Работа №5

«Алгоритмы сортировки»

Цель работы

Знакомство с основными алгоритмами сортировки данных

После выполнения работы студенты могут

- Разрабатывать программный код, сортирующий массивы данных
- Оценивать эффективность работы используемого алгоритма
- Выбирать наилучший алгоритм сортировки, исходя из особенностей задачи
- Комбинировать различные алгоритмы сортировки для достижения наилучшего результата

Теоретическая часть

Задача сортировки данных возникает в программировании сравнительно часто. К примеру, при получении набора данных, когда необходимо выбрать 10 наибольших значений, следует произвести сортировку принятого массива данных по убыванию и выбрать первые 10 значений. Задача сортировки часто возникает при работе с базами данных, файлами на диске и т. д. В то же время, сортировка представляет собой хороший пример задачи, развивающей алгоритмическое мышление и дающей представление о вычислительной сложности.

Сортировка — это упорядочивание элементов в массиве или списке. В случае работы с простыми массивами чисел или строк, сами числа или строки являются значениями, подлежащими сортировке. Если элементами массива являются структуры данных, содержащие несколько полей, то одно из полей, являющееся критерием сортировки, выбирается в качестве **ключа**, а остальные данные перемещаются вслед за ключом, но на поведение алгоритма сортировки не влияют.

Различают сортировку по **неубыванию** и по **невозрастанию**. В первом случае элементы списка должны быть выстроены таким образом, чтобы для любых двух элементов списка выполнялось соотношение $A_i \leq A_{i+1}$, где A_i и A_{i+1} –

два последовательно стоящих элемента списка. Для сортировки по невозрастанию данное соотношение записывается наоборот: $A_i \geq A_{i+1}$.

Один из важных вопросов при сортировке — это скорость работы алгоритма и затраты памяти. Для оценки данных параметров необходимо ввести понятие вычислительной сложности.

Вычислительная сложность — это функция зависимости объема работы, которая выполняется некоторым алгоритмом, от размера входных данных. Объем работы измеряется абстрактными понятиями времени и пространства, называемыми вычислительными ресурсами. Время определяется количеством элементарных шагов, необходимых для выполнения алгоритма, а пространство — объемом памяти, который ему требуется.

Очевидно, что конкретное время выполнения алгоритма будет зависеть от платформы, на которой он выполняется. Так, применение процессора с большей тактовой частотой позволит выполнять больше элементарных операций в единицу времени и сократит время работы. Однако зависимость времени работы алгоритма от объема решаемой задачи будет носить одинаковый характер при выполнении на разных платформах.

Общепринятой формой записи является запись при помощи **асимптотической сложности** алгоритма. Она показывает, какой характер будет иметь вычислительная сложность алгоритма при устремлении объема решаемой задачи к бесконечности. В таких условиях влияние постоянных составляющих и множителей малых порядков оказывается незначительным. На практике характер зависимости вычислительной сложности от объема решаемой задачи сохраняется и на сравнительно малых объемах.

Запись вида O(f(n)) означает, что время работы алгоритма (количество операций) или расход памяти ограничены сверху функцией f(n), где n – объем решаемой задачи. Запись вида Ω(f(n)) означает, что время работы алгоритма ограничено снизу функцией f(n). Таким образом, Ω(f(n)) и O(f(n)) представляют собой наилучший и наихудший случаи соответственно.

Для алгоритмов сортировки, объем решаемой задачи n – это размер сортируемого массива данных, а единственными доступными операциями

являются операции сравнения и перестановки. Если алгоритм в процессе работы совершает k сравнений, то возможно 2^k вариантов комбинаций результатов таких сравнений. Количество перестановок для n элементов равно n!. Для того, чтобы можно было провести сюръекцию из множества комбинаций ответов во множество всех перестановок, количество сравнений должно быть не меньше, чем $\log_2 n!$. Можно показать, что функция $\log_2 n!$ асимптотически ограничена снизу функцией $n \log n$, что дает нижнюю оценку вычислительной сложности любого алгоритма сортировки, основанного на сравнении и перестановке. Таким образом, если и верхняя оценка вычислительной сложности алгоритма может быть оценена как $O(n \log n)$, то данный алгоритм можно считать оптимальным.

На практике запись вида O(f(n)) можно понимать следующим образом: время работы алгоритма будет равно $T = C \cdot f(n)$, где C – некая константа, включающая в себя все программные и аппаратные особенности конкретной платформы, а также особенности конкретного алгоритма.

Рассмотрим пример алгоритма со сложностью $O(n^2)$. Замеры показали, что обработка 10^3 элементов таким алгоритмом занимает около 100 миллисекунд. Значит, обработка $2 \cdot 10^3$ элементов займет в 4 раза больше времени и составит 400 миллисекунд. Применив для той же задачи алгоритм со сложностью $O(n \cdot \log n)$ для объема 10^3 элементов, получаем время работы равное 150 мс. Несмотря на то, что $(10^3)^2$ больше, чем $10^3 \cdot \log 10^3$, константа первого алгоритма оказалась существенно меньше. Тем не менее, увеличение количества элементов с 10^3 до $2 \cdot 10^3$ увеличит время выполнения для второго алгоритма всего в 2.2 раза, до 330 мс. Дальнейшее увеличение объема решаемой задачи будет увеличивать разрыв во времени работы между двумя алгоритмами в пользу последнего.

Следует отметить, что результат работы алгоритма на реальных данных сильно зависит от характера самих данных. Даже алгоритм, имеющий сложность $O(n^2)$, может показать время выполнения порядка O(n) на данных определенного характера (например, на практически отсортированных данных). Для каждого алгоритма выделяется наилучшая и наихудшая ситуация, в зависимости от распределения значений в массиве, которые дают, соответственно, наилучшее и наихудшее время выполнения.

Аналогичная нотация используется при оценке пространства, то есть затрат

памяти. При этом не учитывается расход памяти для хранения исходных данных, а также для исходного кода алгоритма. Обычно затраты памяти не превышают O(n). Алгоритмы, не требующие дополнительной памяти, то есть потребляющие O(1) памяти, также называются алгоритмами сортировки на месте (in-place).

Также одной из характеристик алгоритма сортировки является **устойчивость** или **стабильность**. Алгоритм называется устойчивым, если после сортировки элементы с одинаковыми ключами не меняют свой порядок относительно друг друга. Неустойчивый алгоритм не гарантирует сохранения относительного порядка элементов с одинаковым ключом, что может привести к необходимости сортировки подмассивов из элементов с одинаковым ключом, используя какое-то другое значение в качестве ключа. Пример сортировки, требующей устойчивости, является двукратная сортировка по разным полям. Например, сортировка файлов сначала по имени, а потом по типу.

Идеальный алгоритм сортировки должен обладать следующими свойствами:

- Стабильность
- Работа на месте, используя O(1) дополнительного места
- В наихудшем случае алогоритм должен совершать не более O(n log n) сравнений и не более O(n) перестановок
- Адаптивность: алгоритм должен приближаться к O(n) при работе по практически отсортированным данным, или когда данные содержат небольшое количество уникальных ключей (часто повторяются).

Ни один из алгоритмов сортировки не является идеальным, поэтому выбор конкретного алгоритма зависит от характера данных.

В приведенных далее примерах подразумевается сортировка по возрастанию. В случае сортировки по убыванию меняется только критерий перестановки чисел.

Алгоритм сортировки выбором является наиболее простым для написания алгоритмом, имеющим сложность $O(n^2)$ для любого набора входных данных и дополнительные затраты памяти O(1). Он может быть как устойчивым, так и неустойчивым. Данный алгоритм можно описать следующим образом:

1. Объявляется переменная-индекс. Все элементы слева от индекса считаются отсортированными. Изначально индекс указывает на первый по счету элемент массива.
2. Путем последовательного перебора значений справа от индекса выбирается наименьший элемент. Этот элемент меняется местами с элементом по индексу.
3. Индекс увеличивается на 1, и алгоритм возвращается к пункту 2.
4. Алгоритм продолжает работать до тех пор, пока индекс не сравняется с последним элементом массива.

Алгоритм представлен графически на рисунке 1. Стрелка сверху показывает текущее положение индекса, а двойная стрелка снизу — два элемента, заменяемых между собой.

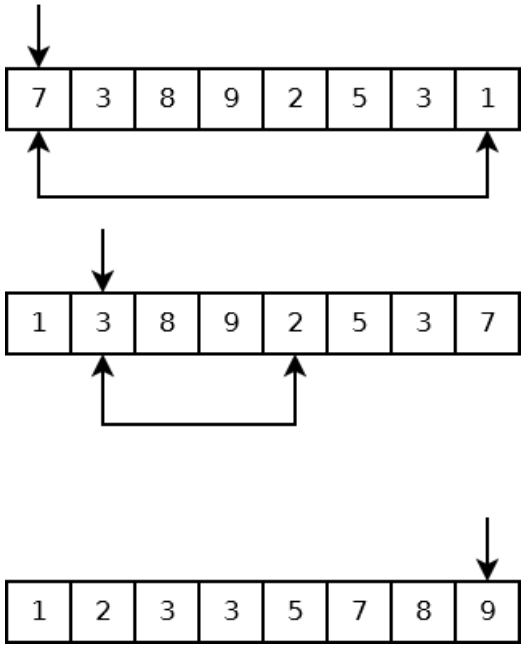

Рисунок 1 - Сортировка выбором

Существенным недостатком данного алгоритма является скорость его работы. Независимо от характера обрабатываемых данных, время его работы всегда оценивается как $O(n^2)$. Тем не менее, он характеризуется наименьшим количеством перестановок, что может быть критичным в некоторых вычислительных системах, а также чрезвычайно прост в написании, что позволяет использовать его при малых значениях n, когда производительность не имеет

принципиального значения.

Приведенная реализация алгоритма является неустойчивой. Для устойчивости следует не менять элементы местами, а вставлять наименьший элемент в конец отсортированных данных, раздвигая массив. Такое поведение вполне оправдано в списках, однако затратно в непрерывных массивах, так как требует перестановки всех элементов правее от того места, куда будет вставлен новый элемент.

Алгоритм сортировки вставками в чем-то похож на сортировку выбором. Алгоритм так же использует указатель, отделяющий отсортированные данные в массиве от неотсортированных, однако вместо поиска наименьшего элемента из неотсортированных данных, алгоритм берет первый неотсортированный элемент и вставляет его в соответствующее место среди отсортированных данных.

Алгоритм записывается следующим образом:

1. Объявляется переменная-индекс. Все элементы слева от индекса считаются отсортированными. Изначально индекс устанавливается в первый по счету элемент массива.
2. Выбирается элемент, следующий за индексом и затем меняется местами с предыдущим элементом до тех пор, пока предыдущий элемент больше обрабатываемого, или пока не достигнуто начало массива.
3. Переменная-индекс увеличивает свое значение на 1.
4. Алгоритм продолжается до тех пор, пока индекс не дойдет до конца массива. После этого данные становятся отсортированными.

Алгоритм представлен графически на рисунке 2. Некоторые шаги алгоритма пропущены, так как на этих шагах не требуется переставлять элементы (промежуточный массив слева от указателя является отсортированным). Стрелка сверху показывает индекс, являющийся границей отсортированных данных, а стрелка снизу — конечное положение элемента после серии перестановок.

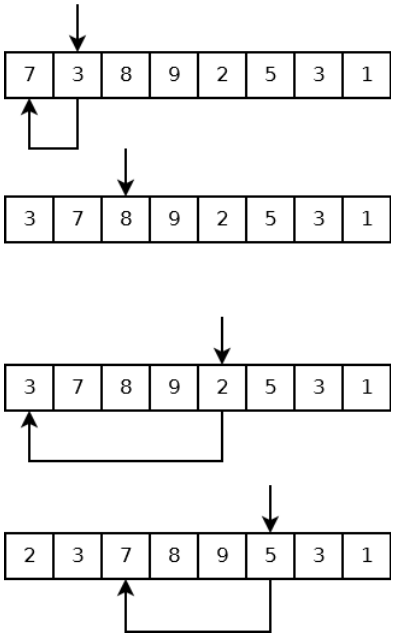

Рисунок 2 - Алгоритм вставки

В среднем вычислительная сложность алгоритма вставки составляет $O(n^2)$, однако, во-первых, данный алгоритм является устойчивым, а, во-вторых, при работе с данными, близкими к отсортированным, в алгоритме резко сокращается количество перестановок, и время работы приближается к O(n). Устойчивость алгоритма обеспечивается тем фактом, что элементы с одинаковым ключом будут выбраны по очереди, без перемешивания, и за счет вставки будут расположены рядом в том же порядке. В силу своей простоты, данный алгоритм используется для небольших массивов или как часть более сложных алгоритмов сортировки. Существуют модификации данного алгоритма, ускоряющие вставку, которые также уменьшают время его работы.

Похожим образом работает алгоритм **пузырьковой сортировки**. Суть этого алгоритма в том, что на каждом этапе сортировки сравниваются два элемента и меняются местами в том случае, если они не совпадают. Проход по массиву можно начинать как с начала, собирая максимальные элементы в конец, так и с конца, собирая минимальные элементы в начале. В обоих случаях самый большой (или самый маленький) элемент «проплывает» через весь массив, как пузырек в жидкости. Отсюда и пошло название метода.

Алгоритм пузырьковой сортировки при сборке больших элементов в конце записывается следующим образом:

1. Объявляется цикл по элементам массива с конца. Переменная цикла определяет, до какого элемента следует двигаться при движении «пузырька». Элементы массива после переменной данного цикла являются отсортированными.
2. Объявляется вложенный цикл от начала массива до значения переменной из предыдущего (внешнего) цикла.
3. Во вложенном цикле меняется местами элемент, на который указывает переменная вложенного цикла, со следующим элементом, если следующий элемент меньше текущего.

На рисунке 3 показано графически несколько шагов работы данного алгоритма. Стрелка показывает текущий предел сортировки (границу отсортированных данных), а двойная стрелка — пару элементов, требующих перестановки.

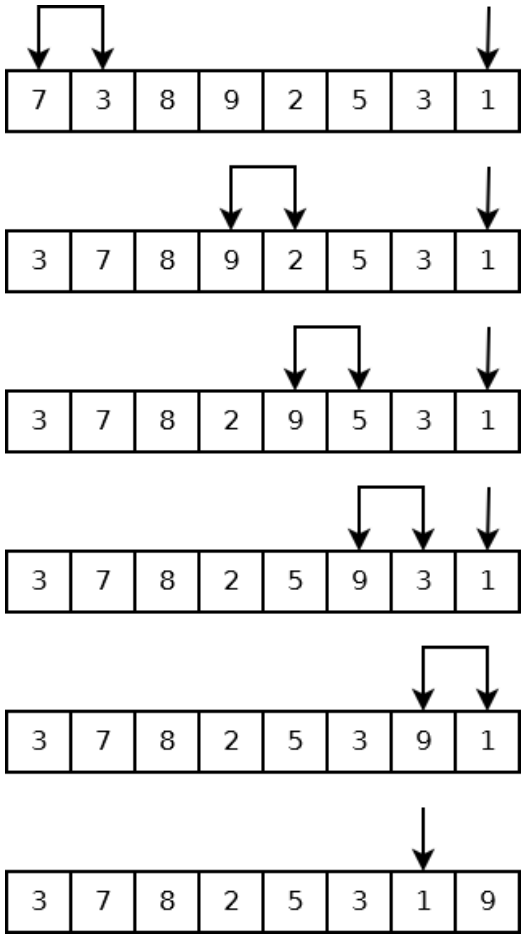

Рисунок 3 - Пузырьковая сортировка

В данном виде алгоритм обладает вычислительной сложностью $O(n^2)$, оказывается устойчивым, но не адаптивным. В частности, на данных, близких к отсортированным, работает так же долго, как и на случайных. Однако, можно добавить флаг, проверяющий во внутреннем цикле, была ли хоть одна перестановка за текущий проход по массиву, и если ни одной перестановки не было, то это может говорить о том, что массив уже полностью отсортирован, и дальнейшая обработка не требуется. Такой подход приближает время выполнения на массивах, близких к отсортированным, к величине $O(n)$.

Пузырьковая сортировка является медленным алгоритмом и на практике применяется крайне редко. Иногда ее совмещают с сортировкой выбором, когда в процессе прохода по массиву и «выдавливания» максимального элемента в конец, также находится минимальный элемент и после прохода ставится в начало. Такой подход уменьшает число итераций циклов, но увеличивает число сравнений и не дает существенного преимущества.

Одним из классических алгоритмов сотрировки является **алгоритм быстрой сортировки**. Он был разработан английским информатиком Чарльзом Хоаром в 1960 году во время его работы в МГУ, а среднее время работы алгоритма оценивается как $O(n \log n)$. Алгоритм представляет собой модификацию алгоритма прямого обмена (один из его вариантов — алгоритм пузырьковой сортировки) путем введения двух простых правил: во-первых, перестановки производятся на максимально возможном расстоянии, а, во-вторых, после каждого прохода элементы делятся на две независимые группы и сортируются по отдельности.

Алгоритм быстрой сортировки можно описать следующим образом:

1. Выбирается опорное значение из массива. От выбора опорного значения сильно зависит скорость работы алгоритма. Хороший результат дает выбор значения элемента из середины массива.
2. Объявляются два указателя: указатель L, который проходит по элементам слева направо, и указатель R, который проходит по элементам справа налево. Конечная цель — разбить массив на две части таким образом, чтобы в левой части находились элементы меньше, а в правой — больше, чем опорное значение.

3. Индекс L постепенно увеличивается на 1 до тех пор, пока элемент, на который он указывает, не окажется больше или равен опорному. Аналогичным образом индекс R уменьшается на 1 до тех пор, пока элемент, на который он указывает, не будет меньше или равен опорному.
4. Если после того, как L и R закончили движение, индекс L не больше, чем R, необходимо поменять местами элементы, на которые указывают L и R, после чего увеличить значение L на единицу и уменьшить значение R на единицу.
5. Пункты 3-4 повторяются до тех пор, пока L не больше, чем R.
6. После того, как элементы L и R пересеклись, полученный массив разделяется на два. Первый из них содержит элементы от первого до R включительно, а второй — от L до последнего включительно. Первый массив включает в себя все элементы меньше опорного, а второй — больше опорного.
7. Каждый из полученных в пункте 6 массивов сортируется тем же методом.

Графическая интерпретация одного такта работы алгоритма быстрой сортировки (от начала работы до первого разделения массива) представлена на рисунке 4. Стрелки с буквами L и R соответствуют одноименным указателям, стрелка M указывает на элемент, значение которого выбрано в качестве опорного, а подсвеченные элементы — это элементы, которые поменялись местами по сравнению с предыдущим шагом.

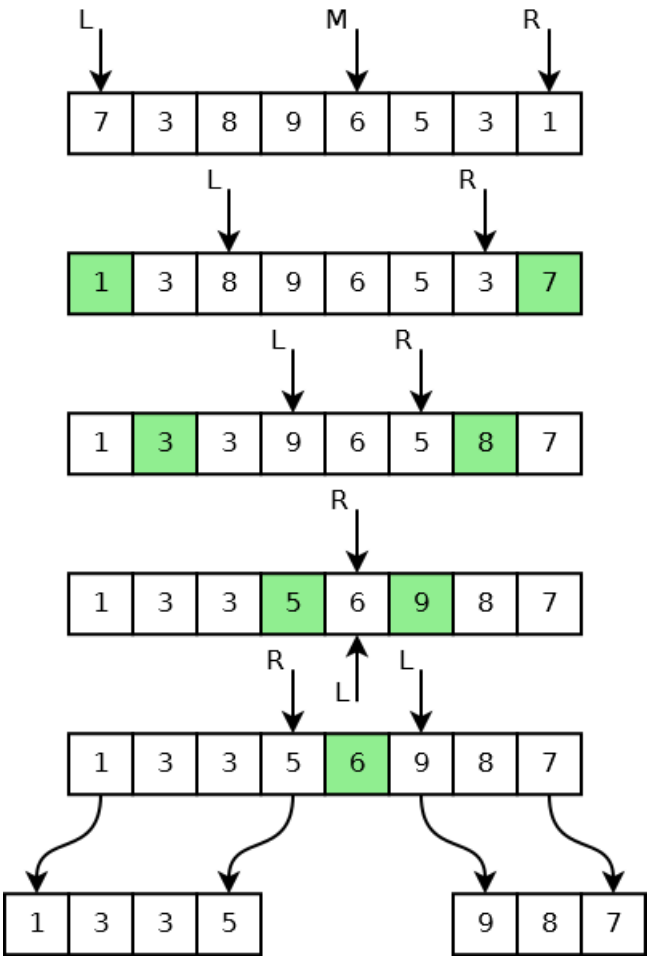

Рисунок 4 - Алгоритм быстрой сортировки

Среднее время работы алгоритма оценивается как O(n log n), что намного быстрее вышеприведенных алгоритмов, однако эффективность его работы очень сильно зависит от характера передаваемых данных и от выбранного опорного элемента. В наихудшем случае после всех перестановок массивы могут разделиться в соотношении 1 и n-1, что практически не даст выигрыша. Если на каждом шаге массивы разбиваются в таком соотношении, алгоритм деградирует до уровня $O(n^2)$, причем константа оказывается больше, чем в других примитивных алгоритмах. На практике, однако, такая ситуация встречается крайне редко, и в среднем алгоритм работает очень быстро.

Также к недостаткам алгоритма следует отнести его неустойчивость и склонность к переполнению стека при рекурсивном вызове (когда два подмассива сортируются путем вызова той же самой функции быстрой сортировки).

Оптимизация данного алгоритма проходит в нескольких направлениях: в направлении устранения его неустойчивости, в направлении выбора наиболее оптимального опорного элемента с целью снижения вероятности получения наихудшего случая, а также в направлении сокращения количества рекурсивных вызовов. Также, алгоритм обладает сравнительно высокой константой и на малых размерах массива оказывается медленнее, чем простые алгоритмы. Поэтому на этапе после разделения массива, каждый из подмассивов можно отсортировать одним из простых алгоритмов с меньшей константой.

Тем не менее, заложенный в алгоритме потенциал позволяет применять его как алгоритм общего назначения для широкого круга задач, и именно этот алгоритм включен в стандартную библиотеку языка C++ (функция `std::sort`) и многих других языков.

Сортировка слиянием представляет собой алгоритм, который показывает несколько худшее время, чем быстрая сортировка, но при этом стабилен и, независимо от характера входных данных, обладает вычислительной сложностью O(n log n), однако требует дополнительных затрат памяти размера O(n). Также сортировка слиянием является устойчивой, что важно в ряде задач. Еще одним плюсом такой сортировки является ее хорошая применимость при работе со списками.

Суть алгоритма описывается так: исходный массив разбивается пополам, его части дальше разбиваются пополам и так далее до тех пор, пока не останутся массивы единичной длины. Массив единичной длины является упорядоченным по определению. Затем массивы начинают объединяться обратно. Процесс объединения двух упорядоченных массивов значительно проще, чем объединение двух неупорядоченных с одновременной сортировкой. В результате количество элементарных действий, необходимое для «пересборки» исходного массива в правильном порядке, оказывается сравнительно небольшим.

Алгоритм сортировки слиянием записывается следующим образом:

1. В случае, если на сортировку передается массив единичной длины, следует выйти, так как такой массив уже отсортирован.
2. Исходный массив разбивается на два массива. В идеале массивы должны быть

одинакового размера, однако допускается их отличие на 1-2 элемента.

3. Два полученных массива отправляются на сортировку тем же методом слияния (путем рекурсивного вызова функции).
4. Создается дополнительный массив для объединения двух отсортированных массивов.

   Рекомендуется создавать такой вспомогательный массив до вызова функции сортировки, чтобы избежать накладных расходов в виде создания и удаления массива на каждом шаге сортировки.

5. Объявляются три индекса, указывающие на текущий обрабатываемый элемент в первом подмассиве, втором подмассиве и во вспомогательном массиве соответственно.
6. В цикле последовательно сравниваются элементы по индексам 1 и 2 и тот из них, который оказывается меньше, записывается во вспомогательный массив по индексу 3, после чего индекс 3 увеличивается на 1, а также увеличивается на 1 тот индекс, элемент по которому оказался меньше. Цикл продолжается до тех пор, пока не закончится один из подмассивов.
7. После окончания одного из подмассивов оставшиеся элементы второго подмассива последовательно записываются во вспомогательный массив.
8. В результате работы вспомогательный массив будет содержать отсортированные данные. Эти данные необходимо скопировать в исходный массив, либо вернуть как результат работы функции.

На рисунке 5 представлена схема объединения двух упорядоченных массивов в один. В левой части показаны два массива, каждый из которых отсортирован по возрастанию, а в правой — массив, в котором будут собираться элементы отсортированного массива. Элементы из массивов сравниваются, и в конечный массив копируется больший из двух, после чего текущий индекс обрабатываемого элемента увеличивается в том массиве, из которого был взят элемент. Стрелки показывают текущий индекс в каждом из массивов.

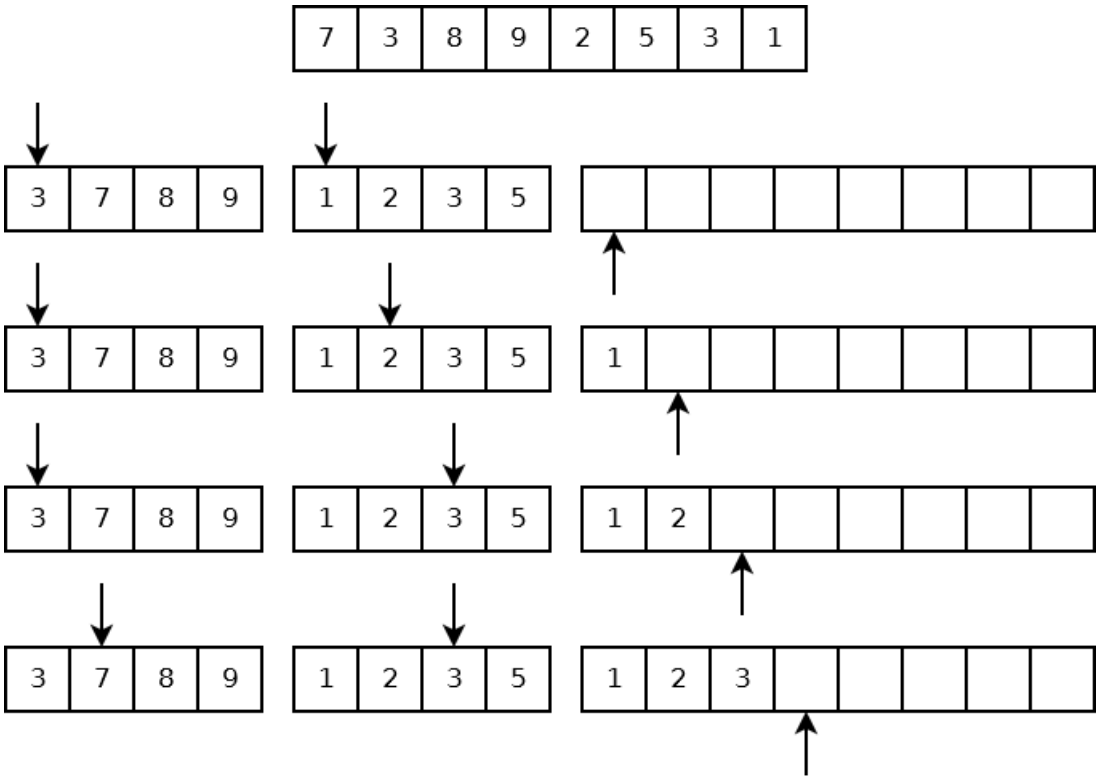

Рисунок 5 - Алгоритм сортировки слиянием

Несмотря на большое количество существующих на сегодня алгоритмов сортировки, ни один из них не является оптимальным по всем возможным критериям. На основе рассмотренных методов, можно резюмировать следующее:

- Для малых массивов и списков (порядка десятков элементов) хорошо подходит сортировка вставками, так как она обладает очень малыми накладными расходами (малой константой), а также является стабильной.
- Для больших массивов данных наибольшим быстродействием обладает быстрая сортировка.
- В случае работы со списками или файлами, а также в случаях, когда стабильность сортировки важнее быстродействия, рекомендуется применять сортировку слиянием.

Помимо приведенных алгоритмов сортировки, можно также встретить следующие:

- **Сортировка Шелла**. Представляет собой усовершенствованный вариант сортировки вставками, при котором сравниваются между собой не только элементы, стоящие рядом, но и отстоящие на определенном расстоянии. При правильном подборе интервалов для сравнения такая сортировка может обладать сложностью $O(n^{4/3})$ в худшем случае и $O(n^{7/6})$ в типовом, и при этом не деградирует так же легко, как быстрая сортировка.
- **Сортировка с помощью двоичного дерева**. Массив элементов перестраивается в двоичное дерево, представляющее собой двухмерный связанный список. Каждый элемент списка может иметь 1 «родителя» и до 2 «детей», причем все потомки «слева» от родительского узла меньше, чем родительский, а «справа» - больше. Пример двоичного дерева представлен на рисунке 6.

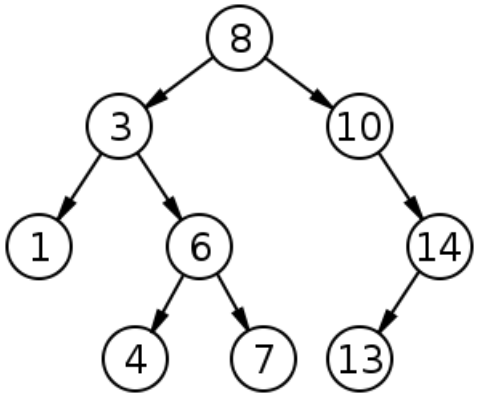

Рисунок 6 - Двоичное дерево

Сортировка осуществляется путем обхода дерева в необходимом порядке и сбора элементов в массив. Алгоритм обладает вычислительной сложностью порядка O(n log n), что относит его к быстрым алгоритмам сортировки, однако хранение дерева в памяти требует O(n) дополнительной памяти, причем для простых элементов вроде чисел перерасход памяти оказывается существенным.

- **Сортировка с помощью двоичной кучи** (также известна как **пирамидальная сортировка**) является модифицированной версией сортировки двоичным деревом. Двоичная куча — это такое двоичное дерево, для которого выполнены следующие условия:
  - Значение в любой вершине не меньше, чем значения ее потомков.
  - Максимальная глубина дерева отличается не более чем на 1.

• Последний слой заполняется слева направо.

Выполнение этих условий позволяет уменьшить расход памяти для хранения такого дерева, так как его можно уместить в обычный массив. При этом потомками элемента A[i] являются элементы A[2i+1] и A[2i+2] при нумерации элементов с нуля. При сортировке сначала формируется двоичная куча, затем корневой элемент кучи перемещается в отсортированные данные (в конец массива), а оставшиеся элементы перестраиваются, чтобы сформировать новую кучу. Постепенно куча сокращается до 1 элемента, и массив оказывается отсортирован.

Сортировка двоичной кучей обеспечивает постоянное время сортировки, равное O(n log n), а также не требует дополнительной памяти (расход памяти O(1)), однако сложнее в реализации и, как правило, в 1.5 раза медленнее быстрой сортировки. Модифицированная версия пирамидальной сортировки, которая приближается к сложности O(n) на почти отсортированных данных, называется **плавная сортировка**.

Порядок проведения работы

В данной работе необходимо написать несколько алгоритмов сортировки и проверить их эффективность. В качестве входных данных для алгоритмов нужно сформировать массив произвольных размеров, заполненный следующими данными:

- **Случайные данные.** Для формирования случайной величины в диапазоне от 0 до N в языке C необходимо написать следующий код:

```
#include <stdlib.h>
#include <time.h>
srand(time(0)); // Инициализация генератора, делается однократно
double r = (double)rand() / RAND_MAX * N
```

  В языке Python эта конструкция будет иметь следующий вид:

```
from random import random, seed
from time import time

seed(time()) # Инициализация таймера, делается однократно
r = random() * N
```

- **Почти отсортированные данные**. Такие данные формируются аналогично случайным, однако диапазон значений случайных чисел меньше, а математическое ожидание линейно зависит от номера элемента.
- Данные, отсортированные **в обратном порядке**.

Для оценки эффективности работы алгоритма следует измерить время его выполнения для двух массивов разных размеров. Для измерения времени выполнения в языке C существует ряд библиотечных функций и средств операционной системы. При использовании компилятора GCC или MinGW доступна следующая конструкция:

```
#include <sys/time.h>

struct timeval tv1, tv2;
```

```
gettimeofday(&tv1, 0);
// Операция, время выполнения которой необходимо измерить
gettimeofday(&tv2, 0);
double elapsed = 1.0e3*(tv2.tv_sec - tv1.tv_sec)
               + 1.0e-3*(tv2.tv_usec - tv1.tv_usec);
```

Переменная `elapsed` будет хранить количество миллисекунд, которые заняла операция. При использовании другого компилятора в среде MS Windows доступен следующий функционал:

```
#include <windows.h>
union
{
    FILETIME ft;
    int64_t tt;
}ft1, ft2;
GetSystemTimeAsFileTime (&ft1.ft);
// Операция, время выполнения которой необходимо измерить
GetSystemTimeAsFileTime (&ft2.ft);
double elapsed = (ft2.tt − ft1.tt) / 1.0e4
```

Аналогичным образом переменная `elapsed` хранит время, затраченное на выполнение операции, в секундах.

В языке C++, начиная со стандарта C++11, доступен заголовочный файл `chrono`, позволяющий рассчитать время выполнения следующим образом:

```
auto t1 = std::chrono::steady_clock::now();
// Операция, время выполнения которой необходимо измерить
auto t2 = std::chrono::steady_clock::now();
auto elapsed =
    std::chrono::duration_cast<std::chrono::milliseconds>(t2-t1);
```

Переменная `elapsed` будет хранить время, затраченное на выполнение операции, в миллисекундах (при использовании `chrono::milliseconds`) или в микросекундах (`chrono::microseconds`).

В языке Python существует функция `time`, находящаяся в одноименном модуле, которая возвращает число секунд с 1 января 1970 года. Используя эту функцию, можно также измерить время выполнения операции:

```
from time import time


t1 = time()
# Операция, время выполнения которой необходимо измерить
t2 = time()
elapsed = 1.0e3*(t2-t2)
```

Переменная `elapsed` точно так же хранит количество миллисекунд, затраченное на выполнение операции.

Для выполнения работы необходимо выполнить следующее:

1. Подготовить основу приложения

   1.1. Создать новый проект консольного приложения.

   1.2. Определить функцию `isSorted`, проверяющую массив на упорядоченность. Аргументы функции — массив, который необходимо проверить, и, при необходимости, его размер. Функция должна возвращать `true`, если каждый следующий элемент массива не меньше предыдущего и `false`, если любые 2 элемента массива нарушают это соотношение.

   1.3. Определить функцию, `printArray`, выводящую массив на экран. Необходимо ограничить количество выводимых элементов массива, а также вывести на экран, отсортирован ли переданный массив.

   1.4. Создать три функции для заполнения массива: `create` для заполнения массива случайными числами, `createNice` для заполнения массива значениями, близкими к отсортированным, и `createReverse` для заполнения массива числами, отсортированными в обратном порядке. В языке C рекомендуется передавать в функцию в качестве аргумента указатель на массив и его размер, а в Python рекомендуется передавать в функцию только размер массива и возвращать созданный заново массив, заполненный значениями.

1.5. Протестировать созданные функции. Для этого необходимо создать несколько массивов и вывести их на экран.

2. Написать и протестировать один из простых алгоритмов сортировки: сортировка выбором, вставками или пузырьковая сортировка.

   2.1. Создать массив данных произвольного размера, заполненный случайными числами, и вывести его на экран.

   2.2. Создать и вызвать функцию, реализующую выбранный алгоритм сортировки. Аргументами функции должны являться массив и, при необходимости, длина этого массива. Функция не должна иметь возвращаемого значения и должна осуществлять преобразование переданного массива.

   2.3. Вывести на экран отсортированный массив данных и убедиться в правильной работе алгоритма.

   2.4. Измерить время выполнения сортировки и также вывести его на экран. Для того, чтобы измерение имело смысл, необходимо увеличивать размер массива до тех пор, пока время сортировки не превысит 150-200 мс.

   2.5. Определить, как изменяется время сортировки при изменении характера исходных данных (случайные, близкие к отсортированным, отсортированные в обратном порядке), а также как изменяется время сортировки при изменении количества элементов массива.

3. ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ. Реализовать один из оптимальных алгоритмов сортировки: быстрая сортировка, сортировка слиянием или другой известный вам алгоритм.

   3.1. Ввести новую функцию, осуществляющую сортировку, и вставить ее вызов вместо вызова функции сортировки, использованной ранее. Убедиться, что алгоритм сортировки работает корректно.

   3.2. Измерить время сортировки и сравнить его с временем сортировки такого же массива использованным ранее способом.

   3.3. Определить, как изменяется время сортировки при изменении характера исходных данных (случайные, близкие к отсортированным, отсортированные

в обратном порядке), а также как изменяется время сортировки при изменении количества элементов массива.

Вопросы для самоконтроля и подготовке к защите работы №5

1. Какие задачи решает сортировка?
2. Каким образом оценивается эффективность алгоритма сортировки?
3. Какие алгоритмы сортировки вы знаете? В чем их отличие?
4. Существует ли алгоритм сортировки, оптимальный для всех случаев?
5. Существуют ли ситуации, в которых неоптимальный алгоритм может оказаться быстрее оптимального?