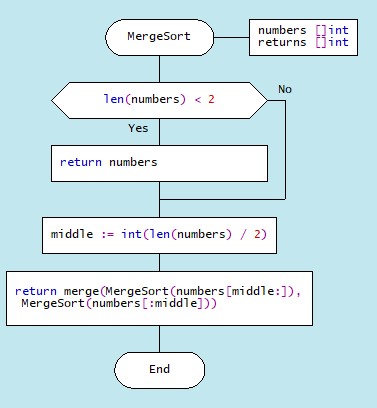
|  |  |
| --- | --- |
| Пути доступа к программным файлам (Quick Sort) | |
| Дракон-диаграмма | <https://github.com/ISA-victory/dsa-dg.git>, |
| Сгенерированный код | https://github.com/ISA-victory/dsa-dg.git |

6.6. Сортировка слиянием (Merge sort)

Сортировка слиянием выполняется с помощью рекурсивного разбиения набора (среза) на фрагменты до тех пор, пока не останется фрагмент, состоящий из двух элементов. Два элемента легко сравнить между собой и упорядочить в зависимости от требования: по возрастанию или убыванию. После разбиения следует обратное слияние, при котором в один момент времени (или за проход цикла) выбираются по одному элементу от каждого фрагмента среза с последующим их сравнением. Наименьший (или наибольший) элемент сохраняется в результирующем наборе, оставшийся элемент остается актуальным для сравнения с элементом из другого фрагмента на следующем шаге (рис.6.8.):

 Рис. 6.8. Визуализация алгоритма сортировки слиянием

Дракон-диаграмма алгоритма сортировки слиянием представлена на рис. 6.9.

а) б)

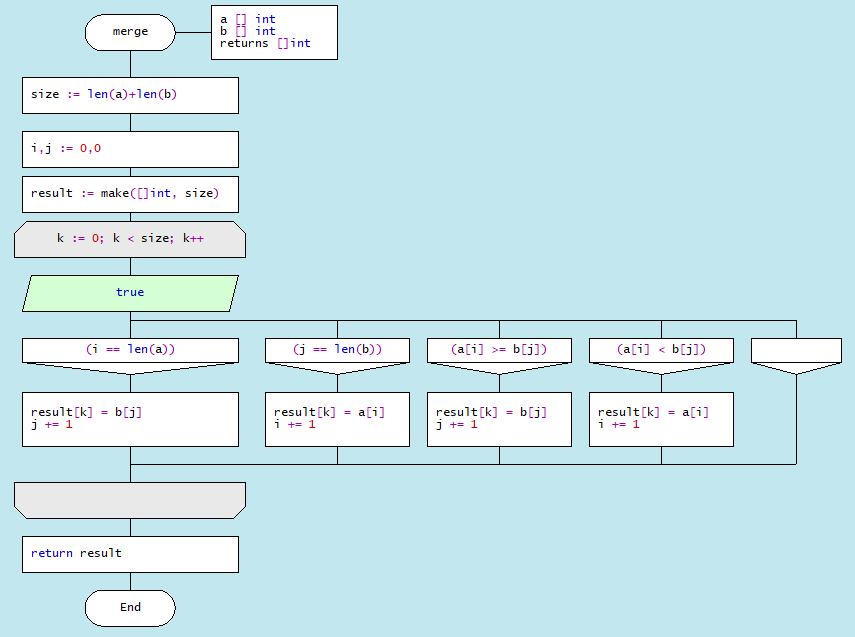


Рис. 6.9. Дракон-диаграмма алгоритма сортировки слиянием:

а) функция main; б) функция MergeSort; d) функция merge

Анализ сложности:

|  |  |
| --- | --- |
| Временнáя сложность: |  |
| Худший случай | O(nlogn) |
| Средний случай | O(nlogn) |
| Лучший случай | O(nlogn) |
| Пространственная сложность: | O(n) |
| Сортировка слиянием - стабильна |  |

|  |  |
| --- | --- |
| Пути доступа к программным файлам (Merge Sort) | |
| Дракон-диаграмма | <https://github.com/ISA-victory/dsa-dg.git>, |
| Сгенерированный код | https://github.com/ISA-victory/dsa-dg.git |

6.6. Shell сортировка

Шелл-сортировка (ShellSort) представляет собой разновидность сортировки вставками (Insertion Sort). При сортировке Шелла сначала сравниваются и сортируются между собой значения, отстоящие один от другого на некотором расстоянии d. После этого процедура повторяется для некоторых меньших значений d до тех пор пока расстояние не станет d=1 (то есть обычной сортировкой вставками). Дракон-диаграмма алгоритма представлена на рис. 6.10.

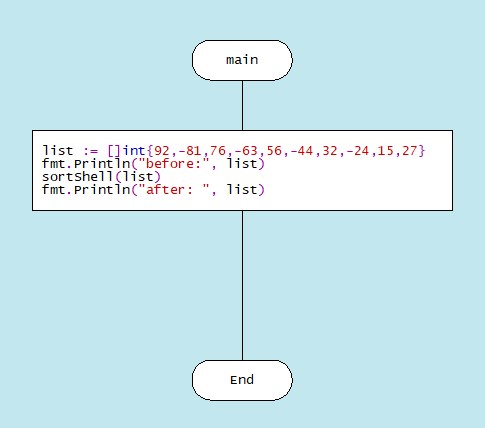
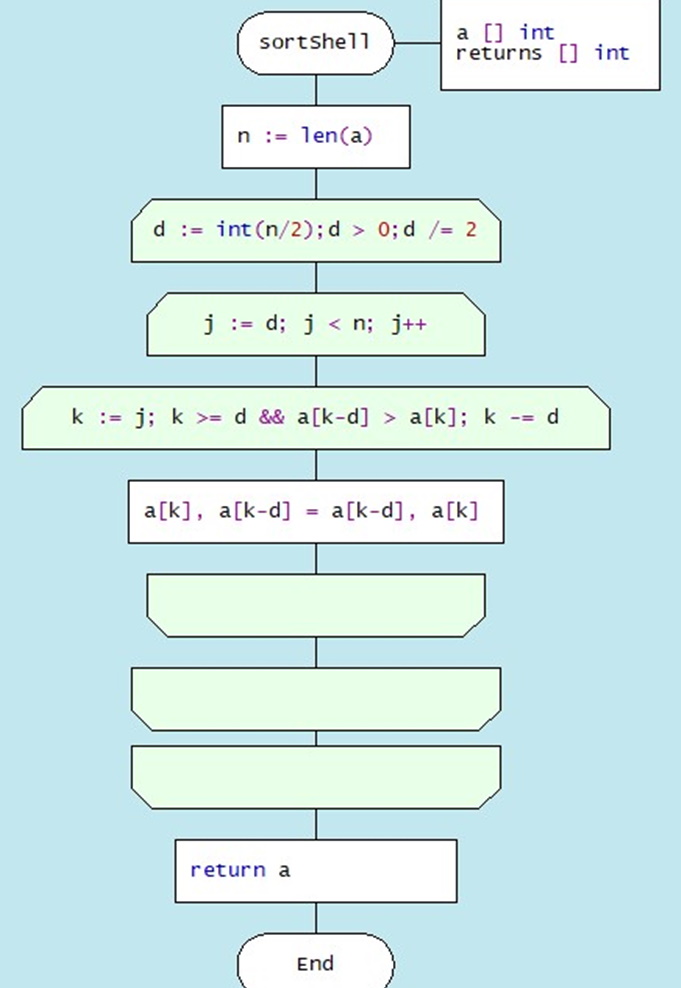
 

Рис. 6.10. Дракон-диаграммы алгоритма Shell-сортировки

Shellsort использует процесс, который лежит в основе многих представленных сортировок: последовательное разбиение данных на сегменты, сортировка этих сегментов и, наконец, объединение в отсортированный набор. Процесс разбиения происходит так, чтобы каждый элемент в сегмента представлял собой фиксированное количество позиций друг от друга. При этом возникает неопределенность в выборе этого числа позиций, иными словами расстояния между элементами в сегменте (d). Самый простой пример это d = n / 2, d2 = d/2 … dn =1.

На рис. 6.11 показан обмен элементами при условии a[k] < a[k-d], где d=5 при первом проходе среза.

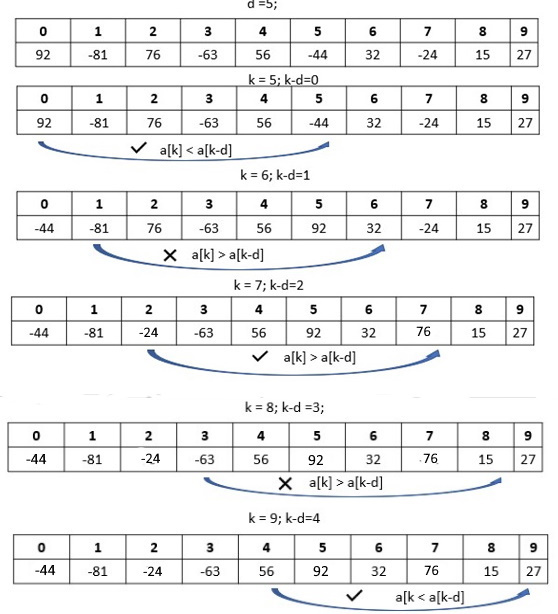


Рис. 6.11. Фрагмент алгоритма обмена элементами набора

Анализ сложности:

|  |  |
| --- | --- |
| Временнáя сложность: |  |
| Худший случай | O(nlogn)2 |
| Средний случай | O(nlogn2) |
| Лучший случай | O(n) |
| Пространственная сложность: | O(1) |
| Сортировка слиянием - нестабильна |  |

|  |  |
| --- | --- |
| Пути доступа к программным файлам (Shell Sort) | |
| Дракон-диаграмма | <https://github.com/ISA-victory/dsa-dg.git>, |
| Сгенерированный код | https://github.com/ISA-victory/dsa-dg.git |

6.7. Пирамидальная сортировка (Heap Sort)

Алгоритм пирамидальной сортировки можно рассматривать как улучшенную версию алгоритма сортировки выбором (Select Sort): он делит входные данные на отсортированную и несортированную области, а затем последовательно уменьшает несортированную область, извлекая самый большой элемент и перемещая его в сортированную область. Улучшение состоит в том, что для нахождения наибольшего значения используется бинарная куча, а не алгоритм линейного поиска. Этот алгоритм выполняется c использованием понятия *кучи,* являющейся полным двоичным деревом (см. подраздел 1.3.). Все узлы кучи либо больше, чем ее дочерние элементы, либо меньше, чем ее дочерние элементы. Кучное двоичное дерево может быть двух типов: минимальная куча (MinHeap), в которой родительский узел всегда меньше дочерних узлов, и максимальная куча (MaxHeap), в которой родительский узел всегда больше или равен дочерним узлам (рис. 6.12).

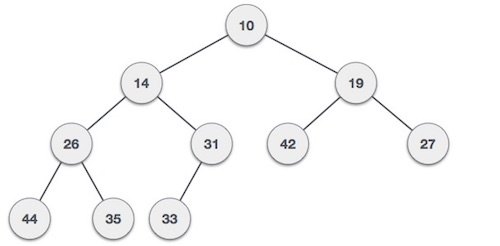
 

Рис. 6.12. Примеры двоичного дерева

Последовательность замены узлов дерева, начиная с корневого узла, осуществляется по формуле:

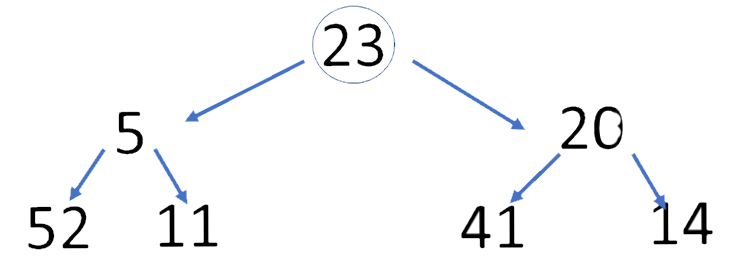
****

Рис. 6.13. Пример построения двоичного дерева

Покажем последовательность замены узлов дерева, начиная с корневого узла, определяемого по формуле:

iн = (размер массива / 2) – 1;

Вначале по алгоритму меняются местами узлы (20) и (41), затем узлы (5) и (52), далее представим этот процесс в таблице:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 23 | 5 | 20 | 52 | 11 | 41 | 14 |
| 23 | 5 | 41 | 52 | 11 | 20 | 14 |
| 23 | 52 | 41 | 5 | 11 | 20 | 14 |
| 52 | 23 | 41 | 5 | 11 | 20 | 14 |
| 14 | 23 | 41 | 5 | 11 | 20 | 52 |
| 41 | 23 | 14 | 5 | 11 | 20 | 52 |
| 20 | 23 | 14 | 5 | 11 | 41 | 52 |
| 23 | 20 | 14 | 5 | 11 | 41 | 52 |
| 11 | 20 | 14 | 5 | 23 | 41 | 52 |
| 20 | 11 | 14 | 5 | 23 | 41 | 52 |
| 5 | 11 | 14 | 20 | 23 | 41 | 52 |
| 11 | 5 | 14 | 20 | 23 | 41 | 52 |
| 5 | 11 | 14 | 20 | 23 | 41 | 52 |

Алгоритм сортировки кучи использует три функции: heap\_Sort, осуществляющая перебор узлов, функция heapify выполняет сравнение смежных узлов, а функция swap меняет местами два узла. Последовательность перемещения узлов в куче показана на рис. 6.14:

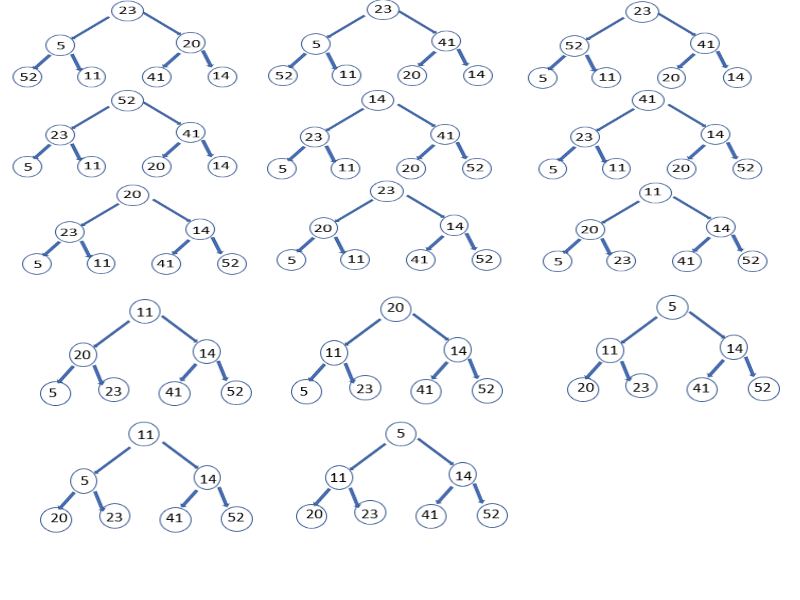
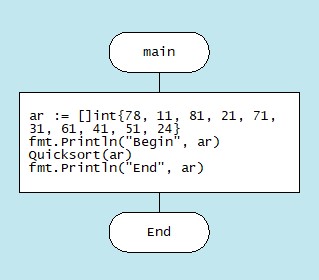


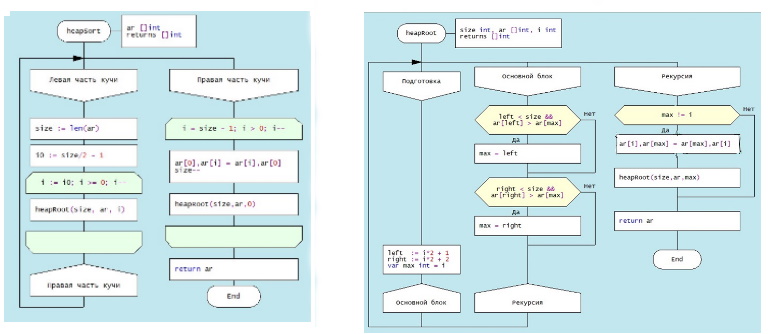
Рис. 6.14. Последовательность перемещения узлов в куче

DRAKON-диаграмма алгоритма кучи представлена на рис. 6.15:





а) функция main



б) функция heapsort в) функция heapRoot

Рис. 6.15. Дракон-диаграммы алгоритма сортировки кучи

Анализ сложности:

|  |  |
| --- | --- |
| Временнáя сложность: |  |
| Худший случай | O(nlogn). |
| Средний случай | O(nlogn). |
| Лучший случай | O(nlogn). |
| Пространственная сложность: | О(1) |
| Сортировка слиянием - нестабильна |  |

|  |  |
| --- | --- |
| Пути доступа к программным файлам (Heap Sort) | |
| Дракон-диаграмма | <https://github.com/ISA-victory/dsa-dg.git>, |
| Сгенерированный код | https://github.com/ISA-victory/dsa-dg.git |

6.8. Сравнительный анализ сортировок

Выбор того или иного алгоритма сортировки определяется следующими факторами:

* + - Временнáя сложность;
    - Пространственная сложность;
    - Стабильность/нестабильность;

Знание сильных и слабых сторон каждого из рассмотренных алгоритмов позволяет сделать выбор в пользу того или иного вида сортировки. Каждый алгоритм уникален и лучше всего работает при определенных условиях.

6.8.1. Сравнение временной сложности

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм сортировки | Средняя | Лучшая | Худшая |
| [Bubble Sort](https://coderslegacy.com/python/bubblesort-algorithm/) | O(n2) | O(n) | O(n2) |
| [Selection Sort](https://coderslegacy.com/python/selection-sort-algorithm/) | O(n2) | O(n2) | O(n2) |
| [Insertion Sort](https://coderslegacy.com/python/insertion-sort-algorithm/) | O(n2) | O(n) | O(n2) |
| [Quick Sort](https://coderslegacy.com/python/quicksort-algorithm/) | O(n.log(n)) | O(n.log(n)) | O(n2) |
| [Merge Sort](https://coderslegacy.com/python/merge-sort-algorithm/) | O(n.log(n)) | O(n.log(n)) | O(n.log(n)) |
| Shell Sort | n(log(n)2 | O(n) | n (log n)2 |
| [Heap Sort](https://coderslegacy.com/python/heap-sort-algorithm/) | O(n.log(n)) | O(n.log(n)) | O(n.log(n)) |

Некоторые распространенные алгоритмы сортировки являются стабильными по своей природе, такие как сортировка слияний, сортировка по количеству, сортировка вставок и сортировка пузырей. Другие, такие как Quicksort, Heapsort и Selection Sort, нестабильны. ... Например, мы можем использовать дополнительное пространство для поддержания стабильности в Quicksort.