

Алгоритмы и структуры данных

Коченюк Анатолий

6 ноября 2020 г.

0.1 Введение

курс будет идти 4 семестра.

1 лекций + 1 практика в неделю

баллы: практика – выходишь к доске и делаешь задание.

практика – до 30 баллов

0-25 – по 5 баллов 25-40 – по 3 балла 40+ – по 1 баллу


лабораторные: 50 баллов

экзамен: в каком-то виде будет. до 20 баллов.

Глава 1

I курс

1.1 Алгоритмы

Алгоритм: входные данные \rightarrow  \rightarrow выходные данные

входной массив $a[0 \dots n-1]$, выходная сумма $\sum a_i$

```
S = 0          \ \ 1
for i = 0 .. n-1 \ \ 1+2n
    S+=a[i]      \ \ 3n

print(S)       \ \ 1
```

Модель вычислений.

RAM - модель. симулирует ПК. За единицу времени можно достать/положить в любое место памяти.

Время работы (число операций) В примере выше $T(n) = 3 + 5n$

мотивация: 3 становится мальньким, а 5 – не свойство алгоритма

$T(n) = O(n)$

$f(n) = O(g(n)) \iff \exists n_0, c \quad \forall n \geq n_0 \quad f(n) \leq c \cdot g(n)$

$n_0 = 4, c = 6 \quad 3 + 5n \leq 6n, n \geq 4 \quad 3 \leq n$

$f(n) = \Omega(g(n)) \iff \exists n_0, c \quad \forall n \geq n_0 \quad f(n) \geq c \cdot g(n)$

$3 + 5n = \Omega(n) \quad n_0 = 1, c = 1$

$$3 + 5n \geq n, n \geq 1$$

$$T(n) = O(n), T(n) = \Omega(n) \iff T(n) = \Theta(n)$$

```
for i = 0 .. n-1
    for j = 0 .. n-1
```

$$- O(n^2)$$

```
for i = 0 .. n-1
    for j = 0 .. i-1
```

$$\sum_{i=0}^{n-1} i = \sum \frac{n \cdot (n-1)}{2} = \Theta(n^2)$$

```
i=1
while i\cdot i<n
    i++
i=1
while i < n
    i=i\cdot \cdot 2
```

$$O(\sqrt{n}), O(\ln n)$$

```
f(n):
    if n=0
        ..
    else
        f(n-1)
```

n рекурсивных вызовов $O(n)$

```
f(n):
    if n=0
        ..
    else
        f(n/2)
        f(n/2)
```

$$2^{\ln n} = n$$

если добавить третий вызов: $2^{\log_2 n} = n^{\log_2 3}$

1.2 Сортировки

1.2.1 Сортировка вставками

Берём массив, идём слева направо: берём очередной элемент и двигаем влево, пока он не упрётся

```
for i = 0 .. n-1
    j=i
    while j>0 and a[j]<a[j-1]
        swap(a[j-1], a[j])
        j--
```

Докажем, что алгоритм работает. по индукции. Если часть отсортирована и мы рассматриваем новый элемент, то он будет двигаться, пока не вставиться на своё место и массив снова будет отсортированным.

Если массив отсортирован $(1, 2, \dots, n) - O(n)$

Если нет $(n, n-1, \dots, 1)$, то $O(n^2)$

Рассматривать мы дальше будем худшие случаи.

1.2.2 Сортировка слияниями

Слияние: из двух отсортированных массивов делает один отсортированный.

как найти первый элемент. Он наименьший, значит либо самый левый в массиве a , либо в массиве b . Мы забыли нужный первый элемент и свели к такой же задаче поменьше.

```
merge(a,b):
    n = a.size()
    m = b.size()
    i=0, j=0
    while i<n or j<m:
        if j==m or (i<n and a[i]<b[j]):
            c[k++] = a[i++]
        else
            c[k++] = b[j++]
    return c
```

$O(n + m)$

Сортировка: берём массив, делим его пополам, рекурсивно сортируем левую и правую часть, а потом сольём их в один отсортированный массив.

```
sort(a):
    n = a.size()
```

```

    if n<=1:
        return a
    al = [0, .. n/2-1]
    ar = [n/2 .. n-1]
    al = sort(al)
    ar = sort(ar)
    return merge(al, ar)

```

порядка n рекурсивных массивов.

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$$

красиво и понятно:

математически и хардкорно: по индукции $T(n) \leq \ln n$

База: $n = 1$ – не взять из-за логарифма, но можно на маленькие n не обращать внимания

Переход:

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq 2 \cdot \frac{n}{2} \ln \frac{n}{2} + n = n(\ln n - 1) + n = n \ln n + n(1 - 1) \leq \ln n$$

Теорема 1 (Мастер-теорема). $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

Если без $f(n) = O(n^{\log_b a - \varepsilon})$, то $a^{\log_b n} = n^{\log_b a}$

Если без $f(n) = O(n^{\log_b a + \varepsilon})$, тогда $T(n) = O(f(n))$

Если без $f(n) = O(n^{\log_b a})$, то $T(n) = n^{\log_b a} \cdot \ln n$

1.3 Структуры данных

Структура, которая хранит данные

Операции: структура данных определяется операциями, которые она умеет исполнять

Массив:

- `get(i)` (`return a[i]`)
- `put(i,v)` (`a[i] = v`)

Время работы на каждую операцию

1.3.1 Двоичная куча

Куча:

-
- храним множество ($x < y$)
 - $\text{insert}(x) \quad A = A \cup \{x\}$
 - $\text{remove_min}()$

Варианты:

1. Массив

- $\text{insert}(x) \quad a[n++] = x \quad (O(1))$
- $\text{remove_min}() \quad (O(n))$

```
j=0
for i=1 .. n-1
    if a[i]<a[j]: j=i
swap(a[j], a[n-1])
return a[--n]
```

2. Отсортированный массив (по убыванию)

- $\text{remove_min}()$

```
return a[--n]
```

- $\text{insert}(x)$

```
a[n++] = x
i=n-1
while i >0 and a[i-1]<a[i]
    swap(a[i], a[i-1])
    i--
```

3. Куча. Двоичное дерево, каждого элемента – 2 ребёнка. У каждого есть один родитель (кроме корня). В каждый узел положим по элементу. Заполняется по слоям. Правило: у дети больше родителя. Минимум в корне – удобно находить.

Занумеруем все элементы слева направо. Из узла i идёт путь в $2i + 1$ и $2i + 2$

```
insert(x)
a[n++] = x
i=n-1
while i>0 and a[i]<a[(i-1)/2]
    swap(a[i], a[(i-1)/2])
    i = (i-1)/2
```

$O(\log n)$

Идея убирания минимума: поставить вверх вместо минимума последний элемент и сделать просеивание вниз.

```

remove_min()
    res = a[i]
    a[0] = a[--n]
    i=0
    while True:
        j=i
        if 2i+1<n and a[2i+1]<a[j]:
            j=2i+1
        if 2i+2<n and a[2i+2]<a[j]:
            k=2i+2
        if j == i: break
        swap(a[i], a[j])
        i=j
    return res

```

1.3.2 Сортировка Кучей (Heap Sort)

```

sort(a):
    for i = 1 .. n-1: insert(a[i])
    for i = 1 .. n-1: remove_min()

heap_sort(a)
    for i = 0 .. n-1
        sift_up(i)
    for i = n-1 .. 0
        swap(a[0], a[i])
        sift_down(0, i) // i -- размер кучи

```

1.4 Быстрая сортировка

1.4.1 Рандомизированные алгоритмы

Алгоритм: Пусть есть массив и все элементы различны. Давайте выберем случайный элемент Поделим массив на две части: $< x$ и $\geq x - O(n)$

Рекурсивно запускаем от каждого куска.

```

a // глобальный массив
sort(l, r):
    x = a[random(l..r-1)]
    if r-l =1:
        return
    m=l

```

```

for i = 1 .. r-1:
    if a[i]<x:
        swap(a[i],a[m])
    m++
sort(l,m)
sort(m,r)

```

Вместо изучения худшего случая рандомизированного алгоритма мы изучаем мат ожидание.

$$E(T(n))$$

$$E(x) = \sum t \cdot p(x = t)$$

Покажем, что мат ожидание времени работы нашего алгоритма $O(n \log n)$

Подход №1: посмотрим. был массив, поделили на две части, от каждой части запустились. каждая часть примерно $\frac{n}{2}$ $T(n) = n + 2T(\frac{n}{2})$ $O(n \log n)$

Скорее всего поделимся не ровно пополам.

Подход №2: поделим на 3 части. средняя – хорошая часть, выбрав элемент в которой части получаются $\leq \frac{2}{3}n$. Каждый третий раз пилим пополам примерно. $E(T(n)) \leq 3 \cdot \log_{\frac{3}{2}} n = O(\log n)$

Определение 1. К-я порядковая статистика: ровно k элементов меньше выбранного.

Сортировкой: $O(n \log n)$

Можно быстрее: Алгоритм Хоара

Возьмём массив a , выберем случайный элемент x . Распилим массив на 2 куска : $< x, \geq x$

Если знаем k , которое ищем, то выбираем одну часть и смотрим там.

```

a // глобальный массив
find(l, r, k): // l<=k<r
    x = a[random(l..r-1)]
    if r-l = 1: // l=k, r = k+1
        return
    m=l
    for i = 1 .. r-1:
        if a[i]<x:
            swap(a[i],a[m])
        m++
    if k<m:
        find(l,m,k)

```

```
else:
    find(m,r,k)
```

1.5 Алгоритм Блут-Флойд-Пратт-Ривест-Тарьян

Разобьём массив на блоки по 5 элементов. $\frac{n}{5}$ блоков. В каждом блоке выбираем медиану. Выбираем медиану среди всех медиан. Если брать медиану из медиан, то это будет неплохой средний элемент

$$T(n) = n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) = O(n)$$

$$T(n) \leq c \cdot n$$

$$T(n) \leq n + c \cdot \frac{n}{5} + c \cdot \frac{7n}{10} = n \left(1 + \frac{9}{10} \cdot c\right) < c \cdot n \quad 10 \leq c$$

Ресар: Изучили

- MergeSort
- HeapSort
- QuickSort

Все за $O(n \log n)$

Что мы можем делать: сравнивать элементы и перекладывать их.

Программа: запустилась, что-то делает, сравнилось: $a[i] < a[j]$. От неё две ветки на два случая. Потом появляются сравнения в подслучаях, дающие больше случаев.

Есть три элемента: x, y, z . Отсортируем их.

$x < y$

$< | x < z$

$< | x$ — минимальный.

$y < z$

$< | xyz$

$\nless | xzy$

$\nless | zxy$

$\nless | y < z$

$< | x < z$

$< | yxz$

$\nless | yzx$

$\nless | zyx$

В листьях перестановки листьев. $n!$ листьев. Глубина хотя бы $\log n!$

$$T \geq \log n! = \sum_{i=1}^n \log i = \Omega(n \log n)$$

Можно ли сделать что-то быстрее не только сравнивая.

Пусть есть массив $a[0 \dots n-1]$. Какие-то маленькие целые числа: $a[i] \in [0 \dots m-1]$, m — маленькое.

1.6 Сортировка подсчётом

$a = [2, 0, 2, 1, 1, 1, 0, 2, 1]$

cnt = [0,0,0] увеличиваем, проходя по массиву

$a' = [0, 0, 1, 1, 1, 1, 2, 2, 2]$

$O(n+m)$

Но часто сортируются не числа, а объекты, к которым приделаны числа.

```
class Item {
    int key; // in [0..m-1]
    Data data;
}
```

Создадим ящики для объектов с ключами 0,1 и 2.

В реальности лучше создавать один массив и просто раскладывать в блоки внутри этого массива.

А дальше заполняем эти ящики проходя по массиву. А потом соберём их в один большой массив.

$x = [0 \dots m^2 - 1]$ $x = a \cdot m + b$ $a, b \in [0 \dots m - 1]$ – двухзначное число в m -ичное число

$$a[i] = b[i] \cdot m + c[i]$$

Если нужно отсортировать $a[i]$, то это то же самое, что отсортировать пары $(b[i], c[i])$ по первому элементу, а при равенстве по второму.

Пусть есть числа: $a = [02\ 21\ 01\ 11\ 21\ 20\ 02\ 00]$

Можно сортировать по первой цифре, а потом по второй. Но это работает довольно долго.

А если сортировать слева направо, то лучше:

```
cnt = [2,4,2]
a' = [20 00 | 21 01 11 21 | 02 02]
cnt = [4,1,3]
a'' = [00 01 02 02 | 11 | 20 21 21]
```

$$O(n + m)$$

Дофиксим: если $a[i] \in [0 \dots m^k - 1]$

$$O(k \cdot (n + m))$$

1.7 Сортирующие сети

Идея: одна операция – компаратор

```
cmp(i, j):
    if a[i] > a[j]
        swap(a[i], a[j])
```

$$n = 2 \quad \text{cmp}(0, 1)$$

$$n = 3 \quad \text{cmp}(0, 1) \quad \text{cmp}(0, 2) \quad \text{cmp}(1, 2)$$

сортировка выбором: на первую позицию ставится минимальный элемент, потом сортируются оставшиеся.

Возьмём сортировку вставками

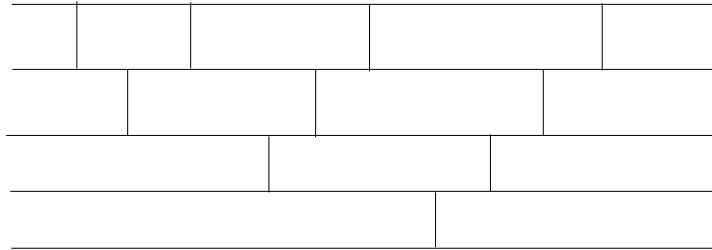


Рис. 1.1: net

Порядка n^2 компараторов нужно. Разрешим делаться несколькими компараторам сразу: $\text{cmp}(0,1)$, $\text{cmp}(2,3)$ друг другу не мешают. Разрешим неконфликтующим компараторам сколько угодно выполняться одновременно.

Элементов уже порядка n

Утверждение 1. Если сеть сортирует любой массив из 0 и 1, то она сортирует любой массив

Доказательство. Пусть сеть сортирует любую последовательность 0 и 1. Дадим ей какой-то массив. Наименьший элемент отметим 0, он придёт наверх в сети. Возьмём следующий элемент, он вылезет следом за 1. ■

1.8 Bitonic sort

Битонная последовательность – сначала возрастает, потом убывает.

Рассмотрим только 0 и 1.

$a = [0\ 0\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0]$

Положим последовательность пополам. Применяем сеть, где сравниваются соответствующие элементы. Обе части битонные, правая больше, чем левая.

В общем случае: сравниваем парами, каждый второй разворачиваем. Получаем битонные последовательности через 4. Применяем к ним Bitonic Sort.

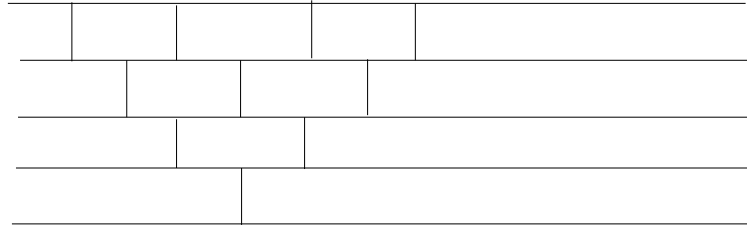


Рис. 1.2: net2

1.9 Двоичный поиск

Есть массив a отсортированный по возрастанию

Как писать НЕ надо:

```
x = 8, i: a[i] = x
a 2 5 8 13 21 27 35
  l           r
x in a[l..r] -- инвариант. Будем сужать.
m=floor((l+r)/2)    l<=m<=r
a[m] >x => a[j] >x, j>m
```

```
l=0, r=n-1
while (r-l+1)>0:
    m=(l+r)/2
    if a[m]>x // a[m..r]>x
        r = m - 1
    else if a[m]<x // a[l..m]<x
        l = m + 1
    else
        return m
```

$O(\log n)$

Задача: найти минимальный $i : a[i] \geq x$ Как писать надо:

```
l, r
```

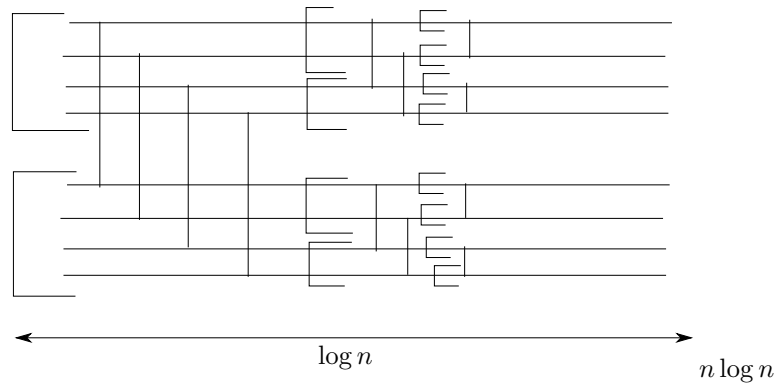


Рис. 1.3: bitonic

```

a[l]<x
a[r]>=x
l=-1, r=n
while (l-r)>1:
    m = (l+r)/2
    if a[m]<x:
        l = m
    else
        r=m
return r // if r=n: такого элемента вообще нет

```

Задача: найти максимальный $i : a[i] \leq x$

```

l, r
a[l]<=x
a[r]>x
l=-1, r=n
while (l-r)>1:
    m = (l+r)/2
    if a[m]<=x:
        l = m
    else
        r=m
return l // if l=-1: такого элемента вообще нет

```

$$x \in \mathbb{N} \begin{cases} \text{хорошие} \\ \text{плохие} \end{cases}$$

$$x - \text{хорошее} \implies x + 1 - \text{хорошее}$$

Задача 1. найти минимальное хорошее число.

n штук прямоугольничков $w \cdot h$. Мы хотим запихать их в квадрат. Вопрос: какой наименьший квадрат подойдёт.

Если можно впихнуть в x^2 , то и в $(x+1)^2$ тоже.

```

l -- плохое
r -- хорошее

l = 0
r = max(h,w)*n

good(l) = 0
good(r) = 1

while (l-r)>1:
    m = (l+r)/2
    if good(m):
        r = m
    else:
        l = m
return r

good(x):
    return (x/h)*(x/w)>=n

```

$$O(\log(l-r)) = O(\log(\max(h,w) \cdot n)) = O(\log(h+w+n))$$

Нужно аккуратно брать правое число, чтобы не было переполнений.

$$r = \max(h, w) \cdot \lceil \sqrt{n} \rceil$$

$$2^k - \text{плохое}, 2^{k+1} - \text{хорошее}$$

$$\log(a+b) = O(\log a + \log b)$$

Задача 2. Есть прямая

На неё в точках x_i живут котики

v_i – скорость

Собратся в одной точке за min время

t – хорошее, если за t можно собратся

Как писать HE надо:

```
r = 0, l = 10^10, EPS = 10^-6
while (r-l)>EPS:
    m = (l+r)/2
    //fix
    if m <= l || m >= r
        break
    //xif
    if good(m):
        r = m
    else:
        l = m
good(x):
    x >= max(li)
    x <= max(ri)
    x -- существует, если max(li) <= min(ri)
```

$O(\log(\frac{r-l}{EPS}) \cdot n)$

так писать не надо, потому что не все числа с нужной точностью можно получить. Точности может не хватать и цикл станет вечным.

Как исправить: можно сделать for

1.10 Троичный поиск

$$m_1 = \frac{2l+r}{3} \quad m_2 = \frac{l+2r}{3}$$

```
if f(m2)>f(m1):
    l = m1
else:
    r = m2
```

$O(\log_{\frac{3}{2}} \frac{r-l}{EPS})$

1.11 Стеки и очереди

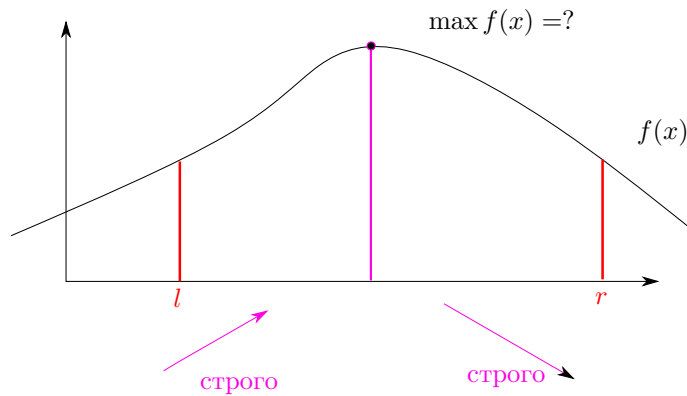


Рис. 1.4: tri

Определение 2. Стек – “стаканчик”.



.push(A) – добавить сверху элемент A

.push(B), .push(C)

.pop() -> C – вернуть самый верхний элемент

LIFO – last in first out

`a = [A,B,C, \ldots]` -- первые n элементов заполнены

`push(x):`

`a[n++] = x`

`pop(x):`

`return a[--n]`

Определение 3. Очередь – “очередь в магазин”

голова очереди – то место, которое обрабатывается

хвост очереди – то, куда добавляются новые элементы

`a = [A,B,C,D,E,F, \ldots]`

`head` -- первый неудалённый элемент

tail -- на первый свободный

```
add(x):  
    a[tail++] = x  
remove():  
    return a[head++]
```

Определение 4. Дек – можно класть и доставать с обоих концов. Когда точно не знаешь нужна тебе очередь или стек.

В обеих реализациях мы считаем, что у нас бесконечно большой массив.

решение 1: Пусть мы знаем, что в очереди n элементов. зациклить массив в очереди, чтобы когда место справа закончится, добавлять в начало.

Но что если мы не знаем сколько элементов максимум.

Сделаем стек, не зная сколько в нём максимум элементов.

```
a = [x, x, x]  
a' = [x, x, x, _, _, _]
```

Если не влезает -- расширять в 2 раза.

```
push(x):  
    if n == a.size():  
        a' = new int[2*n]  
        copy(a[0..n-1] -> a'[0..n-1])  
        a = a'  
    a[n++] = x
```

1.12 Амортизационный анализ

o_1, o_2, \dots, o_k – операции, проделанные в таком порядке

$T(o_i)$ – время работы операции

$\tilde{T}(o_i)$ – амортизированное время работы (выбирается нами)

Амортизированное время хорошее, если $\sum \tilde{T}(o_i) \geq \sum T(o_i)$

Пусть мы хотим $\tilde{T}(o_i) \leq c \implies \sum_k T(o_i) \leq c \cdot k$

Пусть мы делаем k пушей

$$\sum T \leq k + \underbrace{2^p}_{\leq 2k} \leq 3k \quad \tilde{T}(\text{push}) = 3$$

1.12.1 Метод потенциалов

:

Φ – потенциал (выбираемый опять же нами)

$$\tilde{T} = T + \Delta\Phi$$

$$\Phi_0 = 0, \quad \Phi \geq 0$$

$$\sum \tilde{T} = \sum T + \sum \underbrace{\Delta\Phi}_{=\Phi_k=\Phi_0} \geq \sum T$$

Цель: $\Phi = ? \quad \tilde{T} = O(1)$

Φ = (число элементов правой половине массива)·2

$$\tilde{T} = 1 + 1 = O(1)$$

$$\sum \tilde{T} = n - n + 1 = O(1)$$

1.12.2 Метод бухгалтерского учёта (метод с монетами)

```
put_coin(x) // ~T = x
take_coin(x) // ~T = -x
```

```
[2r, 2r, 2r, 2r] -> [2r, 2r, 2r, 2r, _, _, _, _]
```

мы "тратим" 4 рубля, чтобы увеличить массив из 4 элементов в 2 раза. Для копирования: берём +

Теперь про *pop*

```
pop():
    return a[--n]
```

Если сначала много запустить, а потом много заполнить, то останется много свободного места, которое мы не используем

Если массив заполнен меньше, чем на половину, можно сужать его в два раза. Но тогда на границе половины, если чередовать, то он будет постоянно сужаться-расширяться.

```
pop():
    if n <= a.size()/4: ..
```

В правой половине массив лежат монетки по 2 рубля для расширения.

Когда делаем *pop*, то кладём по рублю и копируем для сужения.

```
add(x):
    s2.push(x)
remove(x):
    if s1.empty():
```

```
        while !s2.empty:
            s1.push(s2.pop())
    return s1.pop()
```