Лекция 11. Сортировки

Временная сложность алгоритмов

- Для решения одной и той же задачи можно использовать разные алгоритмы
- Разные алгоритмы отличаются друг от друга по производительности и объему требуемой памяти
- Важной характеристикой алгоритма является **временная сложность**

Временная сложность алгоритмов

- **Временная сложность** – это количество **элементарных операций**, требуемых в ходе выполнения алгоритма
- Элементарными операциями считаются сравнения, присваивания и арифметические операции

Пример – линейный поиск

- Рассмотрим обычный линейный поиск в массиве – мы проходимся по всем элементам от нулевого до последнего и сравниваем с искомым значением
- Пусть длина массива N
- Тогда в худшем случае нам понадобится N сравнений, т.е. временная сложность линейного поиска равна N

Обозначение временной сложности

- Обычно временную сложность смотрят только до порядка
- Пусть, например, она равна $3N^2 + N$
- Тогда откидывают все коэффициенты и оставляют только главный член, который вносит самый большой вклад
- В данном случае получится N^2
- Записывается это так: O(N^2)

Худший случай

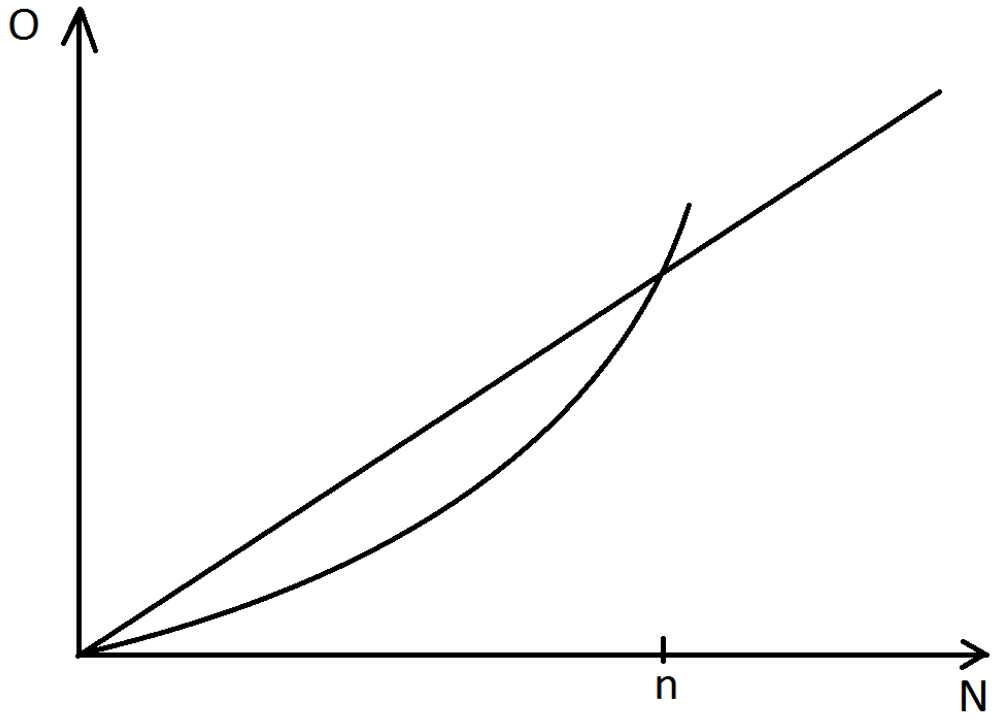
- Временную сложность оценивают для худшего случая – когда алгоритму требуется больше всего операций
- Например, в линейном поиске нужный элемент может оказаться первым, и тогда понадобится всего 1 итерация
- Но в худшем случае придется просмотреть весь массив, поэтому сложность будет O(N)

Пример – бинарный поиск

- Сложность бинарного поиска составляет $O(\log_2 N)$
- Логарифм растет гораздо медленнее линейной функции N
- Например, $\log_2 1024 = 10$
- Получается, бинарный поиск намного эффективнее линейного поиска
- Поэтому важно стараться применять алгоритмы, имеющую меньшую временную сложность

Поведение на малых данных

- Общее правило – выбирать алгоритм с меньшей сложностью, он лучше работает при больших N
- Но на малых данных алгоритм с большей сложностью может быть эффективнее



- На данных размера меньше n этот алгоритм с O(N^2) быстрее, чем с O(N)

Сортировка

- **Сортировка** – это упорядочивание элементов массива в определенном порядке (например, в порядке неубывания)
- Будем рассматривать на примере массивов целых чисел, но алгоритмы верны для массивов любых типов

Простые сортировки

- Алгоритмов сортировки существует просто огромное количество
- **Простыми сортировками** называют несложные алгоритмы сортировки, которые имеют временную сложность $O(N^2)$
- В дальнейшем считаем что хотим упорядочить массив по неубыванию
- Простые сортировки не требуют дополнительной памяти, а переупорядочивают исходный массив

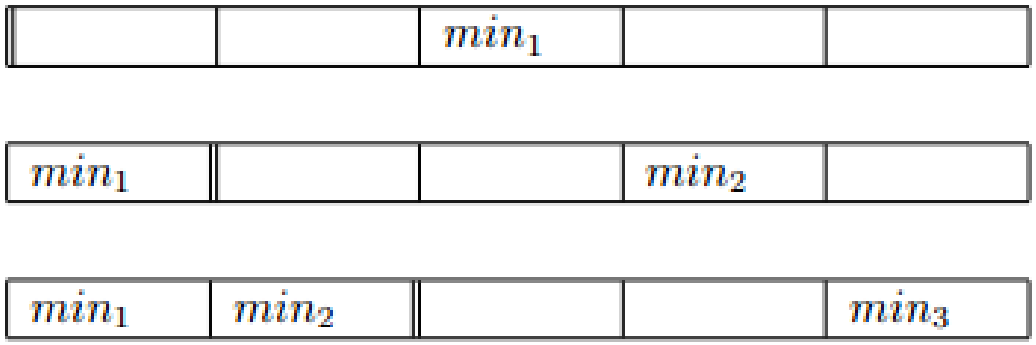
Обмен двух переменных

- Чтобы обменять значения двух переменных, нужно ввести третью вспомогательную переменную
- Например, надо обменять переменные x и y
- Вводим дополнительную переменную temp
- ```
  int temp = x;
  x = y;
  y = temp;
  ```
- Аналогично можно обменивать элементы массива, вместо x и y будут некоторые a[i] и a[j]
  ```

Сортировка выбором

- Шаг 1: линейно ищем минимальный элемент в массиве и обмениваем его с первым элементом
- Шаг 2: линейно ищем минимальный элемент в массиве, начиная со второго элемента, обмениваем его со вторым элементом
- …
- Шаг N-1: ставим минимальный элемент из последних двух на N-1 место

Сортировка выбором



- То есть многократно делаем следующие операции:
  - Ищем индекс минимального элемента в неотсортированной части массива
  - Обмениваем этот элемент с первым элементом неотсортированной части массива

Сортировка выбором

- Временная сложность:
  (N - 1) + (N - 2) + … + 1 $\sim$ $O(N^2)$

Задача

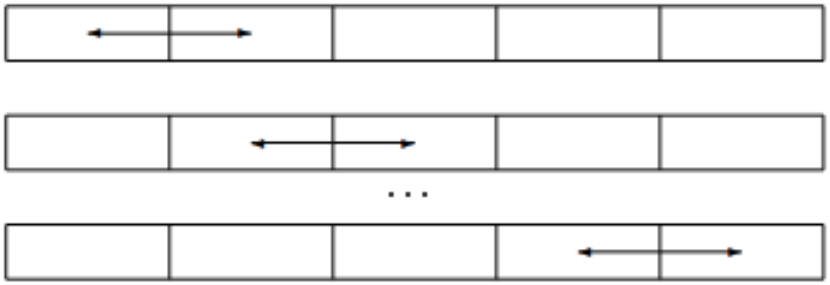
- Реализовать функцию поиска минимума в массиве
- Переделать на функцию, которая ищет индекс, по которому лежит минимум в массиве
- Переделать, чтобы функция поиска индекса минимума работала не по всему массиву, а только в части массива, начинающейся с индекса start

Задача на курс «Сортировка выбором»

- Реализовать сортировку выбором

Сортировка пузырьком

- Выполняем проход по массиву слева направо, сравнивая и при необходимости меняя местами соседние элементы



- После этого максимальный элемент окажется последним
- Повторяем процесс N-1 раз (или меньше, если за некоторую итерацию не произошло ни одного обмена)

Сортировка пузырьком

- В сортировке пузырьком отсортированная часть формируется справа
- Для сортировки пузырьком есть оптимизация – если за полный проход по массиву не было ни одного обмена, то массив уже отсортирован, и алгоритм нужно завершить

Задача на курс «Сортировка пузырьком»

- Реализовать сортировку пузырьком

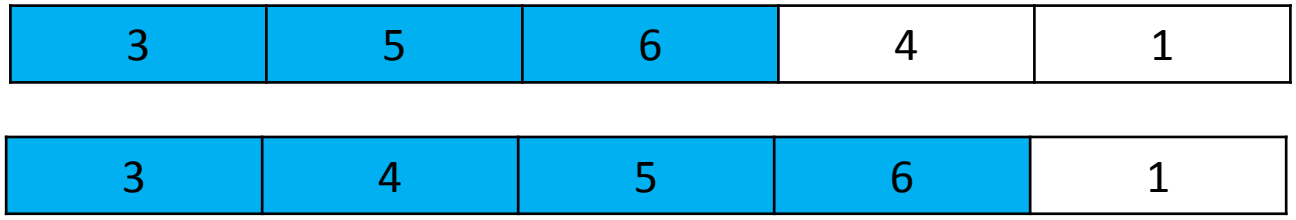
Сортировка вставками

- Так же выполняем N – 1 итераций
- В левой части массива будем выстраивать отсортированную последовательность, на каждой итерации туда будет добавляться один элемент
- Перед первой итерацией считаем, что отсортированная последовательность состоит из первого элемента
- Далее, выполняем для каждого элемента от 2 до N - 1

5	2	1	3	6

Идея итерации алгоритма

- На итерации уже есть какая-то отсортированная часть массива
- И есть первый элемент неотсортированной части
- Надо найти индекс, куда надо вставить этот элемент
- А всё, что правее в отсортированной части – сдвинуть на 1 индекс вправо



- Тут 4 надо вставить по индексу 1, а числа 5 и 6 надо сдвинуть вправо на 1 индекс

Итерация сортировки вставками

- Пусть i – индекс первого элемента неотсортированной части
- Запоминаем в переменную temp элемент array[i]
- Идем справа налево по отсортированной части при помощи счетчика j, сначала он равен i – 1
  - Если j < 0 или temp >= array[j], то заканчиваем идти. Вставляем temp по индексу j + 1. На этом итерация завершена
  - Иначе сдвигаем array[j] вправо: array[j + 1] = array[j]

Задача на курс «Сортировка вставками»

- Реализовать сортировку вставками

Быстрая сортировка

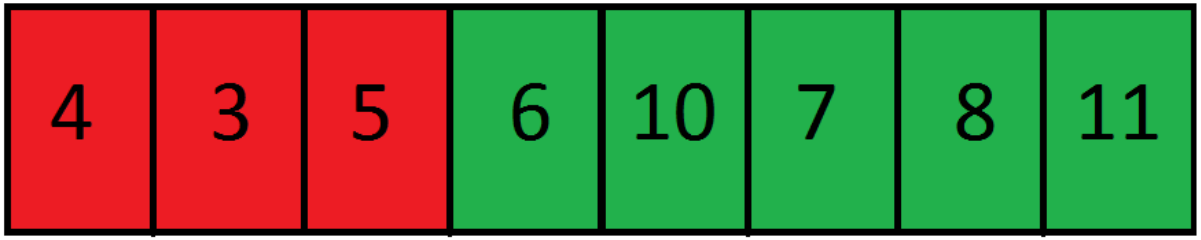
- **Быстрая сортировка** – это уже более сложный алгоритм
- Его спрашивают практически на всех собеседованиях
- Его временная сложность в худшем случае $O(N^2)$, но в среднем $O(\mathrm{N} * \log_2 N)$, что является лучше, чем простые сортировки

Быстрая сортировка

- Быстрая сортировка реализуется с помощью рекурсивной функции:
- static void quickSort(int[] a, int left, int right)
- left и right обозначим индексы границ массива a

Быстрая сортировка

- Выберем некоторое произвольное число x в диапазоне от минимума до максимума по массиву, например, первый (или средний) элемент
- Хотим сделать следующее: чтобы все элементы до некоторого индекса были меньше, либо равны x, а остальные – больше x



x = 5

- После этого рекурсивно вызываем этот же алгоритм для левой части массива и для правой. Но это если эта часть массива содержит как минимум два элемента

Быстрая сортировка

- Как нужным образом поделить массив на две части?

1. Запускаем два счетчика: i слева направо от left до right;
   j – справа налево от right до left
2. Сначала двигаем i, пока не встретим элемент, который >= x.
   После этого начинаем двигать j, пока не встретим элемент,
   который <= x
3. Если i <= j, то делаем обмен элементов по этим индексам,
   затем сдвигаем оба счетчика еще на один элемент
   и на шаг 2. Иначе – завершаем процесс и на шаг 4
4. В этот момент все элементы, которые <= x, находятся левее
   i, а которые >= x – правее j
5. Если i < right, то вызываем рекурсивно для части от i до
   right. Если j > left, то и для части от left до j

Быстрая сортировка

- Для остановки рекурсии надо рассмотреть два выделенных случая:
  - Передали массив длины 1 – можно считать что он уже отсортирован, ничего делать не нужно
  - Передали массив длины 2 – если нужно, меняем эти два элемента местами

Опорный элемент

- Число x называют **опорным элементом**
- Выбирать его можно любым образом из диапазона [min, max], где min и max – минимум и максимум из значений в массиве
- В идеале, опорный элемент должен делить массив на две равные части, тогда скорость работы алгоритма максимальна
- Но чтобы выбрать элемент таким образом, нужно тоже затратить время, что в итоге не окупается, поэтому в качестве опорного элемента часто берут первый или среднее арифметическое первого и последнего элементов

Задача на курс «Быстрая сортировка»

- Реализовать быструю сортировку