Сортировки

August 16, 2016

1 Теорема о нижней оценке времени работы сортировки на произвольном входе

Claim. *Без дополнительной информации о значениях массива, число сравнений при сортировке не менее* $n \log n$

Proof. Сортировка внутри себя делает некоторые сравнения, их результат можно закодировать битовой строкой длины k, где k — число сравнений. На вход нам даётся перестановка массива, всего их $n!$, тогда получаем неравенство $2^k \geq n!$, т.к. иначе по принципу Дирихле мы не сможем отличить две перестановки.

Теперь $2^k \geq n! \Rightarrow k \geq \log n! \sim n \log n \Rightarrow k = \Omega(n \log n)$ □

2 Binary heap

Двоичная куча – структура данных, реализующая интрефейс очереди с приоритетом.

Двоичная куча представляет собой двоичное дерево, где в вершинах дерева лежат значения, так что значение родителя меньше или равно значениям детей (тогда в вершине хранится глобальный минимум, если хотим максимум, то нужно поддерживать свойство другого неравенства).

Операции, которые мы можем реализовать:

- extract min
- add
- sift up
- sift down

Все будут работать за глубину дерева, т.е. $\mathcal{O}(\log n)$

2.1 Sift Down

Заданную вершину мы обмениваем с минимумом/максимумом из детей, пока нарушено свойство кучи.

2.2 Sift Up

Заданную вершину мы обмениваем с родителем пока нарушено свойство кучи.

2.3 Add

Подвешиваем к самой последней вершине и просеиваем вверх.

2.4 Extract Min

Вынимаем вершину и кладём туда нижнюю вершину и просеиваем вниз.

Claim. *Двоичную кучу можно построить за* $\mathcal{O}(n)$

Proof. Сначала поймём как будет выполняться построение.

```
for (int i = n; i > 0; --i) {
    siftDown(i);
}
```

Теперь про время, оценим так:

$$Time = \sum_{i=1}^{n} t_i = \sum_{h=1}^{\log n} h \cdot c_i = \sum_{h=1}^{\log n} h \cdot \frac{n}{2^h} = n \cdot \underbrace{\sum_{h=1}^{\log n} \frac{h}{2^h}}_{\to const \text{ при } n \to \infty} \implies Time = \mathcal{O}(n)$$

h — это расстояние от низа кучи

c_i — это число элементов на одном расстоянии от низа кучи

$c_i = \frac{n}{2^h}$ — $n \sim 2^H$, где H высота кучи, на уровне элементов 2^{H_i}, где H_i расстояние от вершины до уровня, тогда $2^{H-h} = \frac{n}{2^h}$

□

3 Heap sort

3.1 Естественный вариант сортировки

1. строим кучу на массиве за $\mathcal{O}(n)$
2. вынимаем много раз минимум/максимум
3. итоговая ассимптотика $\mathcal{O}(n \log n)$

3.2 Модификация

Мы хотим сейчас изменить сортировку так, чтобы на некоторых входных данных она работала быстрее.

Пусть у нас будет две кучи H — это куча из всех элементов и C — куча кандидатов (изначально состоит из одной вершины).

Теперь алгоритм:

1. вынимаем минимум a из C
2. добавляем в C детей a из H

Теперь, когда это будет работать быстрее. Если минимумы расположены в порядке обхода поиском в глубину, то окажется, что куча C всегда будет содержать не более $\log n$ элементов (глубина кучи H), тогда время работы сортировки будет $\mathcal{O}(n \log \log n)$

4 Merge sort

Merge sort — это сортировка, которая работает за $\mathcal{O}(n \log n)$ времени и $\mathcal{O}(n)$ дополнительной памяти.

4.1 Merge

Merge — это функция, принимающая два отсортированных массива и возвращающая отсортированный массив, состоящий из элементов исходных. Именно эта функция использует дополнительную память.

Как она работает:

1. поддерживаем указатели на текущий элемент в каждом массиве, изначально указывают, на первые элементы
2. теперь пока какой-нибудь из массивов не опустеет выбираем минимум из начальных элементов и ставим его в конец нового массива
3. из оставшегося непустого массива все копируем в конец нового.
4. в новом теперь лежит новый слитый массив

4.2 Sort

1. если размер массива 2, то делаем swap если есть инверсия, если размер массива 1 – то ничего.
2. в остальных случаях делаем рекурсивные вызовы сортировки обеих половин массива.
3. делаем merge отсортированных половинок.

5 Quick sort

5.1 Описание

Сортировка которая в работает за рандомиированный $\mathcal{O}(n \log n)$.

Основная идея взять какой-нибуь элемент и поставить его на правильную позицию, а все остальные расположить правильно относительно позиции выбранного элемента.

Назовём эту операцю `partition`, пример работы:

```
int a[] = { 2, 3, 5, 2, 6, 9, 1, 2};
partition(a, 5);
// a[] = { 2, 3, 2, 1, 2, 5, 6, 9 }
```

Теперь сам алгоритм псевдокодом:

```
qSort(a, len) {
    // base of recursion
    x = random_choice(a);
    t = partition(a, x);
    qSort(a, t);
    qSort(a + t + 1, len - t - 1);
}
```

5.2 Оценка времени работы

Для оценки времени работы нужно воспользоваться магией теории вероятности, нужно дать несколько определений.

5.2.1 Ликбез по терверу

Def:. Ω — *множество событий.*

Def:. $Pr\colon \Omega \to [0,1]$ — *вероятностная функция.*

Def:. $\chi\colon \Omega \to \mathbb{R}$ — *случайная величина.*

Def:. $E(\chi) = \sum\limits_{q\in\Omega} Pr(q)\chi(q)$ — *математическое ожидание случайной величины.*

Claim (О монетке)**.** *Математическое ожидание числа выпадений монетки подряд орлом вверх равно двум.*

Proof. Определимся, что будет являться событиями: Ω — множество выпадений монетки орлом вверх.

Теперь случайная величина: $\chi(q)$ — число бросков монетки.

Теперь, пусть q_k — это событие, что k раз подряд выпал орёл, тогда вероятность этого события можно посчитать по правилу произведения: $Pr(q_k) = \frac{1}{2^k}$

Теперь посчитаем мат. ождание

$$E(\chi) = \sum_{k=0}^{\infty} \frac{k}{2^k} = \frac{1}{2}\sum \frac{1}{2^k} + \frac{1}{4}\sum \frac{1}{2^k} + \frac{1}{8}\sum \frac{1}{2^k} + \cdots =$$
$$\left(\sum \frac{1}{2^k}\right)\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right) = 2 \cdot \frac{1}{2} \cdot 2 = 2$$

P.S.: на самом деле мы в тайне пользуемся абсолютной сходимостью данного ряда, т.к. переставляем его слагаемые. □

5.2.2 Назад к оценке

В первую очередь напишем рекурентное соотношение на время работы:

$$T(n) = T(x) + T(n-x) + \mathcal{O}(n)$$

Первые два слагаемые от рекурсивных вызовов и последнее от операции `partition`.

Интуиция Пусть есть отсортированный массив. Поделим его на 3 части: с крайние части размера $1/4$ от длины массива. Будем считать что середина — это хорошие элементы. Теперь, что плохого в быстрой сортировке? Плохо, если мы выбираем крайние элементы, т.к. тогда рекурента вырождается в следующую вещь:

$$T(n) = T(n-c) + \mathcal{O}(n) \Rightarrow T(n) = n^2$$

Но мат. ожидание выбора плохих элементов подряд — 2, значит большую часть времени мы будем выбирать хорошие элементы и рекурента будет хорошей.

Формализм Давайте посчитаем мат. ожидание времени работы ручками по индукции:

$$E = \sum_x \frac{1}{n} \left(T(x) + T(n-x) + \mathcal{O}(n)\right) = \frac{2}{n} \sum_{x=1}^{n-1} T(x) + \mathcal{O}(n)$$

Применим индукционное предположение:

$$E = \frac{2}{n} \sum_{x=1}^{n-1} \mathcal{O}(x \log x) + \mathcal{O}(n) =$$

$$\frac{2}{n} \sum_{x=1}^{n-1} C \cdot x \log x + \mathcal{O}(n) \leq \frac{2C}{n} \int_1^n x \log x \; dx + \mathcal{O}(n) = \mathcal{O}(n \log n)$$

Посчитанный определённый интеграл можно оценить как $\mathcal{O}(n \log n)$, который съедает лишнее $\mathcal{O}(n)$. Т. о. получили нужную оценку времени работы quick sort.

6 Radix sort

Пусть есть строки/числа. Тогда можно делать сортировку которая будет работать следующим образом: будем сортировать числа/строки по самому младшему разряду, используя стабильную сортировку. Можно использовать ранее упомянутую count sort, тогда время работы такой процедуры будет $\mathcal{O}(nk)$, где n число разрядов, счисления, а k — кол-во чисел.

7 Bucket sort

7.1 Алгоритм

1. найдём границы диапазона
2. если границы равны, то ничего сортировать не нужно
3. разбрасываем элементы массива по группам:

   x_i помещаем в $\left\lceil \frac{x_i - \min}{\max - \min + 1} \cdot n \right\rceil$
4. теперь над каждой группой либо рекурсивно вызваться, либо совершить простую квадратичную сортировку

Грубая оценка времени работы: $\mathcal{O}(n \log(max - min))$, т.к. поделимся хотя бы на две группы.

7.2 Оценки времени для $\mathbb{N}$

Группировать будем по ведущим битам. Тогда получается, что если мы пользуемся только типами `int` или `long long`, то логарифм из предыдущей оценки превращается в 32 или 64 и наша сортирка будет работать за $\mathcal{O}(n)$.