

Алгоритмы и структуры данных. Теория

Александр Сергеев

1 Общие слова и Теоремы

RAM-модель - модель компьютера, в которой обращение к памяти происходит за $O(1)$

Асимптотика:

$$f(n) = O(g(n)) \Leftrightarrow \exists n_0 \in \mathbb{N}, C > 0 : \forall n > n_0 \ f(n) \leq g(n) \cdot C$$

$$f(n) = o(g(n)) \Leftrightarrow \forall C > 0 \ \exists n_0 \in \mathbb{N} : \forall n > n_0 \ f(n) < g(n) \cdot C$$

$$f(n) = \Omega(g(n)) \Leftrightarrow \exists n_0 \in \mathbb{N}, C > 0 : \forall n > n_0 \ f(n) \geq g(n) \cdot C$$

$$f(n) = \omega(g(n)) \Leftrightarrow \forall C > 0 \ \exists n_0 \in \mathbb{N} : \forall n > n_0 \ f(n) > g(n) \cdot C$$

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$

Мастер-теорема

Пусть есть функция, работающая от $T(n)$, $T(n) = a \cdot T(\frac{n}{b}) + n^C$

$$\text{Тогда } T(n) = \begin{cases} O(n^C), C > \log_b a \\ O(n^C \log n), C = \log_b a \\ O(n^{\log_b a}), C < \log_b a \end{cases}$$

Время работы рандомизированного алгоритма - $\max_{input} E(T(input))$, где E - математическое ожидание. T - время для данного $input$.

Теорема о нижней оценке для сортировки сравнениями

Любая сортировка сравнениями работает не быстрее, чем $O(n \log n)$.

Доказательство

Сопоставим сортировке сравнениями дерево, где вершиной будет являться операция сравнения, ребром - результат сравнения, а листом - конечная перестановка элементов

Рассмотрим все перестановки длины n . Всего их будет $n!$. Тогда в нашем дереве будет $n!$ листьев. Тогда глубина дерева $\log_2 n! = \log_2 1 + \log_2 2 + \dots + \log_2 n \geq \frac{n}{2} \log_2 n$.
Отсюда глубина дерева $\Omega(n \log n)$, т.е. необходимо совершить $\Omega(n \log n)$ сравнений, ч.т.д.

Теорема

Для проверки корректности работы сортировки достаточно проверить ее на всех возможных массивах из 0 и 1

2 Структуры

2.1 Двоичная куча

```

1      #Куча хранится в массиве arr, где для вершины v потомками
      являются вершины 2*v+1 и 2*v+2
2
3      #Проверяет, что для вершины v и ее потомков выполнено условие кучи
4      #Иначе опускает значение v вниз по дереву, пока условие не
      выполнится
5      #O(log n)
6      def sift_down(v):
7          while True:
8              m = v
9              if 2*v+1 < len(arr) and arr[2*v+1] < arr[v]:
10                 m = 2*v+1
11                 if 2*v+2 < len(arr) and arr[2*v+2] < arr[v] and
arr[2*v+2] < arr[2*v+1]:
12                     m = 2*v+2
13                 if m == v: break
14                 swap(arr[v], arr[m])
15                 v = m
16
17
18      #Проверяет, что для вершины v и ее родителя выполнено условие кучи
19      #Иначе поднимает значение v вверх по дереву, пока условие не
      выполнится
20      #O(log n)
21      def sift_up(v):
22          while v != 0 and arr[(v-1)//2] > arr[v]:
23              swap(arr[v], arr[(v-1)//2])

```

```

24         v = (v-1)//2
25
26     #Возвращает минимум из кучи и удаляет его
27     #O(log n)
28     def extract_min():
29         res = arr[0]
30         swap(arr[0], arr[-1])
31         del arr[-1]
32         sift_down(0)
33         return res
34
35     #Помещает v в кучу
36     #O(log n)
37     def insert(v):
38         arr.append(v)
39         sift_up(len(arr)-1)

```

Методы построения кучи:

1. Последовательное добавление элементов в кучу через insert
 $O(n \log n)$
2. Построение на том же массиве с начала
 $O(n \log n)$

```

1     for i in 1...n-1:
2         sift_up(i)

```

3. Построение на том же массиве с конца
 $O(n)$

```

1     for i in n-2...0:
2         sift_down(i)

```

Доказательство асимптотики: Рассмотрим каждый уровень нашего дерева. Для уровня i операция `sift_down` выполняется за $\log n - i$. Всего на уровне не более 2^i элементов. Отсюда количество операций

$$\sum_{i=1}^{\log n} 2^i (\log n - i) = \sum_{i=1}^{\log n} 2^{\log n - i} i = \sum_{i=1}^{\log n} \frac{ni}{2^i} < n \sum_{i=1}^{\infty} \frac{i}{2^i} = n \left(\begin{array}{cccccc} \frac{1}{2} & + & & & & \\ \frac{1}{4} & & \frac{1}{4} & + & & \\ \frac{1}{8} & & \frac{1}{8} & & \frac{1}{8} & + \\ \frac{1}{16} & & \frac{1}{16} & & \frac{1}{16} & + \\ \dots & & \dots & & \dots & + \end{array} \right) =$$

$$n \sum_{i=1}^{\infty} \left(\frac{2}{2^i} \sum_{i=1}^{\infty} \frac{1}{2^i} \right) = n \sum_{i=1}^{\infty} \frac{2}{2^i} \cdot 1 = 2n \cdot 1, \text{ ч.т.д.}$$

2.2 Персистентность

Это способ хранения структуры, при котором мы можем получить любое состояние структуры во времени и создать на его основе новое (возможно множественное ветвление) Самый простой способ - хранить копии структуры после каждого изменения

Также можно хранить ссылки на предыдущие состояния:

Пример для стека на односвязном списке: мы храним все состояния переменной-ссылки `top` во времени. При `pop` в новое состояние мы записываем ссылку на `top->prev`. `push` - добавляем вершину, ссылающуюся на `top` и в новое состояние записываем ссылку на этот элемент

Частично-персистентная структура - структура, хранящая все свои состояния, где все состояния кроме последнего - `readonly`

Для частично-персистентного двусвязного списка: в каждой вершине храним две пары (`prev`, `next`) и `t`. Если текущая версия меньше `t`, ходим по первой паре, иначе - по второй. Изначально вторая пара не используется, `t = ∞`. Изменения происходят в момент модификации. Если мы хотим записать во вторую пару значения в момент, когда там уже что-то лежит, то мы копируем этот элемент в новую версию.

3 Алгоритмы

3.1 Heap sort

Суть алгоритма: последовательно вытаскиваем минимум из кучи, получая отсортированный массив

Время $O(n \log n)$

Дополнительная память $O(1)$

3.2 Алгоритм Карацубы

Рассмотрим многочлены

$$P = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x^1 + a_0$$

$$Q = b_n x^n + b_{n-1} x^{n-1} + \dots + b_1 x^1 + b_0$$

Пусть

$$P_0 = a_n x^{n-m} + a_{n-1} x^{n-m-1} + \dots + a_m$$

$$P_1 = a_{m-1} x^{m-1} + a_{m-2} x^{m-2} + \dots + a_1 x^1 + a_0$$

$$\text{Отсюда } P = P_0 x^m + P_1$$

$$Q_0 = b_n x^{n-m} + b_{n-1} x^{n-m-1} + \dots + b_m$$

$$Q_1 = b_{m-1} x^{m-1} + b_{m-2} x^{m-2} + \dots + b_1 x^1 + b_0$$

$$\text{Отсюда } Q = Q_0 x^m + Q_1$$

Тогда

$$P \cdot Q = (P_0 x^m + P_1)(Q_0 \cdot x^m + Q_1) = P_0 Q_0 x^{2m} + (P_1 Q_0 + P_0 Q_1) x^m + P_1 Q_1$$

$$\text{Заметим, что } P_1 Q_0 + P_0 Q_1 = (P_0 + P_1)(Q_0 + Q_1) - P_0 P_1 - Q_0 Q_1$$

$$\text{Отсюда } PQ = P_0 Q_0 x^{2m} + ((P_0 + P_1)(Q_0 + Q_1) - P_0 P_1 - Q_0 Q_1) x^m + P_1 Q_1$$

Т.о. нам для подсчета PQ нам достаточно посчитать $P_0 P_1$, $Q_0 Q_1$ и $(P_0 + P_1)(Q_0 + Q_1)$

$$\text{Тогда при } m = \frac{n}{2}: T(n) = 3T\left(\frac{n}{2}\right)$$

$$\text{Отсюда получаем время } O(n^{\log_2 3})$$

3.3 Quick sort

```
1 #sort [l,r)
2 #Перебрасываем все элементы меньше x налево
3 #Замечание: Нет поддержки одинаковых чисел
4 def sort(l,r):
5     if r-l <= 1:
6         return
7     x = arr[(l+r)//2]
8     m = l
9     for i = l...r-1
10         if arr[i] <= x:
11             swap(arr[i], arr[m])
12             m+=1
13     sort(l,m)
14     sort(m,r)
```

```
1 #Рандомизированная версия
2 #Замечание: Нет поддержки одинаковых чисел
3 def sort(l,r):
```

```

4     if r-l <= 1:
5         return
6     x = arr[randint(1,r)]
7     m = l
8     for i = 1...r-1
9         if arr[i] <= x:
10             swap(arr[i],arr[m])
11             m+=1
12     sort(1,m)
13     sort(m,r)

```

Представим, что с шансом $\frac{1}{3}$ мы разбиваем массив на куски $\frac{1}{3}$ и $\frac{2}{3}$ и с вероятностью $\frac{2}{3}$ длина не меняется. Тогда мат.ожидание количества вызовов для получения "удачного" $1p + 2(1-p)p + 3(1-p)^2p + \dots = \sum_{i=1}^{\infty} i(1-p)^{i-1}p = 3$, где $p = \frac{1}{3}$. Тогда мат.ожидание высоты дерева вызовов $3 \log_{1.5} n$, а средняя асимптотика $O(n \log n)$. В отличие от merge-sort, здесь $O(1)$ дополнительной памяти.

```

1 #Замечание: Есть поддержка одинаковых чисел
2 def sort(l,r):
3     if r-l <= 1:
4         return
5     x = arr[randint(1,r)]
6     m1 = m2 = l
7     for i = 1...r-1
8         if arr[i] > x: pass
9         elif arr[i] == x:
10             swap(arr[m2],arr[i])
11             m2+=1
12         else:
13             swap(arr[i],arr[m2])
14             swap(arr[m2],arr[m1])
15             m1+=1
16             m2+=1
17     sort(1,m1)
18     sort(m2,r)

```

3.4 k-я порядковая статистика

Цель - вывести k-ый элемент отсортированного массива, не сортируя массив.

```

1 #Замечание: Нет поддержки одинаковых чисел
2 def find(l,r,k):
3     if r-l <= 1:
4         return arr[l]
5     x = arr[randint(l,r)]
6     m = l
7     for i = l...r-1
8         if arr[i] <= x:
9             swap(arr[i],arr[m])
10            m+=1
11 if k < m-l:
12     return find(l,m,k)
13 else:
14     return find(m,r,k-(m-l))

```

Асимптотика $3(n + \frac{2}{3}n + (\frac{2}{3})^2n + (\frac{2}{3})^3n + \dots) = 3n \sum_{i=1}^{\infty} (\frac{2}{3})^i = O(n)$

```

1 #Замечание: Есть поддержка одинаковых чисел
2 def find(l,r,k):
3     if r-l <= 1:
4         return arr[l]
5     x = arr[randint(l,r)]
6     m1 = m2 = l
7     for i = l...r-1
8         if arr[i] > x: pass
9         elif arr[i] == x:
10            swap(arr[m2],arr[i])
11            m2+=1
12        else:
13            swap(arr[i],arr[m2])
14            swap(arr[m2],arr[m1])
15            m1+=1
16            m2+=1
17 if k < m1-l:
18     return find(l,m1,k)
19 elif k == m1-l:
20     return arr[k]
21 else:
22     return find(m2,r,k-(m2-l))

```

3.5 k-я порядковая статистика за линейное время

1. Массив делится на группы по 5 элементов

2. В каждой группе берется медиана
3. Из медиан берется медиана x
4. Массив рассекается по медиане x на две части.
5. Функция вызывается от одной из частей массива.

Заметим, что каждая часть массива содержит не более $(\frac{1}{2} + \frac{2}{5} \cdot \frac{1}{2})n = \frac{7}{10}n$ элементов.

Тогда пусть $T(n)$ - время работы для массива длины n .

Разбиение на группы и поиск их медиан происходит за n

Нахождение медианы медиан происходит за $T(\frac{n}{5})$

Вызов от одной части массива происходит за $T(\frac{7n}{10})$

$T(n) = n + T(\frac{n}{5}) + T(\frac{7n}{10})$. Можно доказать по индукции, что $T(n) = O(n)$.

3.6 Сортировка подсчетом

1. Пусть мы пытаемся отсортировать массив целых чисел a длины n , где $m = \max_{0 \leq i \leq n} a[i]$

Заведем массив c размера $m + 1$, где $c[i]$ - количество элементов, равных i

Посчитаем $c[i]$, пройдясь по массиву a

Теперь пройдемся по массиву c слева направо и выпишем в результирующий массив элемент i ровно $c[i]$ раз. В результате мы получим отсортированный массив

Время работы алгоритма - $O(n)$

2. Пусть мы хотим отсортировать массив a по целому ключу $i \in [0, m]$
Заведем массив c размера $m + 1$, где $c[i]$ - количество элементов с ключем i

Посчитаем $c[i]$, пройдясь по массиву a

Заведем результирующий массив b и в нем выделим $c[i]$ элементов под элемент i . Для этого можно хранить массив индексов p размера $m + 1$, где $p[i]$ - первая свободная ячейка под элемент с ключем i

По умолчанию $p[0] = 0, p[i] = p[i - 1] + c[i - 1]$

Пройдемся по a и будем записывать элемент с ключем i в ячейку

$b[p[i]]$ и увеличивать $p[i]$ на 1
Время работы алгоритма - $O(n)$

3.7 Цифровая сортировка

Пусть $0 \leq a_i < n^k$

Представим $a_i = d_{i0} \cdot n^{k-1} + d_{i1} \cdot n^{k-2} + \dots + d_{ik-1}$.

1. a сортируем по d_{k-1}
2. a сортируем по d_i стабильно для $i \in k-2 \dots 0$

3.8 Бинарный поиск

```
1 #Будем считать, что arr[-1] = -inf, arr[n] = +inf
2 #Поиск наименьшего числа не меньше данного
3 def binSearch(x):
4     l = -1
5     r = n
6     while r-l > 1:
7         m = (r+l)//2
8         if arr[m] < x:
9             l = m
10        else:
11            r = m
12    #l - наибольшее <
13    #r - наименьшее >=
14    return r
15
16 #Будем считать, что arr[-1] = -inf, arr[n] = +inf
17 #Поиск наибольшего числа не больше данного
18 def binSearch(x):
19     l = -1
20     r = n
21     while r-l > 1:
22         m = (r+l)//2
23         if arr[m] <= x:
24             l = m
25        else:
26            r = m
27    #l - наибольшее <=
28    #r - наименьшее >
29    return l
```

3.9 Тернарный поиск

Пусть есть функция, которая сначала возрастает, а потом убывает.

```
1 #поиск максимума
2 def search(x):
3     while r-l > e:
4         m2 = l+(r-l)/3*2
5         if f(m1) < f(m2):
6             l = m1
7         else:
8             r = m2
9     return r
```

Соптимизируем вычисление функции

```
1 #поиск максимума
2 def search(x):
3     while r-l > e:
4         m2 = l+(r-l)/3*2
5         if f(m1) < f(m2):
6             l = m1
7             m1 = m2
8             m2 = r-p(r-l) #p - коэффициент золотого сечения
9         else:
10            r = m2
11            m2 = m1
12            m1 = l+p(r-l)
13    return r
```

Также возможен поиск через бинарный поиск $x : f'(x) = 0$