Алгоритмы и структуры данных

Коченюк Анатолий

2 декабря 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 $\rightarrow \boxed{} \rightarrow$ 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

$T(n) = O(n)$

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

$n_0 = 4, c = 6 \quad 3 + 5n \leqslant 6n, n \geqslant 4 \quad 3 \leqslant n$

$f(n) = \Omega(g(n)) \iff - - || - - f(n) \geqslant cg(n)$

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

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

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

```
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 в 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 $(1, 2, \ldots, n)$ – $O(n)$

Если нет(n, n-1, …, 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 первй элемент. Он наименьший, значит либо самый левые 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.

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

красиво и понятно:

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

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

Переход:

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

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 $T(n) = n^{\ln_b a} \cdot \ln n$

1.3 Структуры данных

Структура, которая хранит данные

Операции: структура данных определляется 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 ($x < y$)
- insert(x) $\quad A = A \cup \{x\}$
- remove_min()

Варианты:

1. Массив
   - insert(x) $a[n++] = x$ (O(1))
   - remove_min() (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)
   - remove_min()

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

   - 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 $\geqslant x$ – $O(n)$

Рекурсивно запускаем от каждого куска.

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

```
for i = l .. 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 $\frac{n}{2}$ $\quad T(n) = n + 2T(\frac{n}{2}) \quad O(n \log n)$

Скорее всего поделимся не ровно пополам.

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

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

Сортировкой: $O(n \log n)$

Можно быстрее: Аогоритм Хоара

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

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

```
a // глобальнный 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 = l .. 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 Блют-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

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

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

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

Recap: 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

$\not< |$ xzy

$\not< |$ zxy

$\not< |$ $y < z$

$<|$ $x < z$

$<|$ yxz

$\not< |$ yzx

$\not< |$ zyx

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

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

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.

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

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

Если нужно остортировать $a[i]$, 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])
```

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

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

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

Возьмём сортировку вставками

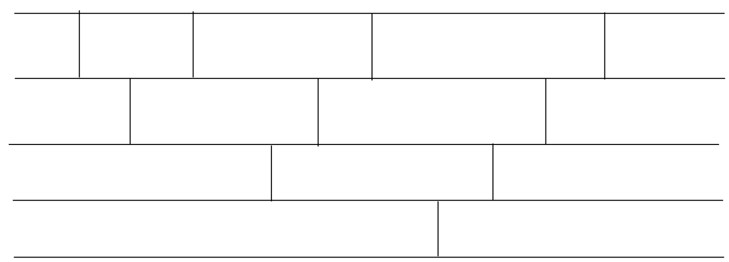


Рис. 1.1: net

Порядка n^2 компараторов нужно. Рзрешим делаться нескольким компараторам сразу: cmp(0,1), cmp(2,3) друг другу не мешают. Разрешим неконфликтующим компараторам скольким угодно выполняться одновременно.

Элементов уже порядка n

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

Доказательство. Пускай сеть сортируют 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]

Пилим 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.

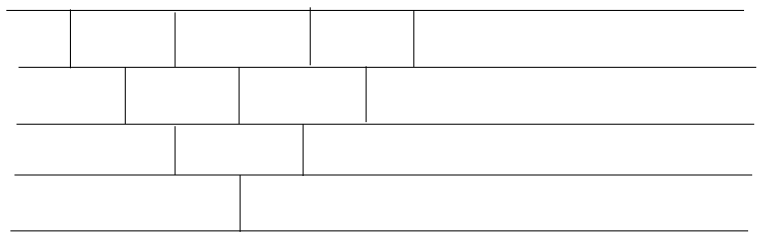


Рис. 1.2: net2

1.9 Двоичный поиск

Есть массив a отсортированный по возрастанию

Как писать <u>НЕ</u> 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 $i : a[i] \geqslant x$ Как писать надо:

```
l, r
```

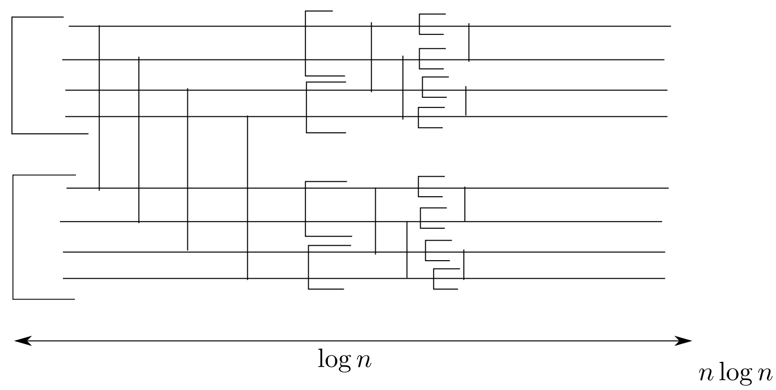


Рис. 1.3: bitonic

```
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 $i : a[i] \leqslant x$

```
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 – хорошее $\implies$ $x+1$ – 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 – плохое, 2^{k+1} – хорошее

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

Задача 2. Есть прямая

На неё в точках x_i живут котики

v_i – скорость

Собраться в одной точке за min время

t – хорошее, если за t можно собраться

Как писать НЕ 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)
```

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

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

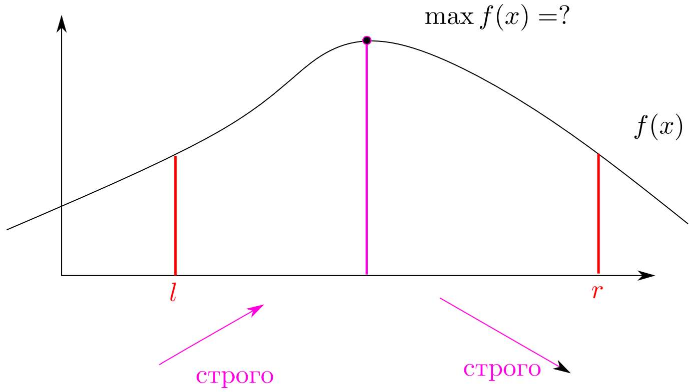
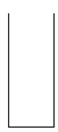


Рис. 1.4: tri

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 обоих 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

$o_1, o_2, \ldots, o_k$ – операции, проделанные в таком порядке

$T(o_i)$ – время работы операции

$\widetilde{T}(o_i)$ – амортизированные время работы (выбирается нами)

Амортизированное время хорошее, если $\sum \widetilde{T}(o_i) \geqslant \sum T(0_i)$

Пусть мы хотим $\widetilde{T}(o_i) \leqslant c \implies \sum\limits_k T(o_i) \leqslant c \cdot k$

Пусть мы делаем k пушей

$\sum T \leqslant k + \underbrace{2^p}_{\leqslant 2k} \leqslant 3k \quad \widetilde{T}(push) = 3$

1.12.1 Метод потенциалов

:

Φ – потенциал (выбираемый опять же нами)

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

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

$\sum \widetilde{T} = \sum T + \sum \underbrace{\Delta\Phi}_{=\Phi_{\text{к}=\Phi_0}} \geqslant \sum T$

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

$\Phi =$ (число элементов правой половине массива)$\cdot 2$

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

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

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 запушить, а потом много запопить, 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 копим для сужения.

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

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

1.13 Фибоначчева куча

1.13.1 Биномиальная куча

Куча: $add(x), remove\_min(x)$

Двоичная куча: обе операции за логарифм

Добавим третью операцию: $merge(H_1, H_2)$ – склеивает две кучи H_1 и H_2 в одну

Биномиальное дерево

B_0 – дерево из одной вершины

B_1 – одна вершина, к которой подвешена друга

B_2 –

1.14 Динамическое программирование

$F_1 =, F_2 = 1 \quad F_n = F_{n-1} + F_{n-2}$

$F_n = ?$

```
f(n):
    if n <= 2:
        return 1
    else:
        return f(n-1)+f(n-2)
```

$T(f(n)) = ?$

$T(n) = 1 + T(n-1) + T(n-2)$

$T(n) \approx F_n$

При вызове $f(10)$ $\quad f(8)$ считается 2 раза, $f(7)$ – три раза, и чем дальше, тем хуже

$res = [1..n]$

```
f(n):
    if res[n] != null:
        return res[n]
    if n <= 2:
        res[n] = 1
```

```
        else:
            res[n] = f(n-1) + f(n-2)
```

$T(n) = O(n)$

$res[1] = 1 \quad res[2] = 1$

```
    for i = 3 .. n:
        res[i] = res[i-1] + res[i-2]
    print(res[n])
```

Задача 3. Есть n кочек и зайчик.

Может прыгнуть на след. кочку, может через одну.

Нужно посчитать число способов попасть на последнюю клетку

Последняя клетка у всех путей – 6. В неё можно было попасть из пятой или из четвёртой.

$d = [1, 1, 2, 3, 5, 8]$

```
    d[1] = d[2] = 1
    for i = 3 .. n:
        d[i] = d[i-1] + d[i-2]
```

Хотим посчитать количеств комбинаторных объектов

Сколько разных векторов из 0 и 1 длины n, в которых нет двух 1 подряд.

Если в конце 0, то перед этим может быть что угодно, если 1, то перед ней стоит 0, а дальше снова что-угодно.

Если 0 в конце, то $d[n-1]$, а если 1, то $d[n-2]$

Теперь, если кролик может прыгать ещё и на 3 кочки. Сколько тогда способов будет? Просто добавить её одно слагаемое $d[i-3]$

Но придётся добавить $d[3] = 3$

А что если, он может прыгать до k клеток вперёд?

```
    d[1]=1
    for i = 2 .. n:
        for j = 1 .. k:
            if i-j >= 1:
                d[i] += d[i-j]
```

Пусть теперь за вставание на кочку нужно платить.

$\cos t = [0, 1, 3, 1, 2, 0]$

Все пути делятся на группы:

$\ldots \to 5 \to 6$

$\ldots \to 4 \to 6$

$d[i]$ – минимальный штраф, чтобы дойти до i

$d = [0, 1, 3, 2, 4, 2]$

```
d[0]=1
for i = 2 .. n:
    d[i] = min(d[i-1], d[i-2]) + cost[i]
```

```
d[1] = 0
for i = 2 .. n:
    d[i] = +inf
    for j = 1 .. k:
        if i-j >= 1:
            d[i] = min(d[i], d[i-j] + cost[i])
```

Восстановим путь: будем хранить ещё один массивчик с предыдущим числом для каждой кочки.

Другая задача: то же самое, только котик в табличке. вместо прямой, собирает баллы по алгосам. Нужно собрать как можно больше.

$d[i, j]$ – максимум до клетки (i, j)

В клетку (i, j) мы придём либо слева, либо сверху.

```
for i = 1 .. n:
    for j = 1 .. n:
        if i = j = 1:
            d[i,j] = 0
        else:
            d[i,j] = -inf
            if j > 1:
                d[i,j] = max(d[i,j], d[i,j-1] + cost[i,j])
            if i > 1:
                d[i,j] = max(d[i,j], d[i-1,j] + cpst[i,j])
```

Вернёмся к самой первой задаче про кролика. Но, он не может тормозить и следующий прыжок должен быть хотя бы на столько же.

$d[i, j]$ – число способов попасть в i с последним прыжком $\leqslant j$

Так никто не пишет:

```
for j = 1 .. n:
    d[1,j] = 1
for i = 2 .. n:
    for j = 2 .. i:
        for k = 1 .. j:
            d[i,j] += d[i-k,k]
```

Так по-человечески:

```
for j = 1 .. n:
    d[1,j] = 1
for i = 2 .. n:
    for j = 1.. n:
        if j > 1:
            d[i,j] += d[i,j-1]
        if i-j >= 1:
            d[i,j] += d[i-j,j]
```

1.14.1 Реальное применение динамического программирования

diff

```
1    res = 0
2    for (int i = 1; i<n; i++) {
3        if (a[i] < res) {
4            res = a[i]
5            break
6        }
7    }
```

А потом кто-то исправляет

```
1    res = INT_MAX
2    for (int i = 0; i < n; i++) {
3        if (a[i] < res) {
4            res = a[i]
5        }
6    }
```

diff:

```
1: -res = 0
   +res = INT_MAX
2: - ..
```

Минимально количество изменений, чтобы из A получить B

```
s = котик
t = коржик

котик
ко(т->р)(+ж)ик
коржик
```

хочется получить вторую из первой:

1. добавить символ
2. удалить символ

3. заменить символ

Минимальное количество действий называется расстоянием Левенштейна.

```
s = .. .. x
t = .. .. x
s->t -- s[:-1] -> t[:-1]
Откидываем общую буквц в конце (D[i-1,j-1])

s = .. .. x
t = .. .. y
1) change(x,y) и снова сводим к меньшему (1 + D[i-1,j-1])
2) Если мы сделали remove(x), тогда из s[:-1] -> t (1 + D[i-1,j])
3) add(y), тогда s -> t[:-1] (1 + D[i,j-1])
```

$D[i,j]$ – минимальное количество действий, чтобы из $s[0..i-1]$ получить $t[0..j-1]$

Пример. `s = котик`
`t = коржик`

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	0	1	2	3	4
2	2	1	0	1	2	3
3	3	2	1	1	2	3
4	4	3	2	2	2	3
5	5	4	3	3	2	3
6	6	5	4	4	3	2

$D[n,m] = 2$

Код:

```
for i = 0 .. |s|
    for j = 0 .. |t|
        if i = 0 or j = 0:
            D[i,j] = i+j
            continue
        if s[i-1] = t[j-1]:
            D[i,j] = D[i-1,j-1]
        else:
            D[i,j] = min(
                D[i-1,j-1] + 1,
                D[i-1, j]  + 1,
                D[i, j-1]  + 1
            )
```

Так мы считаем само расстояние. Давайте для каждого состояния динамики помечать предыдущие состояние динамики. Так мы можем восстановить путь

На самом деле всё делается немножко не так. Так – медленно $O(nm)$. Мысль: если мало изменений, от они находятся близко. Тогда можно хранить значения около главной диагонали. Можно хранить блок строки вокруг оптимума (мин. значения). Блок ширины X – $O(nX)$

Перейлём к другой задаче

Пусть у нас есть текст со словами разной ширины, их надо красиво акууратно разместить на листчке (допустим, пишем свой текстовый редактор)

1. Пихаем слова, пока пихается. Спецэффекты: длинные слова, которые могут некрасиво переноситс\u044f, образуя здоровые конские пробелы. Здоровый конский пробел – это некрасиво.

2. Давайте введём метрику-штраф. Допустим штраф за пробел размера x – x^2

Пусть ширина листочка L

$bad(l, r)$ – штраф, если слова с l по r мы разместим в одной строчке

$bad(l, r) = \left(L - \sum_{i=l}^{r-1} w_i\right)^2$

$bad(l, r) = \left(\frac{L - \sum w_i}{r-l-1}\right) \cdot (r - l - 1) \# ifr - l - 1 > 1$

$D[n]$ – $\min \sum bad$ для слов $0 \ldots n-1$

```
for n = 1 .. n:
    for j = 0 ... i-1:
        if sum(w(j..i-1)) > L:
            break
        cost = bad(k,i) + D[j]
        D[i] = min(D[i],cost)
```

Задача 4. Есть RLE кодирование

2A -> AA

$3(2AB) \to AABAABAAB$

Нужно по строчке получить компактное представление её в RLE

```
s = AAABAABAABABB

A2(2AB)A2B

Если первая буква записана как буква s -> 1 + s[:-1] (1 + D[i+1,j])
```

```
Если повторяющаяся строка длины x

Тогда будет k(s[0..x-1]) s[kx..n-1] (3 + D[i,i+x] + D[i+kx,nj]
```

Пусть $D[i, j]$ – самое короткое представление строки $s[i..j-1]$

Когда восстанавливаем состояние, то находим не один путь, а дерево.

1.15 Алгосики

Задача 5. n предметов, веса w_i, стоимости c_i, рюкзак S

$\sum w_i \leqslant S \quad \sum c_i \to \max$

NP-полная

$c_i = w_i \quad \sum w_i \leqslant S \quad \sum w_i \to \max$

$\sum w_i = S$

Доказательство. 1. w_i, S – маленькие, целые ($S \approx 10^6$)

$d[i, j]$ – можно ли набрать сумму j из предметов $(0 \dots i-1)$

(a) Не берём $i-1 \quad d[i-1, j]$

(b) Берём $i-1 \quad d[i-1, j-w_{i-1}]$

База:

```
$d[0,0] = True$
for i = 1..n:
    for j = 0..S:
        d[i,j] = d[i-1,j]
        if j-w_{i-1} >= 0:
            d[i,j] |= d[i-1,j-w_{i-1}] # |= -- or-равно
```

Теперь сделаем $d[i, j]$ – $\max \sum c$, которую можно набрать, если $\sum w = j$ из предметов $(0, i-1)$

```
d[0,0] = 0, d[o,j] = infty
for i = 1..n:
    for j = 0..S:
        d[i,j] = d[i-1,j]
        if j-w_{i-1} >= 0:
            d[i,j] = max(d[i,j], d[i-1,j-w_{i-1}] + c_{i-1}) # |= -- o
```

n – очень маленькое (можем перебрать все 2^n – небольшое)

Переберём все $X \subseteq \{0 \ldots n-1\}$

Подмножество $\leftrightarrow$ числа.

$n = 5 \quad x = \{0, 2, 3\} \quad 01101 = 13$

Подмножества $\{0 \ldots n-1\} \leftarrow$ числа $0 \ldots 2^n - 1$

$x \cup y = x|y \quad x \cap y = x\&y$

$x \setminus y \quad x\&(\ y)$

$\{i\} \quad 1 << i$

$i \in X \quad (x\&(1 << i)) > 0$

```
for x = 0 .. 2^n-1:
    sw=0, sc=0
    for i = 0 .. n-1:
        if x&(1<<i) > 0:
            sw += w(i)
            sc += c(i)
    if sw <= S:
        res = max(res,sc)
O(2^n * n)
```

Оптимизации:

- Meet in the middle.

  Поделим предметы на левые и правые. Получим $2^{\frac{n}{2}}$ способов с каждой стороны

  $sw[x] + sw[y] \leqslant S \quad sc[x] + sx[y] \to \max$

  ```
  for x = 0 .. 2^{n-1}-1
      sq[y] <= S*sq[x]
      sc[y] -> max (бин поиск)
  ```

- Мультирюкзак. Положить все предметы в минимальное количество рюкзак.

  $d[x]$ – минимальное число рюкзаков, чтобы положить подмножество x

  ```
  for x = 0 .. 2^{n-1}:
      for y \in x:
          if sum w[y] <= S:
              d[x] = min(dx, 1 + d[x-y])
  ```

```
O(4^n)
```

Каждый предмет:

1. В X и в Y
2. В X, но не в Y
3. Ни там, ни там

Так получается 3^n вариантов.

$d[x] = (A, B)$ A – число рюкзаков, B – заполненность последнего рюкзака.

$(A_1, B_1) < (A_2, B_2)$, если $A_1 < A_2$ or $A_1 = A_2 \& B_1 < B_2$

$\blacksquare$