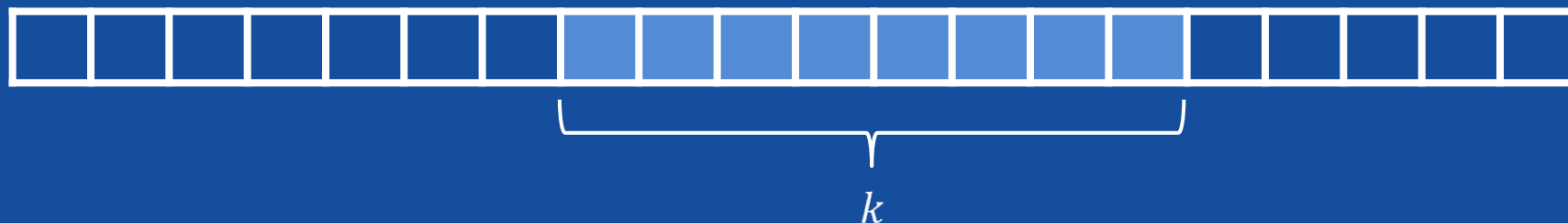


Структуры данных для выполнения интервальных запросов



*С. А. Соболев — магистр математики и информационных технологий
Е.П. Соболевская — доцент кафедры ДМА ФПМИ БГУ*

Пусть в памяти хранится последовательность из n элементов.

Нужно уметь выполнять какую-либо операцию над k последовательными элементами сразу (например, суммировать, находить минимум/максимум).

Такие запросы называют ***интервальными***, потому что они затрагивают целый интервал значений.

- Путаница
 - Промежуток
 - Интервал
 - Отрезок (сегмент)
- Будем использовать в основном целочисленные полуинтервалы (левый конец включён, правый не включён):
$$[l, r) = \{l, l + 1, l + 2, \dots, r - 1\}$$
- В языках программирования часто правый конец диапазона не включается (C++, Python)

Если элементы последовательности не изменяются (не предусмотрено выполнение операции модификации элемента), то в названии задачи фигурирует слово **статическая** (англ. **static**), иначе – **динамическая** (англ. **dynamic**).

Если сначала все интервальные запросы поступили (после чего они все могли быть проанализированы), а только потом формируются ответы на них, то говорят про **офлайн** (англ. **off line**) версию задачи.

Если же поступает запрос, сразу даётся на него ответ и только после ответа на предыдущий запрос идёт следующий запрос, то говорят про **онлайн** (англ. **online**) версию задачи.

Мы будем в нашей лекции рассматривать online версию задачи.

запрос-1	
запрос-2	
...	
запрос-k	ответ-1
	ответ-2
	...
	ответ-k

запрос-1	ответ-1
запрос-2	ответ-2
...	
запрос-k	ответ-k

Выполнить один любой интервальный запрос можно, конечно, «в лоб» циклом по k элементам интервала — это $\Omega(k)$.

Можно ли это сделать
быстрее?

Покажем (на примере задач о сумме и минимуме на интервале), что с помощью специальных структур:

- ❑ **корневая эвристика;**
- ❑ **дерево отрезков;**
- ❑ **разрежённая таблица;**

выполнить интервальный запрос можно быстрее.

При этом сначала над данными выполняют **препроцессинг (предподсчёт)** – предварительная подготовка, которая в последующем позволит эффективно обрабатывать запросы. Очевидно, что время предподсчёта должно быть не больше, чем суммарное время ответа на все запросы.

RSQ — Range Sum Query (запрос суммы на отрезке)

- Задана последовательность из n чисел:

$$A = [a_0, a_1, a_2, \dots, a_{n-1}]$$

- Поступают запросы двух типов:

✓ запрос модификации **Add(i, x)** — прибавить к i -му элементу число x ;

✓ запрос суммы **FindSum(l, r)** — вычислить сумму на $[l, r)$

$r-1$



Задача RSQ. Обычный массив

Используем **обычный массив**, в котором размещены элементы:

Add(3, 2)

Add(2, -5)

FindSum(1, 5)

$$1 + 0 + 11 + 5 = 17$$

2	1	5 0	9 11	5	2	6
0	1	2	3	4	5	6

$n = 7$

Время

работы:

модификация	
сумма	

хотелось бы
быстрее

Выполним предподсчёт

- Введём понятие частичной суммы, или суммы на префиксе:

$$s_k = \sum_{i=0}^{k-1} a_i = a_0 + a_1 + \dots + a_{k-1}$$

По исходной последовательности A массив S строится за $O(n)$, используя следующее рекуррентное соотношение:

$$\begin{aligned} s_0 &:= 0, \\ s_i &:= s_{i-1} + a_{i-1}, \quad i = 1, \dots, n \end{aligned}$$

Сумма на $[l, r)$ равна $s_r - s_l$

	}		}					
A	2	1	9	7	5	2	6	
	0	1	<i>l=2</i>	3	4	5	<i>r=6</i>	7
S	0	2	<u>3</u>	12	19	24	<u>26</u>	32

$$\text{FindSum}(2, 6) = s_6 - s_2 \\ = 26 - 3 = 23$$

Время на запрос
суммы: $O(1)$



Add(i, x)

A	2	1	9 10	7	5	2	6	
	0	1	2	3	4	5	6	7
S	0	2	3	12 13	19 20	24 25	26 27	32 33

Add(2, 1)

Время на запрос
модификации:

$O(n)$



Задача RSQ. Префиксные суммы

A	2	1	9	7	5	2	6	
	0	1	2	3	4	5	6	7
S	0	2	3	12	19	24	26	32

Время на подсчет	
Время на запрос модификации	
Время на запрос суммы	
Объем дополнительной памяти	

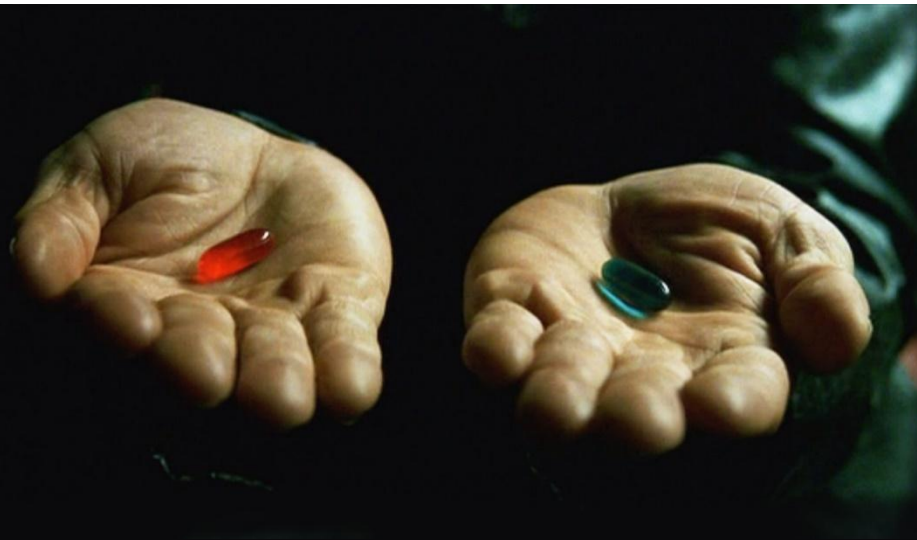
можно $O(1)$, если использовать память, выделенную под массив A



	Обычны й массив	Префиксны е суммы
Время на предподсчёт		$O(n)$
Время на запрос модификаци и	$O(1)$	$O(n)$
	$O(n)$	$O(1)$
Время на запрос суммы		$O(n)$
Объем дополнитель		

$O(1)$, если хранить при предподсчёте массив префиксных сумм на месте исходного массива

- Одна операция быстрая, вторая медленная
- Нужно компромиссное решение



Модификация
Сумма

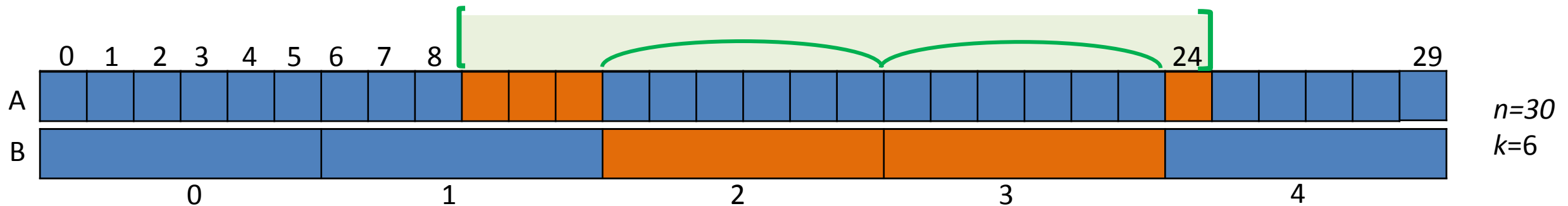
$O(1)$
 $O(n)$

$O(n)$
 $O(1)$



Выполним

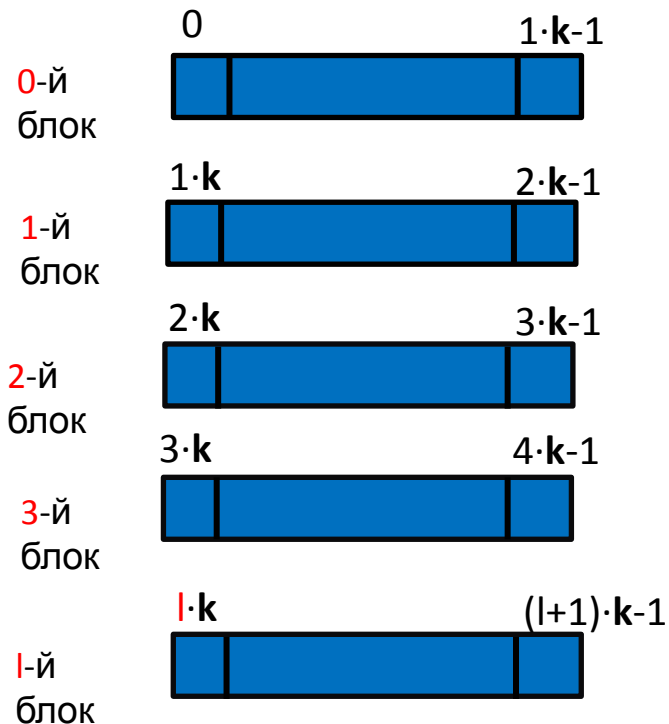
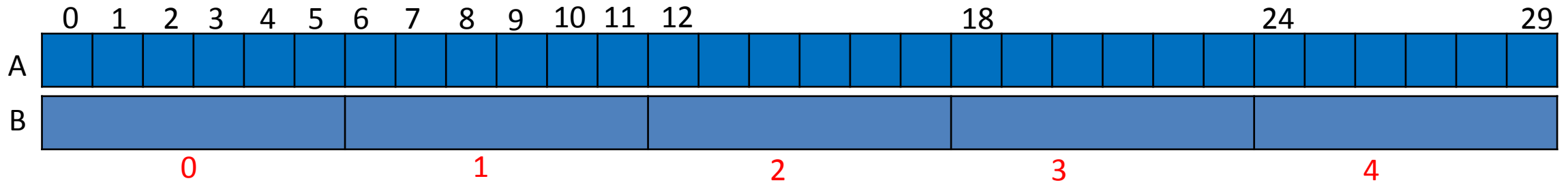
- предподсчёт
- Должно посчитать заранее суммы для блоков определённого размера k (последний блок может быть меньше)
- При суммировании в цикле можно будет «перепрыгивать» блоки



Для этого кроме исходного массива A , создадим массив B размера $\lceil n/k \rceil$ для хранения сумм по блокам.



Удобно нумеровать блоки с нуля (на рис. размер блока $k=6$)



Пусть i индекс элемента в массиве A , в какой блок N_b он попадёт?

$$N_b \cdot k \leq i < (N_b + 1) \cdot k$$

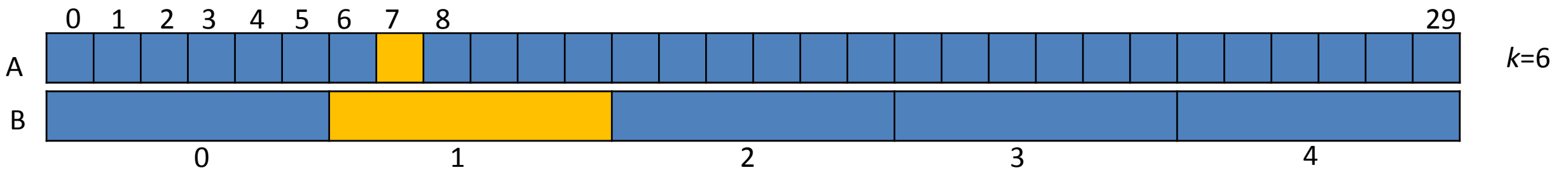
$$\frac{i}{k} - 1 < N_b \leq \frac{i}{k}$$

$$\frac{i}{k} - 1 < \left\lfloor \frac{i}{k} \right\rfloor \leq \frac{i}{k}$$

$$x - 1 < [x] \leq x \leq [x] < x + 1$$

Элемент массива A с индексом i попадает в блок $b_{\lfloor i/k \rfloor}$.

При **модификации** нужно изменить a_i и $b_{\lfloor i/k \rfloor}$.



```
def Add(self, i, x):  
    self.a[i] += x  
    self.b[i // self.k] += x
```

Время выполнения операции модификации - $O(1)$.



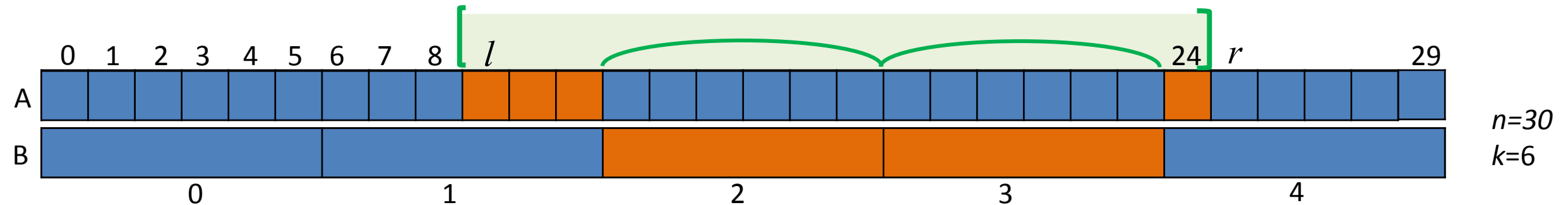
При суммировании на $[l, r)$:

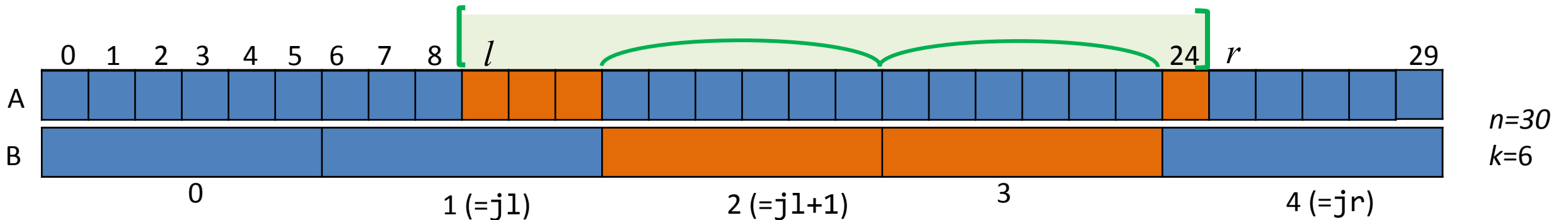
Определяем номера блоков, к которым относятся l и r .

Суммируем от l до границы блока за $O(k)$ по массиву A .

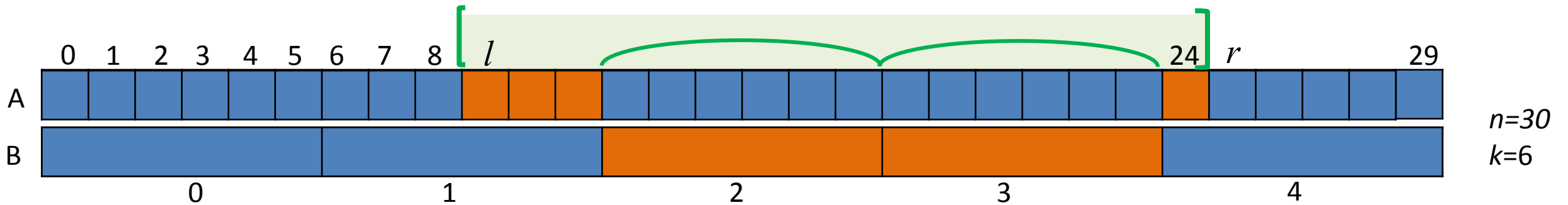
Суммируем блоки за $O(n/k)$ по массиву B .

Суммируем от границы блока до r за $O(k)$ по массиву A .





```
def FindSum(self, l, r):  
    j1 = l // self.k  
    jr = r // self.k  
    if j1 == jr: # same block  
        return sum(self.a[l:r])  
    else:  
        return (  
            sum(self.a[l:((j1 + 1) * self.k)]) +  
            sum(self.b[(j1 + 1):jr]) +  
            sum(self.a[(jr * self.k):r])  
        )
```



При **суммировании** на $[l, r)$ общее время:

$$O(k) + O\left(\frac{n}{k}\right) + O(k) = O\left(k + \frac{n}{k}\right).$$

Каким лучше выбрать размер блока k ?



Оптимальный размер блока

Исследуем на экстремум функцию:

$$f(k) = k + \frac{n}{k}$$

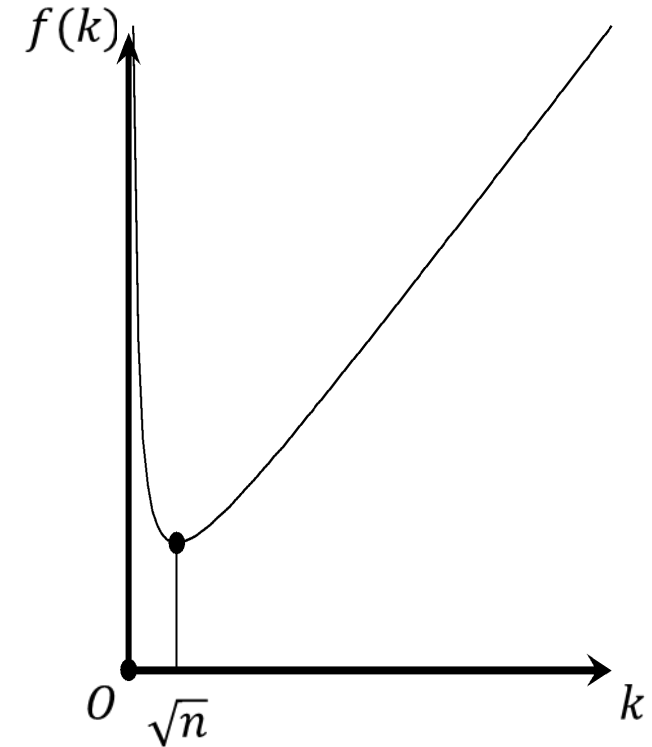
$$f'(k) = 1 - \frac{n}{k^2}$$

$$f'(k) = 0 \Leftrightarrow k = \sqrt{n}$$

Таким образом, оптимально разбивать на $\approx \sqrt{n}$ блоков по $\approx \sqrt{n}$ элементов.

Операция $\text{FindSum}(l, r)$ выполняется за время $O\left(k + \frac{n}{k}\right) = O(\sqrt{n})$.

Приём называется ***sqrt-декомпозицией*** или ***корневой эвристикой***.





Пример реализации

```
class Summator:
    def __init__(self, a):
        self.a = a
        self.k = floor(sqrt(len(a)))

        self.b = []
        for i in range(0, len(a), self.k):
            bsum = sum(self.a[i:(i + self.k)])
            self.b.append(bsum)

    def Add(self, i, x):
        self.a[i] += x
        self.b[i // self.k] += x
```

```
def FindSum(self, l, r):
    jl = l // self.k
    jr = r // self.k
    if jl == jr: # same block
        return sum(self.a[l:r])
    else:
        return (
            sum(self.a[l:((jl + 1) * self.k)]) +
            sum(self.b[(jl + 1):jr]) +
            sum(self.a[(jr * self.k):r]
        )
```



	Обычный массив	Префиксные суммы	Sqrt-декомпозиция
Время на подсчет		$O(n)$	$O(n)$
Время на запрос модификации	$O(1)$	$O(n)$	$O(1)$
	$O(n)$	$O(1)$	$O(\sqrt{n})$
Время на запрос суммы		$O(n)$	$O(\sqrt{n})$
Объем дополнительной памяти			

Дальнейшим развитием идеи разбиения на блоки будет следующее:

- блоки разной длины организуем в виде дерева;
- в каждой вершине дерева содержится сумма элементов массива, индексы которых принадлежат соответствующему отрезку;
- корень соответствует всему массиву $[0, n)$ (т.е. в корне дерева хранится общая сумма всех элементов массива);
- у вершины, соответствующей $[t_l, t_r)$, два сына:

$$[t_l, m) \text{ и } [m, t_r), \text{ где } m = \left\lfloor \frac{t_l + t_r}{2} \right\rfloor$$

Такую структуру будем называть **деревом отрезков**

Терминология не устоялась, под *segment tree* и *interval tree* часто понимают другие структуры

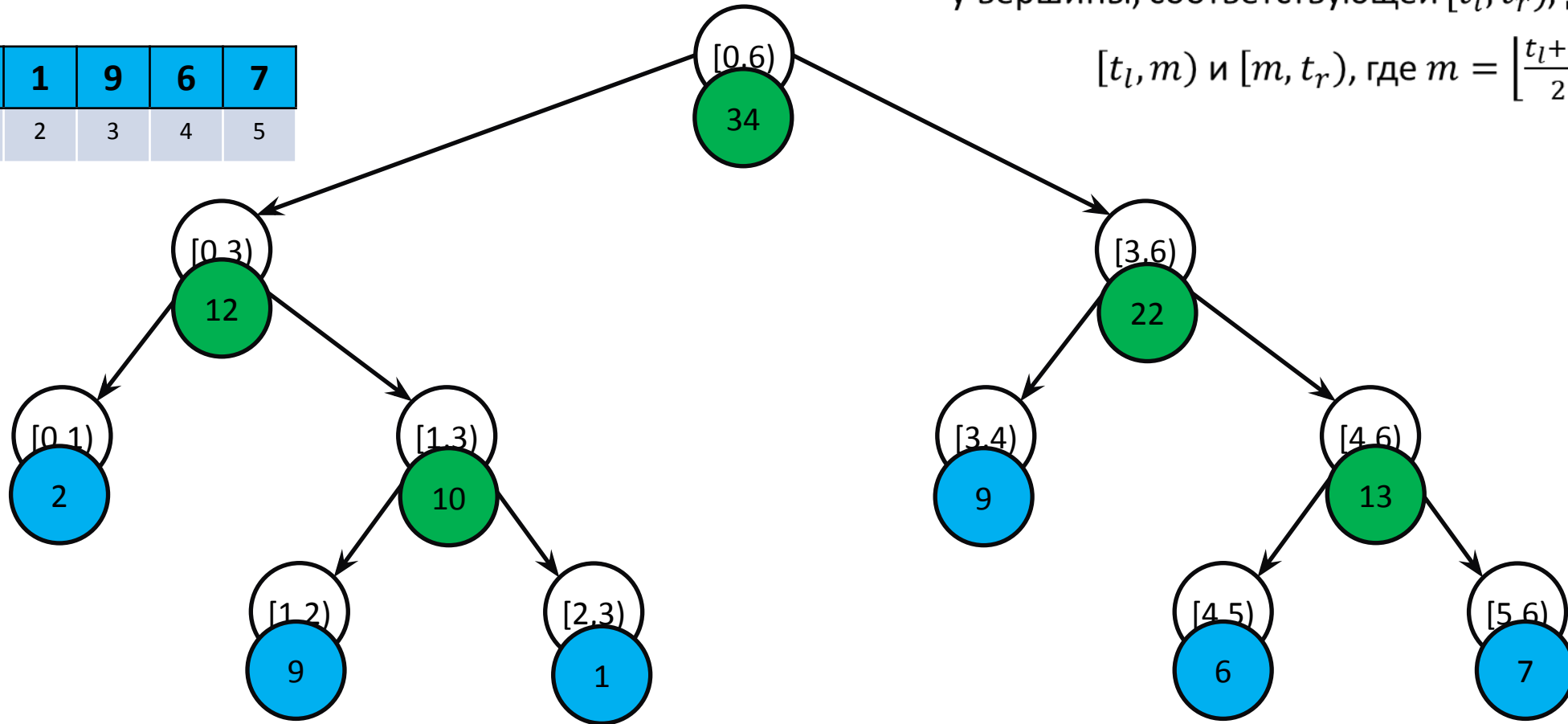


Пример дерева отрезков для задачи RSQ

$n = 6$

a

2	9	1	9	6	7
0	1	2	3	4	5





Теорема

Общее число вершин равно $2n - 1$.

Доказательство.

По индукции.

Если $n = 1$, то есть одна вершина, $2n - 1 = 1$ — верно.

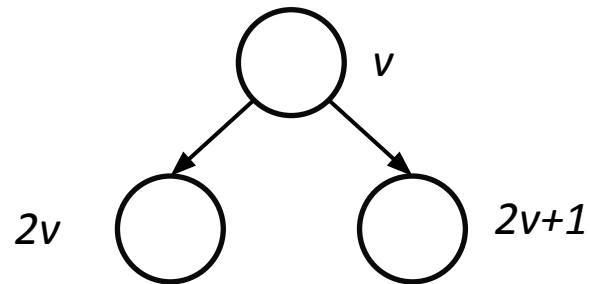
Если $n = n_1 + n_2$, то строим два дерева и добавляем ещё одну вершину:

$$(2n_1 - 1) + (2n_2 - 1) + 1 = 2(n_1 + n_2) - 2 + 1 = 2n - 1$$

Таким образом, в дереве n листьев и $n - 1$ внутренняя вершина.

Высота дерева $O(\log n)$.

- Нет необходимости хранить указатели/ссылки
- Для хранения вершин дерева будем использовать **МАССИВ** (по аналогии с тем, как была реализована на массиве бинарная куча)
- Индексация в массиве с единицы (1 — корень)



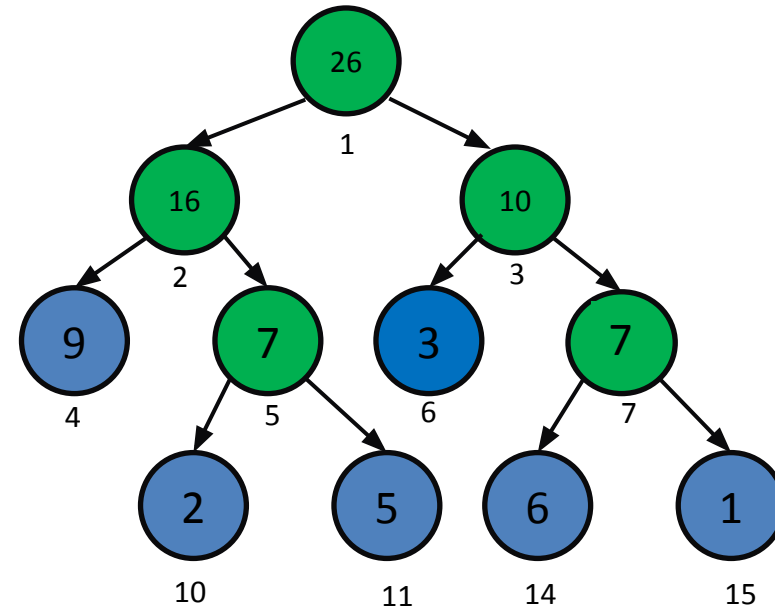


Дерево отрезков. Хранение

n=6

a

9	2	5	3	6	1
0	1	2	3	4	5
26					
16			10		
9	7		3	7	
	2	5		6	1



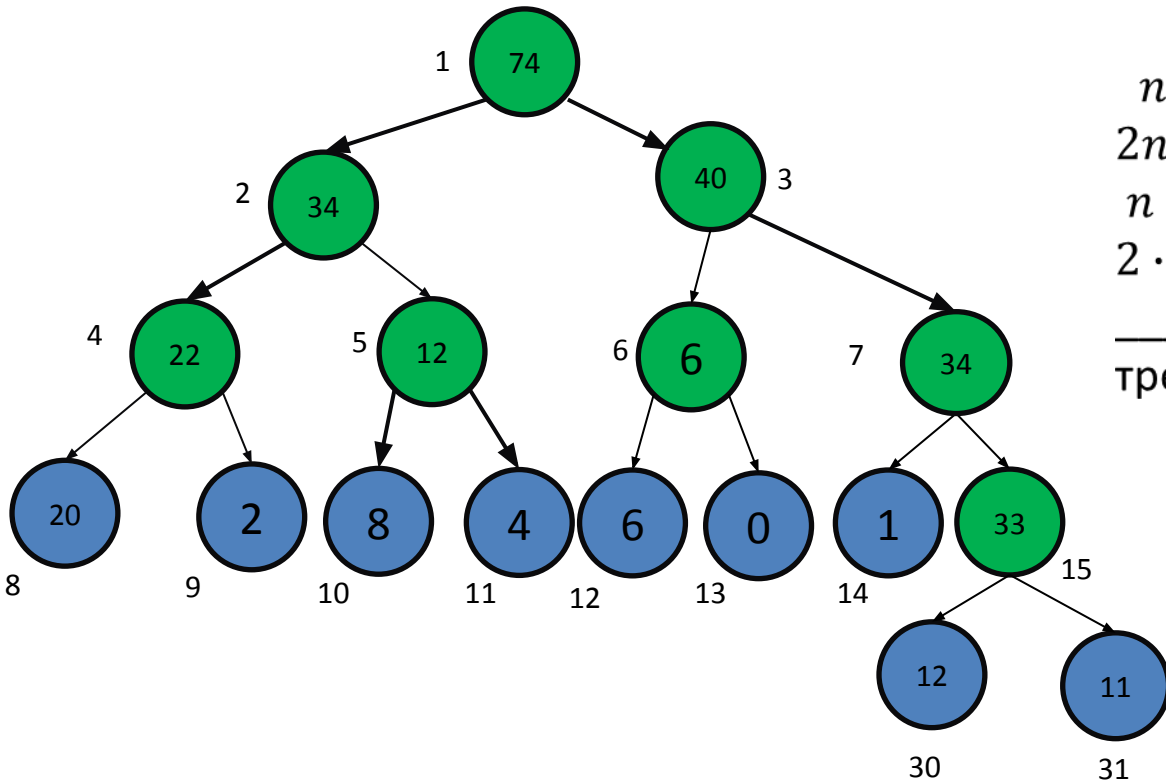
t

26	16	10	9	7	3	7	—	—	2	5	—	—	6	1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Количество реально занятых ячеек равно $2n-1=11$.



Дерево отрезков. Хранение



- n число листьев в дереве (число элементов в массиве)
- $2n - 1$ число реально занятых ячеек массива t (по теореме)
- $n - 2$ число листьев на предпоследнем слое
- $2 \cdot (n - 2)$ число свободных элементов в массиве t

требуемая память для массива t :

$$\underbrace{(2n - 1)}_{\text{используемая память}} + \underbrace{2 \cdot (n - 2)}_{\text{неиспользуемая память}} = 4 \cdot n - 5$$

t	74	34	40	22	12	6	34	20	2	8	4	6	0	1	33	-	-	-	-	-	-	12	11
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		...			29	30	31



В том случае, когда n не является степенью 2, дерево не является полным, из-за чего в массиве могут быть неиспользуемые ячейки.

Чтобы места гарантированно хватило, можно выставить размер массива $4n$.



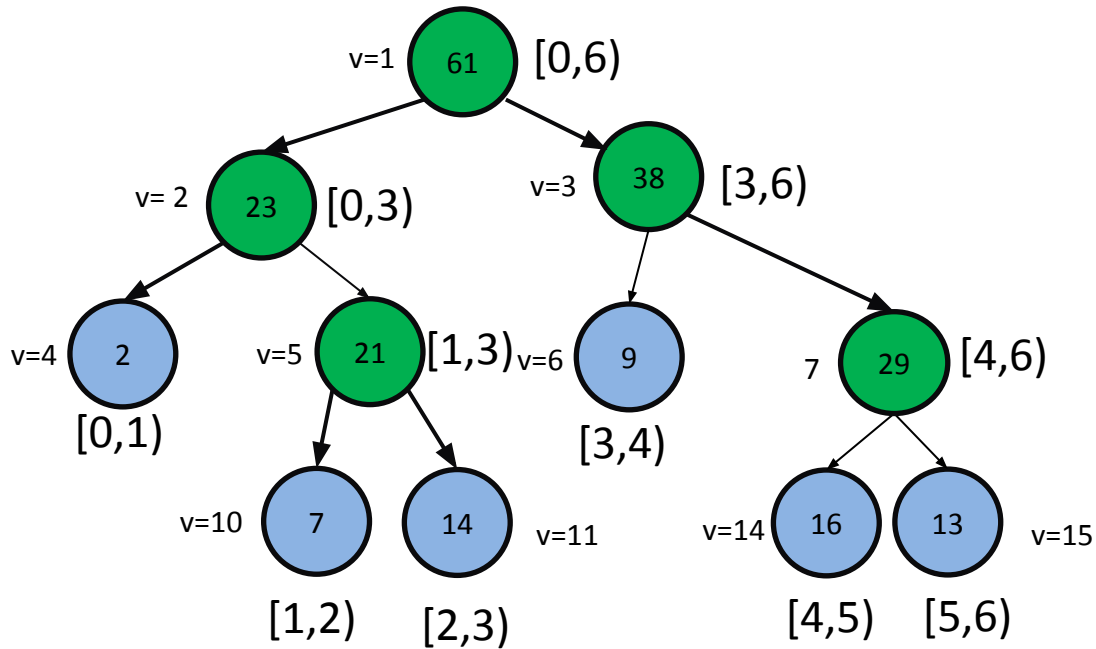


```
def DoBuild(a, t, v, tl, tr):  
    if tr - tl == 1:  
        t[v] = a[tl]  
    else:  
        m = (tl + tr) // 2  
        DoBuild(a, t, v=2*v, tl=tl, tr=m)  
        DoBuild(a, t, v=2*v+1, tl=m, tr=tr)  
        t[v] = t[2*v] + t[2*v+1]
```

```
def Build(a, n):  
    t = [0] * 4*n  
    DoBuild(a, t, v=1, tl=0, tr=n)  
    return t
```




Дерево отрезков. Построение



```
def Build(a, n):
    t = [0] * 4*n
    DoBuild(a, t, v=1, tl=0, tr=n)
    return t
```

```
def DoBuild(a, t, v, tl, tr):
    if tr - tl == 1:
        t[v] = a[tl]
    else:
        m = (tl + tr) // 2
        DoBuild(a, t, v=2*v, tl=tl, tr=m)
        DoBuild(a, t, v=2*v+1, tl=m, tr=tr)
        t[v] = t[2*v] + t[2*v+1]
```

a

2	7	14	9	16	13
0001	211	32	34	45	56

n=6

t

61	23	38	2	21	9	29			7	14			16	13					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20



Add(i, x)

```
def DoAdd(t, v, tl, tr, i, x):  
    if tr - tl == 1:  
        t[v] += x  
        return  
    m = (tl + tr) // 2  
    if i < m:  
        DoAdd(t, v=2*v, tl=tl, tr=m, i=i, x=x)  
    else:  
        DoAdd(t, v=2*v+1, tl=m, tr=tr, i=i, x=x)  
    t[v] = t[2*v] + t[2*v+1]
```

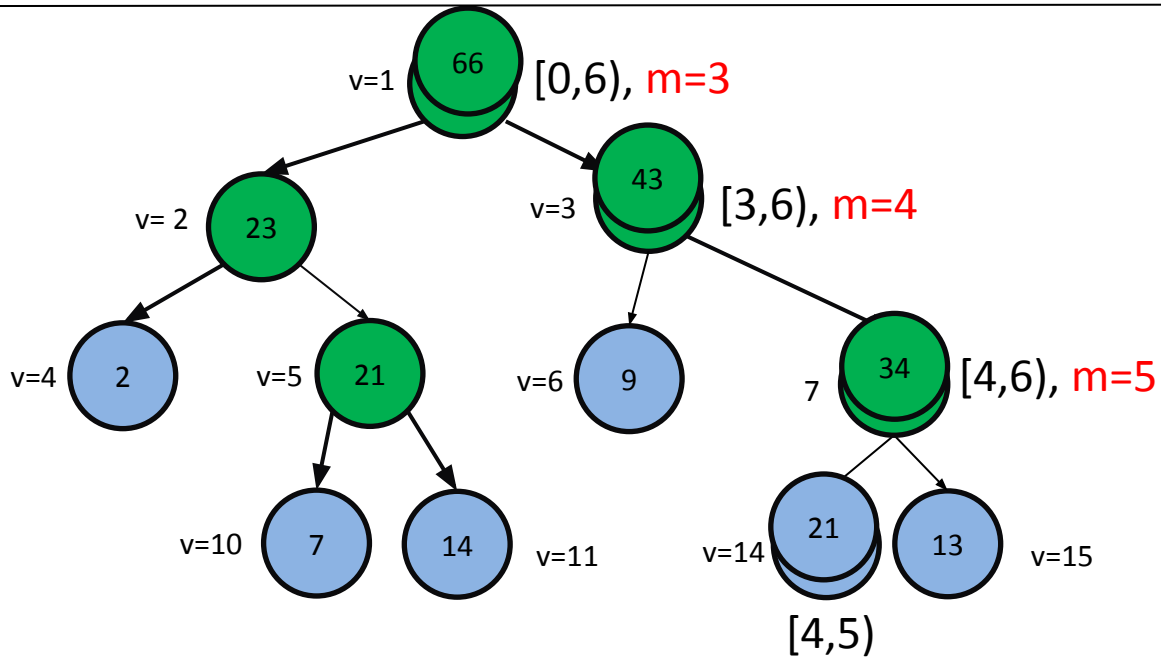
```
def Add(t, n, i, x):  
    DoAdd(t, v=1, tl=0, tr=n, i=i, x=x)
```



Дерево отрезков. Модификация

Add(4,
i=4, x=5

	0001	211	32	34	45	56
a	2	7	14	9	16	13
					21	



```
def Add(t, n, i, x):
    DoAdd(t, v=1, tl=0, tr=n, i=i, x=x)
```

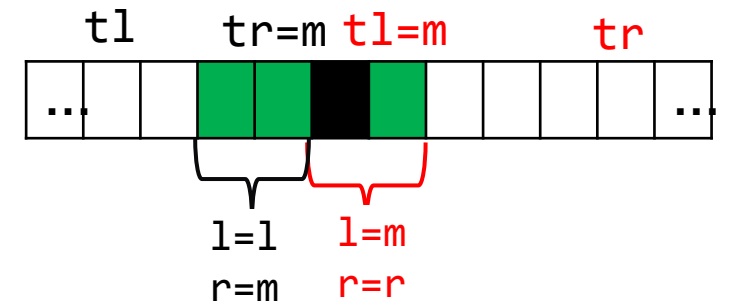
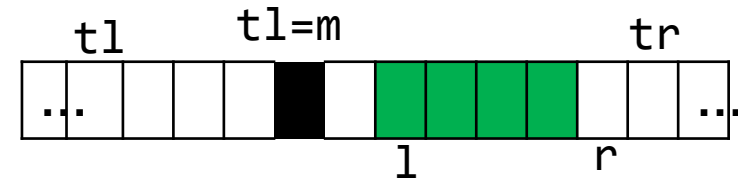
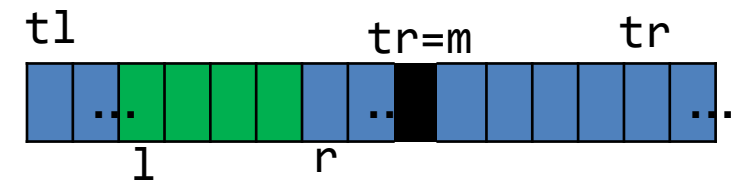
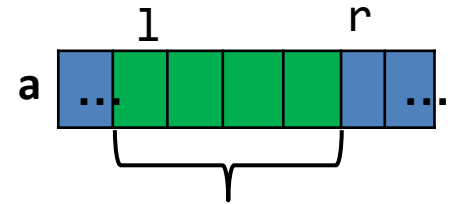
```
def DoAdd(t, v, tl, tr, i, x):
    if tr - tl == 1:
        t[v] += x
        return
    m = (tl + tr) // 2
    if i < m:
        DoAdd(t, v=2*v, tl=tl, tr=m, i=i, x=x)
    else:
        DoAdd(t, v=2*v+1, tl=m, tr=tr, i=i, x=x)
    t[v] = t[2*v] + t[2*v+1]
```

66	43	34	21																
61	23	38	2	21	9	29			7	14			16	13					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20



Дерево отрезков. Сумма

```
def DoFindSum(t, v, tl, tr, l, r):  
    if l == tl and r == tr:  
        return t[v]  
    m = (tl + tr) // 2  
    if r <= m:  
        return DoFindSum(t, v=2*v, tl=tl, tr=m, l=l, r=r)  
    if m <= l:  
        return DoFindSum(t, v=2*v+1, tl=m, tr=tr, l=l, r=r)  
    return (  
        DoFindSum(t, v=2*v, tl=tl, tr=m, l=l, r=m) +  
        DoFindSum(t, v=2*v+1, tl=m, tr=tr, l=m, r=r)  
    )
```





Дерево отрезков. Сумма. Пример

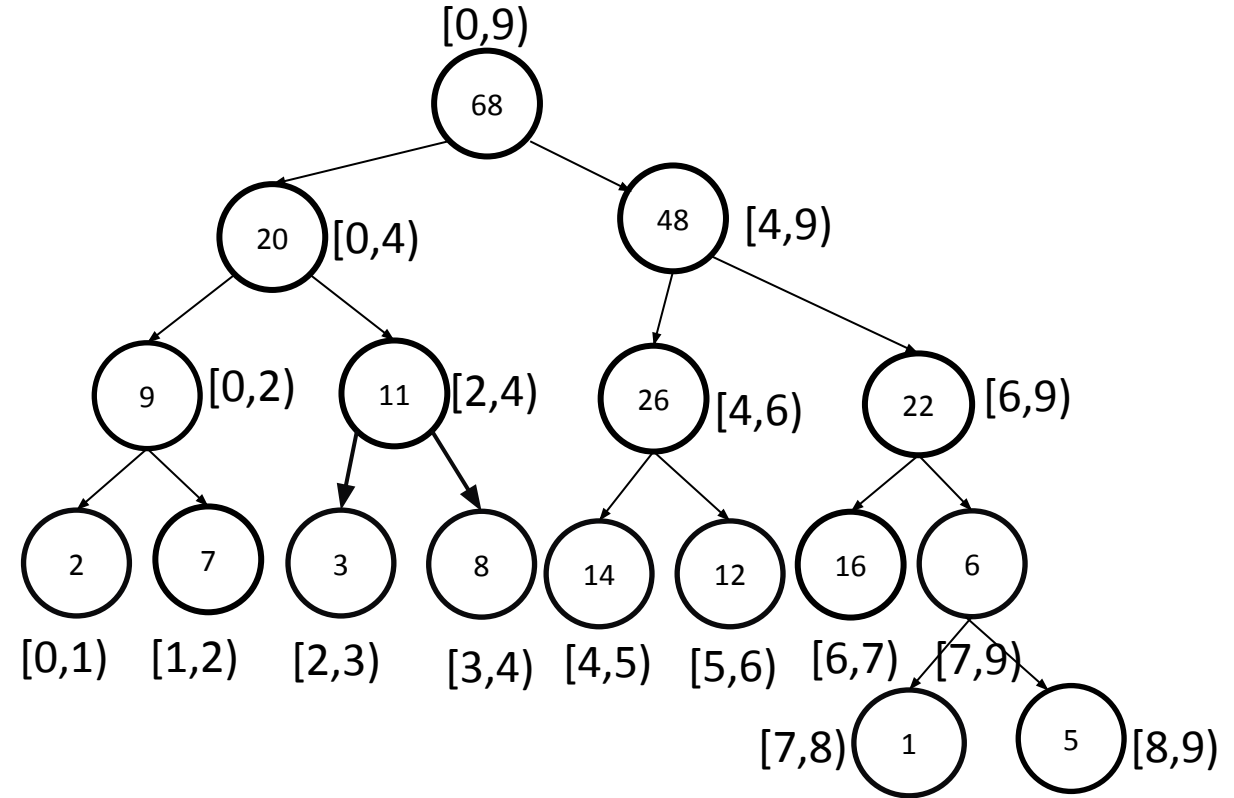
Найти сумму на отрезке

$[1,7)$

	1	2	3	4	5	6	7	8	
a	2	7	3	8	14	12	16	1	5
	2	7	3	8	14	12	16	1	5
	2	7	3	8	14	12	16	1	5
		7					16		

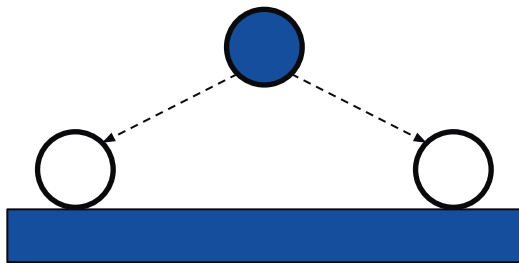
n=9

Сумма
на
отрезке $[1,7)$:
 $7 + 11 + 26 + 16 = 60$

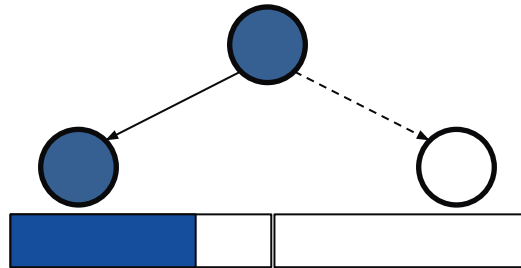


t	68	20	48	9	11	26	22	2	7	3	8	14	12	16	6	-	-	...	-	-	1	5
	[0,9)	[0,4)	[4,9)	[0,2)	[2,4)	[4,6)	[6,9)	[0,1)	[1,2)	[2,3)	[3,4)	[4,5)	[5,6)	[6,7)	[7,9)						[7,8)	[8,9)
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	28	29	30	31

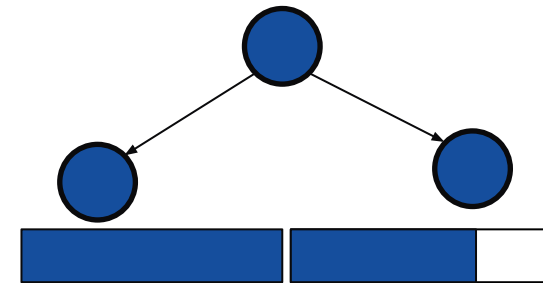
- При вычислении суммы рекурсия иногда уходит сразу в обе ветви
- Нужно доказать, что время работы — $O(\log n)$, не $O(n)$
- Пусть интервал имеет вид $[0, r)$
- Возможна одна из трёх ситуаций:



остановка
а



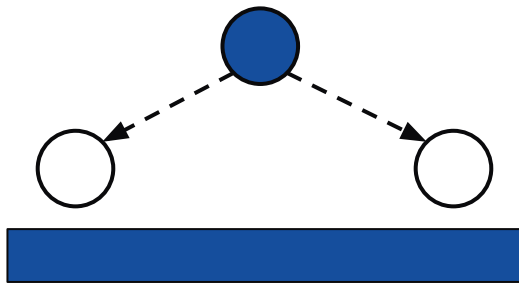
спуск только
влево



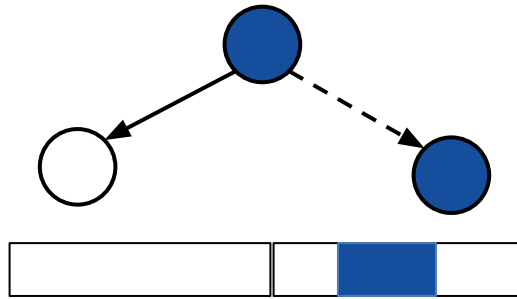
спуск влево и вправо,
но на следующем шаге левая
рекурсия сразу завершится

- Видим, что на каждом уровне активно работает только одна ветвь рекурсии

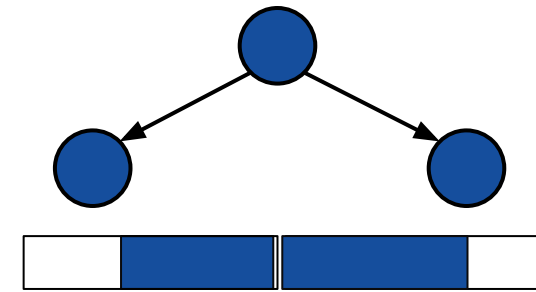
- Пусть теперь интервал $[l, r)$ произвольный
- Возможна одна из трёх ситуаций



остановк
а



спуск только
вправо
(или только влево)



спуск в обе
стороны

- В третьем случае задача сводится к ранее рассмотренной (после разделения на каждом уровне активно работают две рекурсии)

- Нерекурсивная реализация быстрее работает на практике
- Можно добавить поддержку операций на интервале:
 - ✓ $\text{Add}(l, r, x)$ — прибавить x к каждому элементу $[l, r)$
 - ✓ $\text{Set}(l, r, x)$ — установить элементы $[l, r)$ в значение x



	Дерево отрезков
Время на подсчёт	$O(n)$
Время на запрос модификации	$O(\log n)$ $O(\log n)$
Время на запрос суммы	$O(\log n)$
Объём дополнительной памяти	$O(n)$



	Обыч ый массив	Префиксн ые суммы	Sqrt- декомпозици я	Дерево отрезков
Время на предподсчёт		$O(n)$	$O(n)$	$O(n)$
Время на запрос модификаци и	$O(1)$	$O(n)$	$O(1)$	$O(\log n)$
	$O(n)$	$O(1)$	$O(\sqrt{n})$	$O(\log n)$
Время на запрос суммы		$O(1)$	$O(\sqrt{n})$	$O(n)$
Объем дополнител				

Статическая online задача

RMQ — **R**ange **M**inimum **Q**uery

(запрос минимума на отрезке)

- Задана последовательность из n чисел:

$$A = [a_0, a_1, a_2, \dots, a_{n-1}].$$

Поступают запросы *минимума* **FindMin**(l, r) — найти минимум на полуинтервале $[l, r)$.

Запросов модификации нет.

Можно решать корневой декомпозицией или деревом отрезков.
Однако научимся отвечать на каждый запрос за $O(1)$.

Статическая задача RMQ

Сначала рассмотрим решение «в лоб»:

для всех пар (i, j) просчитаем минимум массива A на полуинтервале $[i, j]$ и занесём в таблицу T .

Так как каждый элемент таблицы по рекуррентному соотношению

$$t_{i,j} = \min\{t_{i,j-1}, a_{j-1}\}$$

может быть вычислен за константу, то время подсчёта:

$$\Theta(n^2).$$

	0	1	2	3	4	5
A	1	2	0	9	6	7

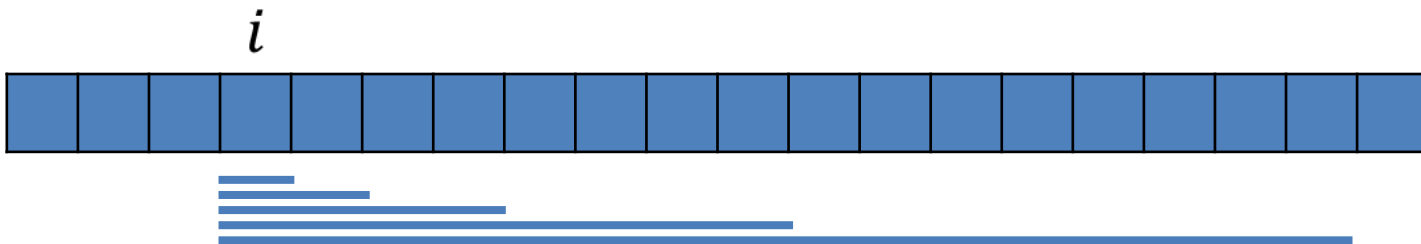
	0	1	2	3	4	5	6
T	0	1	1	0	0	0	0
	1		2	0	0	0	0
	2			0	0	0	0
	3				9	6	6
	4					6	6
	5						7

Затраты на дополнительную память (таблица T):

$$\Theta(n^2).$$

- Идея — разредить таблицу, убрать часть информации
- Будем хранить минимумы для тех интервалов, длины которых являются *степенями двойки*
- Такая структура называется *sparse table* (рус. *разрежённая таблица*)

- Иначе говоря, для каждой позиции i храним минимумы на интервалах длины $1, 2, 4, \dots, 2^k$ вправо от i
- $t_{k,i} = \min\{a_i, a_{i+1}, a_{i+2}, \dots, a_{i+2^k-1}\}$
- $\Theta(n \log n)$ памяти, $\Theta(n \log n)$ времени на построение





Sparse table. Пример

2	9	1	9	6	7	5	2
0	1	2	3	4	5	6	7

$$\begin{cases} t_{0,i} = a_i \\ t_{k,i} = \min \{t_{k-1,i}, t_{k-1,i+2^{k-1}}\} \end{cases}$$

Предположим, $2^{k-1} < n \leq 2^k$.

Что Разреженная таблица занимает $\Theta(n \log n)$ ячеек памяти.

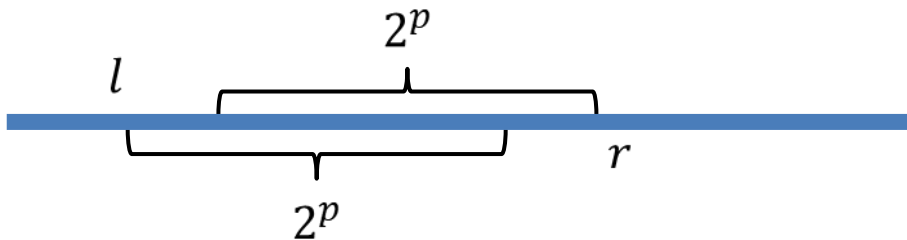
0 (=2 ⁰)	2	9	1	9	6	7	5	2
1 (=2 ¹)	2	1	1	6	6	5	2	
2 (=2 ²)	1	1	1	5	2			
3 (=2 ³)	1							

Оценим число «лишних» ячеек в таблице.

$$(2^1 - 1) + (2^2 - 1) + \dots + (2^k - 1) = 2 \cdot (2^k - 1) - k < 4 \cdot n$$

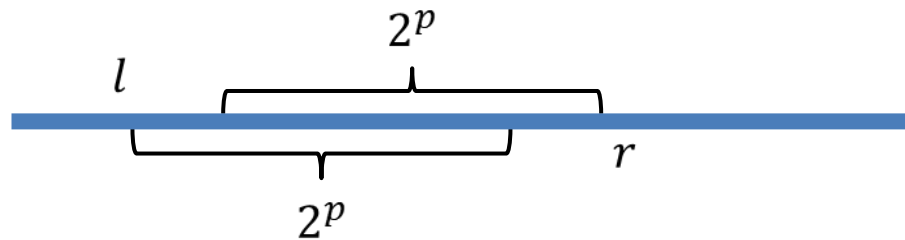
$$(2^1 - 1) + (2^2 - 1) + \dots + (2^k - 1) = 2 \cdot (2^k - 1) - k > 2 \cdot (n - 1) - \lceil \log_2 n \rceil - 1$$

- Нужно найти минимум на $[l, r)$
- Пусть p — максимальная степень такая, что 2^p не превосходит длины интервала: $2^p \leq r - l < 2^{p+1}$
- Интервал $[l, r)$ покрывается не более чем двумя перекрывающимися интервалами длины 2^p (начало первого интервала фиксируется на левой границе l , а второго — на правой границе r)



Ответ на запрос минимума на полуинтервале можно вычислить по формуле:

$$\min \{t_{p, l}, t_{p, r-2^p}\}$$



Формула является корректной, так как бинарная операция минимума обладает свойствами **ассоциативности, коммутативности и идемпотентности**.

1. **Ассоциативность** (выполнять операции можно в произвольном порядке, т.е. допускается любой порядок расстановки скобок):

$$a \boxtimes (b \boxtimes c) = (a \boxtimes b) \boxtimes c$$

2. **Коммутативность** (можно располагать элементы в произвольном порядке, переставляя их):

$$a \boxtimes b = b \boxtimes a$$

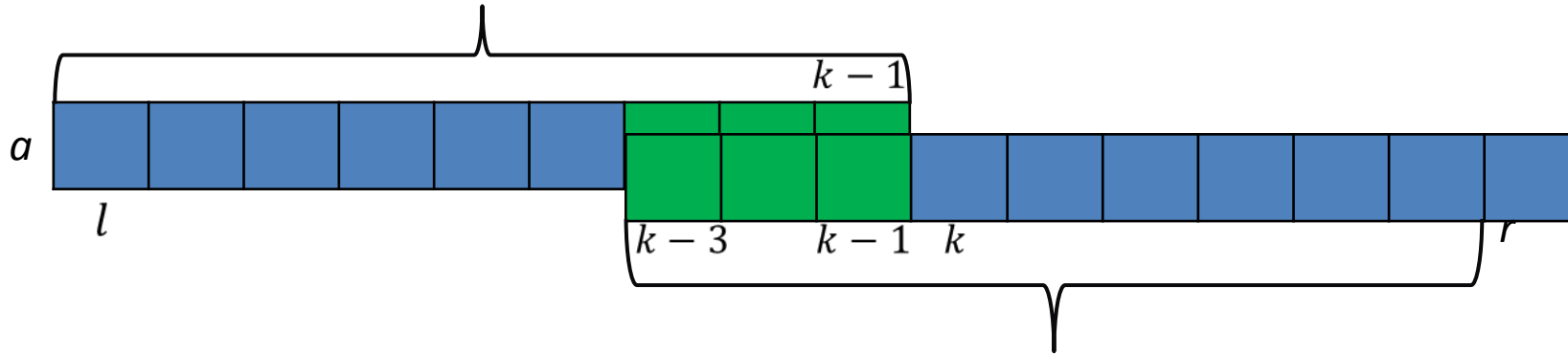
3. **Идемпотентность** (благодаря этому свойству мы сможем работать на пересекающихся отрезках):

$$a \boxtimes a = a$$



Свойства бинарного отношения минимума

Например, покажем, что **минимум** на $[l, r) = \text{минимум} \{ \text{минимум на } [l, k), \text{минимум на } [k - 3, r) \}$



$$\begin{aligned}
 & (a_l \circ a_{l+1} \circ \dots \circ a_{k-3} \circ a_{k-2} \circ a_{k-1}) \circ (a_{k-3} \circ a_{k-2} \circ a_{k-1} \circ a_k \circ \dots \circ a_{r-1}) = [\text{ассоциативность}] \\
 & = a_l \circ a_{l+1} \circ \dots \circ a_{k-3} \circ a_{k-2} \circ a_{k-1} \circ a_{k-3} \circ a_{k-2} \circ a_{k-1} \circ a_k \circ \dots \circ a_{r-1} = [\text{коммутативность}] \\
 & = a_l \circ a_{l+1} \circ \dots \circ a_{k-3} \circ a_{k-3} \circ a_{k-2} \circ a_{k-2} \circ a_{k-1} \circ a_{k-1} \circ a_k \circ \dots \circ a_{r-1} = [\text{ассоциативность}] \\
 & = a_l \circ a_{l+1} \circ \dots \circ (a_{k-3} \circ a_{k-3}) \circ (a_{k-2} \circ a_{k-2}) \circ (a_{k-1} \circ a_{k-1}) \circ a_k \circ \dots \circ a_{r-1} = [\text{идемпотентность}] \\
 & = a_l \circ a_{l+1} \circ \dots \circ a_{k-3} \circ a_{k-2} \circ a_{k-1} \circ a_k \circ \dots \circ a_{r-1}
 \end{aligned}$$



FindMin(1,6)

a	2	9	1	9	6	7	5	2
	0	1	2	3	4	5	6	7
0 (=2 ⁰)	2	9	1	9	6	7	5	2
1 (=2 ¹)	2	1	1	6	6	5	2	
2 (=2 ²)	1	1	1	5	2			
3 (=2 ³)	1							

$$t_{k,i} = \min\{a_i, a_{i+1}, \dots, a_{i+2^k-1}\}$$

$$\max\{p \mid 2^p \leq r - l < 2^{p+1}\} = \max\{p \mid 2^p \leq 6 - 1 < 2^{p+1}\} = \max\{p \mid 2^p \leq 5 < 2^{p+1}\} = 2$$

a	2	9	1	9	6	7	5	2
	0	1	2	3	4	5	6	7

$$\text{FindMin}(1,6) = \min\{t_{p,l}, t_{p,r-2^p}\} = \min\{t_{2,1}, t_{2,6-2^2}\} = \min\{t_{2,1}, t_{2,2}\} = \min\{1,1\} = 1$$

$$\max\{p \mid 2^p \leq r - l < 2^{p+1}\}$$



$$r - l = 3$$



$$r - l = 4$$



$$r - l = 11$$

- Дано целое число x ($1 \leq x \leq n$).
- Найти наибольшее p такое, что $2^p \leq x$, за константное время (для этого можно сделать предподсчёт).
- Подходы
 - Формула
 - Динамическое программирование

- Задана полоска из n клеток, клетки, некоторые клетки покрашены в чёрный цвет, остальные в белый



- Нужно уметь выполнять две операции:
 - $\text{Invert}(l, r)$ — на интервале $[l, r)$ изменить цвет на противоположный
 - $\text{GetColor}(i)$ — получить цвет i -й клетки



iRunner

0.2 Задача о сумме (реализация структур для интервальных запросов - сумма на отрезке)



БЕЛОРУССКИЙ
ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ

Спасибо за внимание!