ЛЕКЦИЯ 11

МЕТОДЫ СОРТИРОВКИ

Настало время отойти от практической направленности лекций и немного заняться теорией. Эта теория в основном будет излагаться доступно, без использования сложных концепций.

Часто данные приходится упорядочивать, прежде чем обрабатывать их. Если набор данных неупорядочен, то чтобы выяснить, есть ли среди них определённый элемент, нужно перебрать все данные. Если данные упорядочены, то можно воспользоваться методом дихотомического поиска, или метода деления пополам, который работает гораздо быстрее, чем полный перебор. Для того, чтобы данные были упорядочены, можно заполнять специальные структуры данных, которые автоматически упорядочивают попадающую в них информацию, а можно и воспользоваться алгоритмами сортировки, чтобы перегруппировать уже записанные в массив неупорядоченные данные.

Алгоритмы сортировки отличаются друг от друга по многим показателям. Одни алгоритмы пользуются дополнительной памятью, а другие — нет. Когда носителями больших объёмов данных были магнитные ленты, было принципиально важно обойтись без дополнительной памяти. В случае жёсткого диска безразлично, далеко или близко находятся обрабатываемые данные, но в случае магнитной ленты это не так. Чтобы сравнить данные, далеко отстоящие друг от друга на ленте, придётся перематывать ленту, а это занимает намного больше времени, чем операции записи и чтения. Поэтому на то, требует определённый алгоритм, чтобы данные лежали близко, или нет, тоже нужно обращать внимание.

Применительно к алгоритмам сортировки введём два понятия: C — **количество сравнений**, M — **количество перестановок**. Эти понятия разделяются, так как иногда операция чтения производится значительно быстрее, чем операция записи. Выбор алгоритма сортировки иногда зависит от специфических требований по объёмам данных или по носителям.

В каждом языке программирования есть реализованная стандартная функция сортировки, которая обычно работает быстро. В языке C это функция `qsort`. С прикладной точки зрения студенту достаточно только знать, как ей пользоваться, а отличать сортировку пузырьком от сортировки Шелла он уметь не обязан. Однако разные алгоритмы стоит пройти, так как некоторым студентам это может быть как полезно, так и инте-

ресно.

Методы сортировки без привлечения дополнительной памяти — это **сортировка включением**, или **сортировка вставками**, **сортировка прямым выбором** и **сортировка пузырьком**. Эти алгоритмы имеют самую плохую временну́ю сложность; их обычно проходят в школе, и они обычно запоминаются лучше всего.

Сортировка включением происходит следующим образом. В начале массив данных пуст, затем начинается его заполнение. Каждый новый элемент входных данных ставится в массив так, чтобы получившийся массив оказался отсортирован. Когда все входные данные окажутся вставлены в массив, тот будет упорядочен.

Можно привести аналогию с колодой карт. Карты по очереди берутся в руку, причём каждая карта ставится на своё место по величине. Когда вся колода окажется в руке, она будет упорядочена, это гарантирует сам метод расположения карт.

Наилучший случай по перестановкам, когда мало количество сдвигов правой части массива (считая от вставляемого элемента), — это уже отсортированный массив входных данных. Тогда каждый новый элемент становится в конец массива, существующего на момент его добавления, и никаких сдвигов не происходит. Этот же случай является наилучшим по сравнениям, так как достаточно одного сравнения, чтобы понять, что новый элемент нужно добавить в конец массива. Худший случай и по сравнениям, и по перестановкам — когда входные данные отсортированы в обратном порядке. Тогда каждый новый элемент нужно сравнивать с каждым уже имеющимся элементом, ставить его в начало массива и сдвигать весь массив вправо.

В среднем, если частично заполненный массив имеет k элементов, на вставку очередного элемента потребуется $\dfrac{k+1}{2}$ сравнений. Среднее количество присвоений на каждом шаге равно $\dfrac{k+1}{2}+2$. Пусть размер конечного массива — n. Тогда минимальные, максимальные и средние временны́е сложности сортировки включением можно представить в таблице 11.1:

$$\begin{array}{ll} \min C = O(n) & \min M = O(1) \\ \max C = O(n^2) & \max M = O(n^2) \\ \operatorname{avr} C = O(n^2) & \operatorname{avr} M = O(n^2) \end{array}$$

Таблица 11.1: Сортировка вставками

Таким образом, временна́я сложность алгоритма почти всегда равна $O(n^2)$. Может повезти, и входные данные будут уже отсортированы, но в общем случае сортировка включением имеет квадратичную от размера входных данных сложность. Это нехорошо, так как количество операций быстро растёт при увеличении n.

Произведём модификацию этого алгоритма. Сразу заметим, что модификацию нужно производить, когда сложность удовлетворительна, нужно только улучшить скорость алгоритма не более, чем в разы. Но если требуется провести сортировку за час, а данному алгоритму нужны для работы трое суток, то модификация бесполезна — нужно менять сам алгоритм.

Поскольку на каждом шаге частичный массив отсортирован, то вовсе не обязательно сравнивать новый элемент со всеми элементами подряд. Методом деления пополам можно значительно уменьшить количество сравнений. Сложность по сравнениям теперь

будет не $O(n^2)$, а $O(n \log n)$. Количество перестановок при этом не затрагивается, так что результирующая сложность алгоритма по-прежнему равна $O(n^2)$.

1. Сортировка прямым выбором

Сортировка прямым выбором, или сортировка выборкой, работает следующим образом. В неупорядоченном массиве входных данных размера n производится цикл по всем элементам. На итерации i выбираем самый маленький элемент из части массива $[i, n]$ и меняем его местами с элементом с номером i. По мере функционирования алгоритма в левой части массива будут накапливаться уже отсортированные элементы.

Для того чтобы найти наименьший элемент из n, требуется $O(n)$ операций сравнения. Количество перестановок всего алгоритма будет равняться $O(n)$. Так как во внешнем цикле алгоритма n итераций, то суммарная сложность алгоритма — $O(n^2)$, как и у алгоритма сортировки включением. Однако времена работы алгоритма будут немного лучше за счёт коэффициента у n^2. В настоящее время алгоритм сортировки выбором используется исключительно в учебных целях.

Лучший случай — когда массив входных данных полностью отсортирован. Тогда количество перестановок равно нулю. Однако количество сравнений при этом не изменится — $O(n^2)$. Худший случай — когда приходится делать перестановку на каждой итерации алгоритма, например, когда весь массив отсортирован, кроме того, что наибольший элемент стоит на первой позиции (рис. 11.1). Тогда $M = O(n)$, а количество сравнений — то же самое.

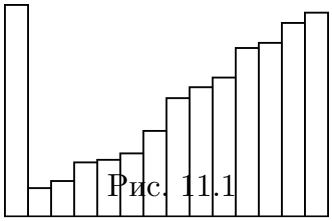

Рис. 11.1

2. Сортировка пузырьком

При сортировке пузырьком сравниваются пары рядом стоящих элементов. Если они стоят в правильном порядке, то ничего не происходит, а если левый больше, чем правый, то они меняются местами. Затем сравнивается пара элементов, смещённая на единицу относительно предыдущей. Когда один проход по массиву заканчивается, начинается следующий, в котором происходит то же самое. Такие проходы производятся до тех пор, пока весь массив не будет отсортирован. Критерием этого является условие, что на очередном проходе не было произведено ни одной перестановки.

Название алгоритма связано с тем, что большой элемент в начале массива в процессе одного прохода как бы «всплывает» вверх (то есть вправо), как пузырь. Маленькое число, наоборот, может сдвинуться влево только на одну позицию за проход. По мере работы алгоритма проходы занимают всё меньше времени, потому что правая часть массива оказывается отсортирована. Этот алгоритм прекрасно подходит для сортировки на магнитных лентах, так как каждый раз сравнивается пара рядом стоящих элементов. Правда, за каждым проходом следует холостая перемотка на начало массива. Так как большие числа «всплывают» быстро, а маленькие «тонут» медленно, то напрашивается модификация алгоритма сортировки пузырьком. Нужно чередовать проходы вправо и влево, тогда эти два процесса происходят одинаково быстро. Такая модификация называется **шейкерной сортировкой**.

Лучший случай — когда массив входных данных изначально отсортирован, тогда производится только один проход, за который программа убеждается, что в массиве ничего менять не нужно. Худший случай — когда массив отсортирован в обратном порядке, тогда за один проход большое число с первой позиции поднимается до своего места в конечном массиве, а количество перестановок максимально.

Временна́я сложность алгоритма сортировки пузырьком даже в среднем случае равна $O(n^2)$. Её модификация, шейкерная сортировка, имеет такую же сложность.

3. Сортировка Шелла

Выбираем из массива подпоследовательность чисел, расположенных друг от друга на расстоянии k. Эта подпоследовательность сортируется тем или иным методом. Затем сортируется подпоследовательность, получающаяся из предыдущей сдвигом вправо на одну позицию. После того, как все возможные подпоследовательности с шагом k отсортированы, выбирается другое значение шага, и процедура повторяется. Каким образом выбирать шаг k — сложный вопрос, на эту тему пишутся научные статьи. В отличие от предыдущих алгоритмов, где всё очень наглядно, для обоснования сложности этого алгоритма нужна математика.

Количество перестановок M у алгоритма Шелла оценивается как $O\left(n^{\frac{4}{3}}\right)$. На больших наборах данных это намного лучше, чем $O(n^2)$.

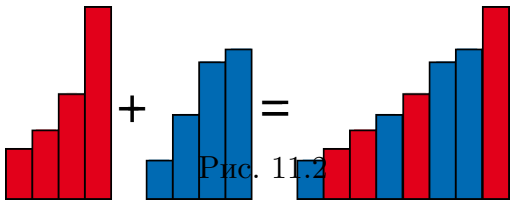

Рис. 11.2

4. Сортировка слиянием

Если имеются две отсортированные подпоследовательности, то чтобы составить из них отсортированную последовательность, нужно попарно сравнивать очередные элементы из них, и перемещать в результирующую последовательность меньший из них. На этом принципе основана сортировка слиянием `mergesort`. Из той подпоследовательности, из которой только что был перемещён элемент, на следующей итерации берётся следующий по величине элемент, а из другой подпоследовательности — тот же самый, что и на прошлой итерации. Если в одной из подпоследовательностей закончились элементы, то остаток другой просто пристыковывается справа к результирующей последовательности. Если в обеих подпоследовательностях содержится по n элементов, то сложность такого слияния этих подпоследовательностей — $O(n)$.

На глобальном уровне принцип данного алгоритма таков. Разбиваем весь массив на две части. После того, как обе части отсортированы, применим вышеописанную процедуру для их слияния в отсортированный массив. Чтобы отсортировать одну такую часть, разобьём и её пополам. После того, как обе четверти отсортированы, применим вышеописанную процедуру слияния, чтобы получить отсортированную половину. Продолжаем дробить части массива до тех пор, пока одна подпоследовательность не достигнет размера в 1 или 2 элемента — такую группу элементов легко отсортировать. Рекурсивно применяя процедуру разбиения пополам и слияния подпоследовательности, можно отсортировать и весь массив.

Сложность алгоритма сортировки слиянием — $O(n \log n)$.

5. Стабильность алгоритма сортировки

Пусть дана таблица студентов, в которой содержатся такие поля, как «фамилия» и «средний балл». Изначально строки отсортированы по фамилии. Требуется отсортировать их по среднему баллу, при этом строки с одинаковыми средними баллами должны оставаться отсортированными по фамилии. Сортировка, которая не меняет местами элементы с одинаковыми значениями, называется **стабильной**. Чтобы решить вышеописанную задачу, нужна стабильная сортировка.

Сортировка выборкой, например, стабильна, потому что если имеются несколько элементов с одинаковыми значениями, то этот алгоритм сначала выберет первый из них, затем второй, и т. д., так что их порядок в массиве сохранится. Однако если изменить алгоритм так, чтобы он выбирал последний из равных элементов, а не первый, то свойство стабильности пропадает. Это можно сделать, изменив в условии выбора текущего элемента строгое неравенство на нестрогое:

```
if(x <= min)
        min=x;
```

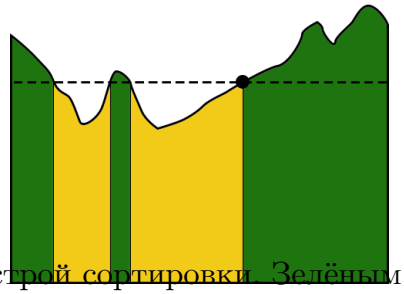

Рис. 11.3: Одна итерация быстрой сортировки. Зелёным цветом обозначены элементы одной части, жёлтым — другой части.

6. Быстрая сортировка

В неотсортированном массиве выбирается какой-то элемент. Проводится цикл по массиву, в котором тот разделяется на две части: в одной из них лежат элементы, меньшие, чем выбранный, а в другом — большие. Группы элементов ставятся одна за другой, между ними вставляется выбранный элемент. Для каждой части эта процедура рекурсивно повторяется, пока весь массив не окажется отсортированным.

Выбор разделяющего элемента является важным этапом алгоритма. Очевидно, что наименьший или наибольший элемент — это плохой выбор, так как все элементы массива окажутся в одной части. Хорошим выбором был бы выбор среднего по значению, но на каждой итерации для его вычисления нужно потратить $O(n)$ операций. Часто делают проще: берут три произвольных элемента в группе и выбирают средний из них.

Сложность этого алгоритма составляет $O(n \log n)$. В общем случае быстрая сортировка является нестабильной: может сложиться ситуация, когда порядок одинаковых по значению элементов нарушится. Например, пусть имеется три одинаковых элемента, а на очередной итерации быстрой сортировки выбран второй из них. Тогда первый и третий элементы окажутся по одну сторону от второго элемента.

7. Сортировка деревом

Для этой сортировки даже не обязательно выучить структуры данных типа «дерево»: дерево можно организовать на массиве.[1] Это дерево — просто удобное представление массива, а не реальная структура данных в памяти компьютера. Такое дерево будет плоским и ветвистым, что, несомненно, эффективно для хранения данных. Будем заполнять дерево следующим образом. Берём элементы входных данных тройками, и максимальный из них ставим над двумя другими. Вставляем эту тройку в массив. Если в воображаемом дереве при этом нарушается порядок следования чисел, то нужно менять местами соответствующие элементы, пока порядок не будет восстановлен. Легче всего это показать на анимированном примере[2].

[1] Здесь должен быть рисунок с пояснениями, как же можно организовать дерево на массиве, но такового не видно на экране.

[2] , которого всё равно не видно на записи

8. Сортировка подсчётом

Сортировка подсчётом также называется `countsort`. Для каждого элемента вводится число, которое хранит количество раз, когда этот элемент встретился в массиве. Пройдёмся в цикле по массиву и заполним все такие числа. После этого наберём новый массив, заполненный соответствующими количествами элементов. Такая процедура эквивалентна сортировке исходного массива.

Пусть исходный массив состоит из чисел от 0 до 19. Для того, чтобы его отсортировать, заведём ещё один массив из 20 целых чисел. Заполним его так, как только что было описано. После этого запишем в новый массив столько чисел от 0 до 19, сколько их было подсчитано при заполнении массива из 20 элементов. Например, если нулей встретилось 4, то в позиции 0, 1, 2 и 3 нового массива нужно поставить нули. Если число единиц равно 3, то следующие три позиции будут заняты единицами, и т. д.

Сортировка подсчётом эффективна тогда, когда элементы массива входных данных имеют ограниченный диапазон и повторяются. Если же элементы могут принимать широкий диапазон значений, или количество повторов чисел мало, то эта сортировка невыгодна. Стабильность алгоритма может быть достигнута, если использовать дополнительную память, в противном случае алгоритм нестабилен. Временна́я сложность алгоритма — $O(n)$.

9. Поразрядная сортировка

Поразрядная сортировка (`radixsort`) имеет два варианта:

1. если сначала сортировка производится по младшим разрядам, затем по старшим, то это LSD (least significant digit);

2. если сначала сортировка производится по старшим разрядам, а затем по младшим, то это MSD (most significant digit).

Эта сортировка применима в случае, когда, например, нужно отсортировать миллион трёхзначных чисел. В случае LSD применим сортировку подсчётом сначала по единицам, потом — по десяткам, потом — по сотням. При этом нужен вспомогательный массив размером в 10 элементов. После того, как произведена сортировка по единицам, взаимное расположение чисел по единицам в дальнейшем сохраняется, то же самое и для десятков.

Можно взять не десятичную разрядность, а, например, шестнадцатеричную. Тогда вспомогательный массив вводится не на 10, а на 16 элементов.

Алгоритм поразрядной сортировки, как и сортировка подсчётом, работает за линейное время.

10. Сортировка корзинками

Сортировка корзинками также называется `basketsort`. Этот алгоритм удобно проиллюстрировать на примере стирки в семье с 12 детьми. Если постирать чьи-то джинсы с чьим-то белым платьем, то после этого начнутся внутрисемейные конфликты. Нужно

распределить всю одежду для стирки на несколько корзинок. В одну корзину пойдут грязные немаркие вещи, в другую — белую одежду.

Пусть все элементы (для определённости, числа) могут принимать малый диапазон значений. Разобьём числа на «корзинки»: от 0 до 2, от 3 до 5, от 6 до 8. Для каждой корзинки заводим свой упорядоченный список чисел. Каждый элемент из входного массива попадает в одну корзинку и занимает своё место в соответствующем списке. После этого списки всех корзинок склеиваются вместе, и получается отсортированный массив.

Этот алгоритм лучше всего работает, когда в каждую корзинку попадает примерно одинаковое количество элементов. Если же входные данные разбросаны неравномерно, например, большое количество данных в одной-двух корзинках, то сортировка работает медленно.

		Time Complexity					
		Best	Worst	Avg.	Space	Stable	Comments
Comparison Sort	Bubble Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Yes	*For each pair of indices, swap the elements if they are out of order*
	Modified Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes	*At each Pass check if the Array is already sorted. Best Case-Array Already sorted*
	Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Yes	*Swap happens only when once in a Single pass*
	Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes	*Very small constant factor even if the complexity is O(n^2).* ***Best Case:*** *Array already sorted* ***Worst Case:*** *sorted in reverse order*
	Quick Sort	O(n.lg(n))	O(n^2)	O(n.lg(n))	O(1)	Yes	*Best Case: when pivot divide in 2 equal halves Worst Case: Array already sorted - 1/n-1 partition*
	Randomized Quick Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(1)	Yes	*Pivot chosen randomly*
	Merge Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(n)	Yes	*Best to sort linked-list (constant extra space). Best for very large number of elements which cannot fit in memory (External sorting)*
	Heap Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(1)	No	
Non-Comparison Sort	Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(n+2^k)	Yes	*k = Range of Numbers in the list*
	Radix Sort	O(n.k/s)	O(2^s.n.k/s)	O(n.k/s)	O(n)	No	
	Bucket Sort	O(n.k)	O(n^2.k)	O(n.k)	O(n.k)	Yes	

Рис. 11.4

Сложности алгоритмов удобно изобразить в виде таблицы (см. рис. 11.4). Заметим, что в худшем случае быстрая сортировка `quicksort` даёт сложность $O(n^2)$.

Если производительность библиотечного алгоритма сортировки имеет значение, то нужно ознакомиться с тем, как он устроен, прежде чем использовать в программе. За стандартной функцией `qsort` в некоторых языках скрывается не быстрая сортировка, а другие алгоритмы. При выборе алгоритма сортировки нужно учитывать специфику конкретной задачи, понимать, какие данные нужно сортировать, и насколько важны скорость алгоритма и объём дополнительной памяти, которые он требует.