ТЕОРИЯ И ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Лекция 3. Алгоритмы сортировки

Глухих Михаил Игоревич, к.т.н., доц.

mailto: glukhikh@mail.ru

Основные характеристики алгоритмов сортировки

- Трудоёмкость T=O(…) – в среднем и худшем случае
- Ресурсоёмкость R=O(…)
  - Для R=O(1) говорят «сортировка на месте»
- Устойчивость – изменяется ли порядок равных элементов в списке
  - Бывает важна в некоторых случаях, например, при индексации баз данных, или при сортировке по нескольким ключам
  - Точно неважна, если элемент – это только ключ сравнения
  - Какие из известных вам алгоритмов сортировки устойчивы?
  - NB: строго говоря, любой алгоритм сортировки можно реализовать так, что он станет неустойчивым
  - Но не у любого алгоритма существует устойчивая реализация
- Операции сравнения – используются или нет

Устойчивые сортировки

- Пузырьком $T=O(N^2)$, $R=O(1)$
- Вставками $T=O(N^2)$, $R=O(1)$
- Слиянием $T=O(N \log N)$, $R=O(N)$
- Описание алгоритмов см. лекцию 20 из курса по C++ (для удобства также выложена на страницу данного курса как лекция 3A)

Неустойчивые сортировки

- Выбором $T=O(N^2)$, $R=O(1)$
  - Этот алгоритм уже на первом шаге меняет местами минимальный элемент в списке с первым, что и приводит к неустойчивости
  - Пример: (2A, 2B, 1) → (1, 2B, 2A) (считаем 2A == 2B)
- Пирамидальная сортировка (сортировка двоичной кучей, Heap Sort) $T=O(N \log N)$, $R=O(1)$
  - Неустойчива примерно по тем же причинам – вершина в процессе сортировки «уезжает» в некоторое место кучи

Пирамидальная сортировка -- напоминание

- Основывается на структуре под названием «бинарная пирамида» или «бинарная куча» (binary heap)
- Это бинарное дерево (НЕ бинарное дерево сортировки), у которого значение ключа предка всегда больше либо равно значению ключа потомка
- Бинарное дерево на основе массива: если предок имеет #J, то потомки имеют #2J+1 (левый) и #2J+2 (правый)
- Для подготовки такой структуры выполняется процедура просеивания (heapify), меняющая местами значения предка и большего из его потомков в случае необходимости, и затем повторяющая такую операцию на уровень ниже

Просеивание

- MAX-HEAPIFY(A, J, S)
  - L = 2 * J + 1
  - R = 2 * J + 2
  - Max = J
  - if L < S and A[L] > A[Max]:
    - Max = L
  - if R < S and A[R] > A[Max]:
    - Max = R
  - if Max != J:
    - Swap A[J] with A[Max]
    - MAX-HEAPIFY(A, Max, S)
-

Подготовка кучи

- BUILD-MAX-HEAP(A)
  - for J = A.length / 2 – 1 downto 0
    - MAX-HEAPIFY(A, J, A.length)
-
- Трудоёмкость: O(N) просеиваний, каждое из которых имеет трудоёмкость O(logN)
- Корректность:
  - Инвариант: перед каждой итерацией цикла узлы с номером больше J являются корнями корректной бинарной пирамиды

Пирамидальная сортировка

- HEAP-SORT(A)
  - BUILD-MAX-HEAP(A)
  - for J = A.length – 1 downto 0
    - Swap A[0] with A[J]
    - MAX-HEAPIFY(A, 0, J)s
- Трудоёмкость: O(N) просеиваний, каждое из которых имеет трудоёмкость O(logN)
- Корректность: следует из того, что после Swap только корень пирамиды нарушает её свойство, и из инварианта

Неустойчивые сортировки

- Быстрая (Хоара) T=O(N logN), worst(T)=O(N^2), R=O(1)
  - Может переставлять местами элементы, равные опорному
  - Выбор опорного элемента:
    - Идеальный – медиана (если бы мы могли найти её быстро)
    - Худший – наименьший или наибольший элемент
    - Что будет, если опорный элемент всегда делит массив в соотношении 20/80?
    - Что будет, если его выбирать случайным образом?
    - Что будет, если выбирать элемент посередине?

Быстрая сортировка -- напоминание

- Используется принцип декомпозиции «Разделяй и Властвуй»
- На каждом шаге массив A[Min…Max] путём перестановки элементов разбивается на два подмассива A[Min…R-1] и A[R+1…Max], таких, что
  - A[J] <= A[R] для J < R
  - A[J] >= A[R] для J > R
- Каждый из подмассивов сортируется рекурсивно

Собственно быстрая сортировка

- QUICK-SORT(A, Min, Max)
  - if (Min < Max)
    - R = PARTITION(A, Min, Max)
    - QUICK-SORT(A, Min, R-1)
    - QUICK-SORT(A, R+1, Max)

Производительность быстрой сортировки

- Наихудший случай: R==Min или R==Max
  - T(N) = T(N-1) + O(N)
- Наилучший случай: R==(Min+Max)/2
  - T(N) = 2T(N/2) + O(N)
- 20 / 80: R=0.2Min+0.8Max
  - T(N) = T(0.8N) + T(0.2N) + O(N)
- «Средний» ~ чередование наилучшего и наихудшего

Процедура разбиения

- `PARTITION(A, Min, Max)`
  - `X = A[Max]`
  - `L = Min - 1`
  - `for R = Min to Max`
    - `if A[R] <= X`
      - `L++`
      - `Swap A[L] with A[R]`
  - `return L+1`
  -
- Инвариант: в начале каждой итерации элементы #Min…#L <= X, #L+1…#R-1 >= X

Сортировки сравнениями

- Трудоёмкость в худшем случае O(N logN) или хуже
- Бинарное дерево решений:
  - Внутренние узлы – сравнения между двумя элементами
  - Листья – всевозможные перестановки списка (их N!)
  - Отсюда минимальное число сравнений lg(N!) = O(N logN)
- Тем не менее, существуют сортировки за линейное время...

Сортировки за линейное время

- Все сортировки за линейное время основаны не на сравнениях и предполагают какие-то дополнительные требования к исходным данным
- Сортировка подсчётом
- Поразрядная сортировка
- Карманная сортировка

Сортировка подсчётом

- Работает для целых чисел в интервале от 0 до K, где K = O(N)
  - Также работает для данных, сводимых к таким целым числам, например ...

Сортировка подсчётом

- Работает для целых чисел в интервале от 0 до K, где K = O(N)
  - Также работает для данных, сводимых к таким целым числам, например для элементов перечислений
- Трудоёмкость O(N), ресурсоёмкость O(N), устойчива
- Идея
  - Вначале подсчитать, сколько в списке целых чисел, равных J (для всех J от 0 до K): EqCount(J)
  - Потом подсчитать, сколько в списке целых чисел, меньших J (опять-таки для всех J): LessCount(J)
  - Затем мы размещаем число, равное J, по индексу LessCount(J)
  - Если в списке могут быть равные числа, схема чуть-чуть модифицируется

Сортировка подсчётом

```
COUNTING-SORT(In, Out, K)
  • for J = 0 to K                    // Clear
    • Count[J] = 0
  • for J = 0 to In.length – 1 // Count equals
    • Count[In[j]] ++
  • for J = 1 to K                    // Count less or equals
    • Count[J] += Count[J-1]
  • for J = In.length – 1 downto 0
    • Out[Count[A[J]] – 1] = A[J]
    • Count[A[J]] --
```

Карманная сортировка

- Она же – корзинная (bucket sort)
- Предполагает, что мы имеем числа, распределенные равномерно в некотором интервале
- Трудоёмкость O(N), ресурсоёмкость O(N)
- Идея
  - Разбить интервал на **N** карманов равного размера
  - Распределить числа по карманам в соответствии с их значениями, получив O(1) чисел в каждом кармане
  - Отсортировать числа в каждом кармане отдельно любым простым способом, например, сортировкой вставками
  - Соединить карманы

Карманная сортировка

- Псевдокод для вещественных чисел в интервале [0, 1)
- BUCKET-SORT(In, Out)
  - `N = In.length`
  - `let B: array of lists`
  - `for J = 0 to N - 1`
    - `B[J] = emptyList()`
  - `for J = 0 to N - 1`
    - `K = floor(N*In[J])`
    - `B[K] += In[J]`
  - `Out = emptyList()`
  - `for J = 0 to N - 1`
    - `SORT(B[K])`
    - `Out += B[K]`

Итоги

- Рассмотрены
  - Характеристики сортировок: трудоёмкость, ресурсоёмкость, устойчивость
  - Анализ корректности и трудоёмкости для пирамидальной и быстрой сортировки
  - Показана нижняя граница трудоёмкости для сортировок, основанных на сравнениях
  - Сортировки за линейное время
- Далее
  - Простые структуры данных