Мини-задача #43 (1 балл)

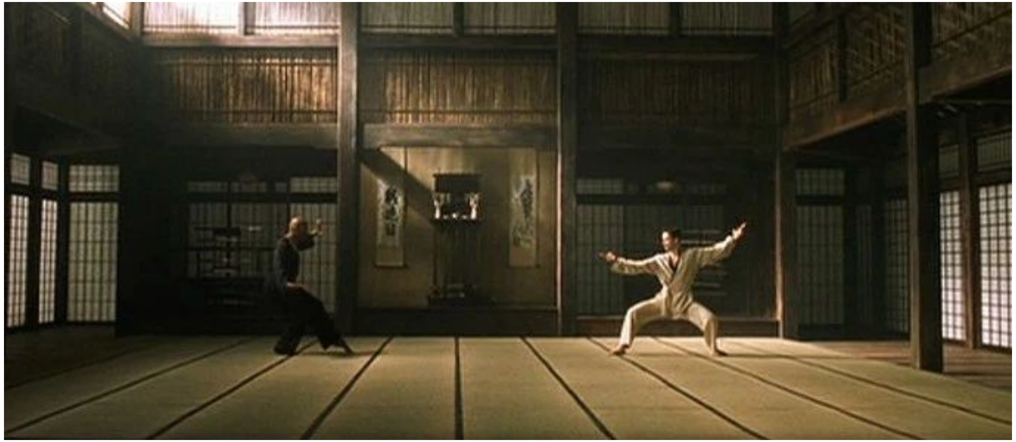
https://leetcode.com/problems/range-sum-query-mutable

Решите задачу, используя базовое дерево отрезков!

Мини-задача #44 (1 балл)

https://leetcode.com/problems/count-of-smaller-numbers-after-self/

Решите задачу, используя (небольшую вариацию) дерева отрезков.



Алгоритмы и структуры данных

Деревья отрезков

Путаница с названиями

Путаница с названиями

В англоязычной литературе есть такая структура данных: interval search tree.

Путаница с названиями

В англоязычной литературе есть такая структура данных: interval search tree.

Хранит некоторые интервалы.

Операции:

1) поиск интервала в дереве,
2) добавление/удаление интервала,
3) поиск по интервалу всех, кто с ним пересекается.

Путаница с названиями

В англоязычной литературе есть такая структура данных: interval search tree.

Хранит некоторые интервалы.

Операции:

1) поиск интервала в дереве,
2) добавление/удаление интервала,
3) поиск по интервалу всех, кто с ним пересекается.

Реализация:

Сбалансированное дерево поиска по левой границе с доп. информацией: максимальной правой границей в поддереве.

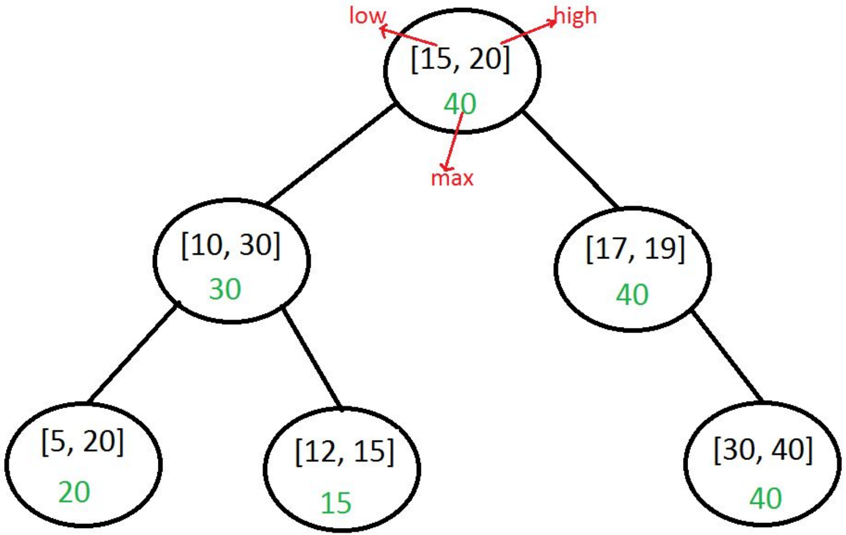
Путаница с названиями

В англоязычной литературе есть такая структура данных: interval search tree.

Хранит некоторые интервалы.

Реализация:

Сбалансированное дерево поиска по левой границе с доп. информацией: максимальная правая граница в поддереве.



Путаница с названиями

Мы же сегодня будем говорить про совсем другую структуру данных: segment tree.

Путаница с названиями

Мы же сегодня будем говорить про совсем другую структуру данных: segment tree.

К сожалению, в русской литературе иногда переводят и interval tree и segment tree, как "как дерево отрезков".

Путаница с названиями

Мы же сегодня будем говорить про совсем другую структуру данных: segment tree.

К сожалению, в русской литературе иногда переводят и interval tree и segment tree, как "как дерево отрезков".

Сегодня имеем в виду именно что segment tree.

Сумма на отрезке

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

$$a_1, a_2, a_3, \ldots, a_n$$

Сумма на отрезке

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать операцию следующего вида:

- sum(l, r) - найти сумму элементов: a_l + … _ a_r

Сумма на отрезке

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

$$a_1, a_2, a_3, \dots, a_n$$

Необходимо поддержать операцию следующего вида:

- sum(l, r) - найти сумму элементов: $a_l + \dots \_ \ a_r$

Как решать?

Сумма на отрезке

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать операцию следующего вида:

- sum(l, r) - найти сумму элементов: a_l + … _ a_r

Как решать?

Наивный алгоритм за O(N), как быстрее?

Сумма на отрезке

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать операцию следующего вида:

- sum(l, r) - найти сумму элементов: a_l + … _ a_r

sum(3, 5) = ?

46 11 40 [8 2 42] 65 10

Сумма на отрезке

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать операцию следующего вида:

- sum(l, r) - найти сумму элементов: a_l + … _ a_r

sum(0, 2) = 97

46 11 40 8 2 42 65 10

sum(0, 5) = 149

Сумма на отрезке

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать операцию следующего вида:

- sum(l, r) - найти сумму элементов: a_l + … _ a_r

sum(0, 2) = 97

46 11 40 8 2 42 65 10   sum(3, 5) = 149 - 97 = 52

sum(0, 5) = 149

Prefix sum

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать операцию следующего вида:

- sum(l, r) - найти сумму элементов: a_l + … _ a_r

Т.е. задача сводится к тому, чтобы вычесть две **префиксные суммы** до l-1 и до r.

Prefix sum

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать операцию следующего вида:

- sum(l, r) - найти сумму элементов: a_l + … _ a_r

Т.е. задача сводится к тому, чтобы вычесть две **префиксные суммы** до l-1 и до r. Но ведь их можно насчитать заранее!

Prefix sum

```
def buildPrefixes(n: int, arr: int[]) -> int[]:
  prefixes = int[n]

  currentSum = 0
  for i in 0..n-1:
    currentSum += arr[i]
    prefixes[i] = currentSum

  return prefixes
```

Prefix sum

```
def buildPrefixes(n: int, arr: int[]) -> int[]:
  prefixes = int[n]

  currentSum = 0
  for i in 0..n-1:
    currentSum += arr[i]
    prefixes[i] = currentSum

  return prefixes
```

Работает за O(N) времени и O(N) памяти

Prefix sum

```
def buildPrefixes(n: int, arr: int[]) -> int[]:
  prefixes = int[n]

  currentSum = 0
  for i in 0..n-1:
    currentSum += arr[i]
    prefixes[i] = currentSum

  return prefixes



def sum(l, r: int, prefixes: int[]) -> int:
```

Работает за O(N) времени и O(N) памяти

Prefix sum

```
def buildPrefixes(n: int, arr: int[]) -> int[]:
  prefixes = int[n]

  currentSum = 0
  for i in 0..n-1:
    currentSum += arr[i]
    prefixes[i] = currentSum

  return prefixes



def sum(l, r: int, prefixes: int[]) -> int:
  return prefixes[r] - prefixes[l - 1]
```

Работает за O(N) времени и O(N) памяти

Prefix sum

```
def buildPrefixes(n: int, arr: int[]) -> int[]:
  prefixes = int[n]

  currentSum = 0
  for i in 0..n-1:
    currentSum += arr[i]
    prefixes[i] = currentSum

  return prefixes
```

Работает за O(N) времени и O(N) памяти

```
def sum(l, r: int, prefixes: int[]) -> int:
  return prefixes[r] - prefixes[l - 1]
```

Работает за O(1)

Prefix sum

```
def buildPrefixes(n: int, arr: int[]) -> int[]:
  prefixes = int[n]

  currentSum = 0
  for i in 0..n-1:
    currentSum += arr[i]
    prefixes[i] = currentSum

  return prefixes
```

Работает за O(N) времени и O(N) памяти

Препроцессинг запустили всего один раз, а запросов может быть много!

```
def sum(l, r: int, prefixes: int[]) -> int:
  return prefixes[r] - prefixes[l - 1]
```

Работает за O(1)

Prefix sum

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать операцию следующего вида:

- sum(l, r) - найти сумму элементов: a_l + … _ a_r

Дерево отрезков

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать новые операции следующего вида:

1. update(pos, val) - обновляем значение элемента a_pos
2. sum(l, r) - найти сумму элементов: a_l + … _ a_r

Дерево отрезков

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать новые операции следующего вида:

1. update(pos, val) - обновляем значение элемента a_pos

2. sum(l, r) - найти сумму элементов: a_l + … _ a_r

Префиксные суммы, получается, придется обновлять?
Магия с O(1) пропадает, нужно думать.

Дерево отрезков

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать новые операции следующего вида:

1. update(pos, val) - обновляем значение элемента a_pos
2. sum(l, r) - найти сумму элементов: a_l + … _ a_r

Но мы уже знаем один вариант решения!

Дерево отрезков

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать новые операции следующего вида:

1. update(pos, val) - обновляем значение элемента a_pos
2. sum(l, r) - найти сумму элементов: a_l + … _ a_r

Но мы уже знаем один вариант решения! Декартовы деревья. Обе операции будут стоит O(logN) **в среднем** .

Дерево отрезков

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать новые операции следующего вида:

1. update(pos, val) - обновляем значение элемента a_pos

2. sum(l, r) - найти сумму элементов: a_l + … _ a_r

Но мы уже знаем один вариант решения! Декартовы деревья. Обе операции будут стоит O(logN) **в среднем** .

Поговорим про другой вариант, со сложностью в худшем.

Дерево отрезков

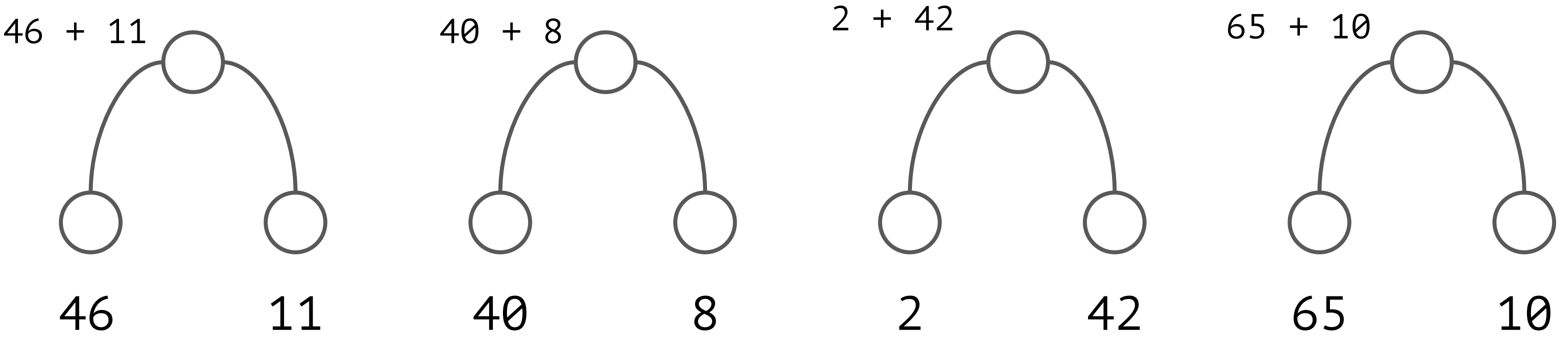
46 11 40 8 2 42 65 10

Дерево отрезков

46 11 40 8 2 42 65 10

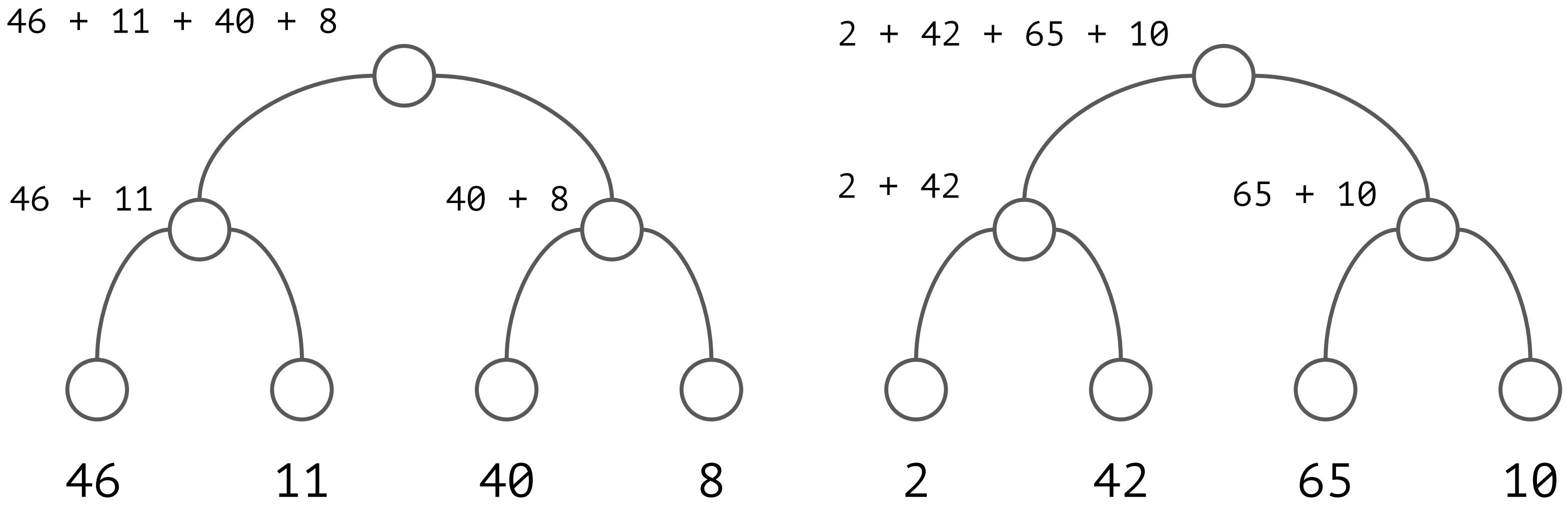
Дерево отрезков

46 11 40 8 2 42 65 10



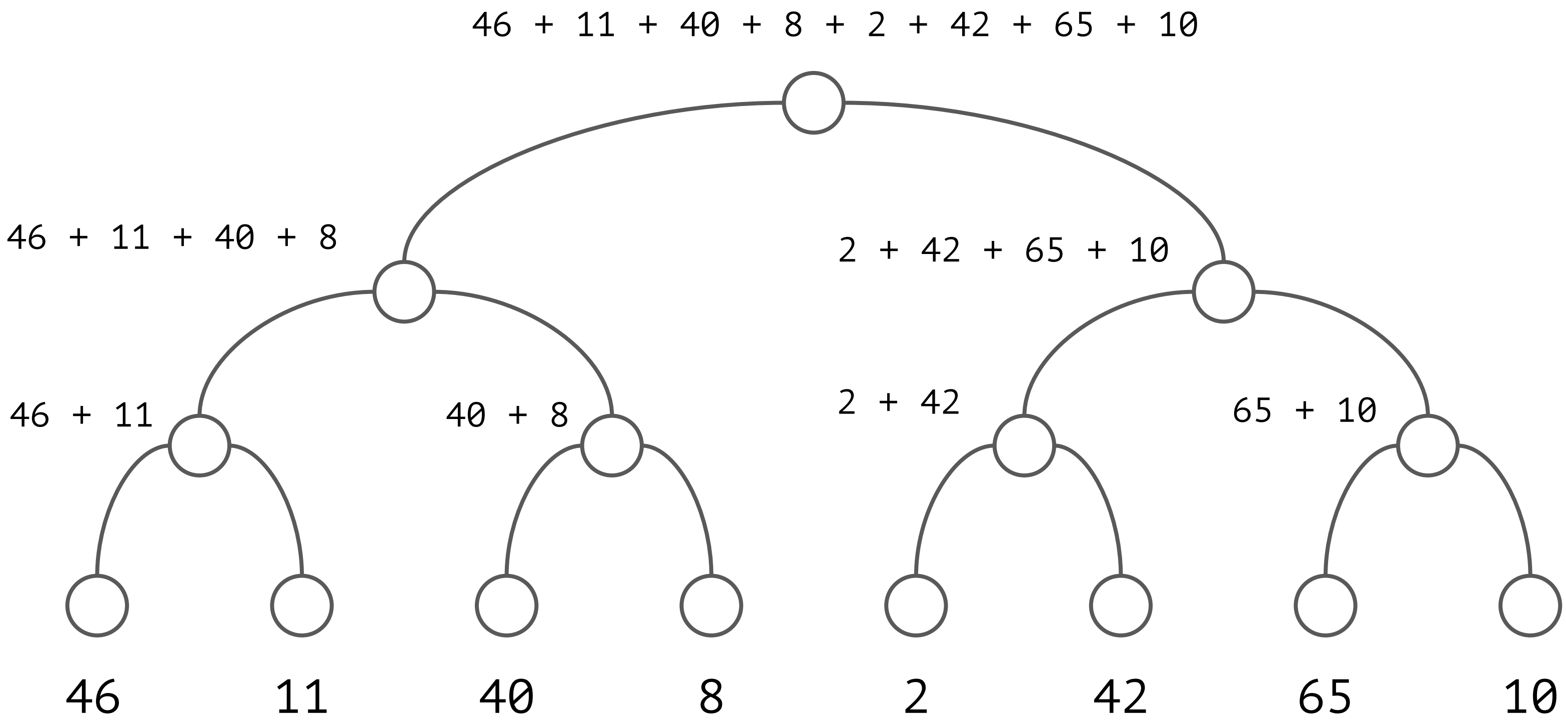
Дерево отрезков

46 11 40 8 2 42 65 10



Дерево отрезков

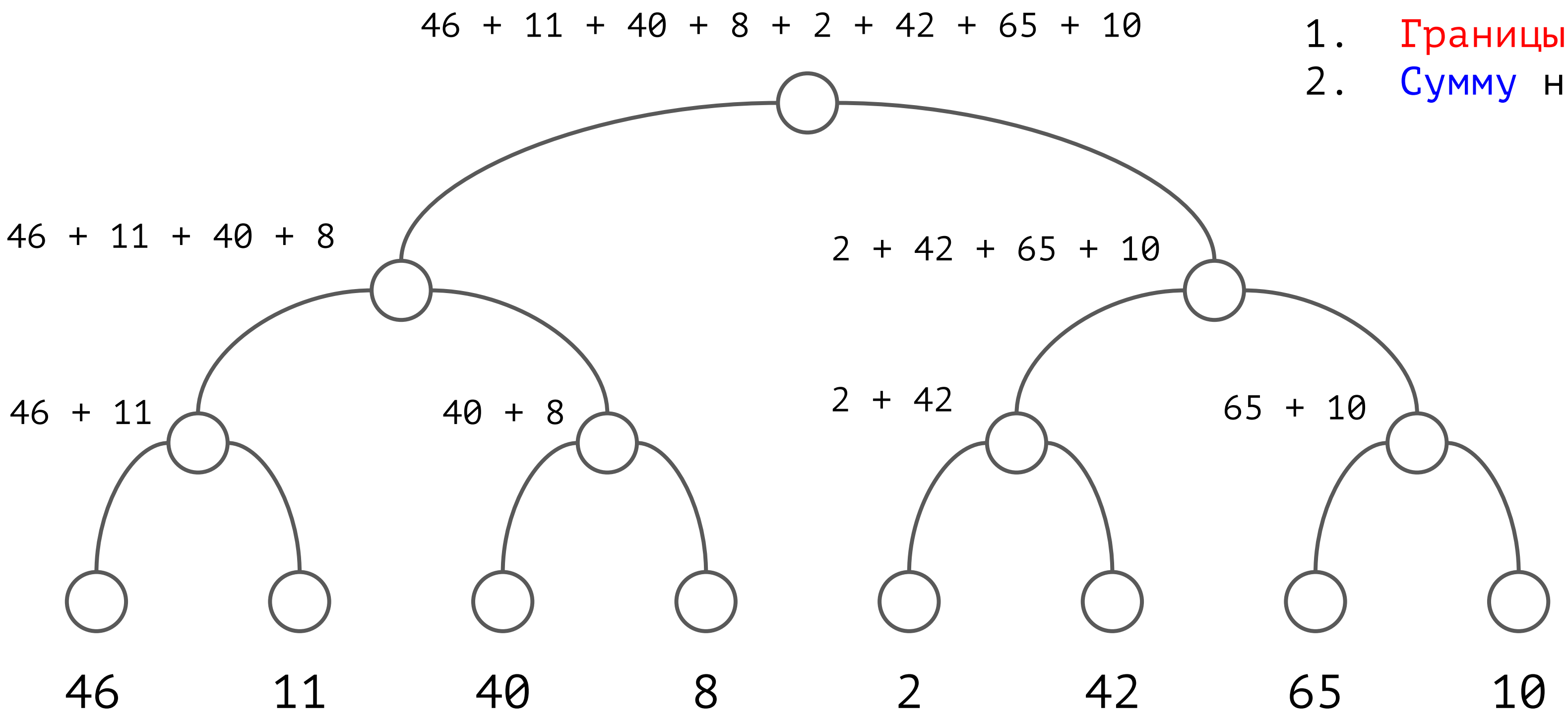
46 11 40 8 2 42 65 10



46 11 40 8 2 42 65 10

На самом деле в каждой вершине будем хранить:

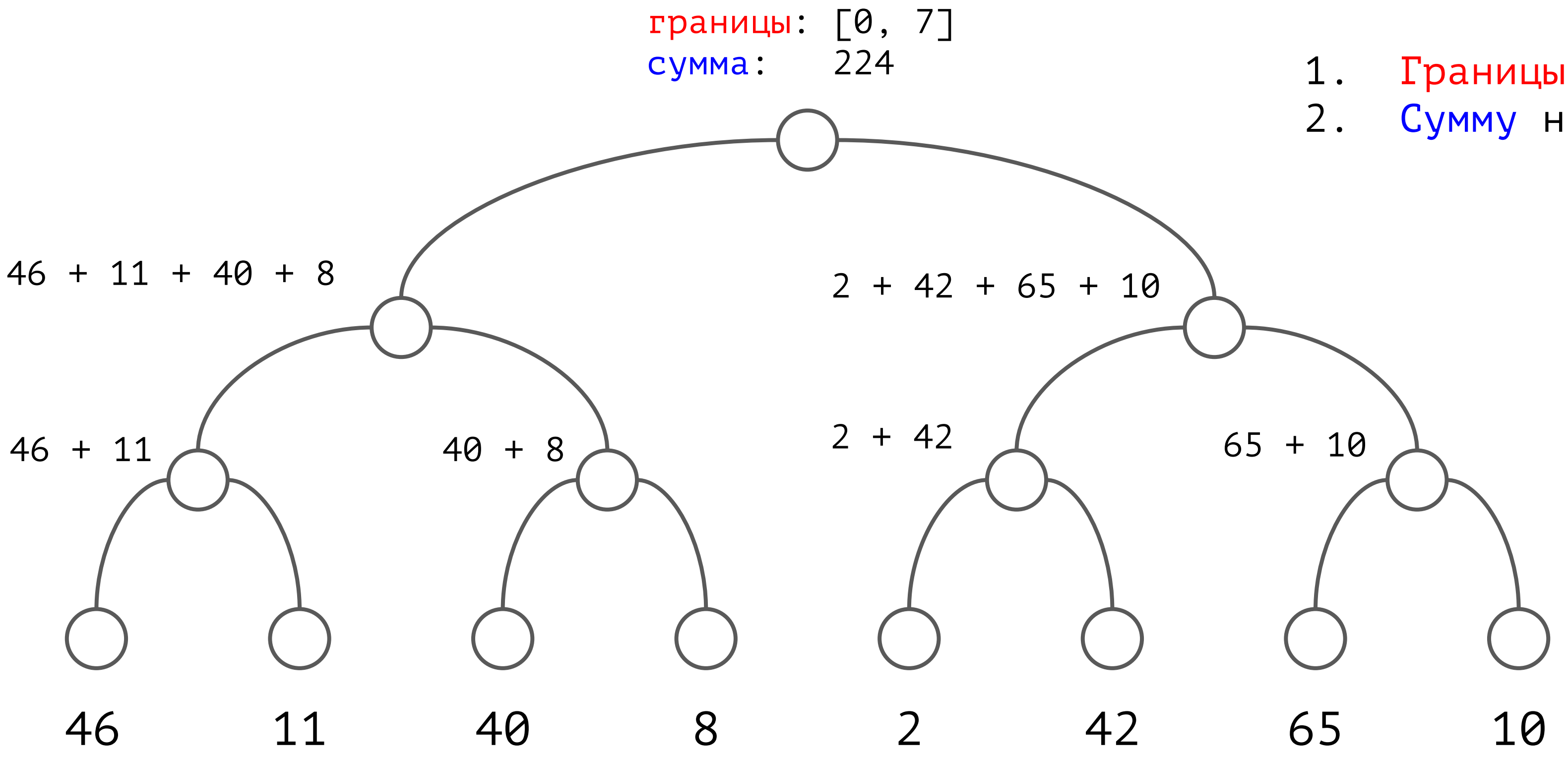
1. Границы отрезка
2. Сумму на нем



46 11 40 8 2 42 65 10

На самом деле в каждой вершине будем хранить:

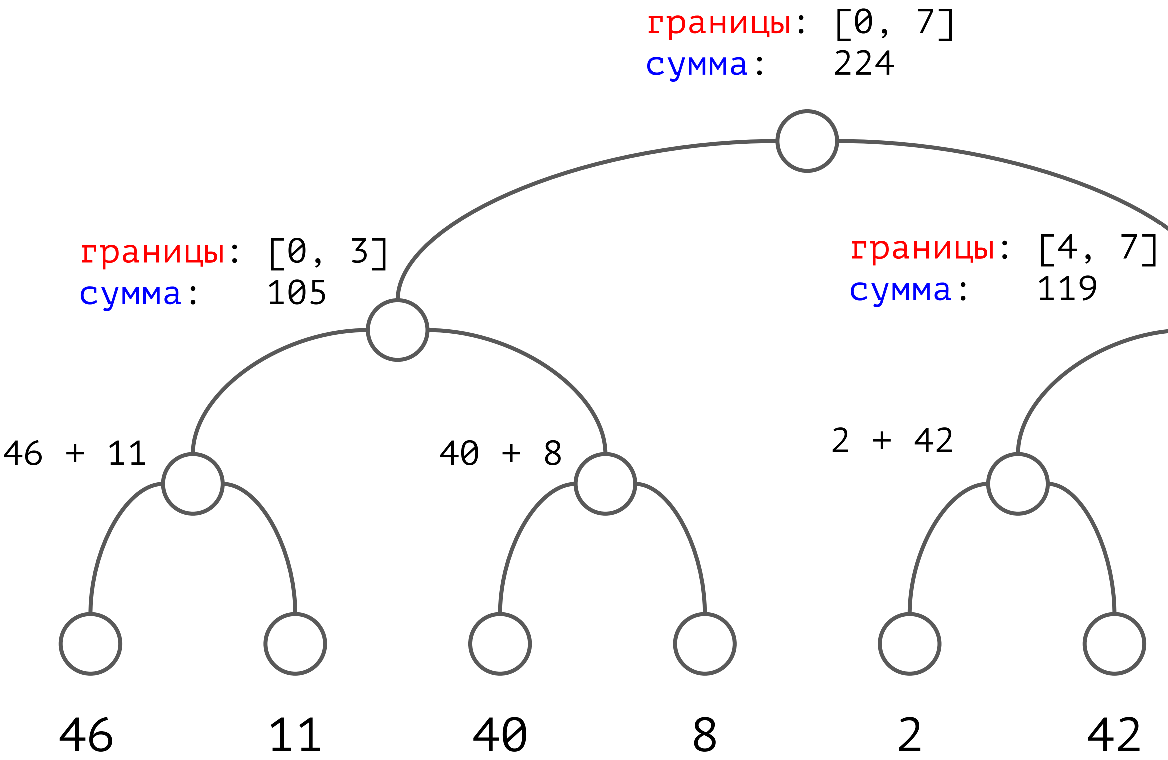
1. Границы отрезка
2. Сумму на нем



46 11 40 8 2 42 65 10

На самом деле в каждой вершине будем хранить:

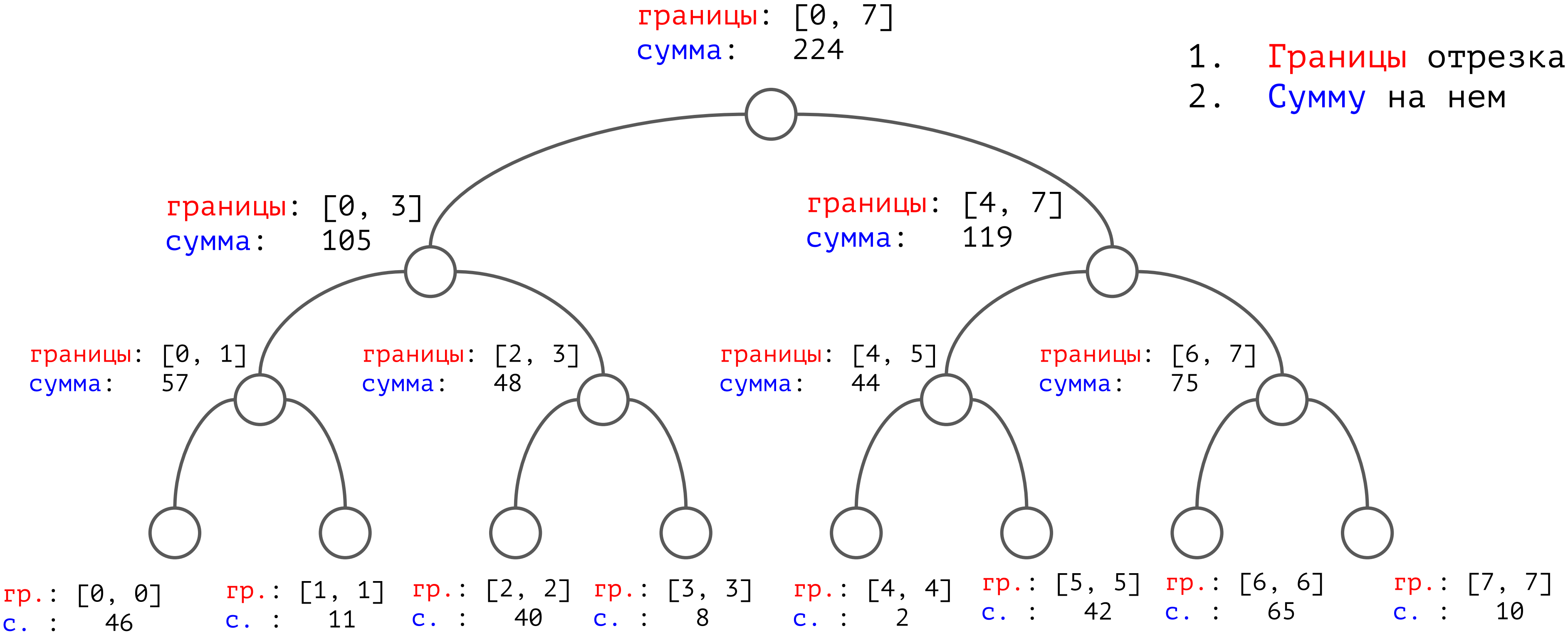
1. Границы отрезка
2. Сумму на нем



46 11 40 8 2 42 65 10

На самом деле в каждой вершине будем хранить:

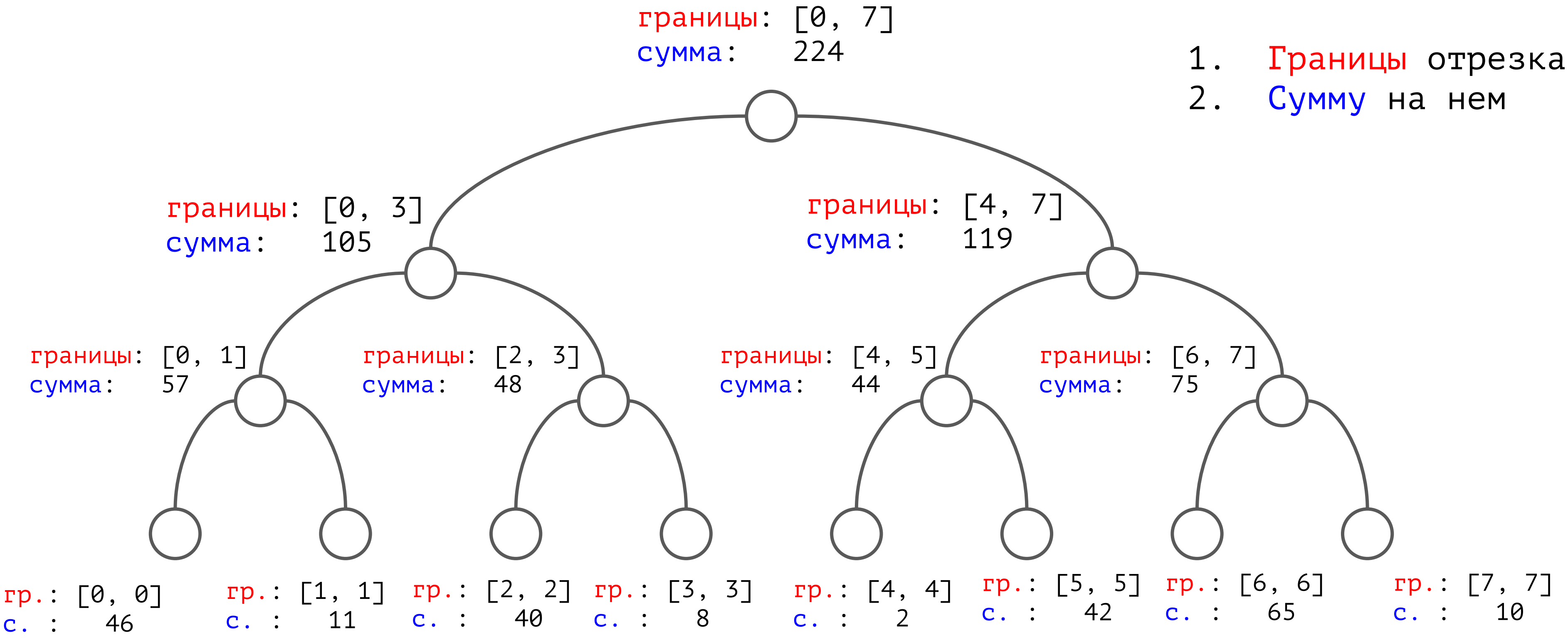
1. Границы отрезка
2. Сумму на нем



46 11 40 8 2 42 65 10

На самом деле в каждой вершине будем хранить:

1. Границы отрезка
2. Сумму на нем



Строится за O(N), т.к. в дереве 2N - 1 вершин (идем снизу, мерж за O(1))

Дерево отрезков

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

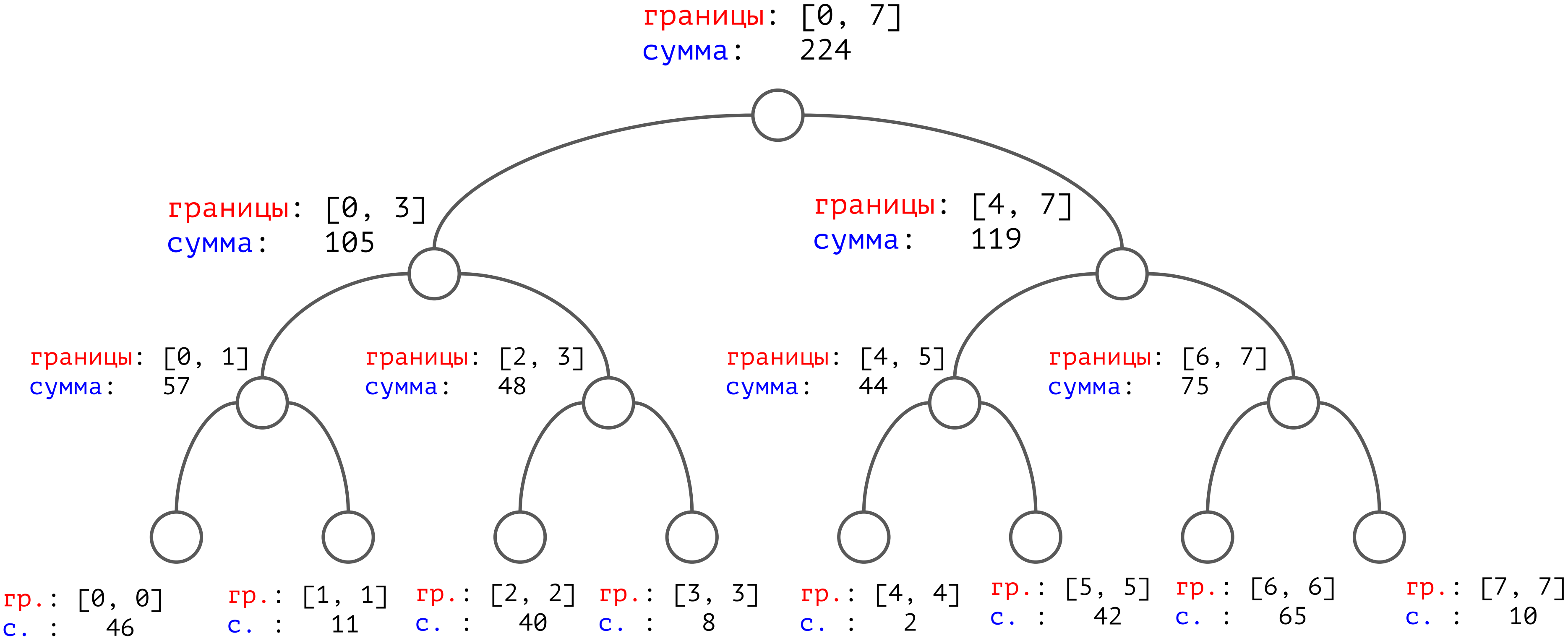
Необходимо поддержать новые операции следующего вида:

1. update(pos, val) - обновляем значение элемента a_pos

2. sum(l, r) - найти сумму элементов: a_l + … _ a_r

46 11 40 8 2 42 65 10

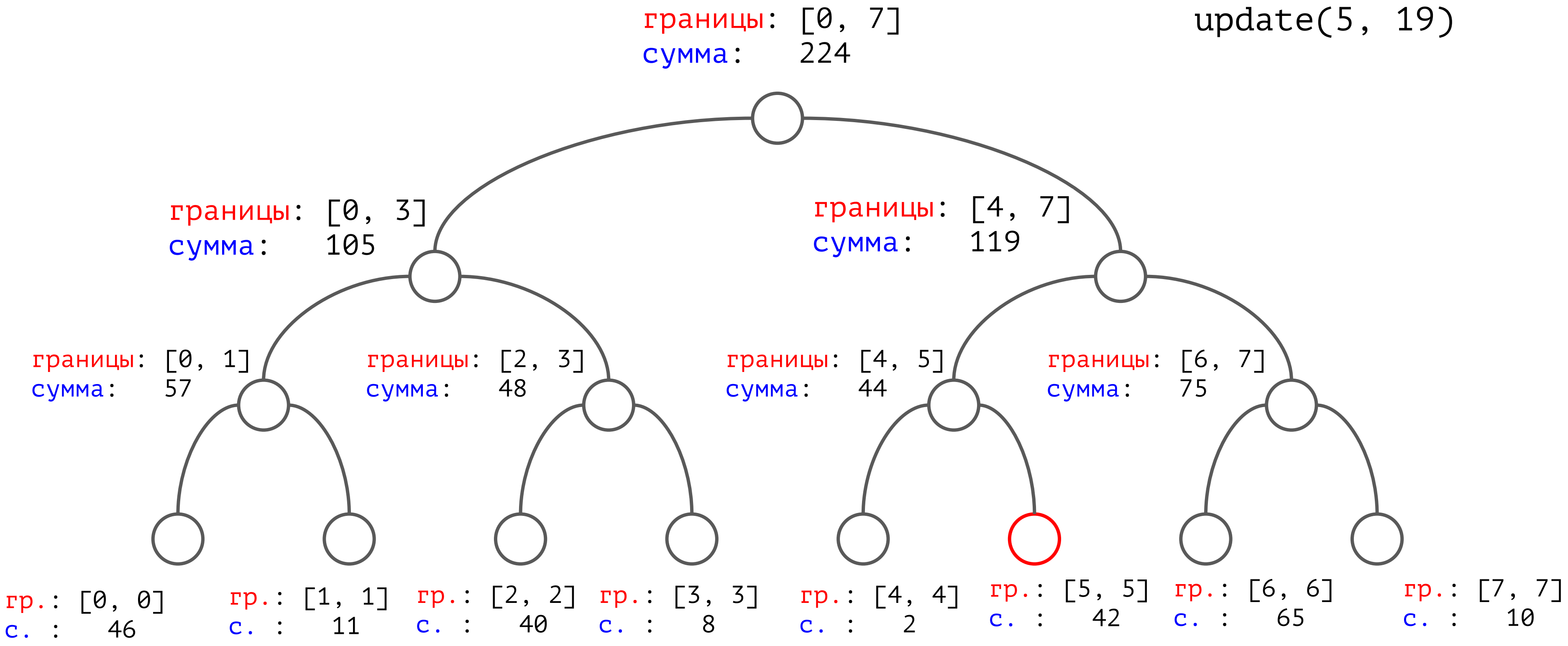
Как реализовать update(pos, x)?



46 11 40 8 2 42 65 10

Как реализовать update(pos, x)?

update(5, 19)

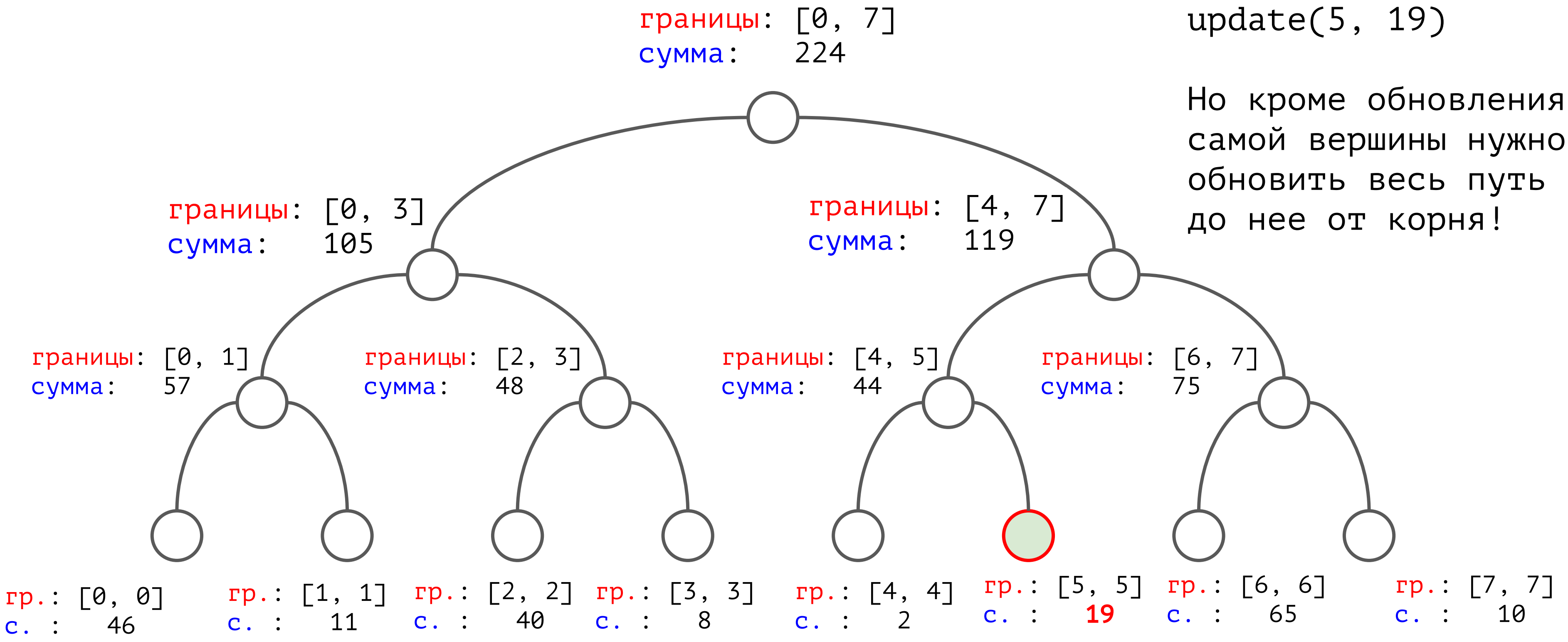


46 11 40 8 2 19 65 10

Как реализовать update(pos, x)?

update(5, 19)

Но кроме обновления самой вершины нужно обновить весь путь до нее от корня!

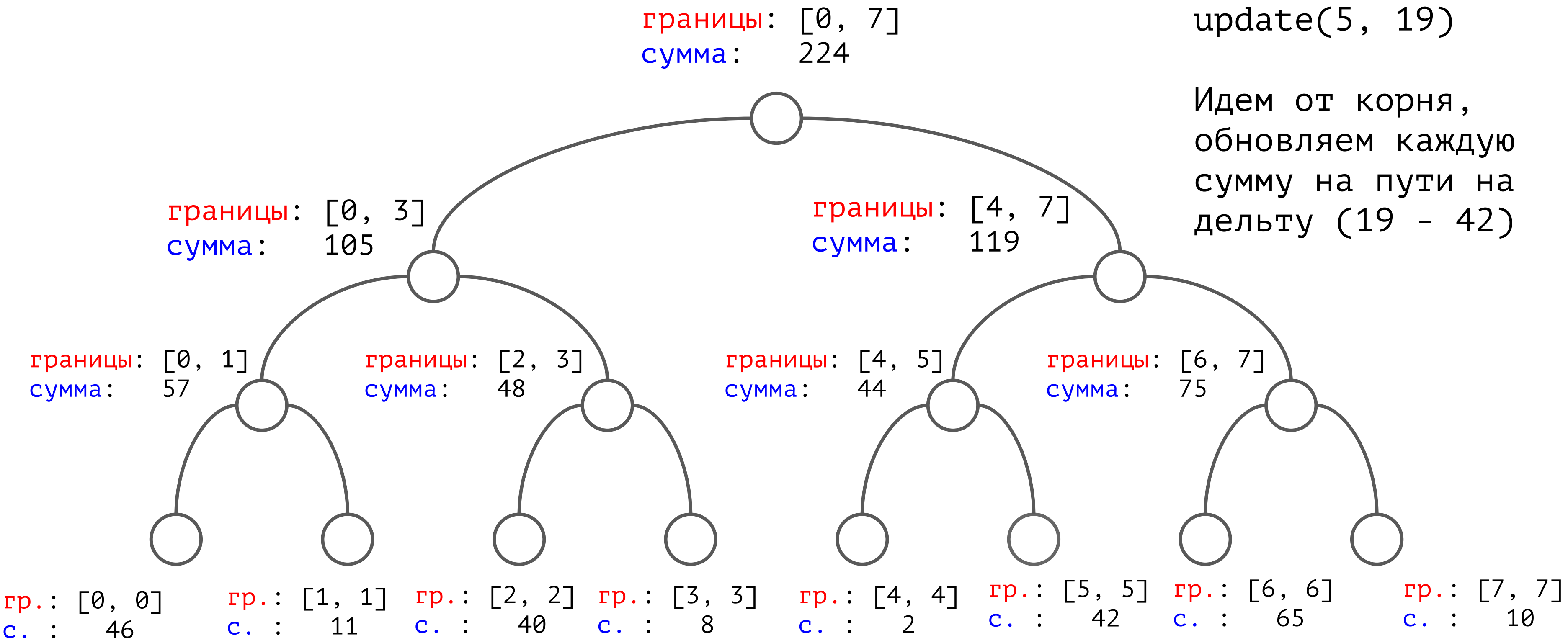


46 11 40 8 2 42 65 10

Как реализовать update(pos, x)?

update(5, 19)

Идем от корня, обновляем каждую сумму на пути на дельту (19 - 42)

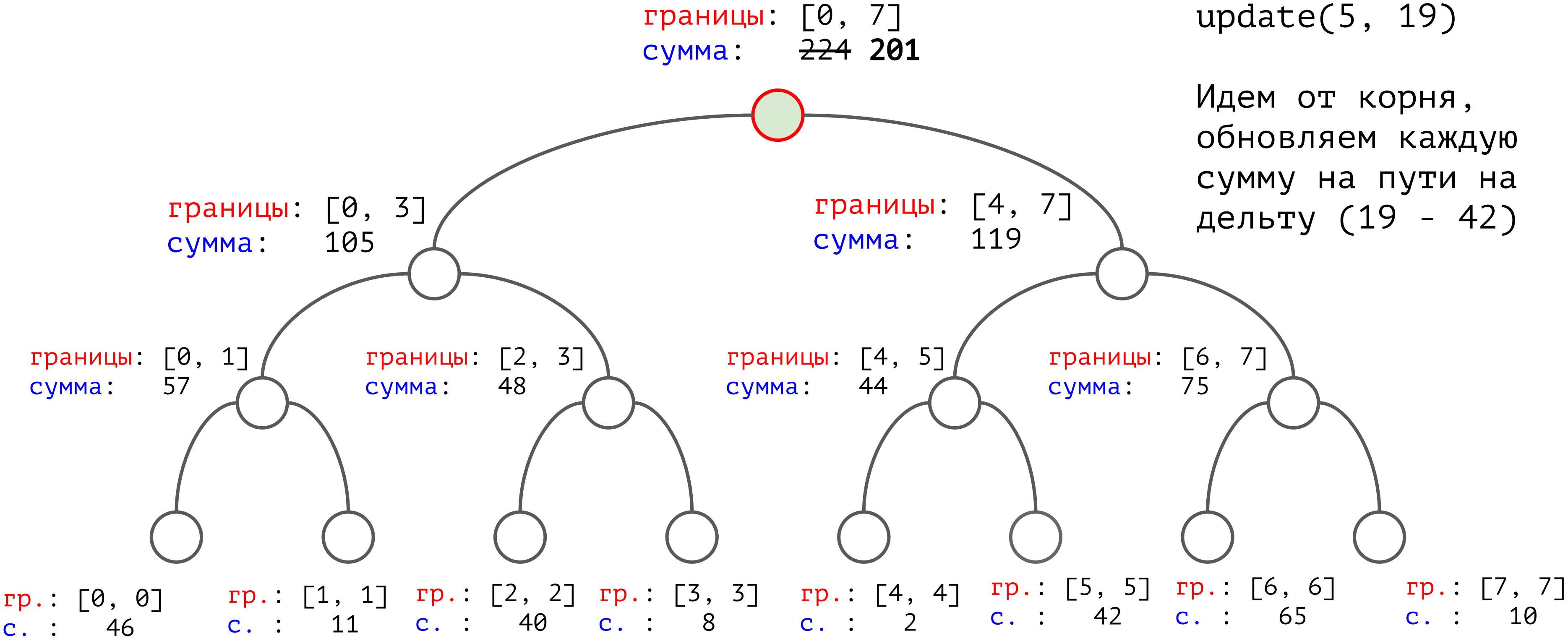


46 11 40 8 2 42 65 10

Как реализовать update(pos, x)?

update(5, 19)

Идем от корня, обновляем каждую сумму на пути на дельту (19 - 42)

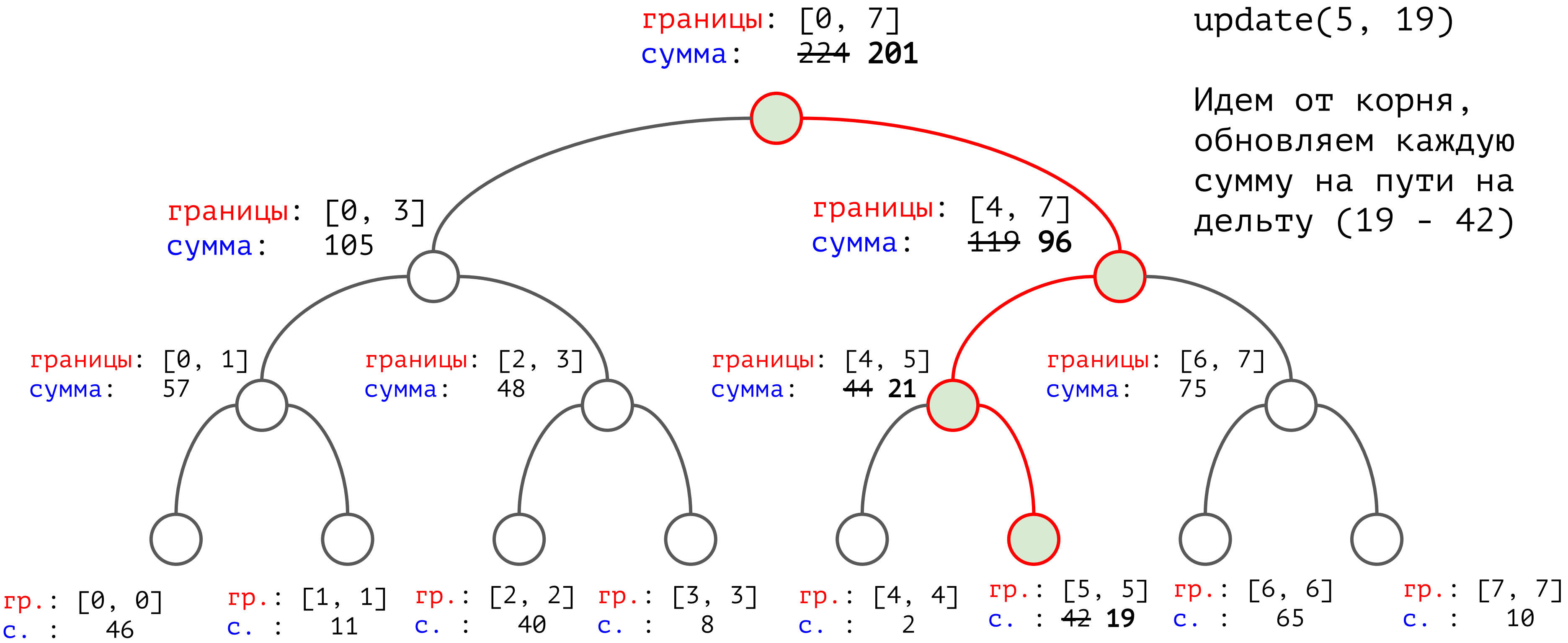


46 11 40 8 2 19 65 10

Как реализовать update(pos, x)?

update(5, 19)

Идем от корня, обновляем каждую сумму на пути на дельту (19 - 42)



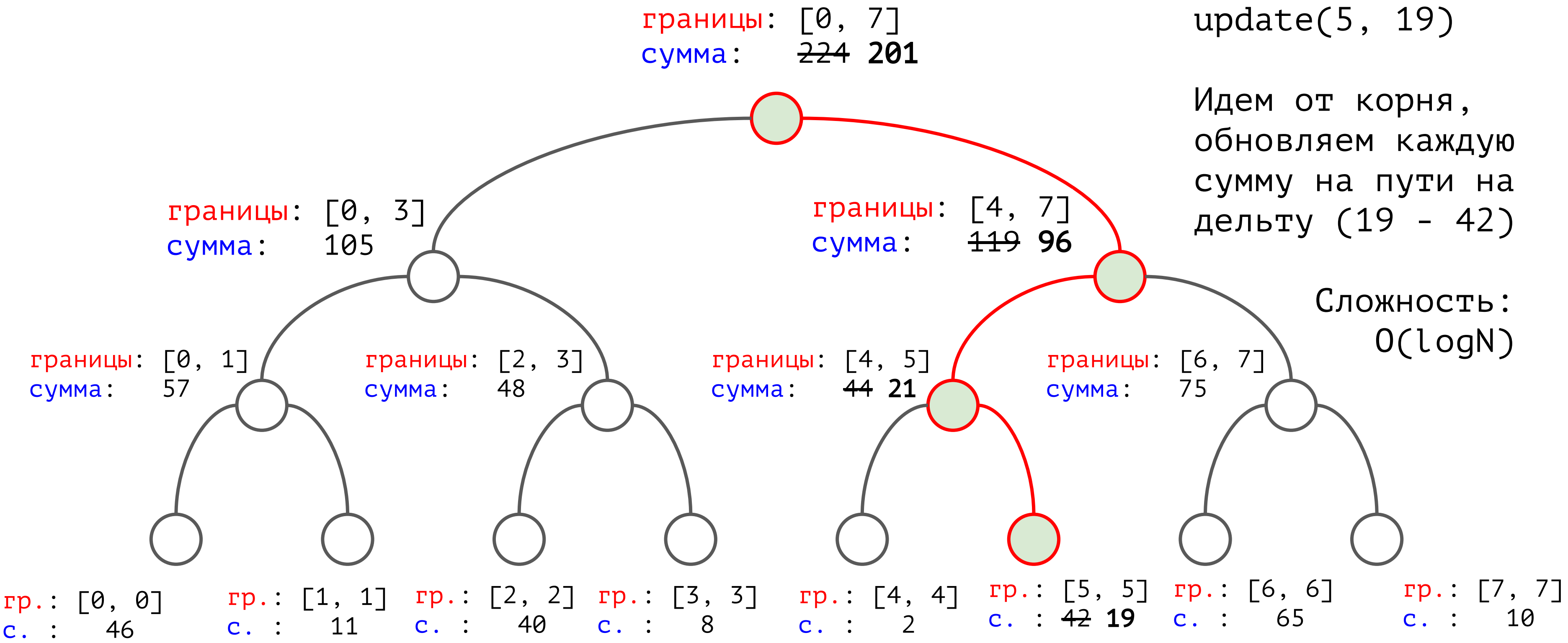
46 11 40 8 2 19 65 10

Как реализовать update(pos, x)?

update(5, 19)

Идем от корня, обновляем каждую сумму на пути на дельту (19 - 42)

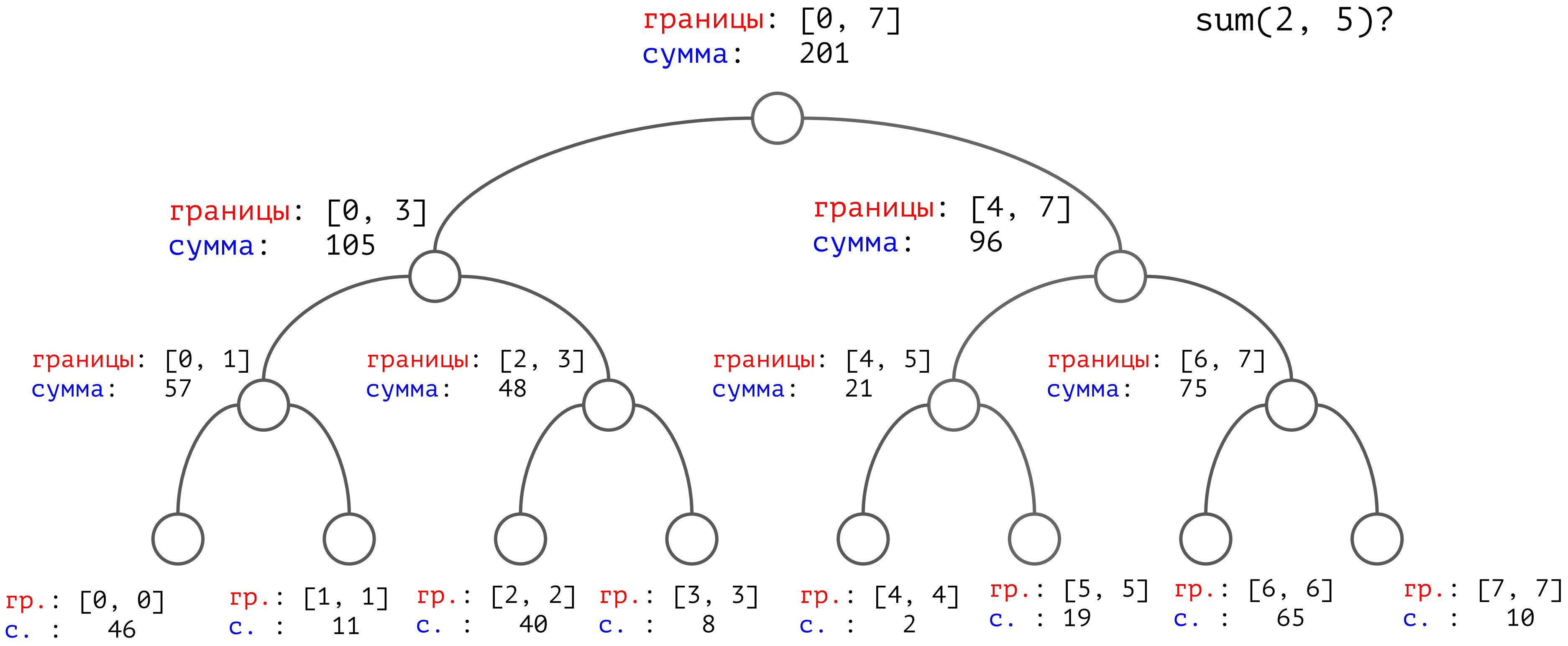
Сложность: O(logN)



46 11 40 8 2 19 65 10

Как реализовать sum(l, r)?

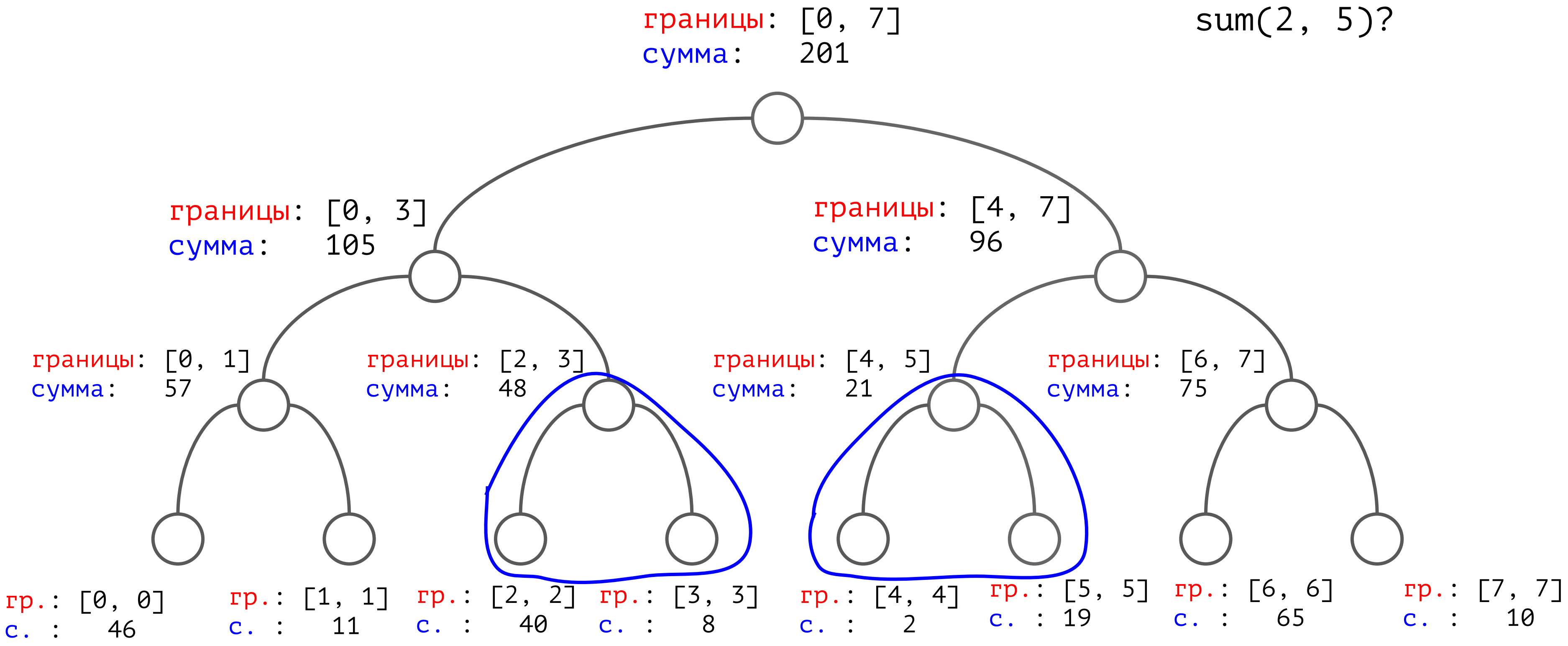
sum(2, 5)?



46 11 40 8 2 19 65 10

Как реализовать sum(l, r)?

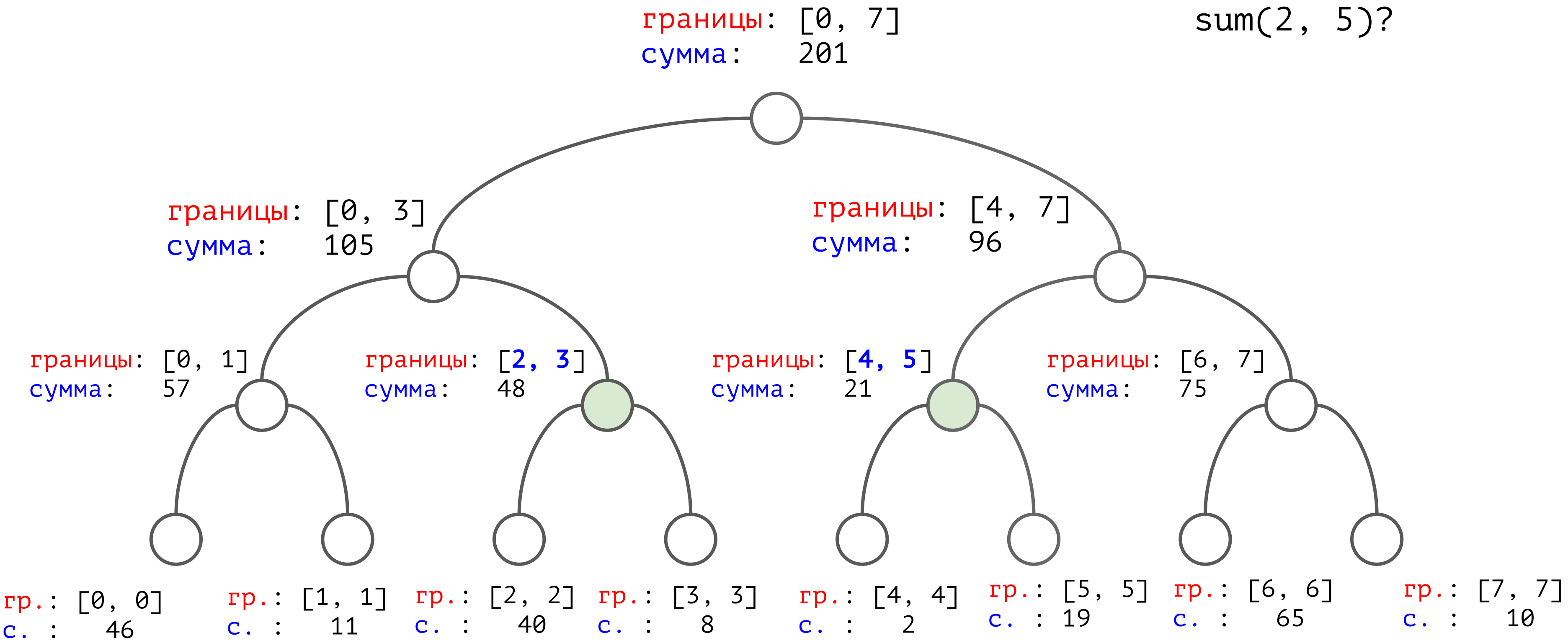
sum(2, 5)?



46 11 40 8 2 19 65 10

Как реализовать sum(l, r)?

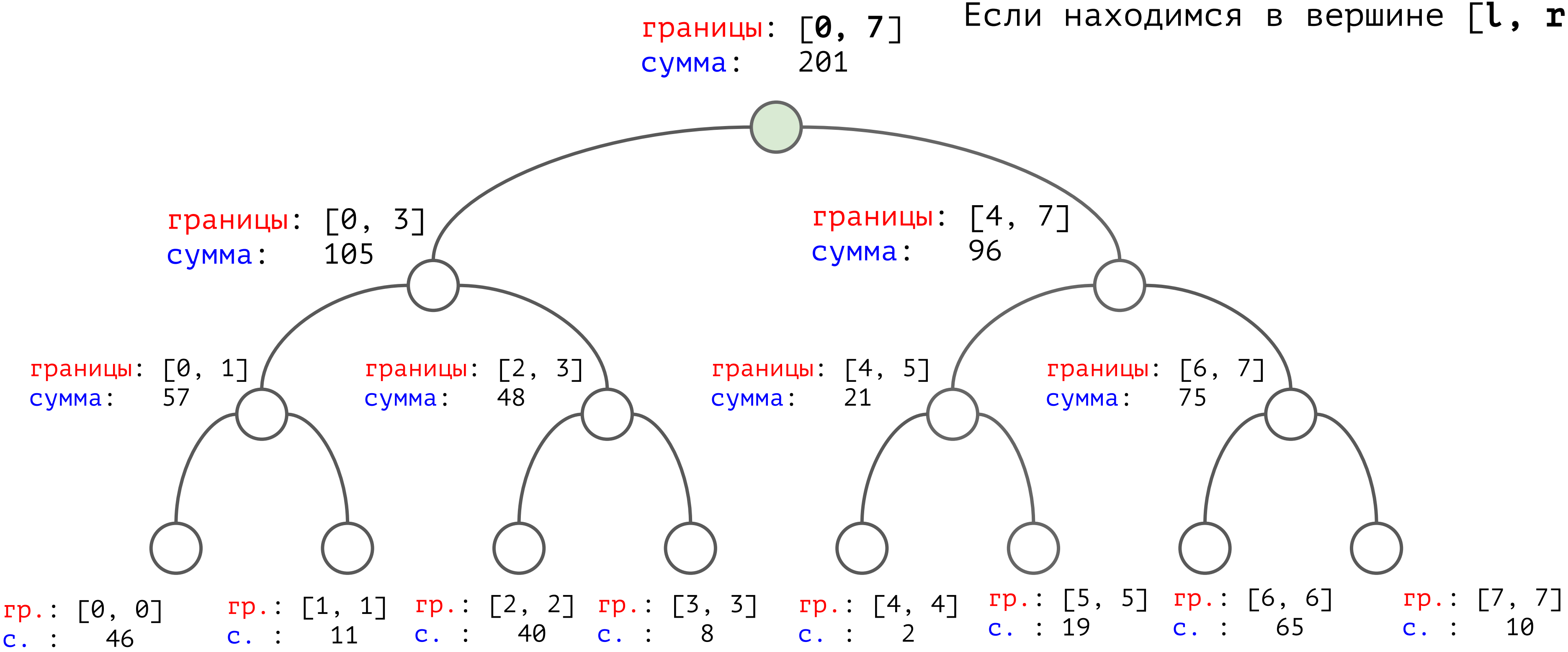
sum(2, 5)?



46 11 40 8 2 19 65 10

Как реализовать sum(l, r)?

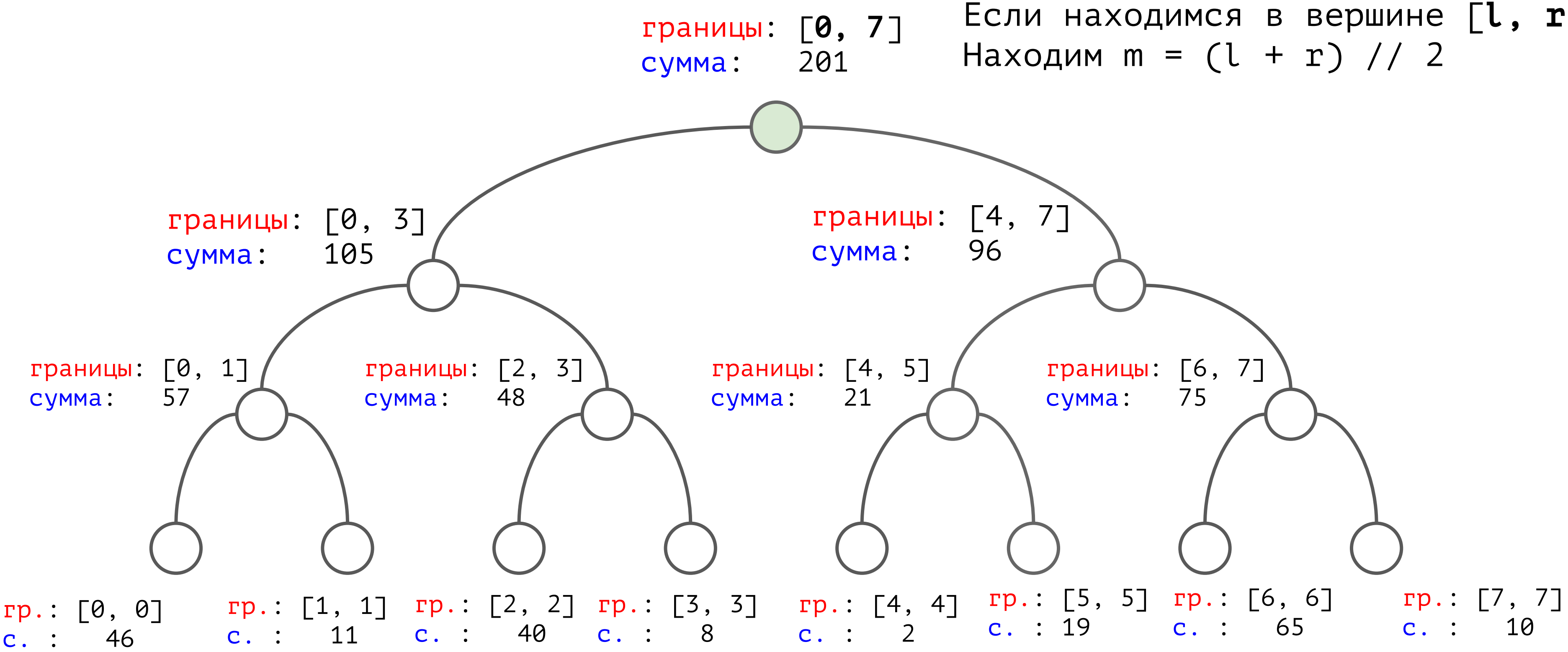
Если находимся в вершине [**l**, **r**]



46 11 40 8 2 19 65 10

Как реализовать sum(l, r)?

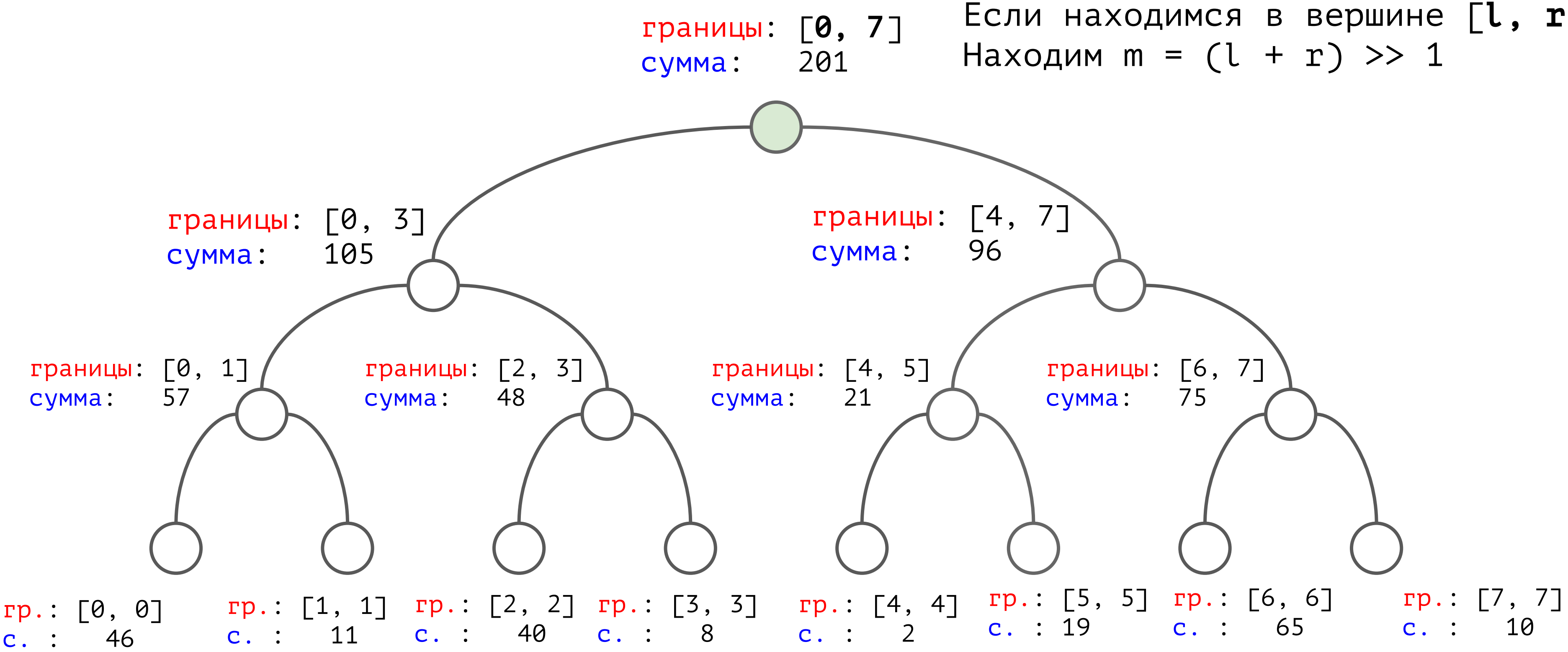
Если находимся в вершине **[l, r]**
Находим m = (l + r) // 2



46 11 40 8 2 19 65 10

Как реализовать sum(l, r)?

Если находимся в вершине [**l**, **r**]
Находим m = (l + r) >> 1

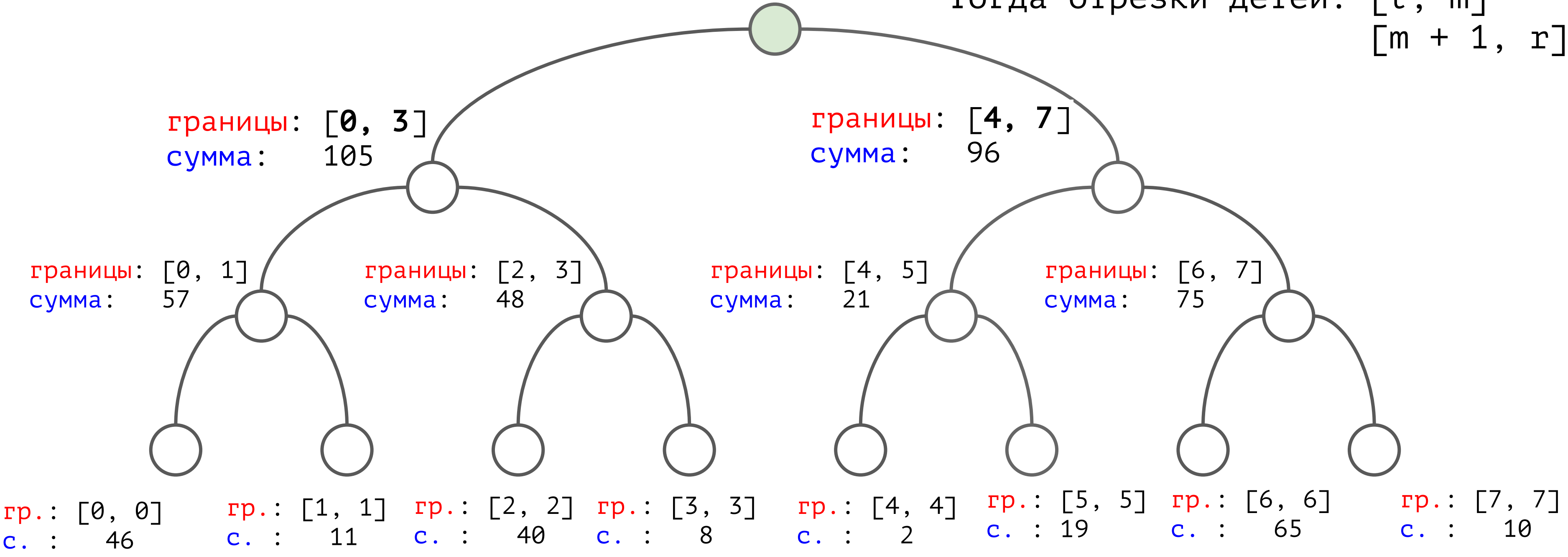


46 11 40 8 2 19 65 10

Как реализовать sum(l, r)?

Если находимся в вершине [**l**, **r**]
Находим m = (l + r) >> 1
Тогда отрезки детей: [l, m]
[m + 1, r]

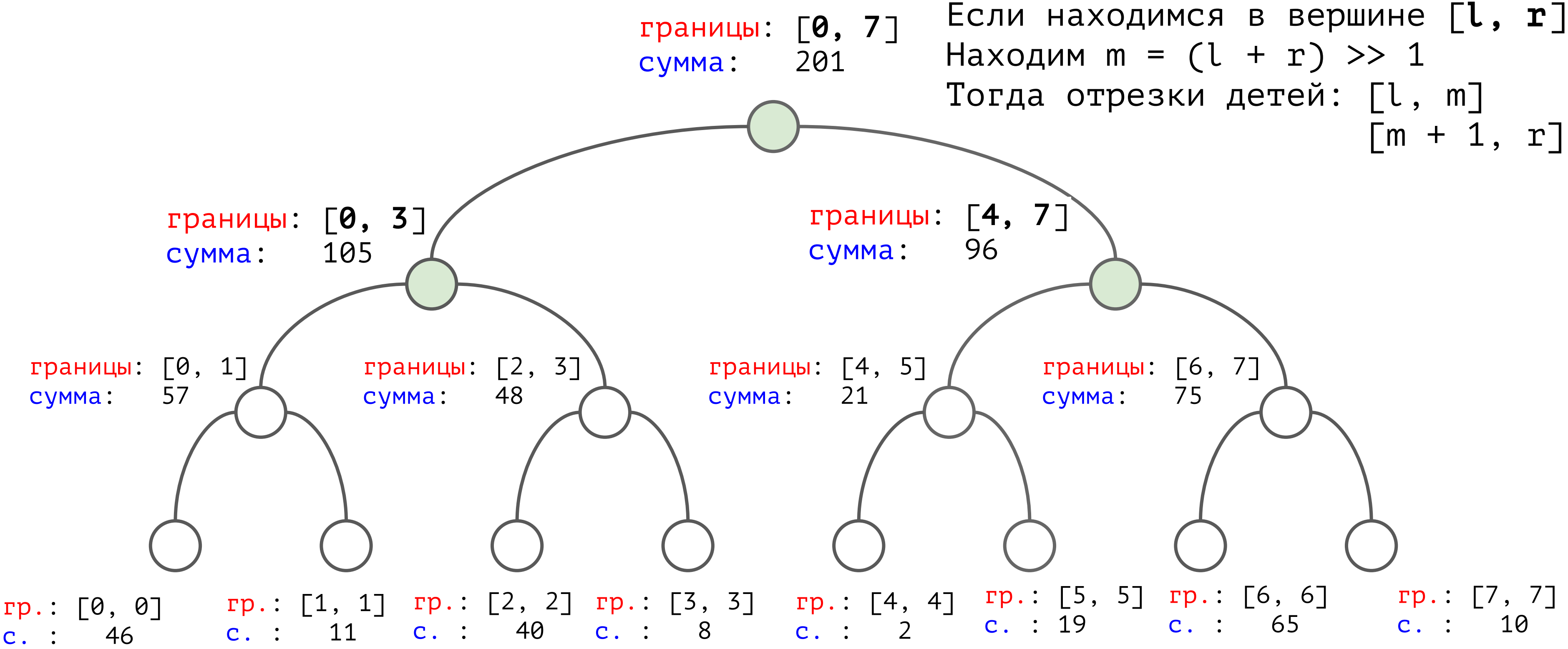
границы: [**0**, **7**]
сумма: 201



46 11 40 8 2 19 65 10

Как реализовать sum(l, r)?

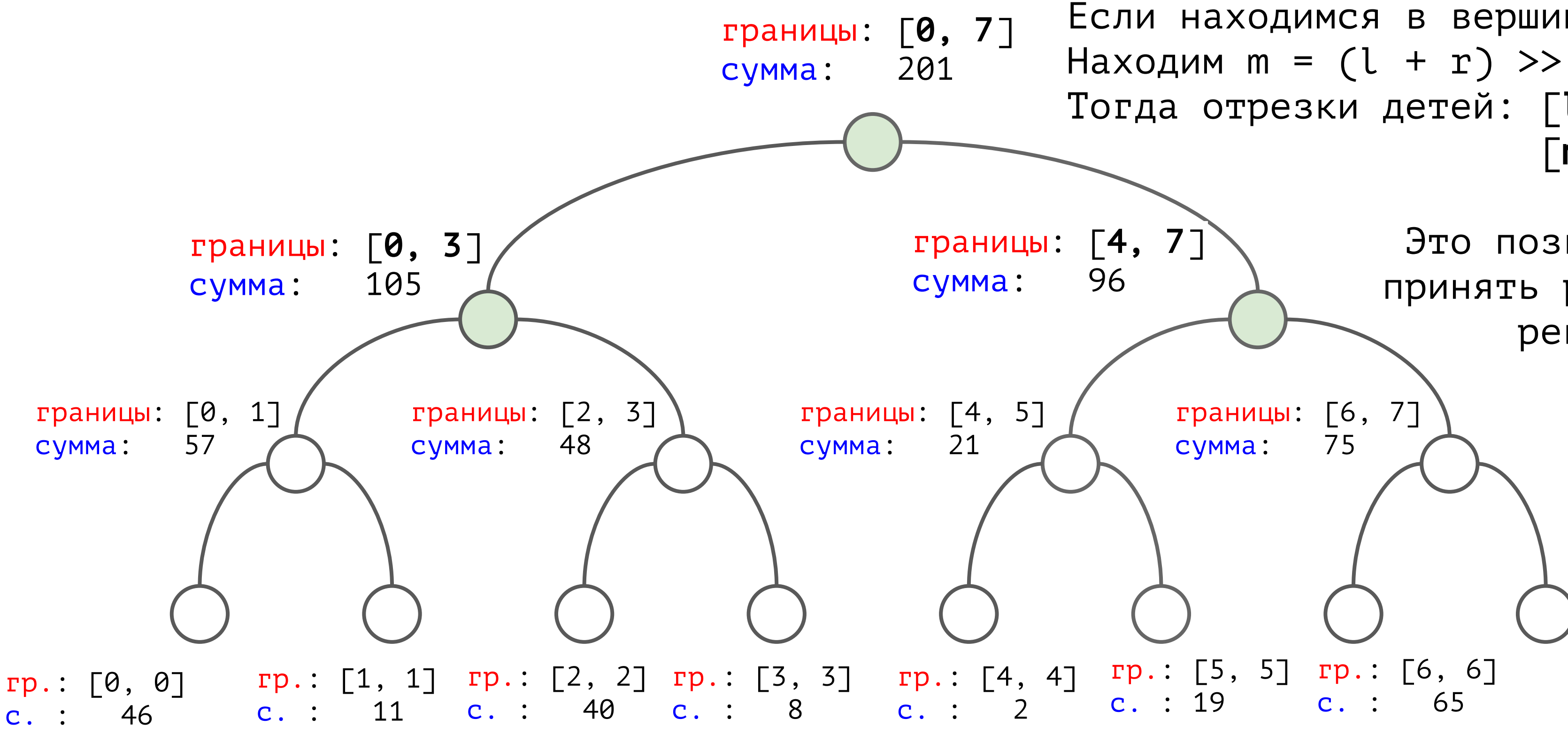
Если находимся в вершине [**l**, **r**]
Находим m = (l + r) >> 1
Тогда отрезки детей: [l, m]
[m + 1, r]



46 11 40 8 2 19 65 10

Как реализовать sum(l, r)?

Если находимся в вершине **[l, r]**
Находим m = (l + r) >> 1
Тогда отрезки детей: **[l, m]**
[m + 1, r]

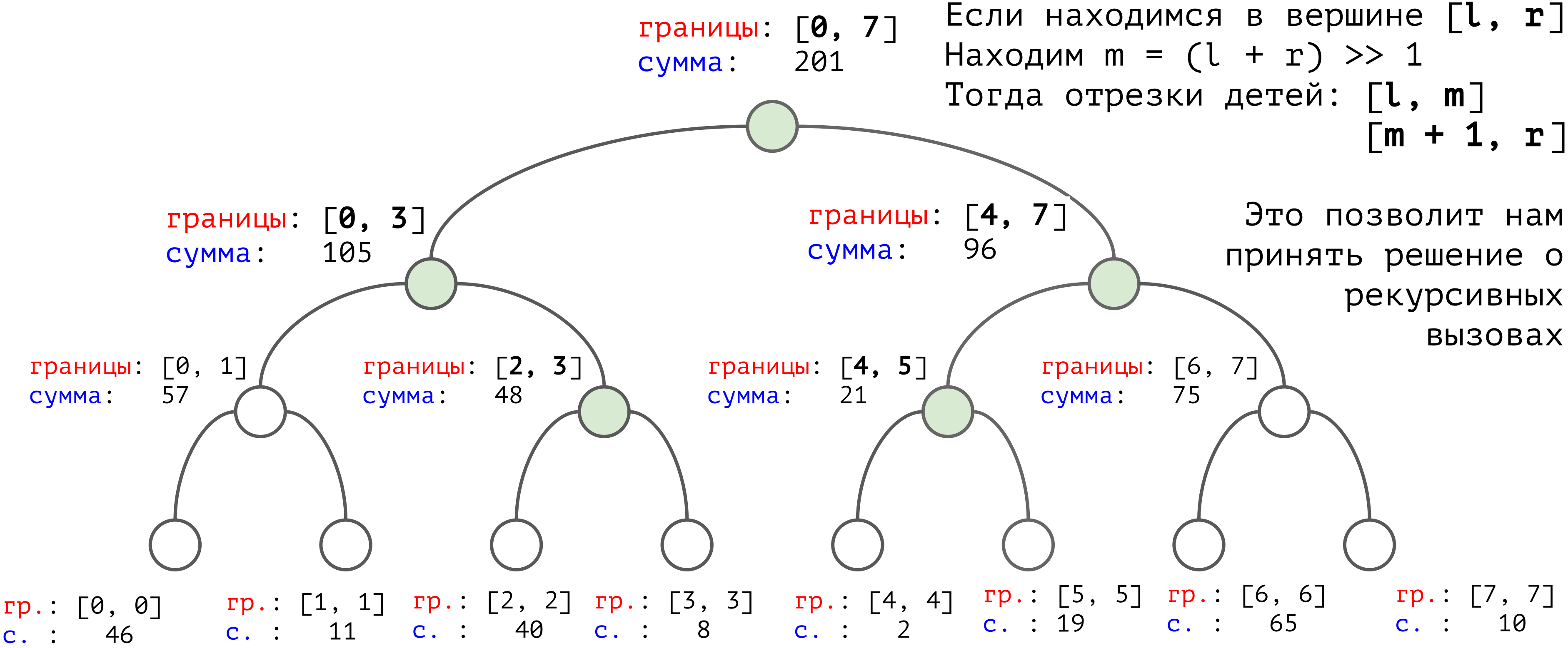


Это позволит нам принять решение о рекурсивных вызовах

46 11 40 8 2 19 65 10

Как реализовать sum(l, r)?

Если находимся в вершине **[l, r]**
Находим m = (l + r) >> 1
Тогда отрезки детей: **[l, m]**
[m + 1, r]



Это позволит нам принять решение о рекурсивных вызовах

Дерево отрезков: getSum

Пусть запросили [l, r], а мы находимся в вершине root.
У вершины есть свой отрезок: [root.lb, root.rb], сумма:
root.sum, и дети: root.left, root.right.

```
def getSum(root: Node, l, r: int) -> int:
```

Дерево отрезков: getSum

Пусть запросили [l, r], а мы находимся в вершине root.
У вершины есть свой отрезок: [root.lb, root.rb], сумма: root.sum, и дети: root.left, root.right.

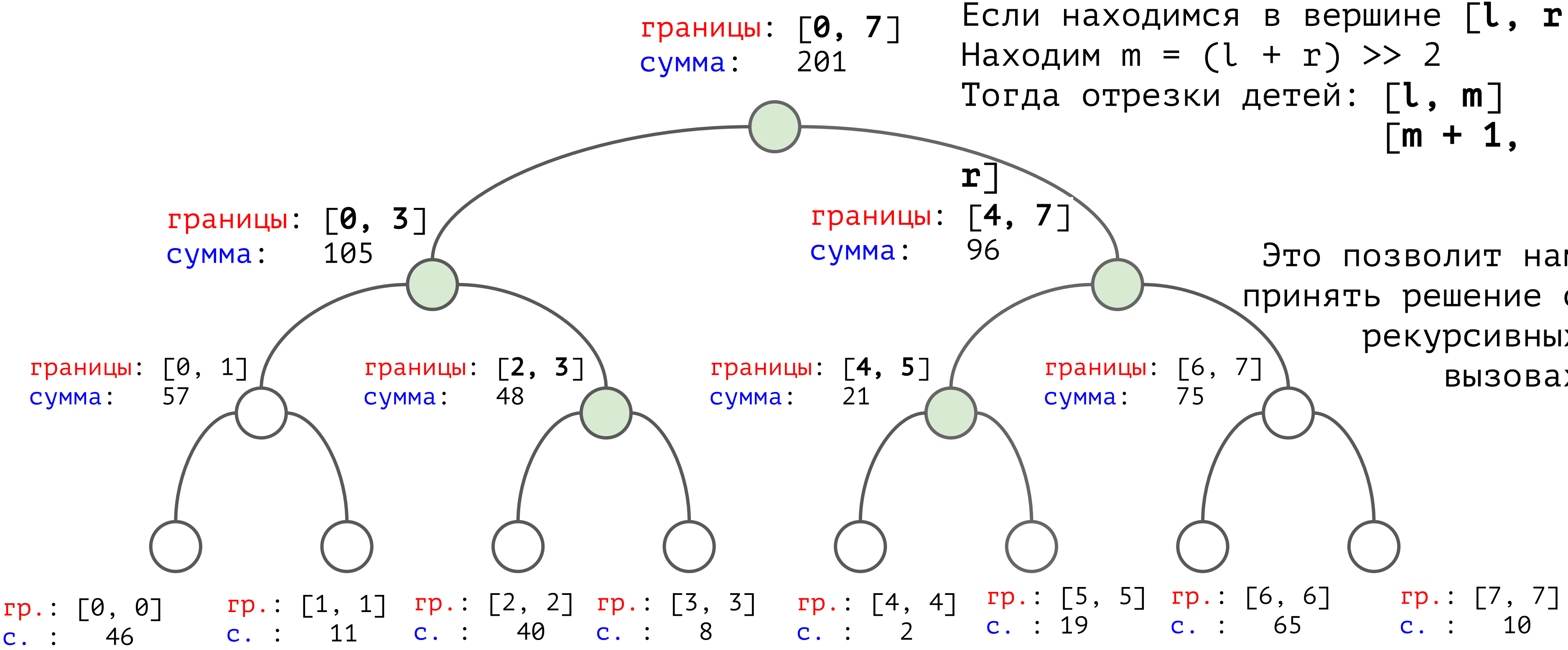
```
def getSum(root: Node, l, r: int) -> int:
  if l == root.lb and r == root.rb:
    return root.sum
```

46 11 40 8 2 19 65 10

Как реализовать sum(l, r)?

Если находимся в вершине **[l, r]**
Находим m = (l + r) >> 2
Тогда отрезки детей: **[l, m]**
[m + 1, r]

Это позволит нам принять решение о рекурсивных вызовах



Дерево отрезков: getSum

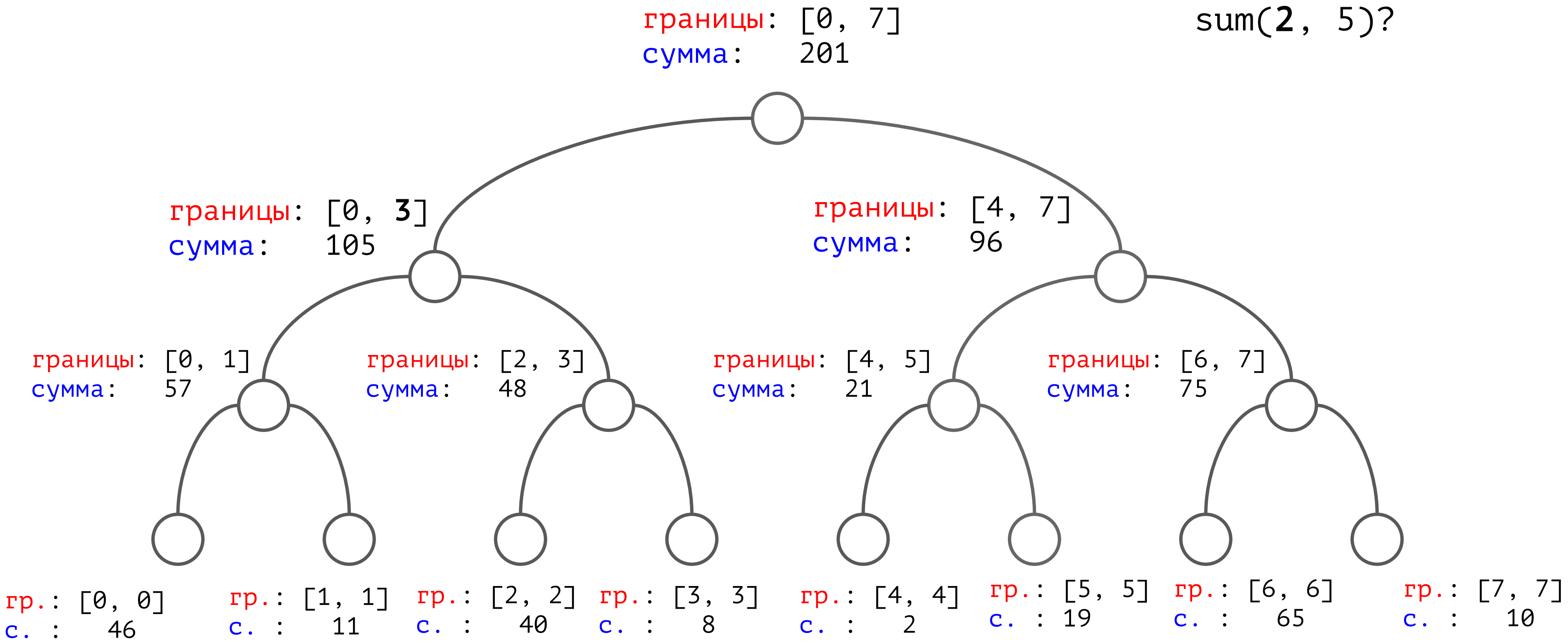
Пусть запросили [l, r], а мы находимся в вершине root.
У вершины есть свой отрезок: [root.lb, root.rb], сумма:
root.sum, и дети: root.left, root.right.

```
def getSum(root: Node, l, r: int) -> int:
  if l == root.lb and r == root.rb:
    return root.sum

  m = (root.lb + root.rb) >> 1
  res = 0









  return res
```

46 11 40 8 2 19 65 10

Как реализовать sum(l, r)?

sum(**2**, 5)?

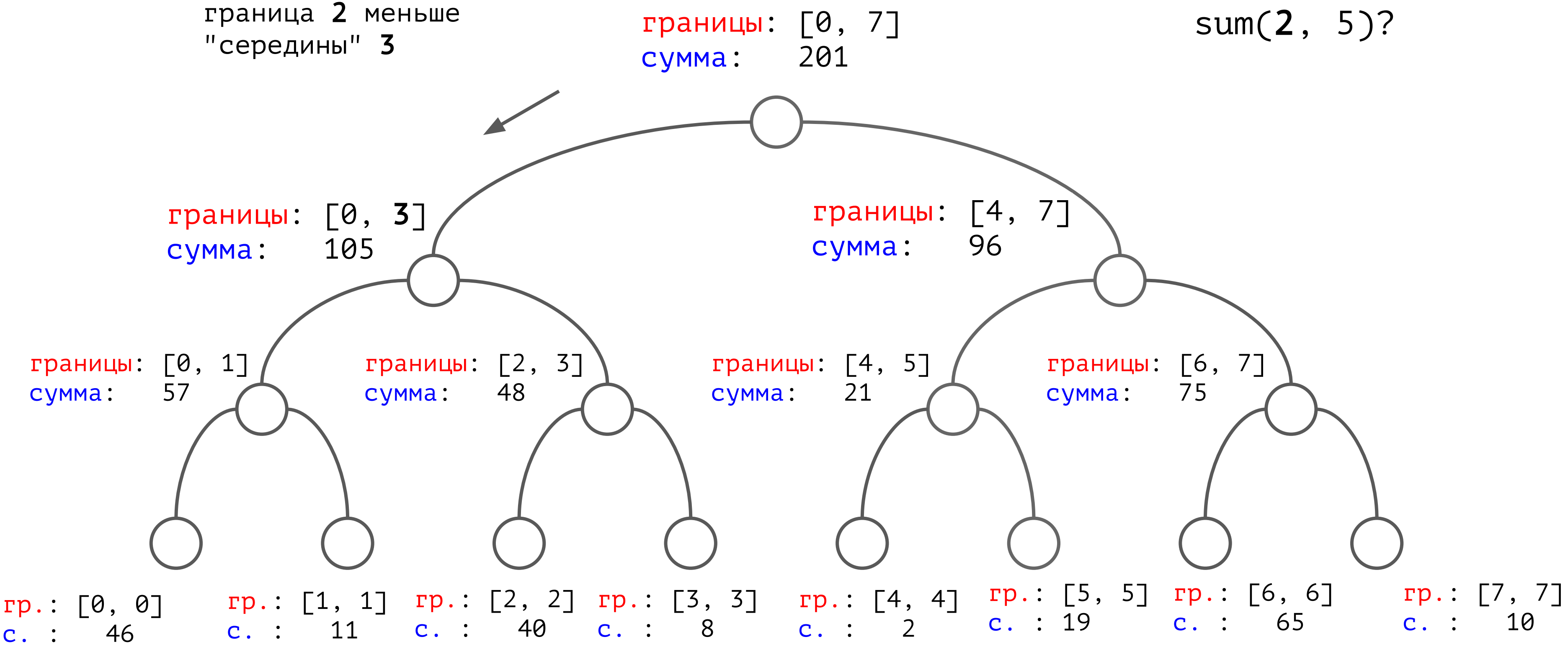


46 11 40 8 2 19 65 10

Как реализовать sum(l, r)?

sum(**2**, 5)?

запрашиваемая левая граница **2** меньше "середины" **3**



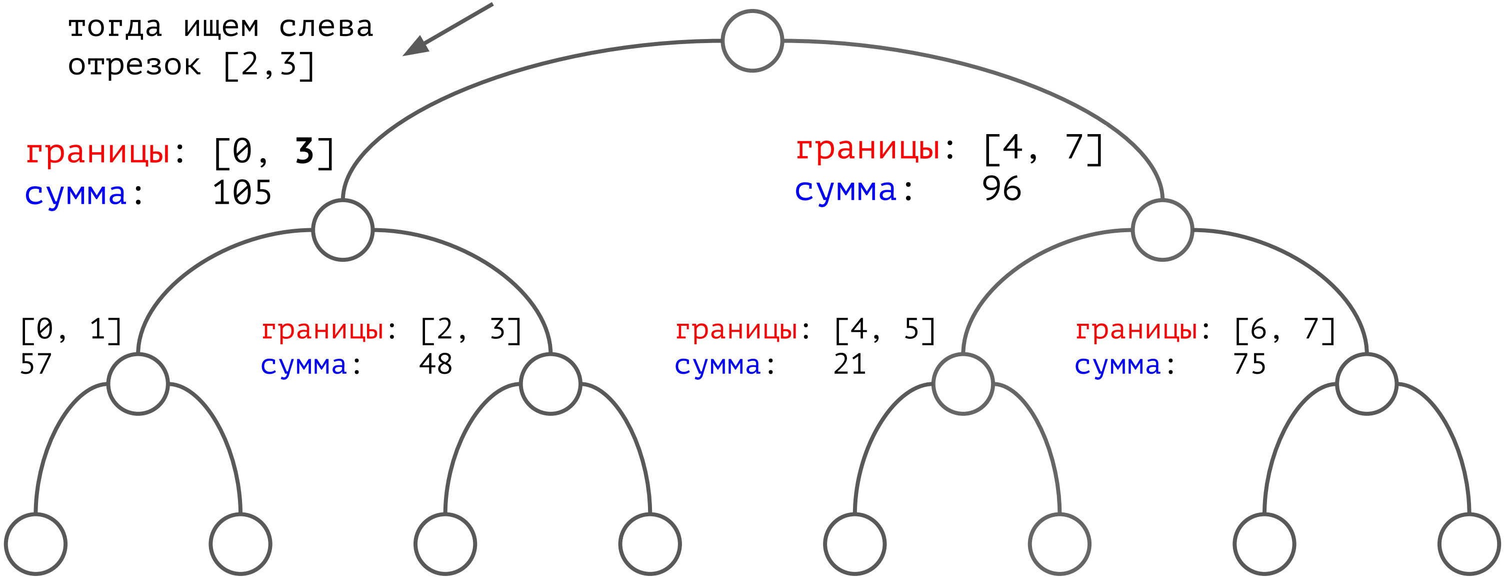
46 11 40 8 2 19 65 10

Как реализовать sum(l, r)?

sum(**2**, 5)?

запрашиваемая левая граница **2** меньше "середины" **3**

тогда ищем слева отрезок [2,3]



Дерево отрезков: getSum

Пусть запросили [l, r], а мы находимся в вершине root.
У вершины есть свой отрезок: [root.lb, root.rb], сумма: root.sum, и дети: root.left, root.right.

```
def getSum(root: Node, l, r: int) -> int:
  if l == root.lb and r == root.rb:
    return root.sum

  m = (root.lb + root.rb) >> 1
  res = 0

  if l <= m:
      res += getSum(root.left, l, min(r, m))




  return res
```

Дерево отрезков: getSum

Пусть запросили [l, r], а мы находимся в вершине root.
У вершины есть свой отрезок: [root.lb, root.rb], сумма:
root.sum, и дети: root.left, root.right.

```
def getSum(root: Node, l, r: int) -> int:
  if l == root.lb and r == root.rb:
    return root.sum

  m = (root.lb + root.rb) >> 1
  res = 0

  if l <= m:
      res += getSum(root.left, l, min(r, m))



  return res
```

min, чтобы понять, весь отрезок ищем слева или только до m

Дерево отрезков: getSum

Пусть запросили [l, r], а мы находимся в вершине root.
У вершины есть свой отрезок: [root.lb, root.rb], сумма: root.sum, и дети: root.left, root.right.

```
def getSum(root: Node, l, r: int) -> int:
  if l == root.lb and r == root.rb:
    return root.sum

  m = (root.lb + root.rb) >> 1
  res = 0

  if l <= m:
      res += getSum(root.left, l, min(r, m))

  if r >= m + 1:
      res += getSum(root.right, max(l, m + 1), r)

  return res
```

Дерево отрезков: getSum

Пусть запросили [l, r], а мы находимся в вершине root.
У вершины есть свой отрезок: [root.lb, root.rb], сумма: root.sum, и дети: root.left, root.right.

Сложность?

```
def getSum(root: Node, l, r: int) -> int:
  if l == root.lb and r == root.rb:
    return root.sum

  m = (root.lb + root.rb) >> 1
  res = 0

  if l <= m:
      res += getSum(root.left, l, min(r, m))

  if r >= m + 1:
      res += getSum(root.right, max(l, m + 1), r)

  return res
```

Дерево отрезков: getSum

Пусть запросили [l, r], а мы находимся в вершине root.
У вершины есть свой отрезок: [root.lb, root.rb], сумма: root.sum, и дети: root.left, root.right.

```
def getSum(root: Node, l, r: int) -> int:
  if l == root.lb and r == root.rb:
    return root.sum

  m = (root.lb + root.rb) >> 1
  res = 0

  if l <= m:
      res += getSum(root.left, l, min(r, m))

  if r >= m + 1:
      res += getSum(root.right, max(l, m + 1), r)

  return res
```

Сложность?

Кажется, что логарифм, но мы ведь можем делать и по два рекурсивных вызова!

Дерево отрезков: getSum

Утверждение: во время исполнения getSum(root, l, r) на каждом уровне дерева отрезков рассматривается не больше 4-ех отрезков.

Дерево отрезков: getSum

Утверждение: во время исполнения getSum(root, l, r) на каждом уровне дерева отрезков рассматривается не больше 4-ех отрезков.

Следствие: тогда сложность getSum составляет O(logN), т.к. высота дерева отрезков - logN.

Дерево отрезков: getSum

Утверждение: во время исполнения getSum(root, l, r) на каждом уровне дерева отрезков рассматривается не больше 4-ех отрезков.

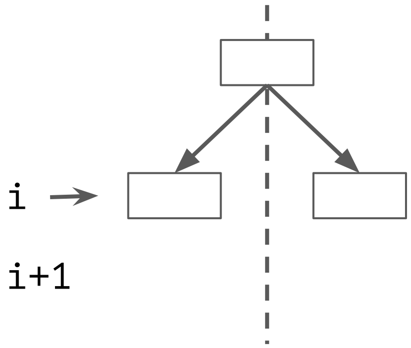
Док-во: индукция по номеру уровня. На первом уровне есть только корень, поэтому рассматривается одна вершина, что меньше 4.

Дерево отрезков: getSum

Утверждение: во время исполнения getSum(root, l, r) на каждом уровне дерева отрезков рассматривается не больше 4-ех отрезков.

Док-во: индукция по номеру уровня. На первом уровне есть только корень, поэтому рассматривается одна вершина, что меньше 4.

Шаг: пусть на i-ом уровне мы рассматриваем <= 4 вершин. Пусть рассматривается 1 или 2 вершины: из каждого сделаем не больше двух рекурсивных вызовов => на уровне i + 1 будет рассмотрено <= 4 вершин.

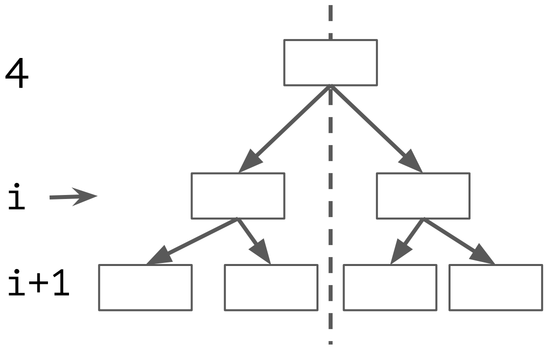


Дерево отрезков: getSum

Утверждение: во время исполнения getSum(root, l, r) на каждом уровне дерева отрезков рассматривается не больше 4-ех отрезков.

Док-во: индукция по номеру уровня. На первом уровне есть только корень, поэтому рассматривается одна вершина, что меньше 4.

Шаг: пусть на i-ом уровне мы рассматриваем <= 4 вершин. Пусть рассматривается 1 или 2 вершины: из каждого сделаем не больше двух рекурсивных вызовов => на уровне i + 1 будет рассмотрено <= 4 вершин.

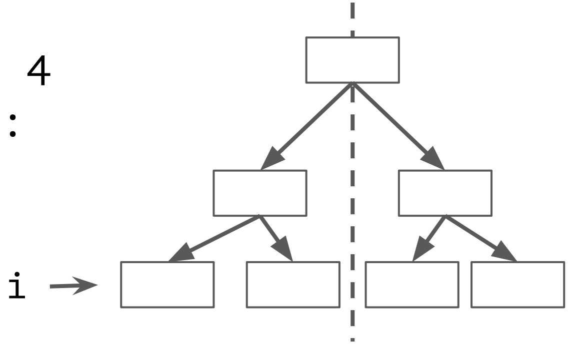


Дерево отрезков: getSum

Утверждение: во время исполнения getSum(root, l, r) на каждом уровне дерева отрезков рассматривается не больше 4-ех отрезков.

Док-во: индукция по номеру уровня. На первом уровне есть только корень, поэтому рассматривается одна вершина, что меньше 4.

Шаг: пусть на i-ом уровне мы рассматриваем <= 4 вершин. Пусть рассматривается 1 или 2 вершины: из каждого сделаем не больше двух рекурсивных вызовов => на уровне i + 1 будет рассмотрено <= 4 вершин.



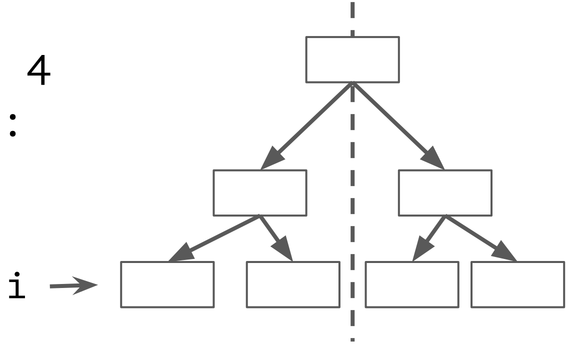
Рассмотрим случай, когда на i-ом уровне 4 вызова.

Дерево отрезков: getSum

Утверждение: во время исполнения getSum(root, l, r) на каждом уровне дерева отрезков рассматривается не больше 4-ех отрезков.

Док-во: индукция по номеру уровня. На первом уровне есть только корень, поэтому рассматривается одна вершина, что меньше 4.

Шаг: пусть на i-ом уровне мы рассматриваем <= 4 вершин. Пусть рассматривается 1 или 2 вершины: из каждого сделаем не больше двух рекурсивных вызовов => на уровне i + 1 будет рассмотрено <= 4 вершин.



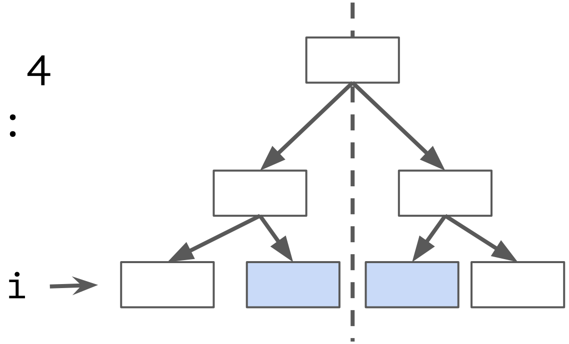
Рассмотрим случай, когда на i-ом уровне 4 вызова. Мы ищем непрерывный отрезок в дереве.

Дерево отрезков: getSum

Утверждение: во время исполнения getSum(root, l, r) на каждом уровне дерева отрезков рассматривается не больше 4-ех отрезков.

Док-во: индукция по номеру уровня. На первом уровне есть только корень, поэтому рассматривается одна вершина, что меньше 4.

Шаг: пусть на i-ом уровне мы рассматриваем <= 4 вершин. Пусть рассматривается 1 или 2 вершины: из каждого сделаем не больше двух рекурсивных вызовов => на уровне i + 1 будет рассмотрено <= 4 вершин.



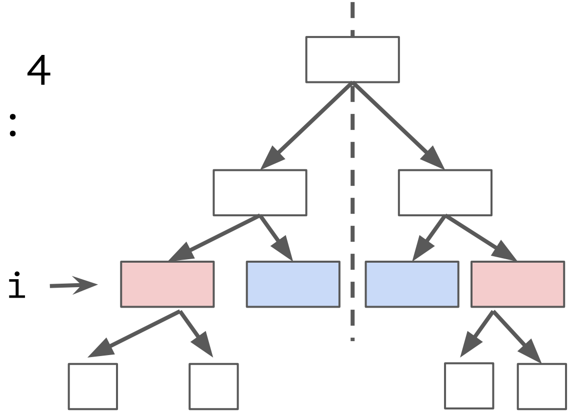
Рассмотрим случай, когда на i-ом уровне 4 вызова. Мы ищем непрерывный отрезок в дереве. Это значит, что средние отрезки войдут целиком, никакого ветвления там не будет!

Дерево отрезков: getSum

Утверждение: во время исполнения getSum(root, l, r) на каждом уровне дерева отрезков рассматривается не больше 4-ех отрезков.

Док-во: индукция по номеру уровня. На первом уровне есть только корень, поэтому рассматривается одна вершина, что меньше 4.

Шаг: пусть на i-ом уровне мы рассматриваем <= 4 вершин. Пусть рассматривается 1 или 2 вершины: из каждого сделаем не больше двух рекурсивных вызовов => на уровне i + 1 будет рассмотрено <= 4 вершин.



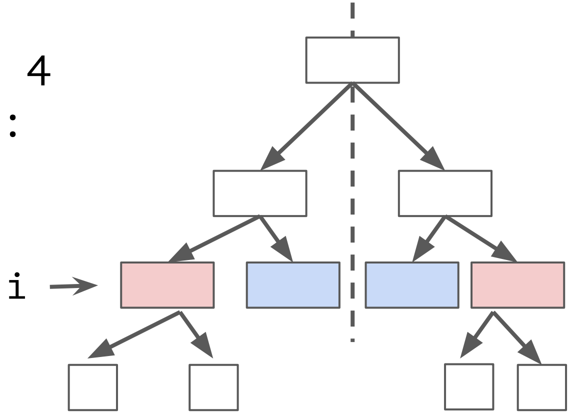
Рассмотрим случай, когда на i-ом уровне 4 вызова. Мы ищем непрерывный отрезок в дереве. Это значит, что средние отрезки войдут целиком, никакого ветвления там не будет! Рекурсивные вызовы будут только у крайних отрезков.

Дерево отрезков: getSum

Утверждение: во время исполнения getSum(root, l, r) на каждом уровне дерева отрезков рассматривается не больше 4-ех отрезков.

Док-во: индукция по номеру уровня. На первом уровне есть только корень, поэтому рассматривается одна вершина, что меньше 4.

Шаг: пусть на i-ом уровне мы рассматриваем <= 4 вершин. Пусть рассматривается 1 или 2 вершины: из каждого сделаем не больше двух рекурсивных вызовов => на уровне i + 1 будет рассмотрено <= 4 вершин.



Рассмотрим случай, когда на i-ом уровне 4 вызова. Мы ищем непрерывный отрезок в дереве. Это значит, что средние отрезки войдут целиком, никакого ветвления там не будет! Рекурсивные вызовы будут только у крайних отрезков. Поэтому на i+1 уровне <= 4 вершин рассмотрим. □

Дерево отрезков

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать новые операции следующего вида:

1. update(pos, val) - обновляем значение элемента a_pos

2. sum(l, r) - найти сумму элементов: a_l + … _ a_r

Дерево отрезков

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать новые операции следующего вида:

1. update(pos, val) - обновляем значение элемента a_pos
2. sum(l, r) - найти сумму элементов: a_l + … _ a_r

Получилось все сделать за O(logN) в худшем и за O(N) памяти.

Дерево отрезков

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать новые операции следующего вида:

1. update(pos, val) - обновляем значение элемента a_pos
2. sum(l, r) - найти сумму элементов: a_l + … _ a_r

А только ли для суммы это работает? 🤔

Дерево отрезков

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать новые операции следующего вида:

1. update(pos, val) - обновляем значение элемента a_pos
2. sum(l, r) - найти сумму элементов: a_l + … _ a_r

А только ли для суммы это работает? 🤔

Легко заменяется на ассоциативные операции с нейтральным элементом: умножение, умножение матриц, нахождение минимума, нахождение gcd и т.д. (т.е. на моноиде, узнаете на ФП!)

Дерево отрезков: детали реализации

Дерево отрезков: детали реализации

Совершенно необязательно хранить именно дерево, можно обойтись массивом.

Дерево отрезков: детали реализации

Совершенно необязательно хранить именно дерево, можно обойтись массивом.

- Заводим массив t[4*N] (так точно хватит)

Дерево отрезков: детали реализации

Совершенно необязательно хранить именно дерево, можно обойтись массивом.

- Заводим массив t[4*N] (так точно хватит). t[i] хранит сумму в соответствующей вершине. t[2*i] и t[2*i + 1] - дети этой вершины.

Дерево отрезков: детали реализации

Совершенно необязательно хранить именно дерево, можно обойтись массивом.

- Заводим массив t[4*N] (так точно хватит). t[i] хранит сумму в соответствующей вершине. t[2*i] и t[2*i + 1] - дети этой вершины.

- Границы вообще не хранятся в явном виде. Во все операции они передаются в качестве аргументов.

Дерево отрезков: детали реализации

```
t = [0] * (4*N)

def build(a: int[], v, tl, tr: int):
```

Дерево отрезков: детали реализации

```
t = [0] * (4*N)


def build(a: int[], v, tl, tr: int):
  if tl == tr:
    t[v] = a[tl]
    return
```

Дерево отрезков: детали реализации

```
t = [0] * (4*N)


def build(a: int[], v, tl, tr: int):
  if tl == tr:
    t[v] = a[tl]
    return

  tm = (tl + tr) >> 1
  build(a, v*2, tl, tm)
  build(a, v*2 + 1, tm + 1, tr)
```

Дерево отрезков: детали реализации

Построение за O(N)

```
t = [0] * (4*N)

def build(a: int[], v, tl, tr: int):
  if tl == tr:
    t[v] = a[tl]
    return

  tm = (tl + tr) >> 1
  build(a, v*2, tl, tm)
  build(a, v*2 + 1, tm + 1, tr)

  t[v] = t[v*2] + t[v*2 + 1]
```

Дерево отрезков: детали реализации

```
def getSum(root: Node, l, r: int) -> int:
  if l == root.lb and r == root.rb:
    return root.sum

  m = (root.lb + root.rb) >> 1
  res = 0

  if l <= m:
      res += getSum(root.left, l, min(r, m))

  if r >= m + 1:
      res += getSum(root.right, max(l, m + 1), r)

  return res
```

Дерево отрезков: детали реализации

```
t = [0] * (4*N)
build(a, 1, 0, N - 1)


def getSum(v, tl, tr, l, r: int) -> int:
  if l == tl and r == tr:
    return t[v]

  tm = (tl + tr) >> 1
  res = 0

  if l <= tm:
      res += getSum(v*2, tl, tm, l, min(r, tm))

  if r >= tm + 1:
      res += getSum(v*2 + 1, tm + 1, tr, max(l, tm + 1), r)

  return res
```

Дерево отрезков: детали реализации

```
t = [0] * (4*N)
build(a, 1, 0, N - 1)


def getSum(v, tl, tr, l, r: int) -> int:
  if l == tl and r == tr:
    return t[v]

  tm = (tl + tr) >> 1
  res = 0

  if l <= tm:
      res += getSum(v*2, tl, tm, l, min(r, tm))

  if r >= tm + 1:
      res += getSum(v*2 + 1, tm + 1, tr, max(l, tm + 1), r)

  return res
```

аргументы tl/tr - замена полей вершин

Мини-задача #43 (1 балл)

https://leetcode.com/problems/range-sum-query-mutable

Решите задачу, используя базовое дерево отрезков!

Дерево отрезков: поиск нулей

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

$$a_1, a_2, a_3, \ldots, a_n$$

Необходимо поддержать новые операции следующего вида:

1. update(pos, val) - обновляем значение элемента a_pos
2. k_zero(l, r, k) - индекс k-ого нуля на отрезке [l, r]

Дерево отрезков: поиск нулей

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать новые операции следующего вида:

1. update(pos, val) - обновляем значение элемента a_pos
2. k_zero(l, r, k) - индекс k-ого нуля на отрезке [l, r]

4 23 1 2 5 0 21 [24 0 2 5 0 23] 515 0

Дерево отрезков: поиск нулей

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать новые операции следующего вида:

1. update(pos, val) - обновляем значение элемента a_pos
2. k_zero(l, r, k) - индекс k-ого нуля на отрезке [l, r]

4 23 1 2 5 0 21 [24 0 2 5 0 23] 515 0   k_zero(7, 12, 2)

Дерево отрезков: поиск нулей

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать новые операции следующего вида:

1. update(pos, val) - обновляем значение элемента a_pos
2. k_zero(l, r, k) - индекс k-ого нуля на отрезке [l, r]

4 23 1 2 5 0 21 [24 0 2 5 0 23] 515 0   k_zero(7, 12, 2)

Как решать?

Дерево отрезков: поиск нулей

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать новые операции следующего вида:

1. update(pos, val) - обновляем значение элемента a_pos
2. k_zero(l, r, k) - индекс k-ого нуля на отрезке [l, r]

4 23 1 2 5 0 21 [24 0 2 5 0 23] 515 0 k_zero(7, 12, 2)

Как решать? Дерево отрезков! Храним теперь количество нулей на контролируемом отрезке.

Дерево отрезков: поиск нулей

k_zero(l, r, k) - индекс k-ого нуля на отрезке [l, r]

4 23 1 2 5 0 21 24 0 2 5 0 23 515 0 k_zero(7, 12, 2)

Заметим:

1) Нам достаточно научиться искать k-ый ноль, начиная с l.

Дерево отрезков: поиск нулей

k_zero(l, r, k) - индекс k-ого нуля на отрезке [l, r]

4 23 1 2 5 0 21 24 0 2 5 0 23 515 0 k_zero(7, 12, 2)

Заметим:

1) Нам достаточно научиться искать k-ый ноль, начиная с l.
(если он оказался правее r => не нашли)

Дерево отрезков: поиск нулей

k_zero(l, r, k) - индекс k-ого нуля на отрезке [l, r]

4 23 1 2 5 0 21 24 0 2 5 0 23 515 0   k_zero(7, 12, 2)

Заметим:

1) Нам достаточно научиться искать k-ый ноль, начиная с l.
   (если он оказался правее r => не нашли)

2) Нам достаточно уметь k-ый ноль с самого начала массива!

Дерево отрезков: поиск нулей

k_zero(l, r, k) - индекс k-ого нуля на отрезке [l, r]

4 23 1 2 5 0 21 24 0 2 5 0 23 515 0 k_zero(7, 12, 2)

Заметим:

1) Нам достаточно научиться искать k-ый ноль, начиная с l.
   (если он оказался правее r => не нашли)

2) Нам достаточно уметь k-ый ноль с самого начала массива!

   4 23 1 2 5 0 [21 24 0 2 5 0] 23 515 0

Дерево отрезков: поиск нулей

k_zero(l, r, k) - индекс k-ого нуля на отрезке [l, r]

4 23 1 2 5 0 21 24 0 2 5 0 23 515 0   k_zero(7, 12, 2)

Заметим:

1) Нам достаточно научиться искать k-ый ноль, начиная с l.
   (если он оказался правее r => не нашли)

2) Нам достаточно уметь k-ый ноль с самого начала массива!

$\underbrace{4\ 23\ 1\ 2\ 5\ 0}_{\text{Пусть здесь p нулей (нашли деревом отрезков)}}\ [21\ 24\ 0\ 2\ 5\ 0]\ 23\ 515\ 0$

Дерево отрезков: поиск нулей

k_zero(l, r, k) - индекс k-ого нуля на отрезке [l, r]

4 23 1 2 5 0 21 24 0 2 5 0 23 515 0   k_zero(7, 12, 2)

Заметим:

1) Нам достаточно научиться искать k-ый ноль, начиная с l.
   (если он оказался правее r => не нашли)

2) Нам достаточно уметь k-ый ноль с самого начала массива!

$\underbrace{4\ 23\ 1\ 2\ 5\ 0}_{}$ [21 24 0 2 5 0] 23 515 0

Пусть здесь p нулей
(нашли деревом отрезков)

Тогда чтобы найти второй ноль здесь, нужно
найти p + 2 ноль с начала массива

Дерево отрезков: поиск нулей

k_zero(l, r, k) - индекс k-ого нуля на отрезке [l, r]

4 23 1 2 5 0 21 24 0 2 5 0 23 515 0   k_zero(7, 12, 2)

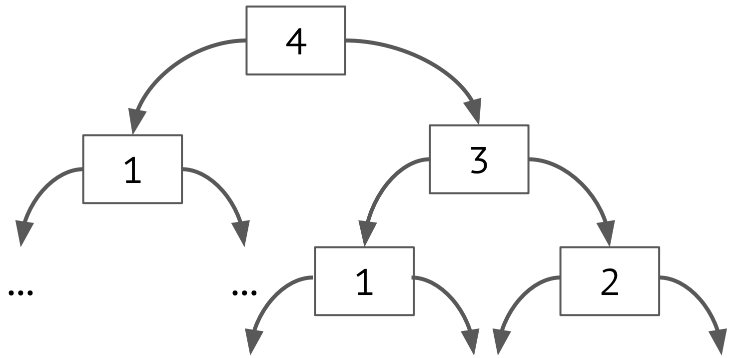
Как найти k-ый ноль с начала массива?

Дерево отрезков: поиск нулей

k_zero(l, r, k) - индекс k-ого нуля на отрезке [l, r]

4 23 1 2 5 0 21 24 0 2 5 0 23 515 0   k_zero(7, 12, 2)

Как найти k-ый ноль с начала массива?



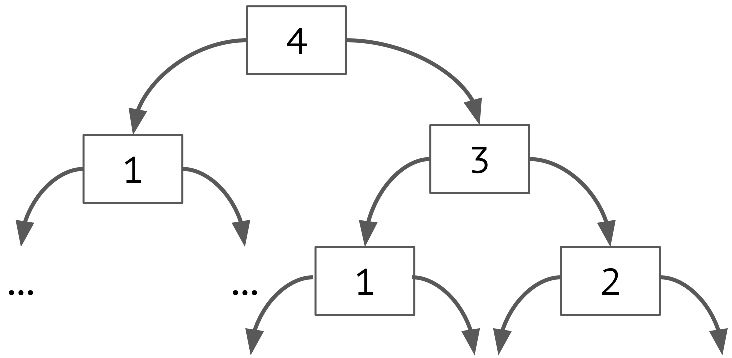
Дерево отрезков, где в вершинах храним количество нулей

Дерево отрезков: поиск нулей

k_zero(l, r, k) - индекс k-ого нуля на отрезке [l, r]

4 23 1 2 5 0 21 24 0 2 5 0 23 515 0 k_zero(7, 12, 2)

Как найти k-ый ноль с начала массива?



Когда ищем k-ый ноль рекурсивно идем либо влево (если там больше нулей, чем мы ищем), либо вправо (но искать начинаем k-{количество нулей в левом} по порядку ноль)

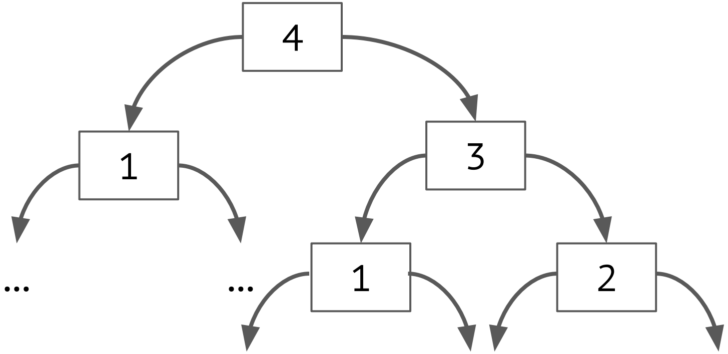
Дерево отрезков, где в вершинах храним количество нулей

Дерево отрезков: поиск нулей

k_zero(l, r, k) - индекс k-ого нуля на отрезке [l, r]

4 23 1 2 5 0 21 24 0 2 5 0 23 515 0   k_zero(7, 12, 2)

Как найти k-ый ноль с начала массива?



Когда ищем k-ый ноль рекурсивно идем либо влево (если там больше нулей, чем мы ищем), либо вправо (но искать начинаем k-{количество нулей в левом} по порядку ноль). Сложность?

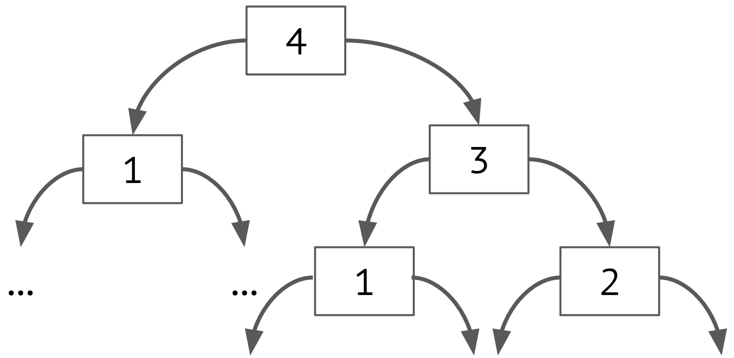
Дерево отрезков, где в вершинах храним количество нулей

Дерево отрезков: поиск нулей

k_zero(l, r, k) - индекс k-ого нуля на отрезке [l, r]

4 23 1 2 5 0 21 24 0 2 5 0 23 515 0 k_zero(7, 12, 2)

Как найти k-ый ноль с начала массива?



Когда ищем k-ый ноль рекурсивно идем либо влево (если там больше нулей, чем мы ищем), либо вправо (но искать начинаем k-{количество нулей в левом} по порядку ноль). Сложность? O(logN)!

Дерево отрезков, где в вершинах храним количество нулей

Дерево отрезков: присваивание на диапазоне

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Как решать?

Дерево отрезков: присваивание на диапазоне

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r

2. assign(l, r, x) - заменить весь отрезок на x

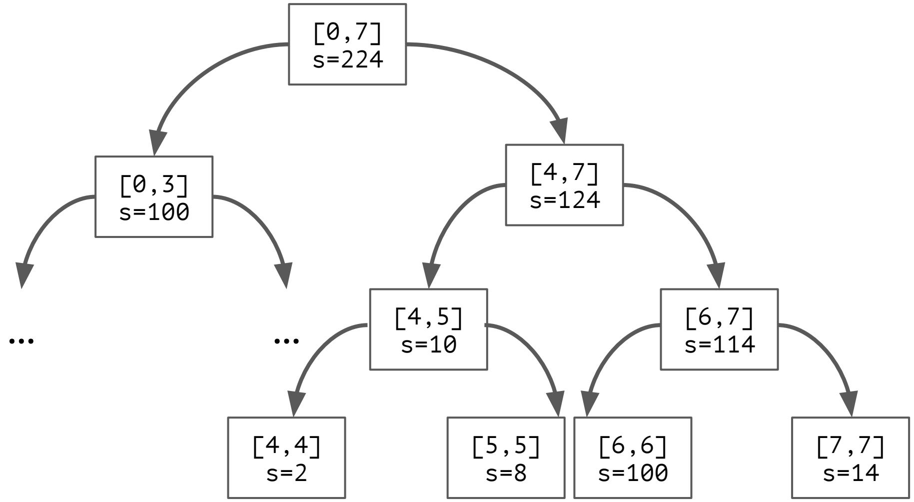
Как решать? Можно просто (r-l) раз сделать update, но тогда сложность будет O(N*logN)

Дерево отрезков: присваивание на диапазоне

Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

assign(4, 7, 9)



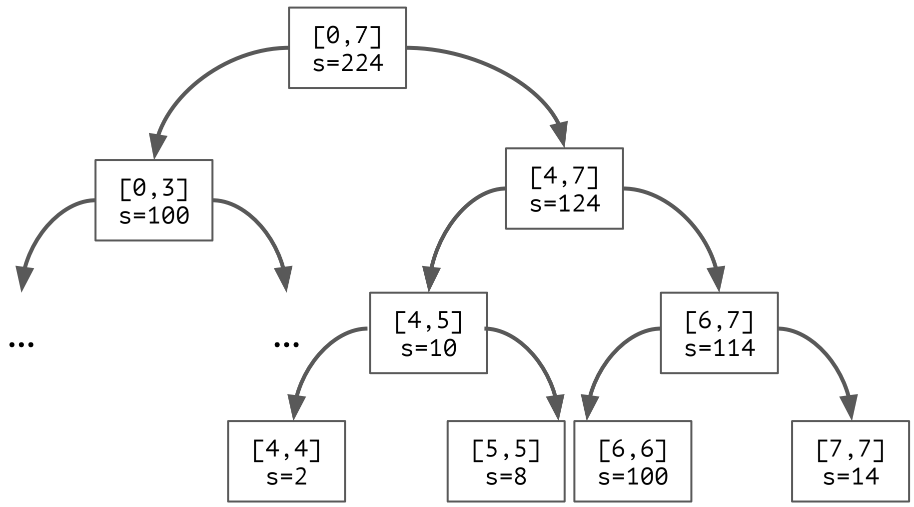
Дерево отрезков: присваивание на диапазоне

Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

assign(4, 7, 9)

Что в целом то нужно сделать?



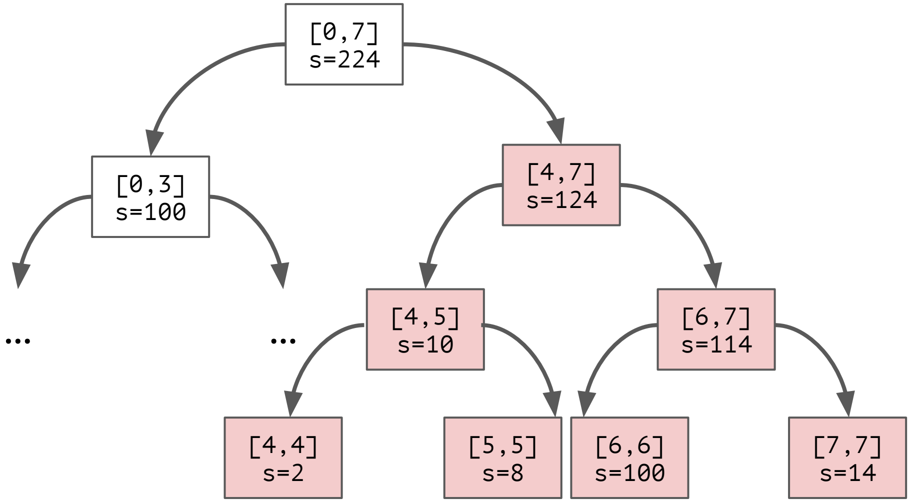
Дерево отрезков: присваивание на диапазоне

Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

assign(4, 7, 9)

Что в целом то нужно сделать? Обновить соответствующее поддерево (поддеревья)



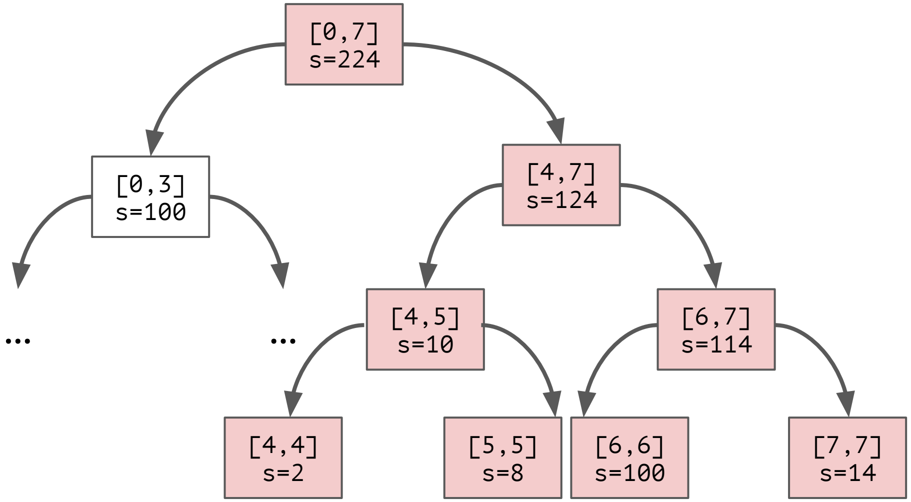
Дерево отрезков: присваивание на диапазоне

Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

assign(4, 7, 9)

Что в целом то нужно сделать? Обновить соответствующее поддерево (поддеревья)



Дерево отрезков: присваивание на диапазоне

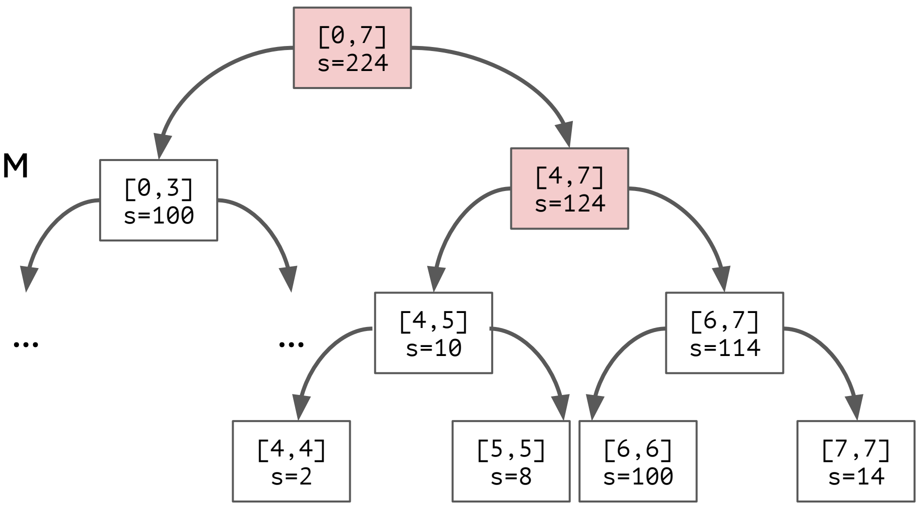
Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

В assign-е для начала обновим только точно интересные нам отрезки

assign(4, 7, 9)



Дерево отрезков: присваивание на диапазоне

Необходимо поддержать новые операции следующего вида:

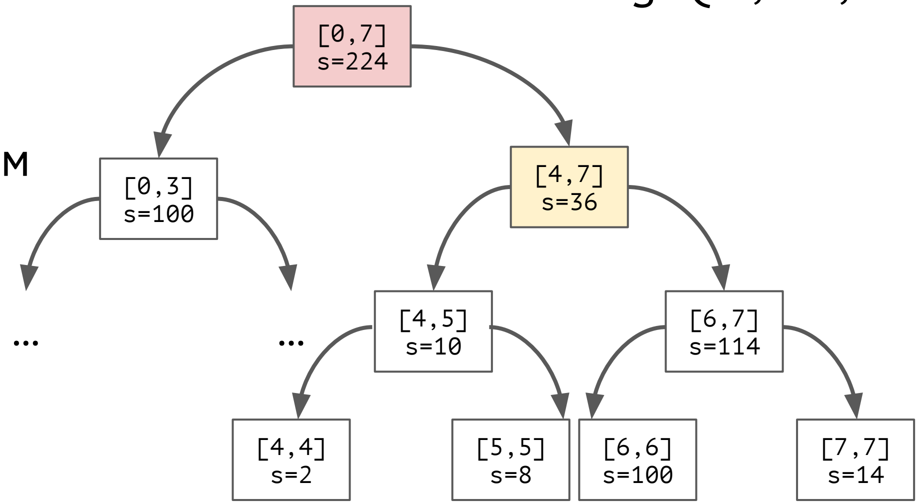
1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

В assign-е для начала обновим только точно интересные нам отрезки

В [4,7] сумма понятна - это (7- 4 + 1) * 9

assign(4, 7, 9)



Необходимо поддержать новые операции следующего вида:

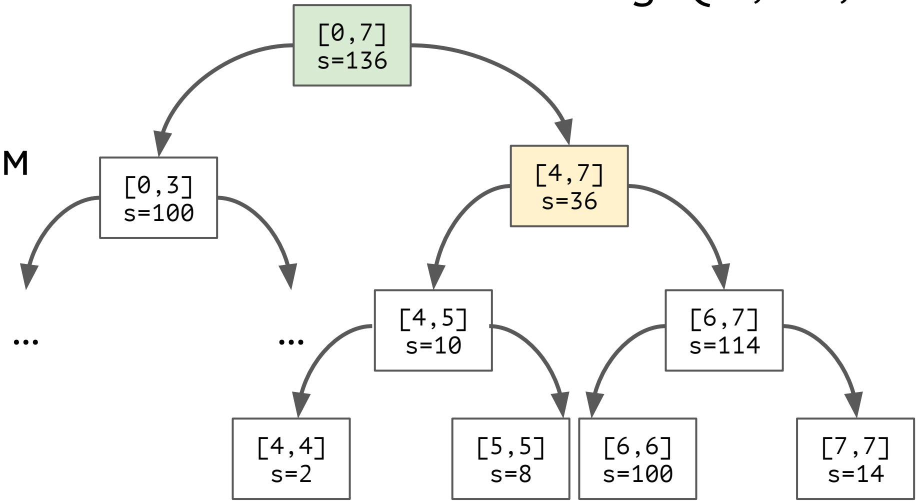
1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

В assign-е для начала обновим только точно интересные нам отрезки

В [4,7] сумма понятна - это (7- 4 + 1) * 9

assign(4, 7, 9)



Как обновить предка - тоже понятно, просто учтем новое значение в сыне.

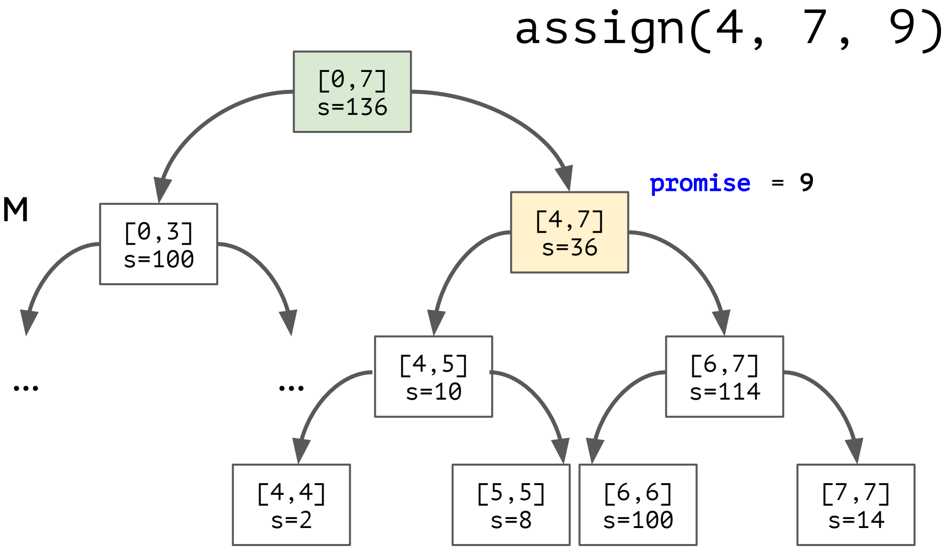
Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

В assign-е для начала обновим только точно интересные нам отрезки

В [4,7] сумма понятна - это (7- 4 + 1) * 9



Как обновить предка - тоже понятно, просто учтем новое значение в сыне. Осталось обновить сыновей. Сделаем это потом, пока просто запомним это желание.

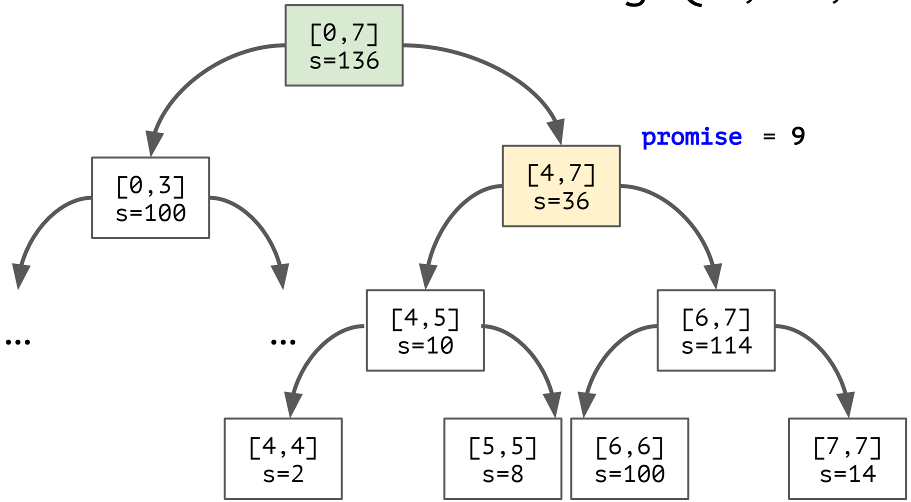
Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

А когда будем этот promise учитывать?

assign(4, 7, 9)



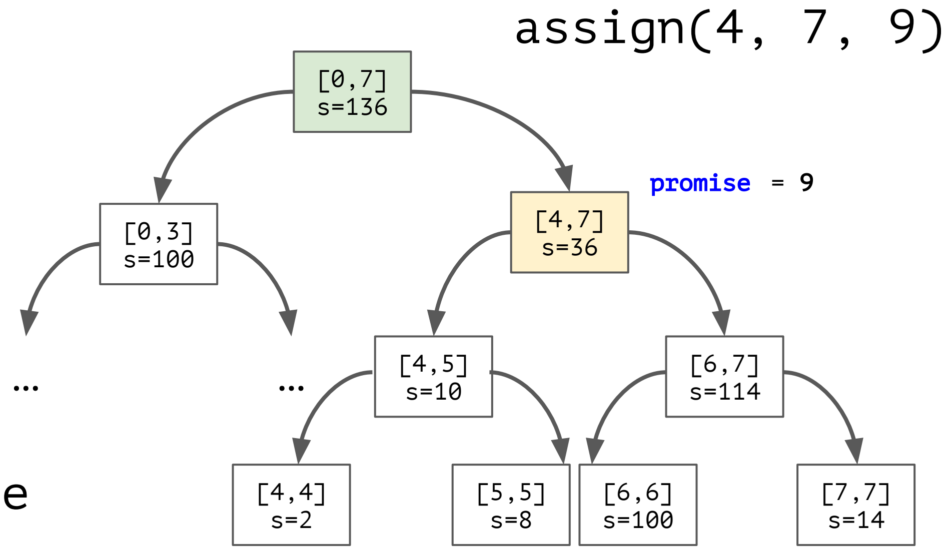
Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

А когда будем этот promise учитывать?

Когда считаем сумму! Если нам все равно спускаться от корня, то вот этот promise и протолкнем вниз.



Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

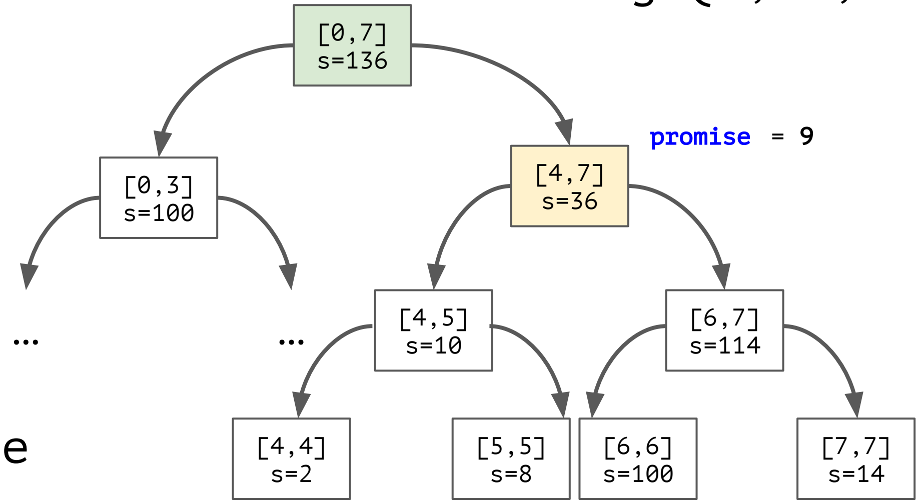
Идея: давайте обновлять не моментально, а лениво.

А когда будем этот promise учитывать?

Когда считаем сумму! Если нам все равно спускаться от корня, то вот этот promise и протолкнем вниз.

sum(6, 7)

assign(4, 7, 9)



Необходимо поддержать новые операции следующего вида:

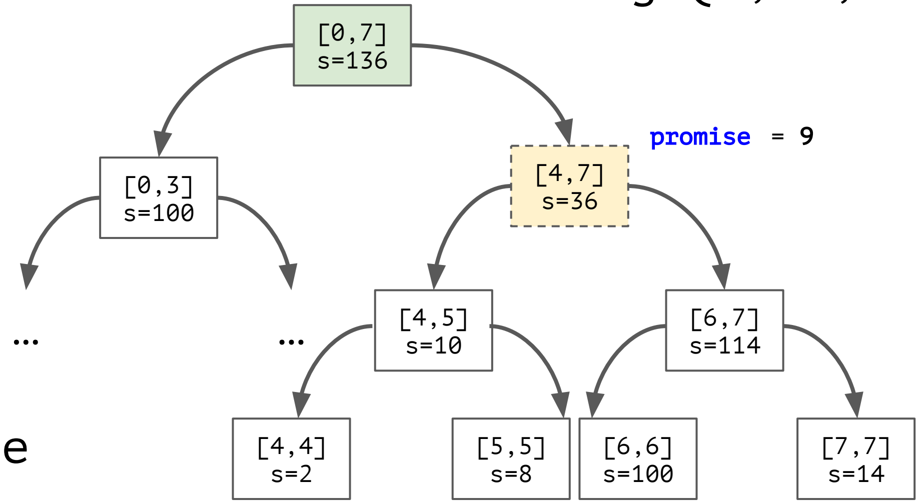
1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

А когда будем этот promise учитывать?

Когда считаем сумму! Если нам все равно спускаться от корня, то вот этот promise и протолкнем вниз.

assign(4, 7, 9)



sum(6, 7) -> дошли до вершины с promise ->

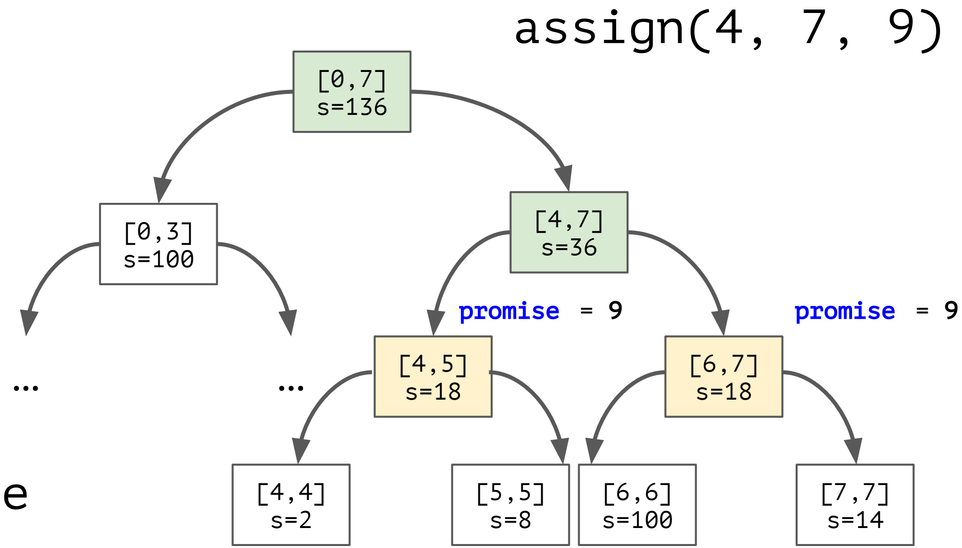
Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

А когда будем этот promise учитывать?

Когда считаем сумму! Если нам все равно спускаться от корня, то вот этот promise и протолкнем вниз.



sum(6, 7) -> дошли до вершины с promise -> проталкиваем в детей обещание

Необходимо поддержать новые операции следующего вида:

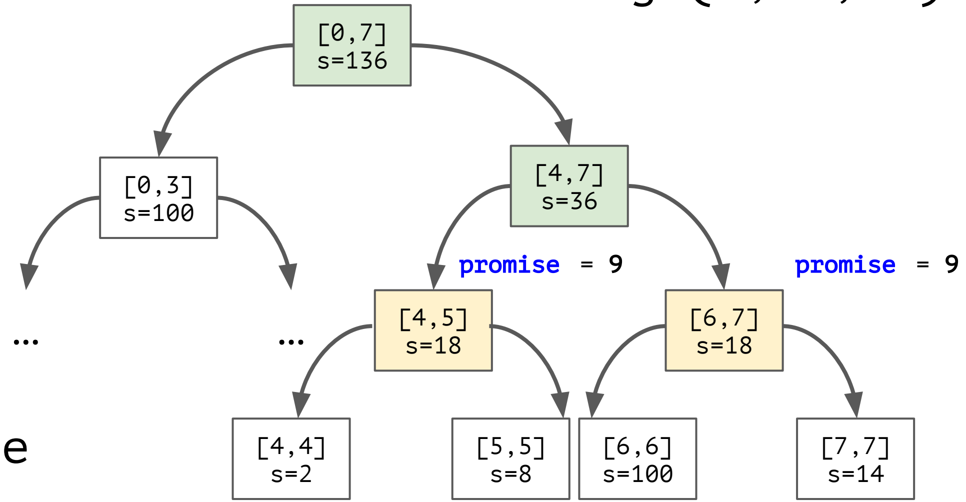
1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

А когда будем этот promise учитывать?

Когда считаем сумму! Если нам все равно спускаться от корня, то вот этот promise и протолкнем вниз.

assign(4, 7, 9)



sum(6, 7) -> дошли до вершины с promise -> проталкиваем в детей обещание -> на запрос sum(6, 7) уже можете дать ответ.

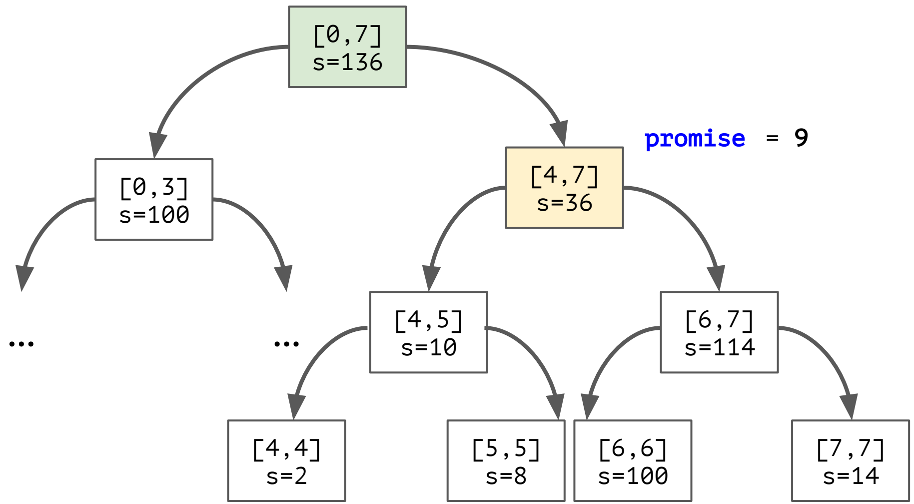
Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

Маленькая проблема:

assign(4, 7, 9)



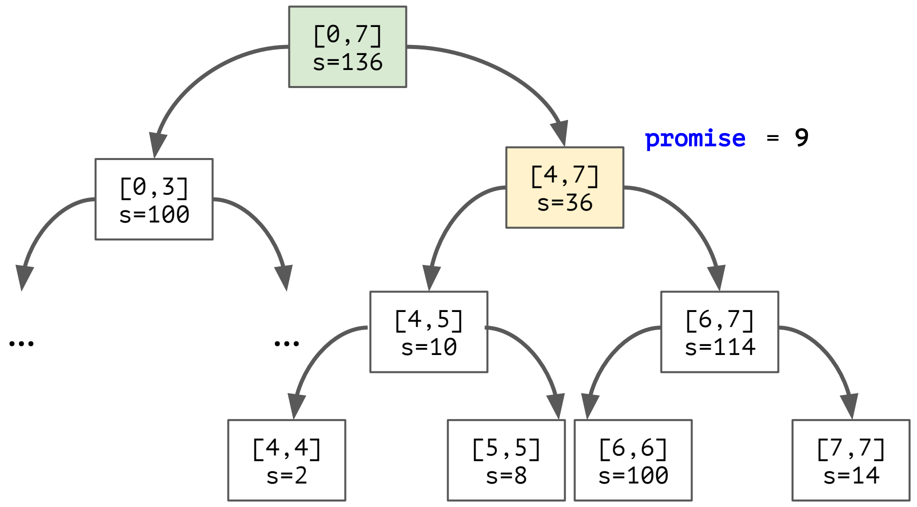
Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

Маленькая проблема:

assign(4, 7, 9)
assign(6, 7, 13)



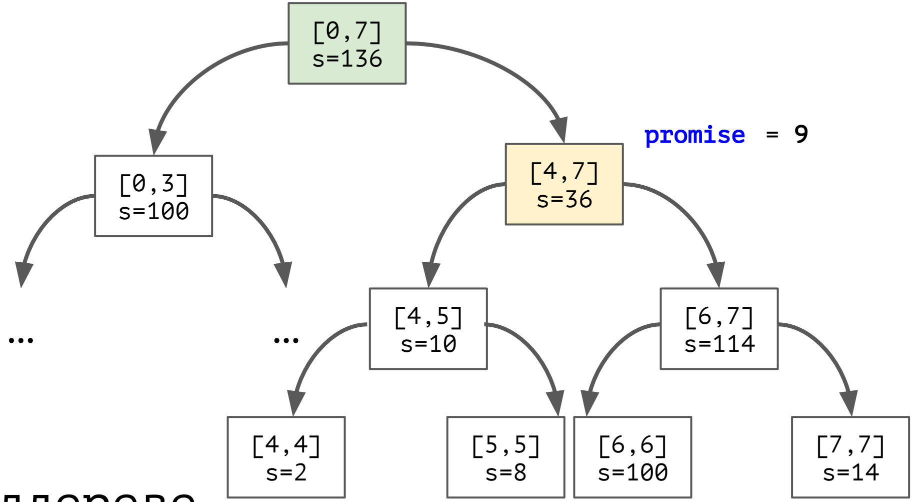
Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

Маленькая проблема:

assign(4, 7, 9)
assign(6, 7, 13)



Забыть о promise = 9 нельзя, он еще пригодится в левом поддереве.

Но уже новый promise на подходе. Что делать?

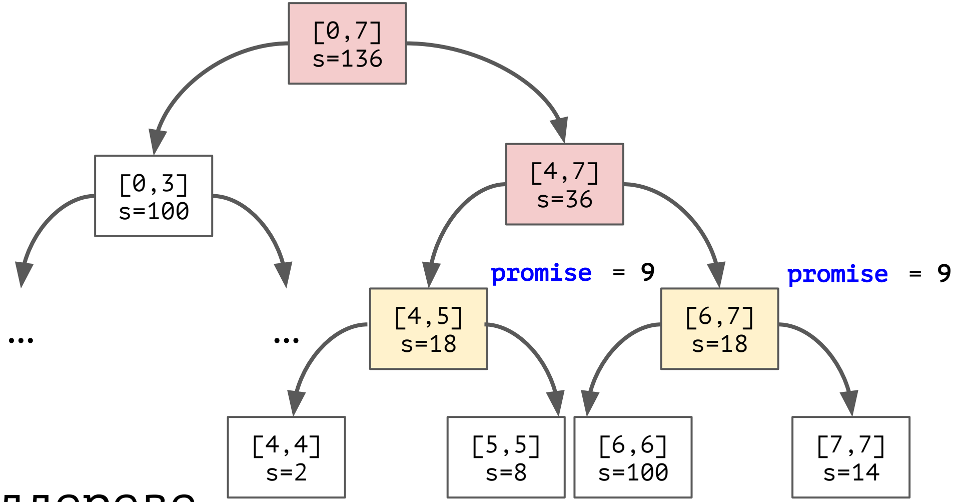
Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

Маленькая проблема:

assign(4, 7, 9)
assign(6, 7, 13)



Забыть о promise = 9 нельзя, он еще пригодится в левом поддереве.

Но уже новый promise на подходе. Что делать? Протолкнем старый promise дальше.

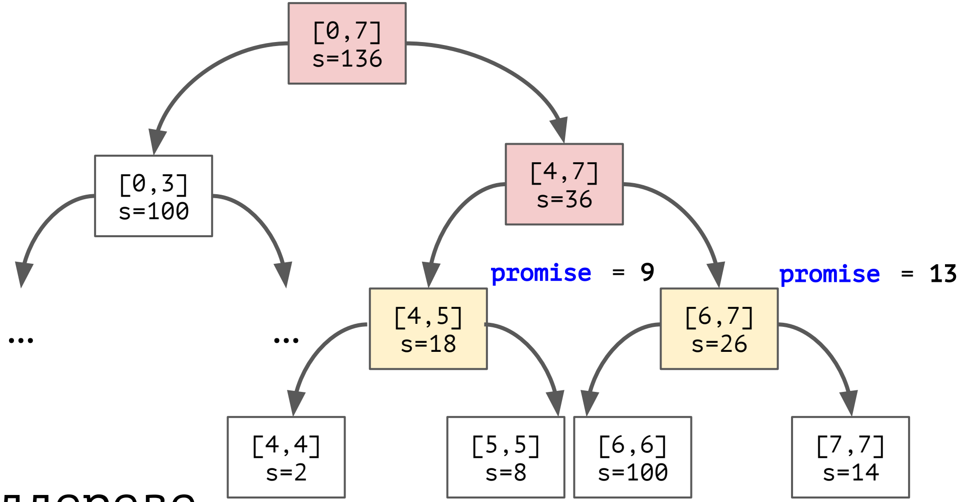
Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

Маленькая проблема:

assign(4, 7, 9)
assign(6, 7, 13)



Забыть о promise = 9 нельзя, он еще пригодится в левом поддереве.

Но уже новый promise на подходе. Что делать?
Протолкнем старый promise дальше. А дальше обновляем promise

Необходимо поддержать новые операции следующего вида:

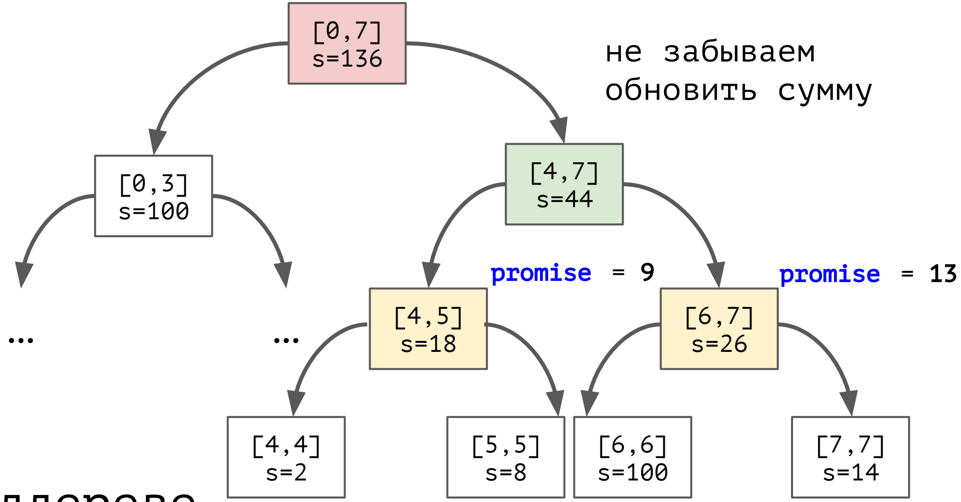
1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

Маленькая проблема:

assign(4, 7, 9)
assign(6, 7, 13)

Забыть о promise = 9 нельзя, он еще пригодится в левом поддереве.



Но уже новый promise на подходе. Что делать?
Протолкнем старый promise дальше. А дальше обновляем promise

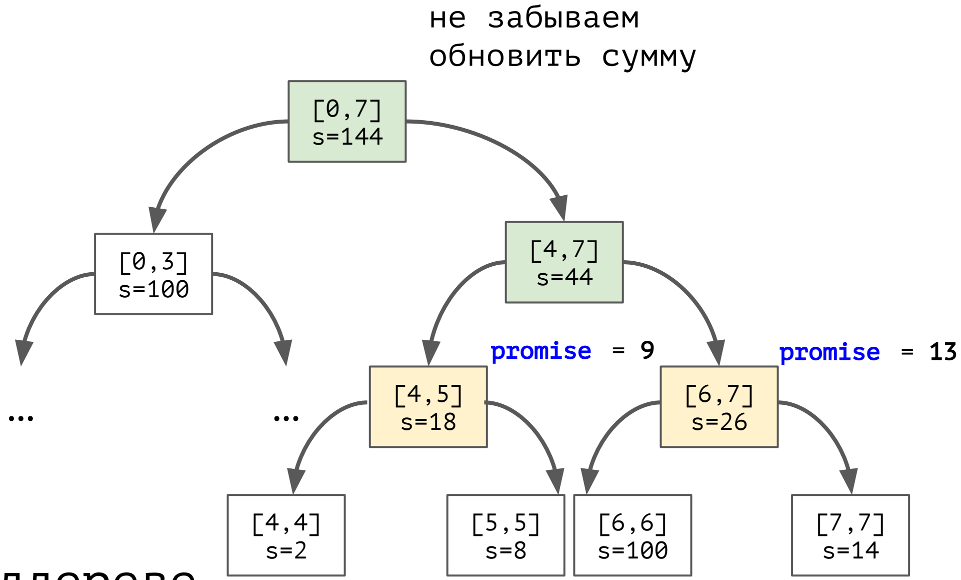
Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

Маленькая проблема:

assign(4, 7, 9)
assign(6, 7, 13)



Забыть о promise = 9 нельзя,
он еще пригодится в левом поддереве.

Но уже новый promise на подходе. Что делать?
Протолкнем старый promise дальше. А дальше обновляем promise

Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

```
def push(root: Node):
  if root.promise is None:
    return
```

Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

```
def push(root: Node):
  if root.promise is None:
    return

  m = (root.lb + root.rb) >> 1

  root.left.sum = root.promise * (m - l + 1)
  root.left.promise = root.promise
```

Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

```
def push(root: Node):
  if root.promise is None:
    return

  m = (root.lb + root.rb) >> 1

  root.left.sum = root.promise * (m - l + 1)
  root.left.promise = root.promise

  root.right.sum = root.promise * (r - m)
  root.right.promise = root.promise

  root.promise = None
```

Утилитарная процедура для проталкивания promise (работает за O(1))

Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

```
def assign(root: Node, l, r, x: int):
```

Дерево отрезков: присваивание на диапазоне

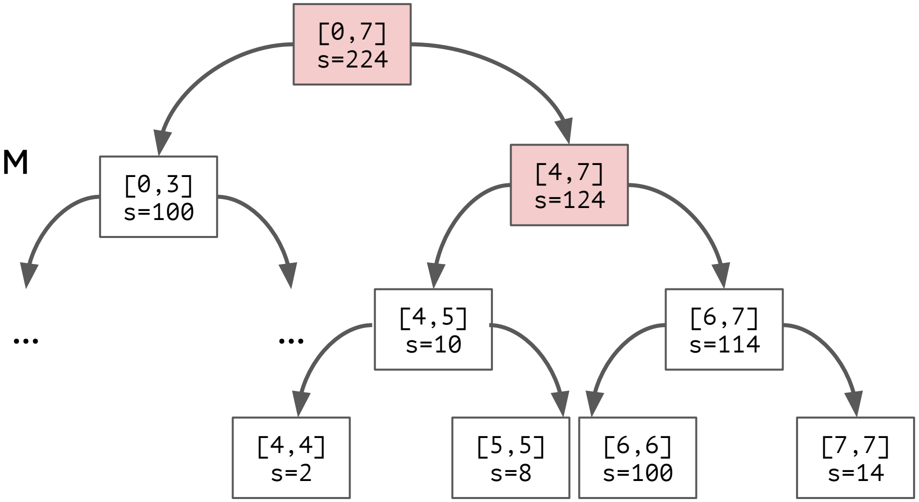
Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

В assign-е для начала обновим только точно интересные нам отрезки

assign(4, 7, 9)



Дерево отрезков: присваивание на диапазоне

Необходимо поддержать новые операции следующего вида:

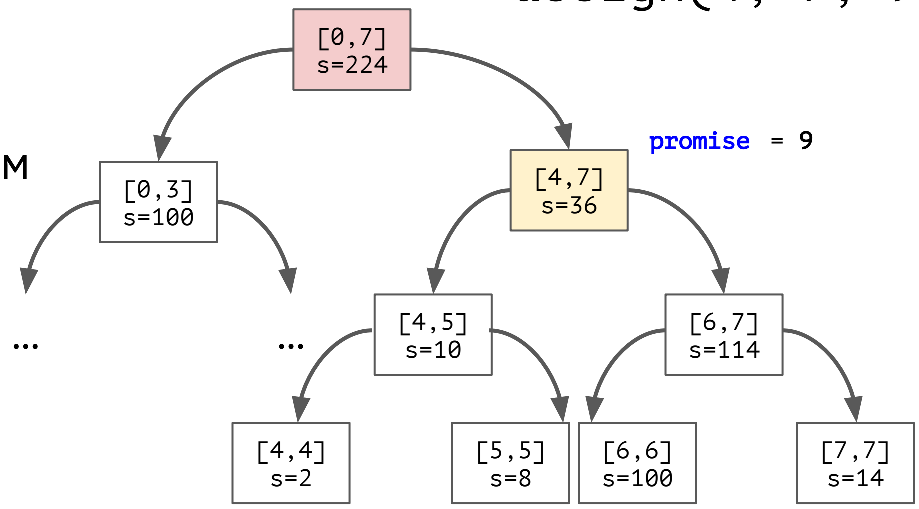
1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

В assign-е для начала обновим только точно интересные нам отрезки

В [4,7] сумма понятна - это (7- 4 + 1) * 9

assign(4, 7, 9)



Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

```
def assign(root: Node, l, r, x: int):
  if root.lb == l and root.rb == r:
    root.promise = x
    root.sum = x * (root.rb - root.lb + 1)
    return
```

Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

```
def assign(root: Node, l, r, x: int):
  if root.lb == l and root.rb == r:
    root.promise = x
    root.sum = x * (root.rb - root.lb + 1)
    return

  push(root) // проталкиваем на случай, если был старый promise
```

Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

```
def assign(root: Node, l, r, x: int):
  if root.lb == l and root.rb == r:
    root.promise = x
    root.sum = x * (root.rb - root.lb + 1)
    return

  push(root) // проталкиваем на случай, если был старый promise
  m = (root.lb + root.rb) >> 1
```

Дерево отрезков: getSum

Пусть запросили [l, r], а мы находимся в вершине root.
У вершины есть свой отрезок: [root.lb, root.rb], сумма: root.sum, и дети: root.left, root.right.

Сложность?

```
def getSum(root: Node, l, r: int) -> int:
  if l == root.lb and r == root.rb:
    return root.sum

  m = (root.lb + root.rb) >> 1
  res = 0

  if l <= m:
      res += getSum(root.left, l, min(r, m))

  if r >= m + 1:
      res += getSum(root.right, max(l, m + 1), r)

  return res
```

Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

```
def assign(root: Node, l, r, x: int):
  if root.lb == l and root.rb == r:
    root.promise = x
    root.sum = x * (root.rb - root.lb + 1)
    return

  push(root) // проталкиваем на случай, если был старый promise
  m = (root.lb + root.rb) >> 1

  if l <= m:
      assign(root.left, l, min(r, m), x)

  if r >= m + 1:
      assign(root.right, max(l, m + 1), r, x)
```

Дерево отрезков: присваивание на диапазоне

Необходимо поддержать новые операции следующего вида:

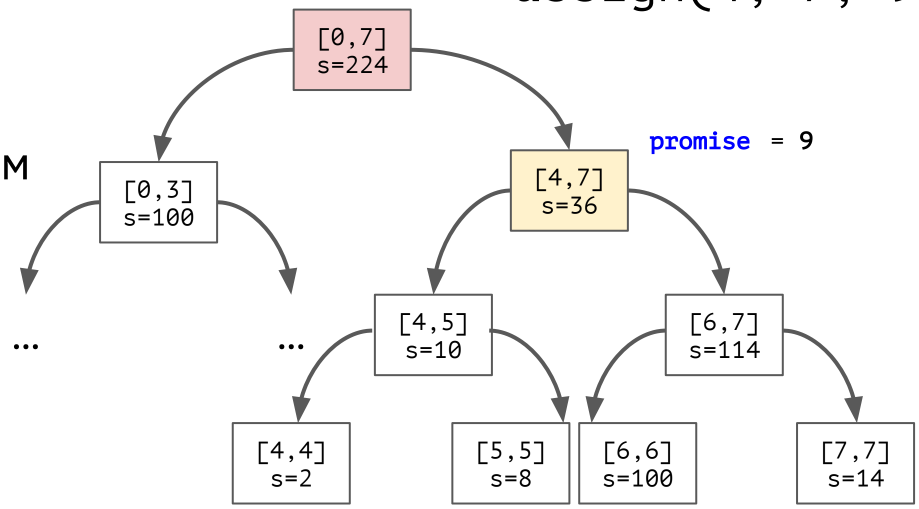
1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

В assign-е для начала обновим только точно интересные нам отрезки

В [4,7] сумма понятна - это (7- 4 + 1) * 9

assign(4, 7, 9)



Дерево отрезков: присваивание на диапазоне

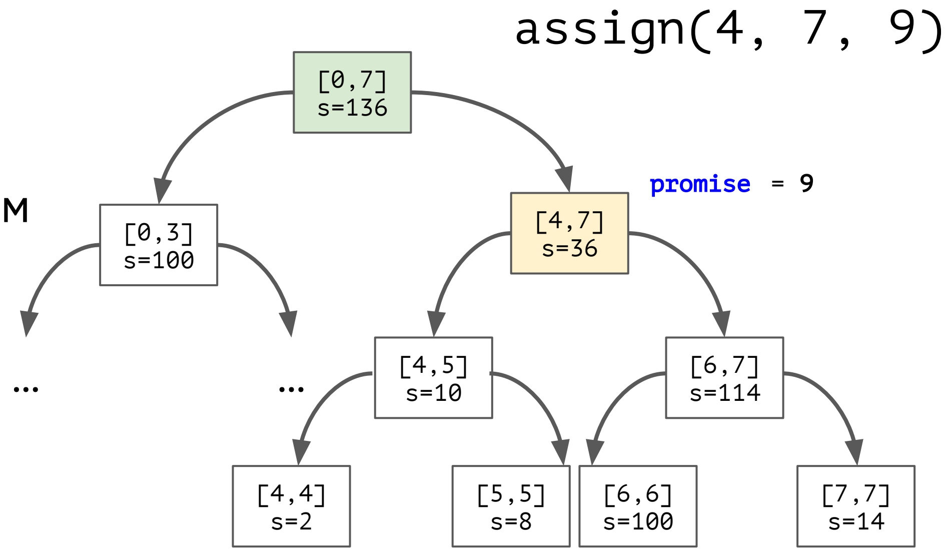
Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Идея: давайте обновлять не моментально, а лениво.

В assign-е для начала обновим только точно интересные нам отрезки

В [4,7] сумма понятна - это (7- 4 + 1) * 9



Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

```
def assign(root: Node, l, r, x: int):
  if root.lb == l and root.rb == r:
    root.promise = x
    root.sum = x * (root.rb - root.lb + 1)
    return

  push(root) // проталкиваем на случай, если был старый promise
  m = (root.lb + root.rb) >> 1

  if l <= m:
      assign(root.left, l, min(r, m), x)

  if r >= m + 1:
      assign(root.right, max(l, m + 1), r, x)

  root.sum = root.left.sum + root.right.sum
```

Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

```
def assign(root: Node, l, r, x: int):
  if root.lb == l and root.rb == r:
    root.promise = x
    root.sum = x * (root.rb - root.lb + 1)
    return

  push(root) // проталкиваем на случай, если был старый promise
  m = (root.lb + root.rb) >> 1

  if l <= m:
      assign(root.left, l, min(r, m), x)

  if r >= m + 1:
      assign(root.right, max(l, m + 1), r, x)

  root.sum = root.left.sum + root.right.sum
```

Сложность опять logN, рассуждения аналогичные getSum

Дерево отрезков: getSum

```python
def getSum(root: Node, l, r: int) -> int:
  if l == root.lb and r == root.rb:
    return root.sum

  m = (root.lb + root.rb) >> 1
  res = 0

  if l <= m:
      res += getSum(root.left, l, min(r, m))

  if r >= m + 1:
      res += getSum(root.right, max(l, m + 1), r)

  return res
```

Дерево отрезков: getSum

```python
def getSum(root: Node, l, r: int) -> int:
  push(root)
  if l == root.lb and r == root.rb:
    return root.sum

  m = (root.lb + root.rb) >> 1
  res = 0

  if l <= m:
      res += getSum(root.left, l, min(r, m))

  if r >= m + 1:
      res += getSum(root.right, max(l, m + 1), r)

  return res
```

Проталкиваем обещания вниз, сложность не меняется

Дерево отрезков: присваивание на диапазоне

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Дерево отрезков: присваивание на диапазоне

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать новые операции следующего вида:

1. sum(l, r) - найти сумму элементов: a_l + … _ a_r
2. assign(l, r, x) - заменить весь отрезок на x

Сложность обеих операций: O(logN)

Поиск количества чисел на отрезке >= k

Поиск количества чисел на отрезке >= k

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать операцию следующего вида:

- gte(l, r, k) - получить количество элементов на отрезке, которые больше либо равны k

Поиск количества чисел на отрезке >= k

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать операцию следующего вида:

- gte(l, r, k) - получить количество элементов на отрезке, которые больше либо равны k

46 11 [40 8 2 42] 65 10

Поиск количества чисел на отрезке >= k

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать операцию следующего вида:

- gte(l, r, k) - получить количество элементов на отрезке, которые больше либо равны k

46 11 [40 8 2 42] 65 10 gte(2, 5, 10) -> 2

Поиск количества чисел на отрезке >= k

Задача: пусть есть массив фиксированного размера n. Пусть для простоты n - это степень двойки.

a_1, a_2, a_3, …, a_n

Необходимо поддержать операцию следующего вида:

- gte(l, r, k) - получить количество элементов на отрезке, которые больше либо равны k

46 11 [40 8 2 42] 65 10   gte(2, 5, 10) -> 2

Хочется использовать дерево отрезков, но как?

Поиск количества чисел на отрезке >= k

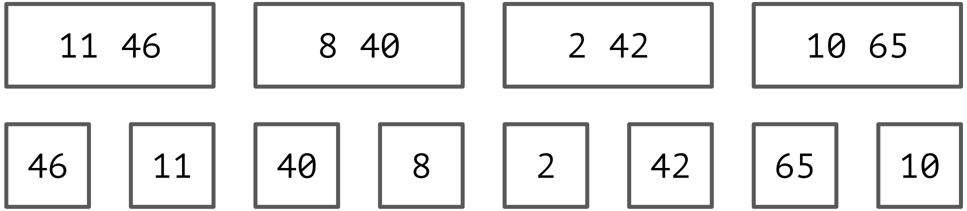
Поиск количества чисел на отрезке >= k

46 11 40 8 2 42 65 10

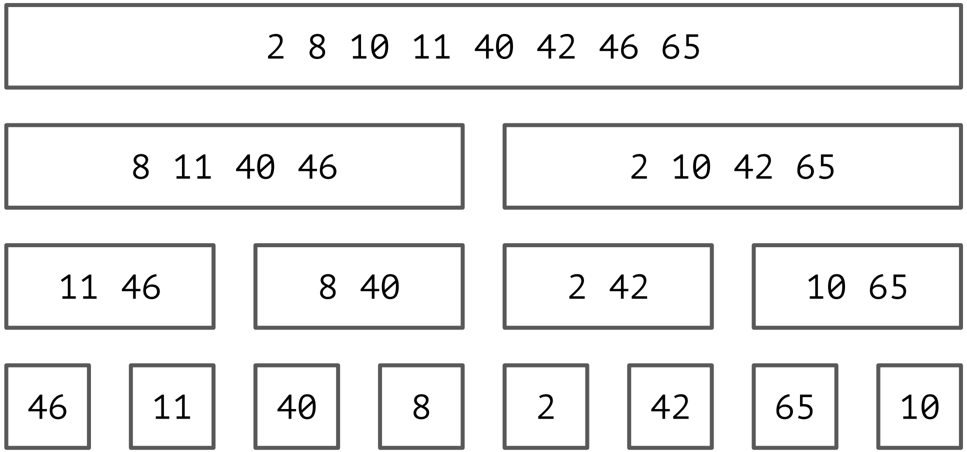
Поиск количества чисел на отрезке >= k

46	11	40	8	2	42	65	10

Поиск количества чисел на отрезке >= k

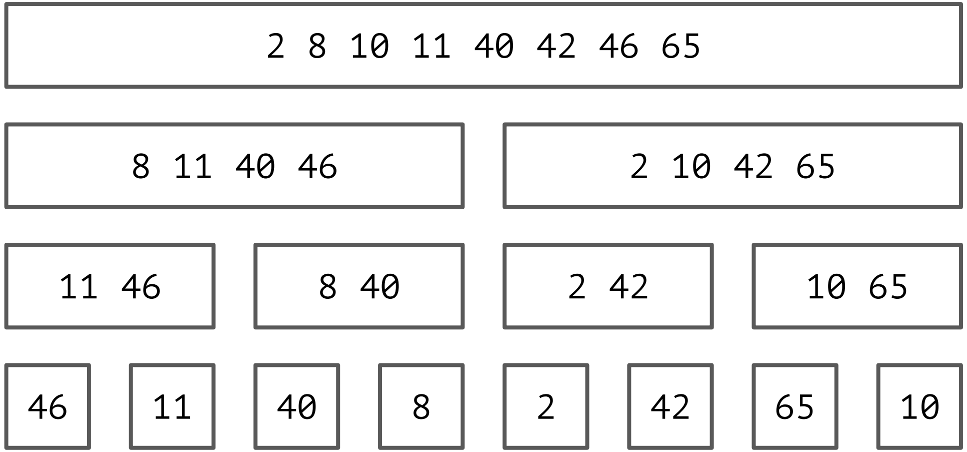


Поиск количества чисел на отрезке >= k



Что напоминает?

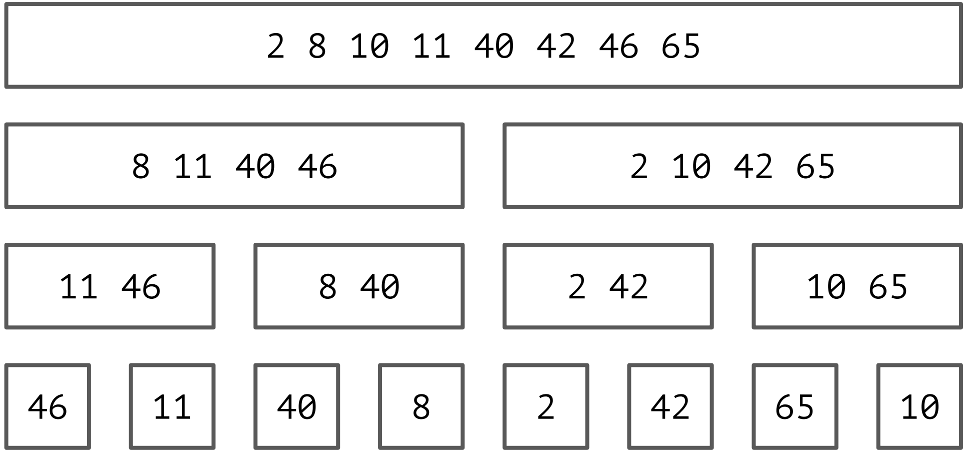
Merge sort tree



Что напоминает?

Ход работы merge sort!

Merge sort tree

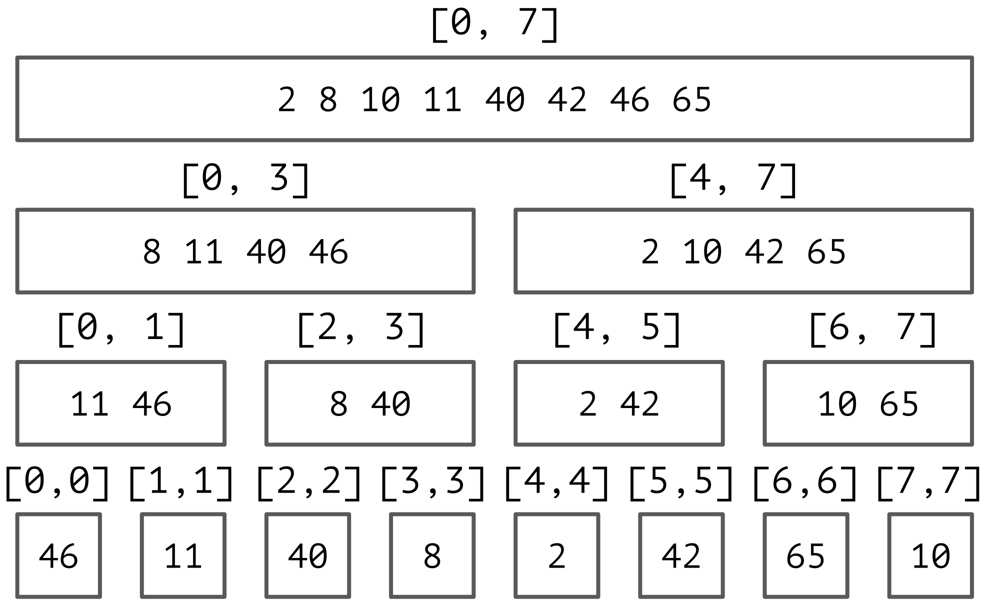


Что напоминает?

Ход работы merge sort!

А еще это вполне себе дерево отрезков, просто вместо суммы храним отсортированный массив

Merge sort tree

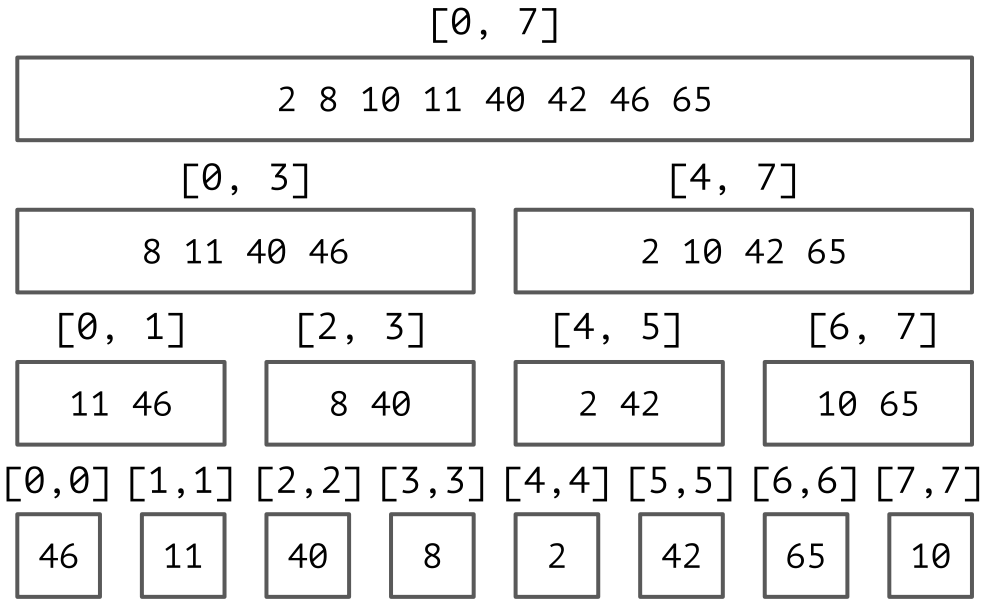


Что напоминает?

Ход работы merge sort!

А еще это вполне себе дерево отрезков, просто вместо суммы храним отсортированный массив

Merge sort tree



Что напоминает?

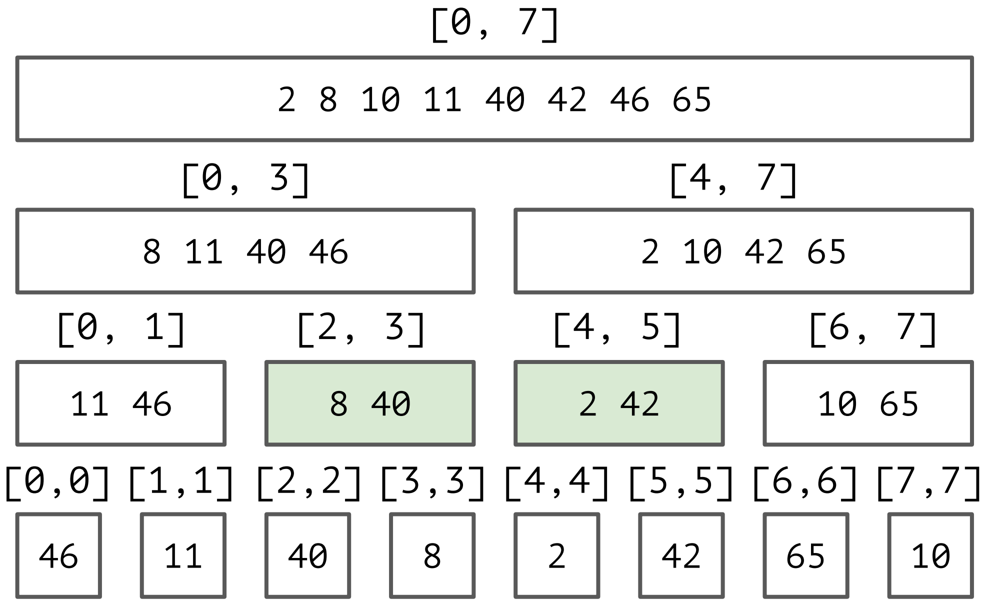
Ход работы merge sort!

А еще это вполне себе дерево отрезков, просто вместо суммы храним отсортированный массив

Как теперь найти количество элементов на [l,r] больших k?

Merge sort tree

gte(2, 5, 10) = ?



Что напоминает?

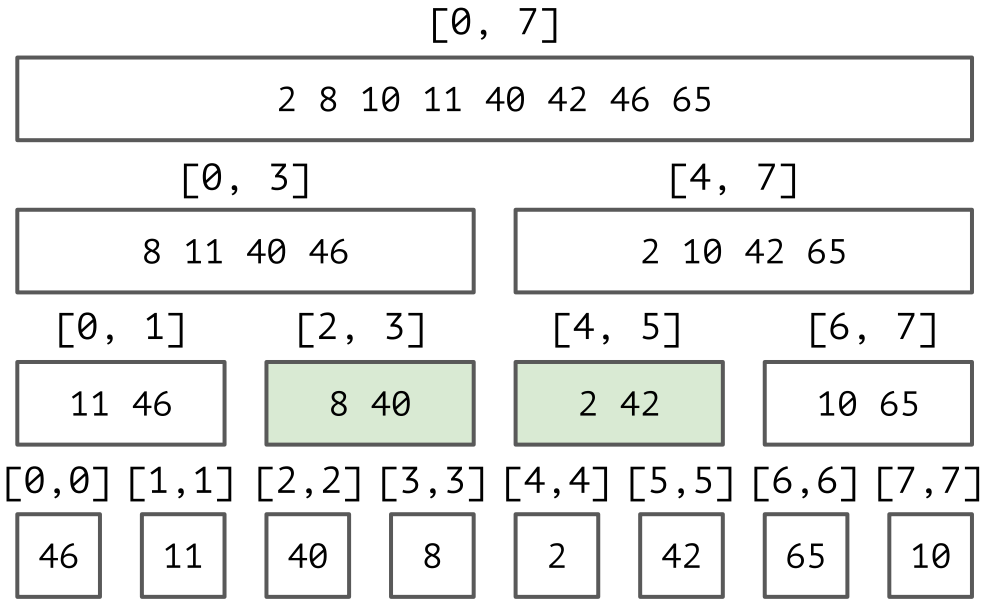
Ход работы merge sort!

А еще это вполне себе дерево отрезков, просто вместо суммы храним отсортированный массив

Как теперь найти количество элементов на [l,r] больших k?
Как в getSum ищем соответствующие отрезки.

Merge sort tree

gte(2, 5, 10) = ?



Что напоминает?

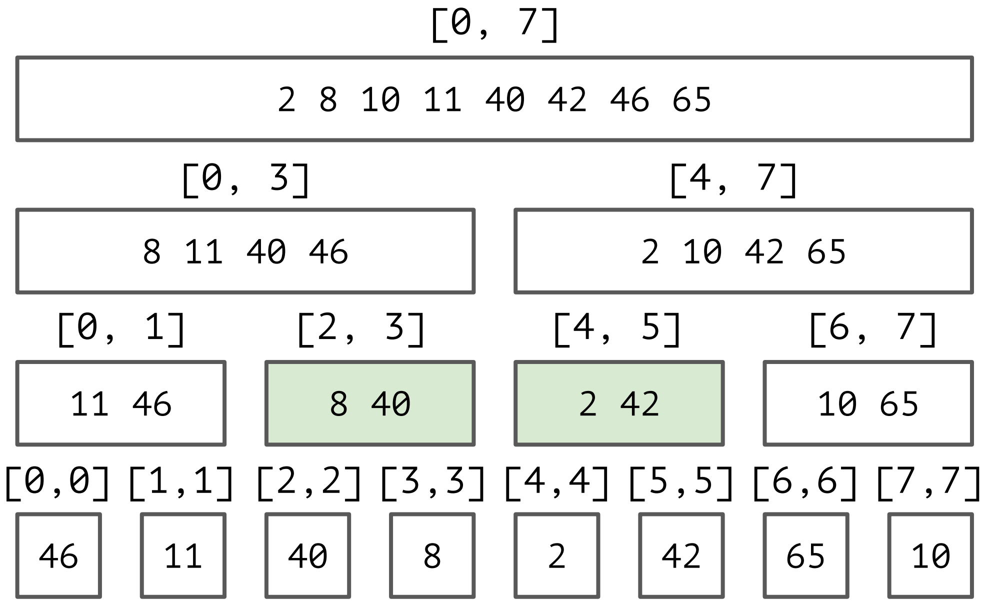
Ход работы merge sort!

А еще это вполне себе дерево отрезков, просто вместо суммы храним отсортированный массив

Как теперь найти количество элементов на [l,r] больших k?
Как в getSum ищем соответствующие отрезки. А как потом в каждом отрезке найти количество больше искомого?

Merge sort tree

gte(2, 5, 10) = ?



Что напоминает?

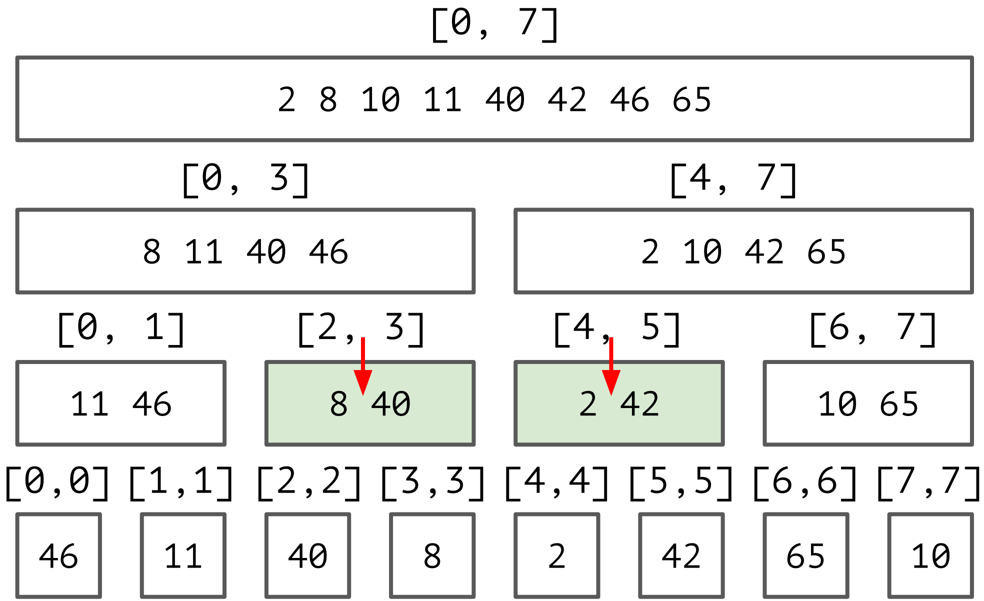
Ход работы merge sort!

А еще это вполне себе дерево отрезков, просто вместо суммы храним отсортированный массив

Как теперь найти количество элементов на [l,r] больших k? Как в getSum ищем соответствующие отрезки. А как потом в каждом отрезке найти количество больше искомого? **Бинарный поиск**! Массив от отсортирован!

Merge sort tree

gte(2, 5, 10) = ?



Что напоминает?

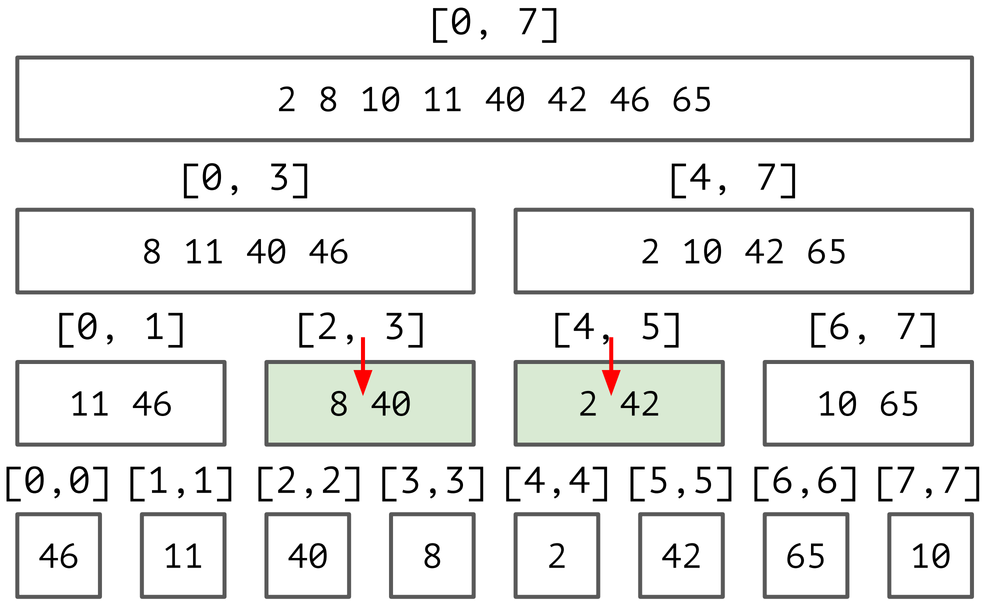
Ход работы merge sort!

А еще это вполне себе дерево отрезков, просто вместо суммы храним отсортированный массив

Как теперь найти количество элементов на [l,r] больших k? Как в getSum ищем соответствующие отрезки. А как потом в каждом отрезке найти количество больше искомого? **Бинарный поиск**! Массив от отсортирован!

Merge sort tree

gte(2, 5, 10) = 2



Что напоминает?

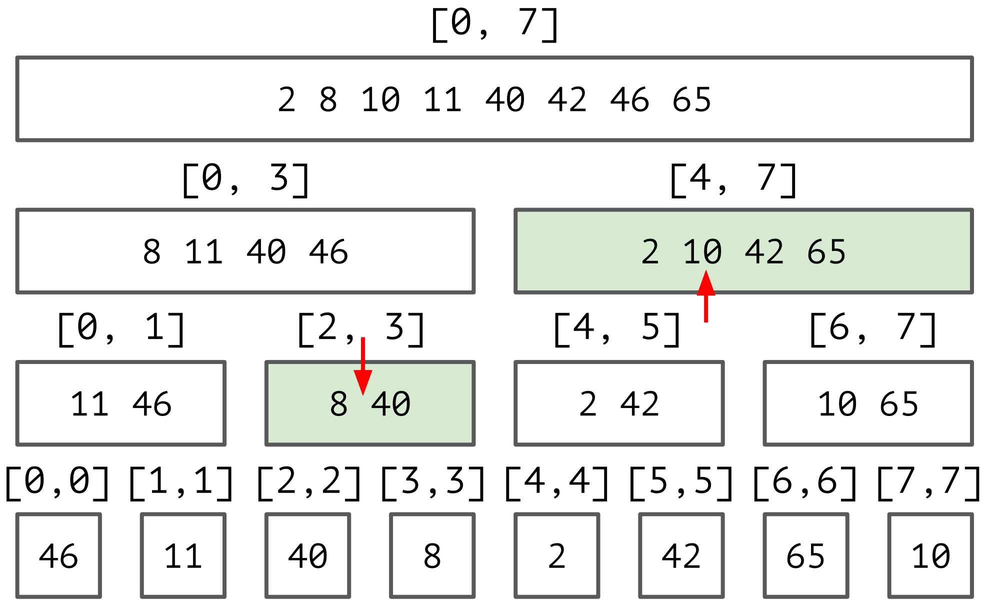
Ход работы merge sort!

А еще это вполне себе дерево отрезков, просто вместо суммы храним отсортированный массив

Как теперь найти количество элементов на [l,r] больших k? Как в getSum ищем соответствующие отрезки. А как потом в каждом отрезке найти количество больше искомого? **Бинарный поиск**! Массив от отсортирован! Сколько осталось справа - часть ответа из этого отрезка.

Merge sort tree

gte(2, 7, 10) = 4



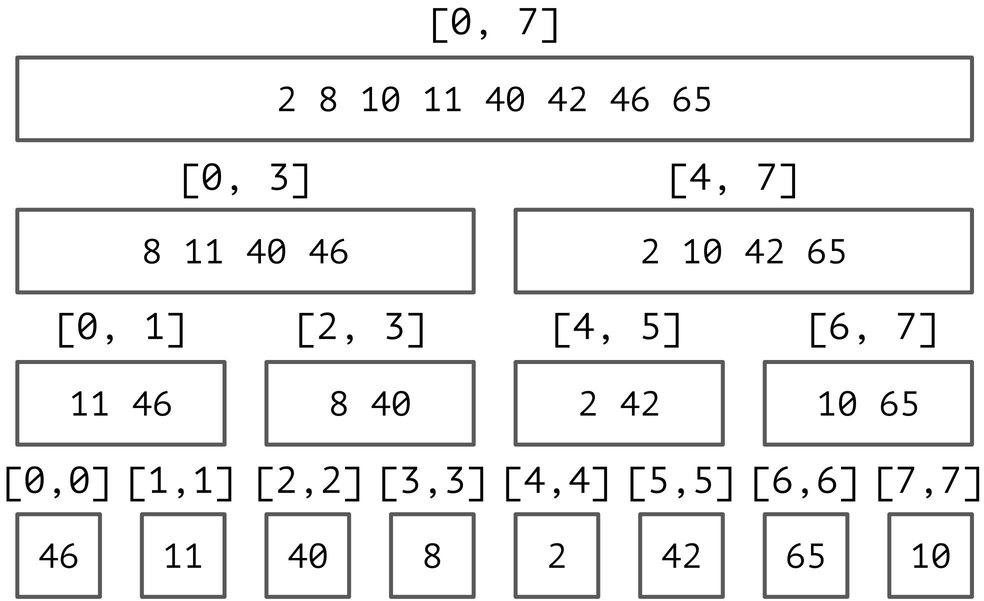
Что напоминает?

Ход работы merge sort!

А еще это вполне себе дерево отрезков, просто вместо суммы храним отсортированный массив

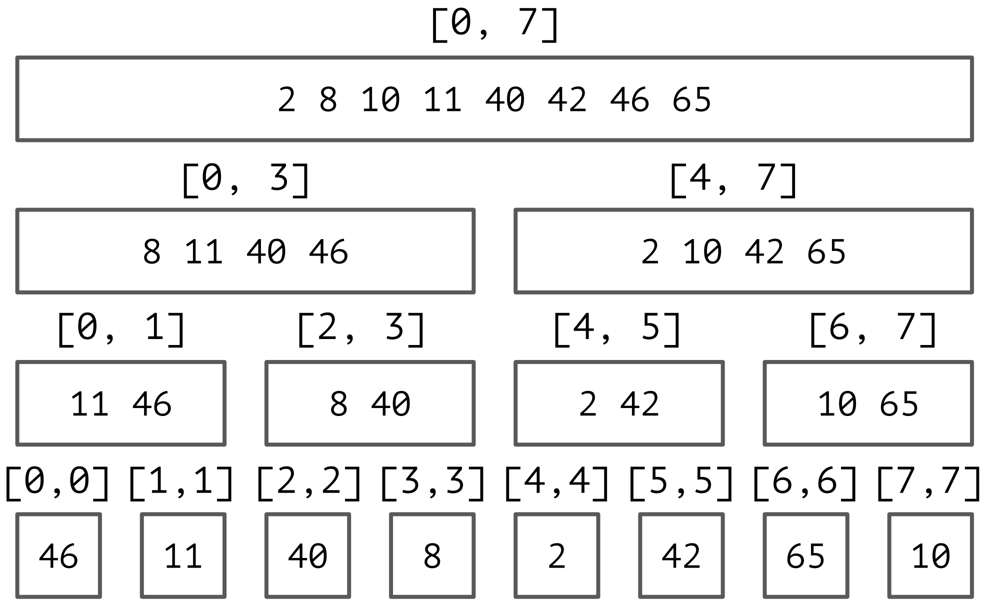
Как теперь найти количество элементов на [l,r] больших k? Как в getSum ищем соответствующие отрезки. А как потом в каждом отрезке найти количество больше искомого? **Бинарный поиск**! Массив от отсортирован! (Правая граница - найденный индекс) = часть ответа из этого отрезка

Merge sort tree



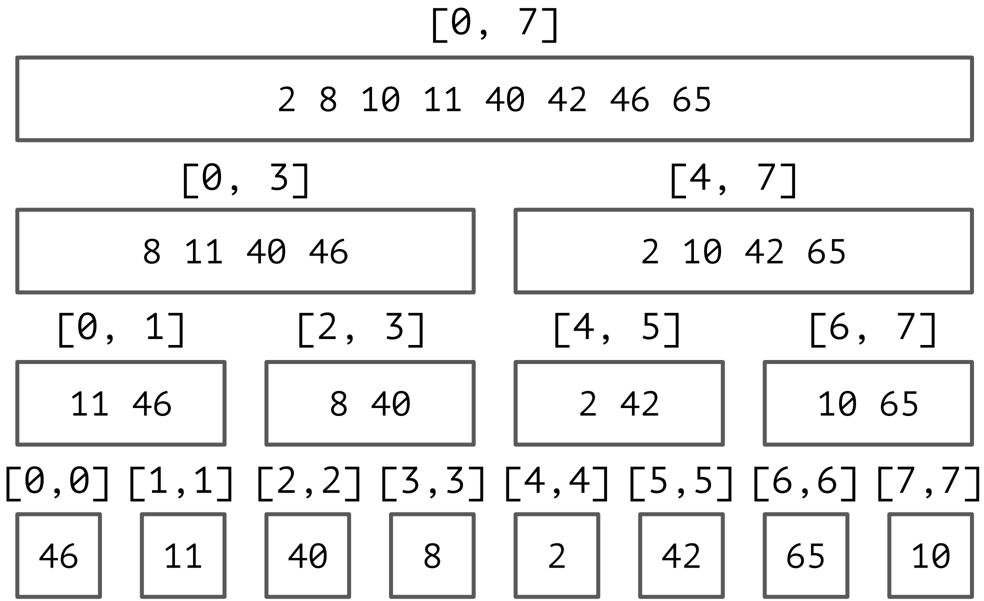
Как построить такое дерево?

Merge sort tree



Как построить такое дерево? Запустить merge sort и сохранять подмассивы.

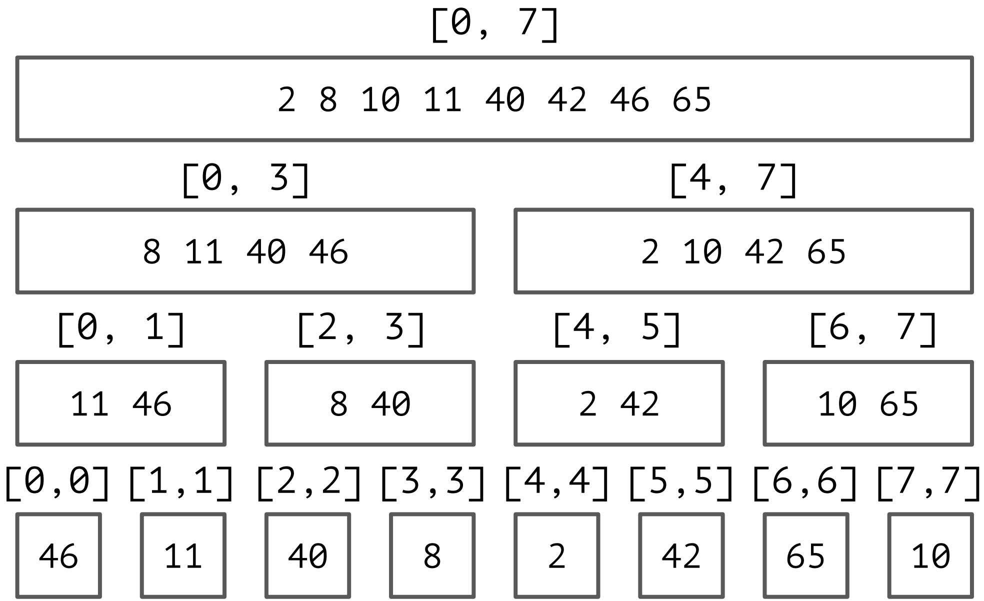
Merge sort tree



Как построить такое дерево? Запустить merge sort и сохранять подмассивы.

Сложность (построения) O(N*logN), память - ?

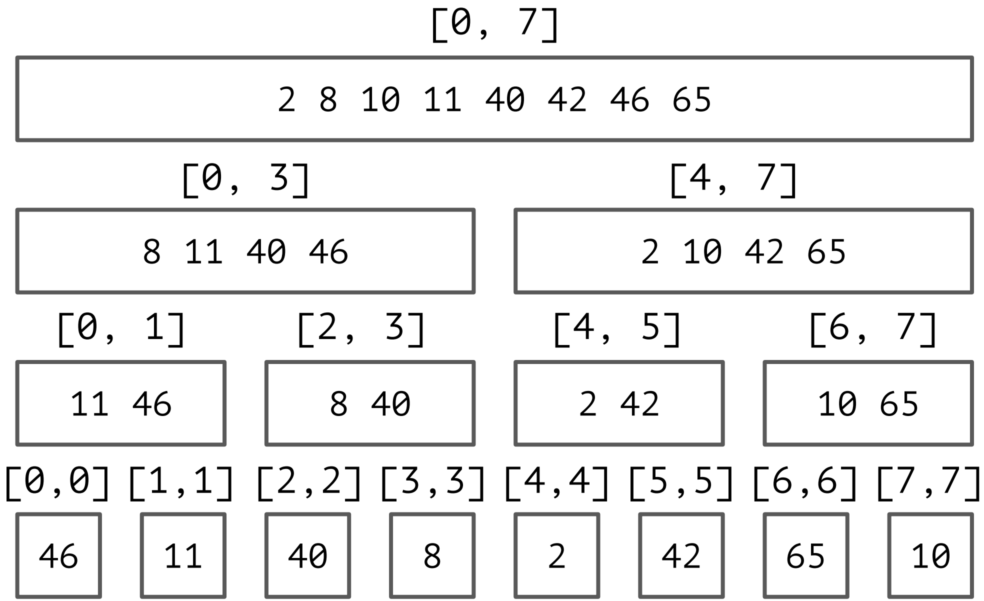
Merge sort tree



Как построить такое дерево? Запустить merge sort и сохранять подмассивы.

Сложность (построения) O(N*logN), память - O(N*logN), т.к. каждый элемент входит O(logN) раз

Merge sort tree

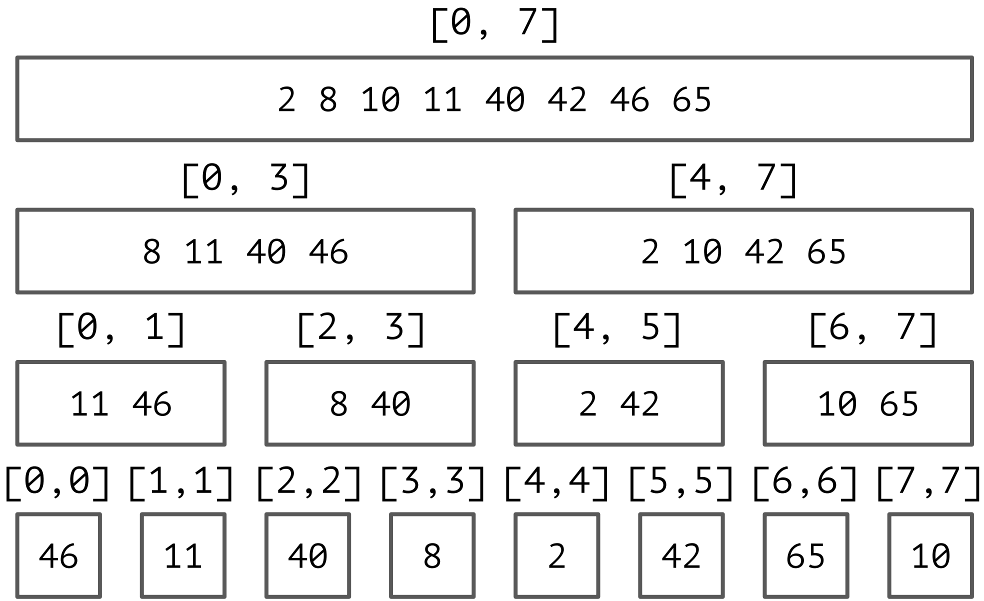


Как построить такое дерево? Запустить merge sort и сохранять подмассивы.

Сложность (построения) O(N*logN), память - O(N*logN), т.к. каждый элемент входит O(logN) раз

Сложность запроса?

Merge sort tree

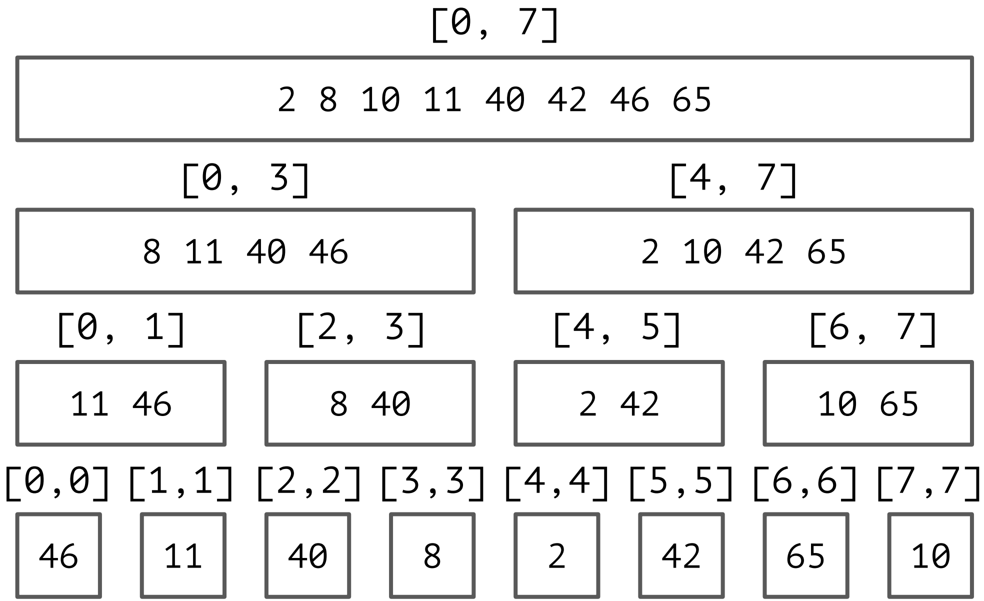


Как построить такое дерево? Запустить merge sort и сохранять подмассивы.

Сложность (построения) O(N*logN), память - O(N*logN), т.к. каждый элемент входит O(logN) раз

Сложность запроса? logN бинарных поисков => O(logN * logN)

Merge sort tree



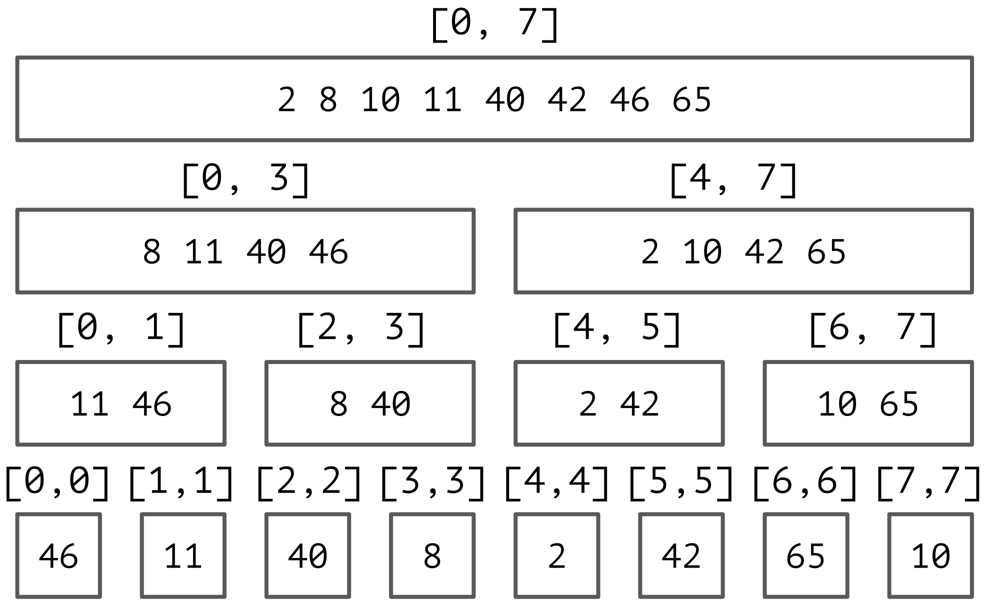
Как построить такое дерево? Запустить merge sort и сохранять подмассивы.

Сложность (построения) O(N*logN), память - O(N*logN), т.к. каждый элемент входит O(logN) раз

Сложность запроса? logN бинарных поисков => O(logN * logN)

Можно ли лучше?

Merge sort tree



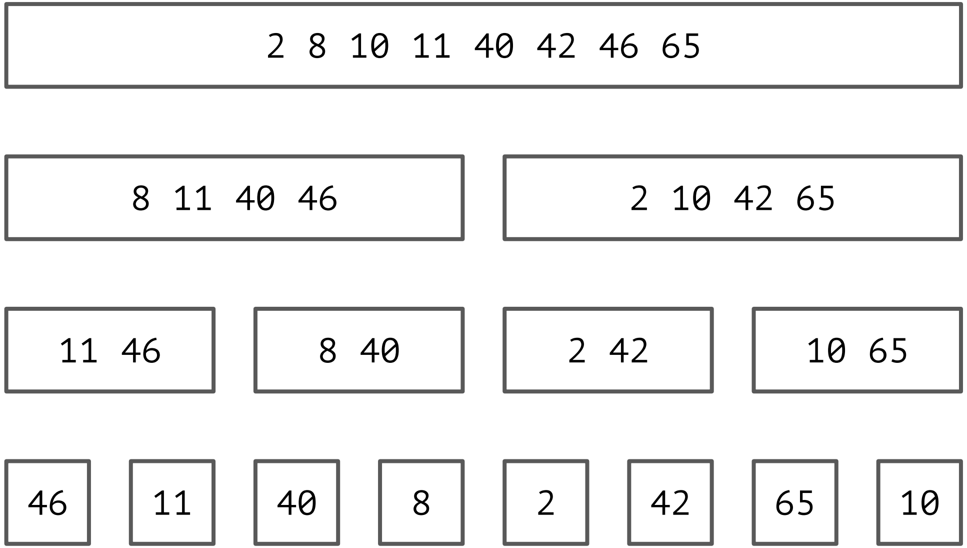
Как построить такое дерево? Запустить merge sort и сохранять подмассивы.

Сложность (построения) O(N*logN), память - O(N*logN), т.к. каждый элемент входит O(logN) раз

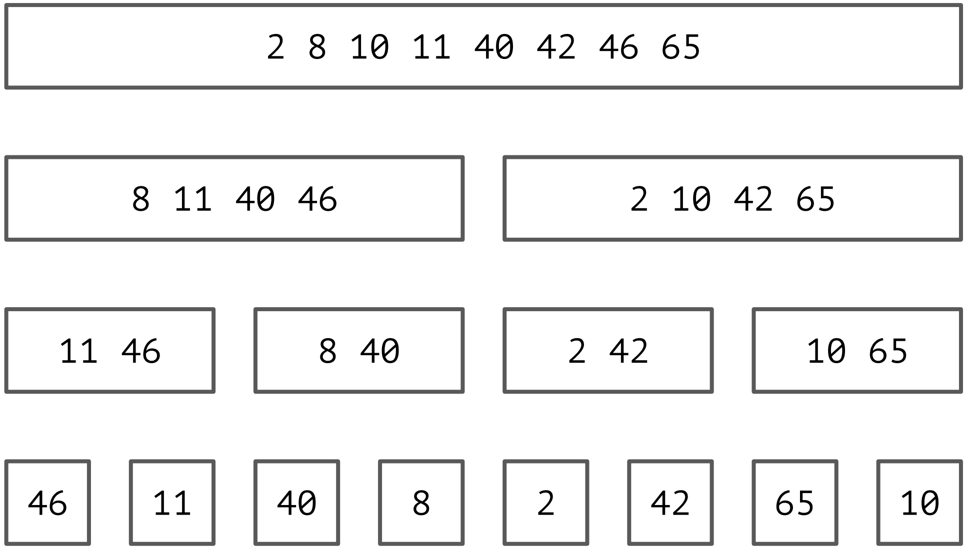
Сложность запроса? logN бинарных поисков => O(logN * logN)

Можно ли лучше? Да! Но ценой памяти.

Fractal cascading

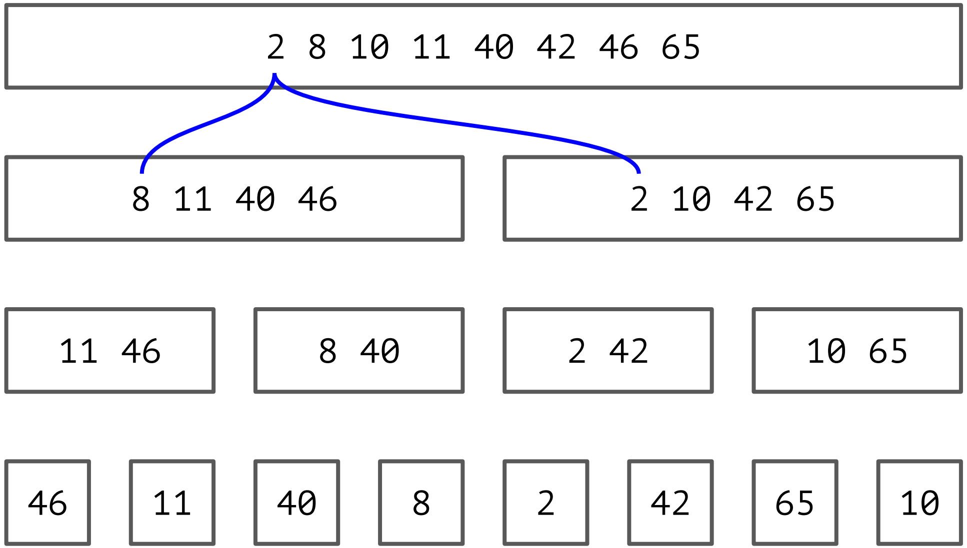


Fractal cascading



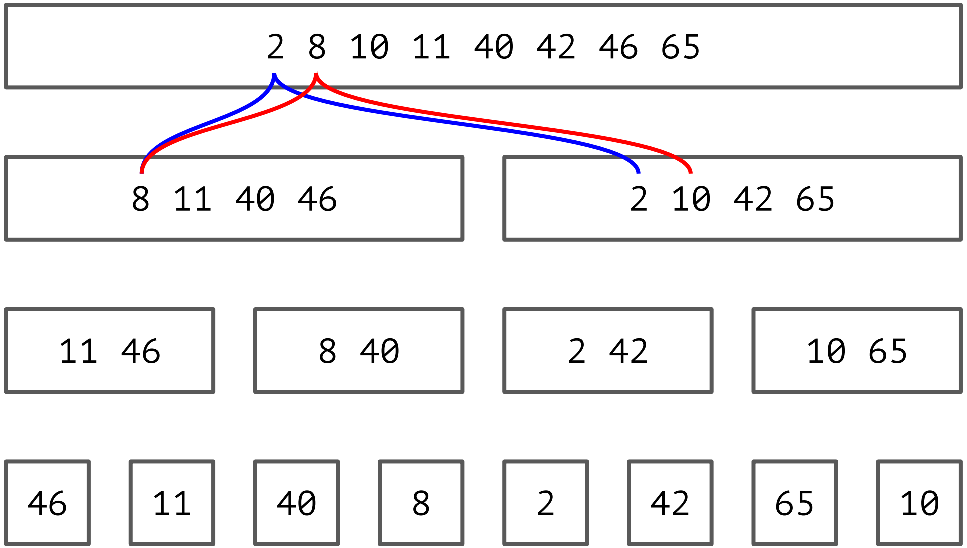
Пусть теперь каждый элемент в каждом массиве хранит индексы на первого, кто >= чем он, в детях

Fractal cascading



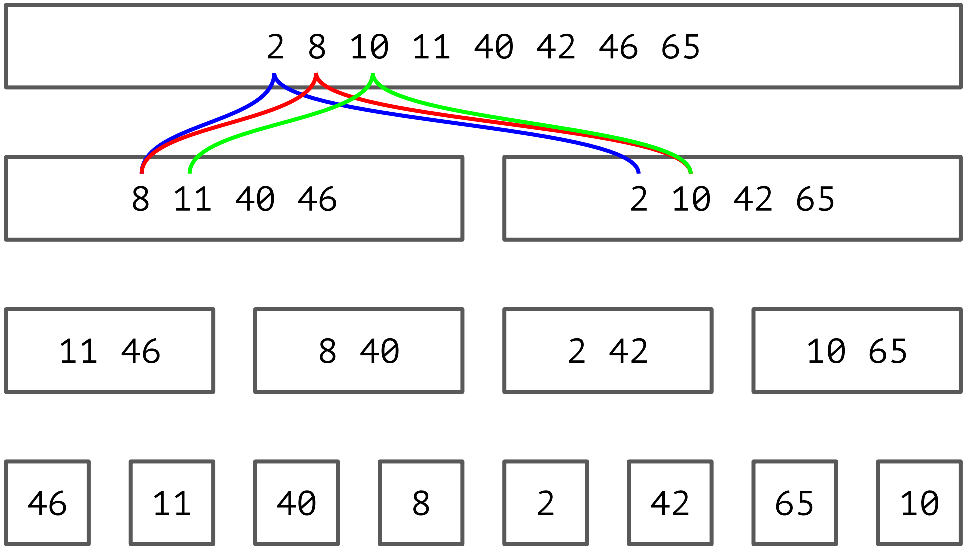
Пусть теперь каждый элемент в каждом массиве хранит индексы на первого, кто >= чем он, в детях

Fractal cascading



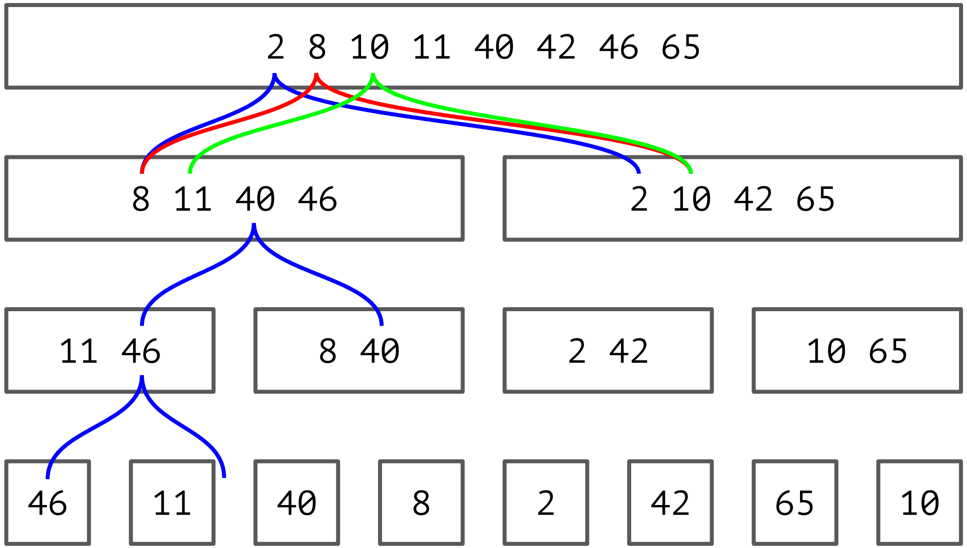
Пусть теперь каждый элемент в каждом массиве хранит индексы на первого, кто >= чем он, в детях

Fractal cascading



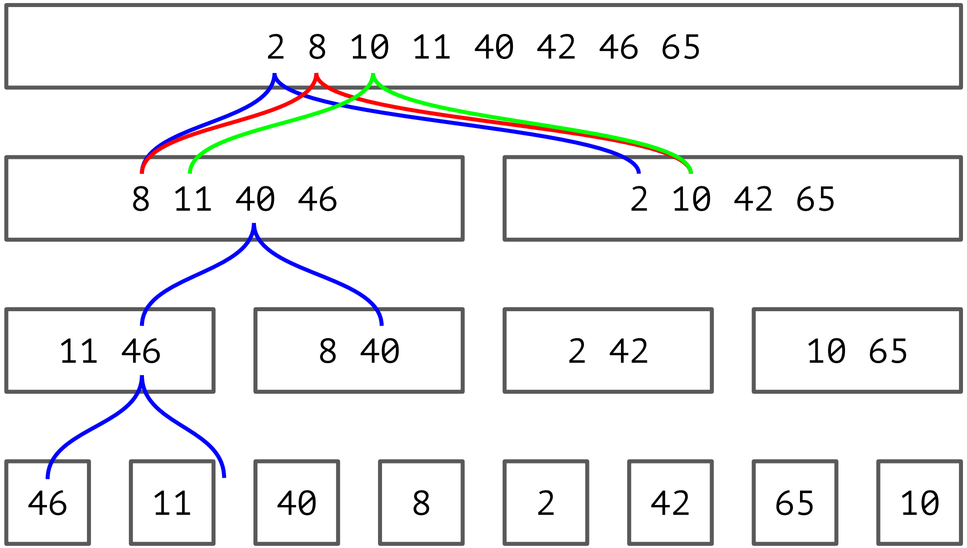
Пусть теперь каждый элемент в каждом массиве хранит индексы на первого, кто >= чем он, в детях

Fractal cascading



Пусть теперь каждый элемент в каждом массиве хранит индексы на первого, кто >= чем он, в детях

Fractal cascading

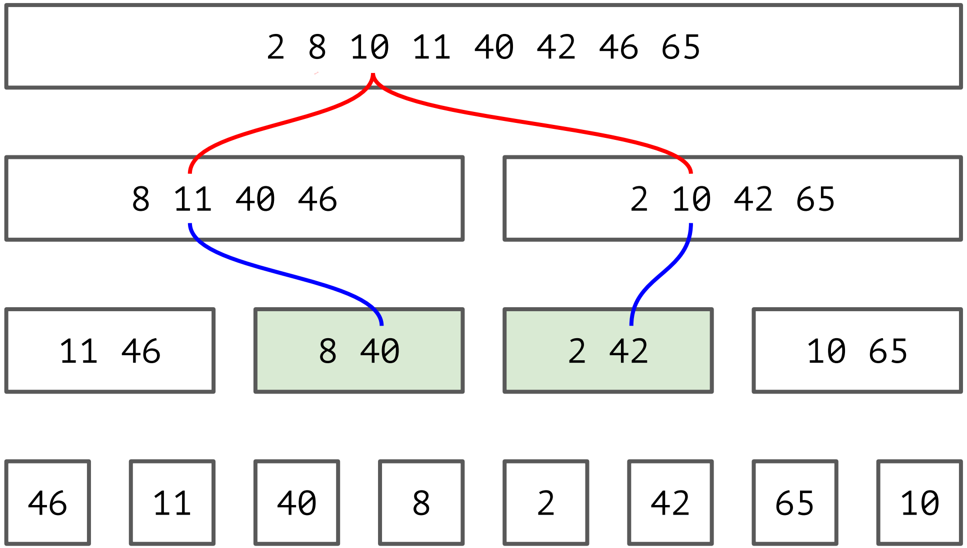


Пусть теперь каждый элемент в каждом массиве хранит индексы на первого, кто >= чем он, в детях

Тогда больше не нужно каждый раз запускать бинарный поиск! Только один раз в начале.

Fractal cascading

`gte(2, 5, 10) = 2`

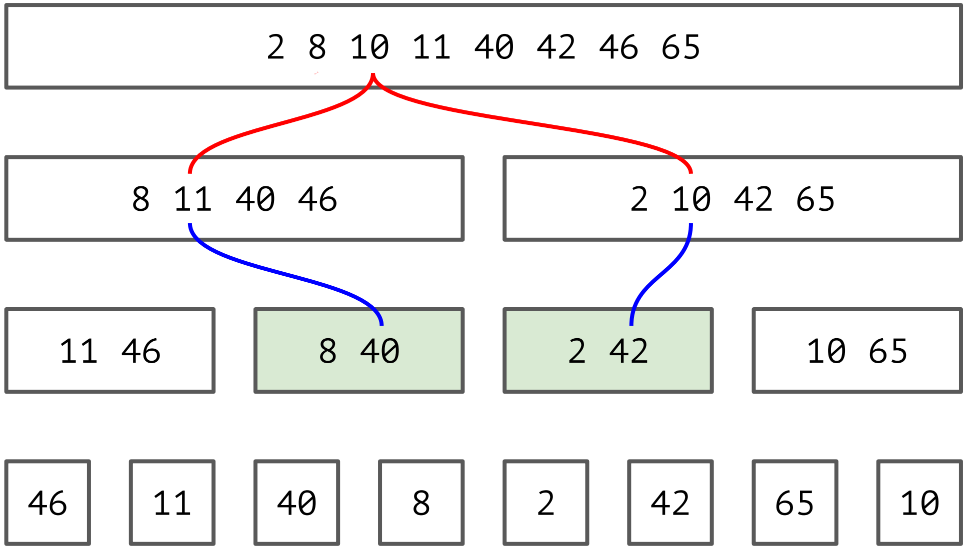


Пусть теперь каждый элемент в каждом массиве хранит индексы на первого, кто >= чем он, в детях

Тогда больше не нужно каждый раз запускать бинарный поиск! Только один раз в начале.

Fractal cascading

gte(2, 5, 10) = 2

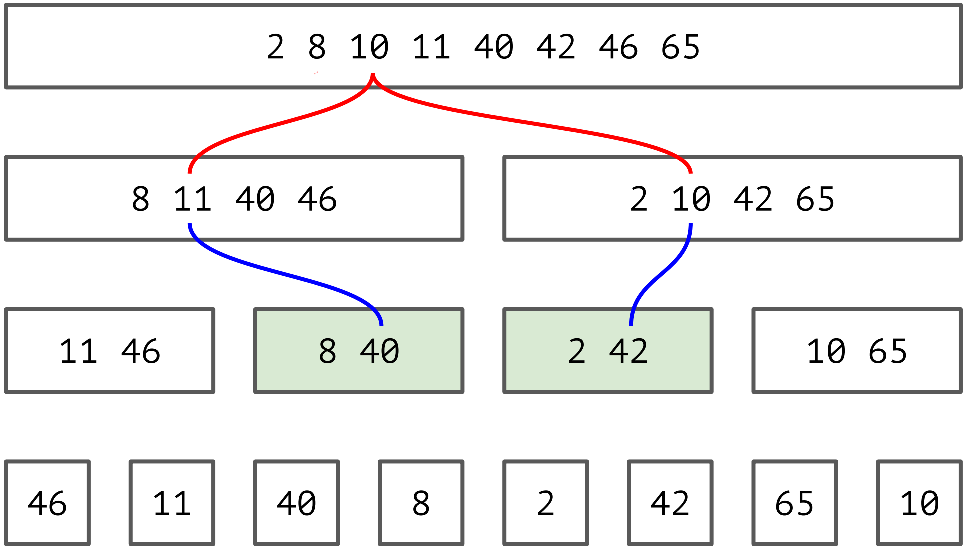


Пусть теперь каждый элемент в каждом массиве хранит индексы на первого, кто >= чем он, в детях

Тогда больше не нужно каждый раз запускать бинарный поиск! Только один раз в начале.

Первый бинарный поиск ищет позицию в отсортированном массиве. Он хранит нужные нам элемент в сыновьях. Дальше при спуске используем уже именно их (и так далее, каждый раз ищем в детях подходящий элемент за O(1)).

Fractal cascading

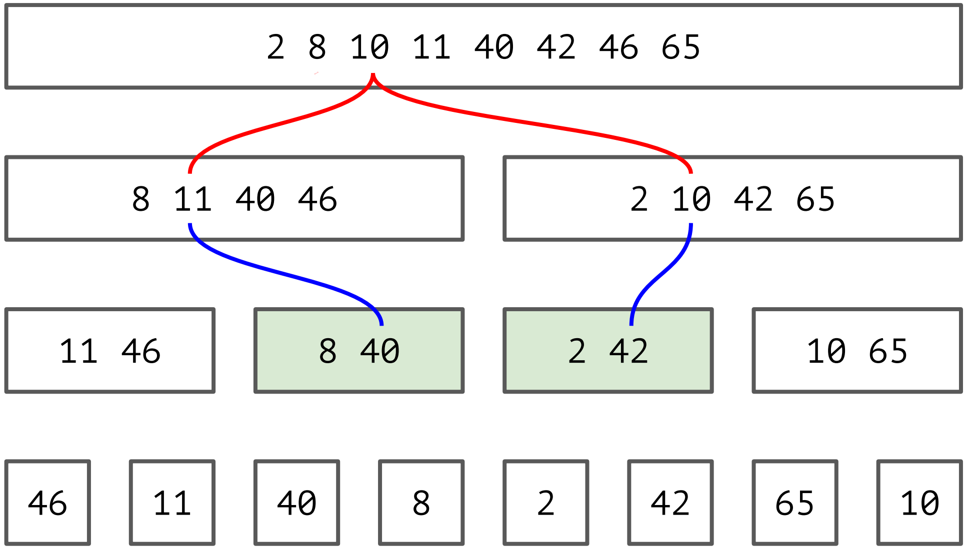


Пусть теперь каждый элемент в каждом массиве хранит индексы на первого, кто >= чем он, в детях

Тогда больше не нужно каждый раз запускать бинарный поиск! Только один раз в начале.

Сложность построения: не меняется O(N*logN)

Fractal cascading

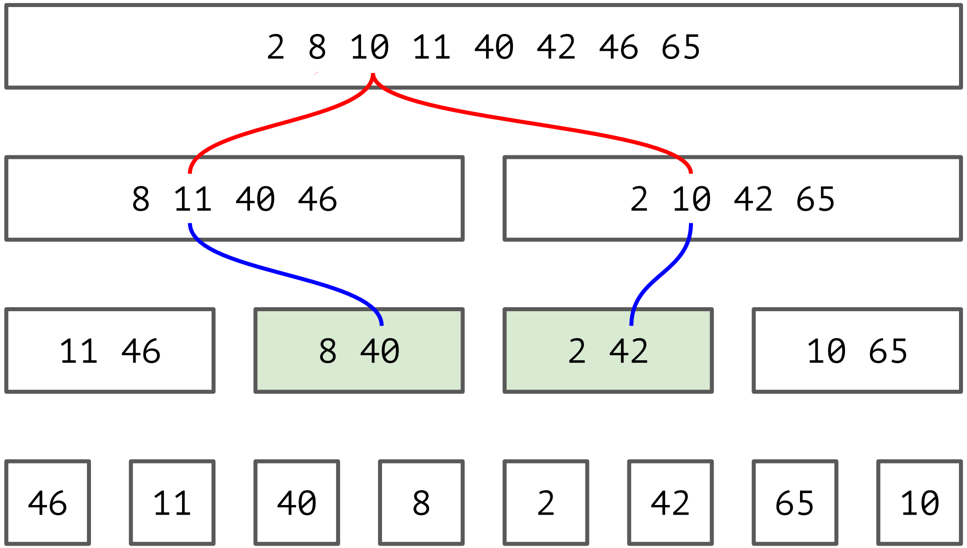


Пусть теперь каждый элемент в каждом массиве хранит индексы на первого, кто >= чем он, в детях

Тогда больше не нужно каждый раз запускать бинарный поиск! Только один раз в начале.

Сложность построения: не меняется O(N*logN)
Память: x3, но остается O(N*logN)

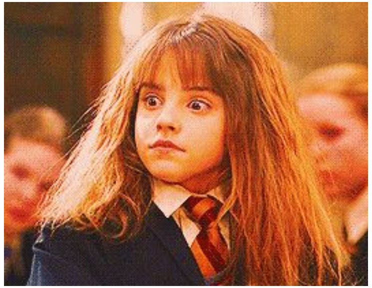
Fractal cascading



Пусть теперь каждый элемент в каждом массиве хранит индексы на первого, кто >= чем он, в детях

Тогда больше не нужно каждый раз запускать бинарный поиск! Только один раз в начале.

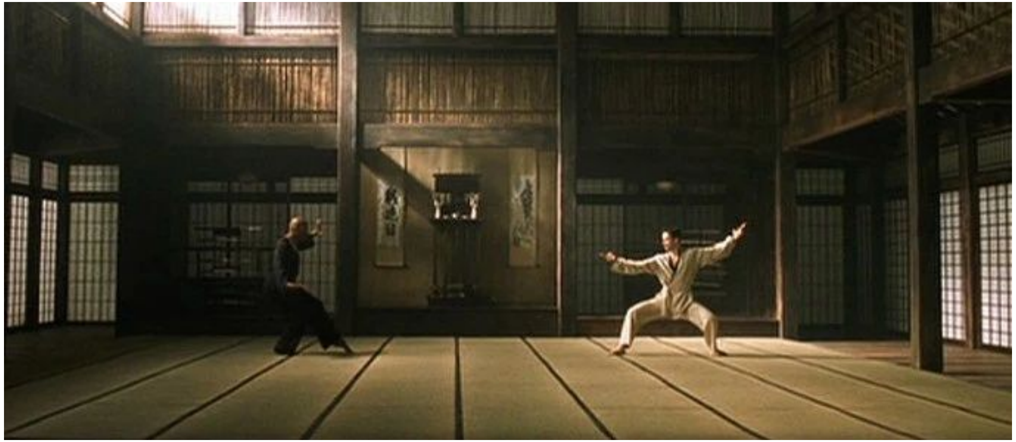
Сложность построения: не меняется O(N*logN)
Память: x3, но остается O(N*logN)
Сложность запроса: O(logN) !!!



Мини-задача #44 (1 балл)

https://leetcode.com/problems/count-of-smaller-numbers-after-self/

Решите задачу, используя (небольшую вариацию) дерева отрезков.



Динамическое дерево отрезков

Динамическое дерево отрезков

Задача: пусть индексы от 0 до 10^18 (какой-то очень большой константы C).

Динамическое дерево отрезков

Задача: пусть индексы от 0 до 10^18 (какой-то очень большой константы C).

Изначально по каждому индексы "записан" 0.

Динамическое дерево отрезков

Задача: пусть индексы от 0 до 10^18 (какой-то очень большой константы C).

Изначально по каждому индексы "записан" 0.

Динамическое дерево отрезков

Задача: пусть индексы от 0 до 10^18 (какой-то очень большой константы C).

Изначально по каждому индексы "записан" 0.

Необходимо снова поддержать операции следующего вида:

1. update(pos, val) - обновляем значение элемента a_pos
2. sum(l, r) - найти сумму элементов: a_l + … _ a_r

Динамическое дерево отрезков

Задача: пусть индексы от 0 до 10^18 (какой-то очень большой константы C).

Изначально по каждому индексы "записан" 0.

Необходимо снова поддержать операции следующего вида:

1. update(pos, val) - обновляем значение элемента a_pos
2. sum(l, r) - найти сумму элементов: a_l + … _ a_r

Сделать обычное дерево отрезков тяжеловато, слишком уж много элементов. Зато они пустые изначально! Идеи?

Задача: индексы от 0 до C. Изначально по каждому индексы "записан" 0. Поддержать update и getSum

Идея: давайте делать вид, что у нас дерево отрезков, но выделять память только тогда, когда без этого не обойтись.

Задача: индексы от 0 до C. Изначально по каждому индексы "записан" 0. Поддержать update и getSum

Идея: давайте делать вид, что у нас дерево отрезков, но выделять память только тогда, когда без этого не обойтись.

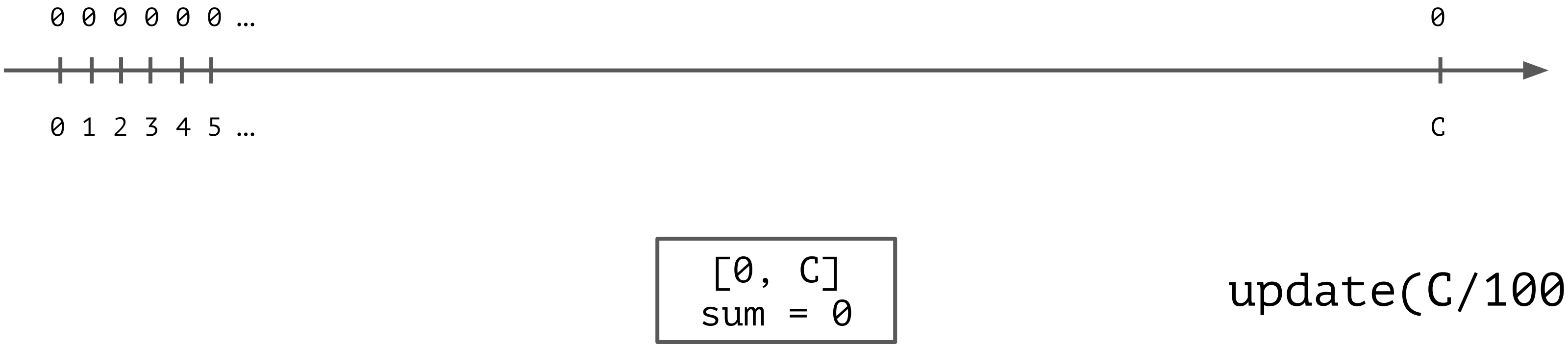
А когда без памяти не обойтись?

Задача: индексы от 0 до C. Изначально по каждому индексы "записан" 0. Поддержать update и getSum

Идея: давайте делать вид, что у нас дерево отрезков, но выделять память только тогда, когда без этого не обойтись.

А когда без памяти не обойтись? Когда делаете update!

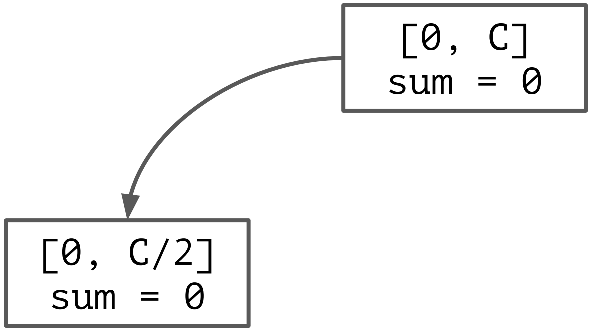
Задача: индексы от 0 до C. Изначально по каждому индексы "записан" 0. Поддержать update и getSum



update(C/100, 42)

Задача: индексы от 0 до C. Изначально по каждому индексы "записан" 0. Поддержать update и getSum

левого сына просто
не было, заметили
это и создали его

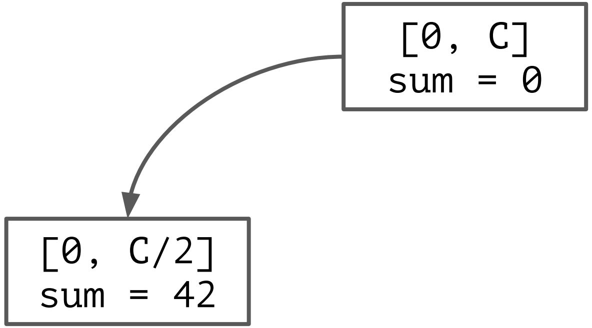


update(C/100, 42)

Задача: индексы от 0 до C. Изначально по каждому индексы "записан" 0. Поддержать update и getSum

левого сына просто не было, заметили это и создали его

sum тоже сразу можем выставить



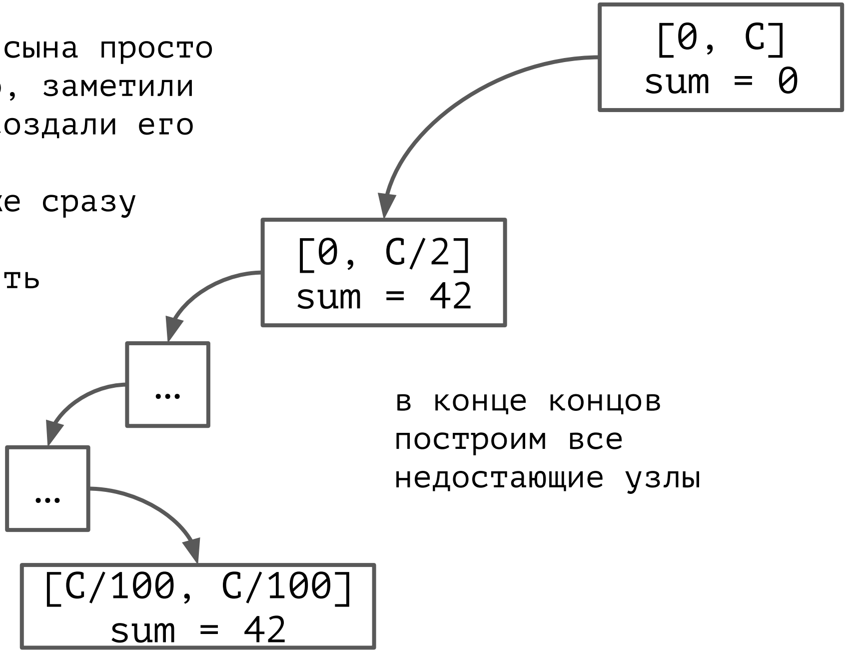
update(C/100, 42)

Задача: индексы от 0 до C. Изначально по каждому индексы "записан" 0. Поддержать update и getSum

update(C/100, 42)

левого сына просто не было, заметили это и создали его

sum тоже сразу можем выставить



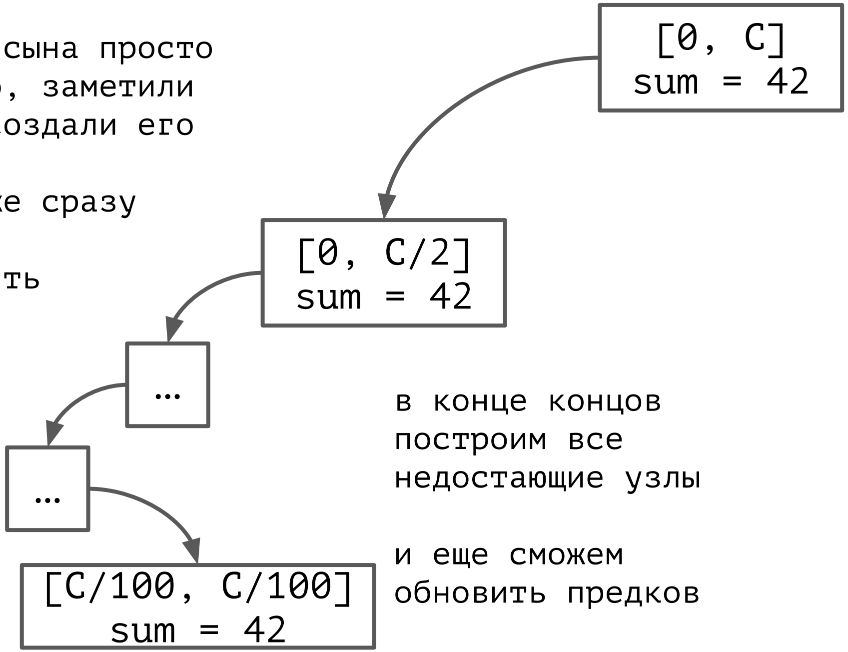
в конце концов построим все недостающие узлы

Задача: индексы от 0 до C. Изначально по каждому индексы "записан" 0. Поддержать update и getSum

update(C/100, 42)

левого сына просто не было, заметили это и создали его

sum тоже сразу можем выставить



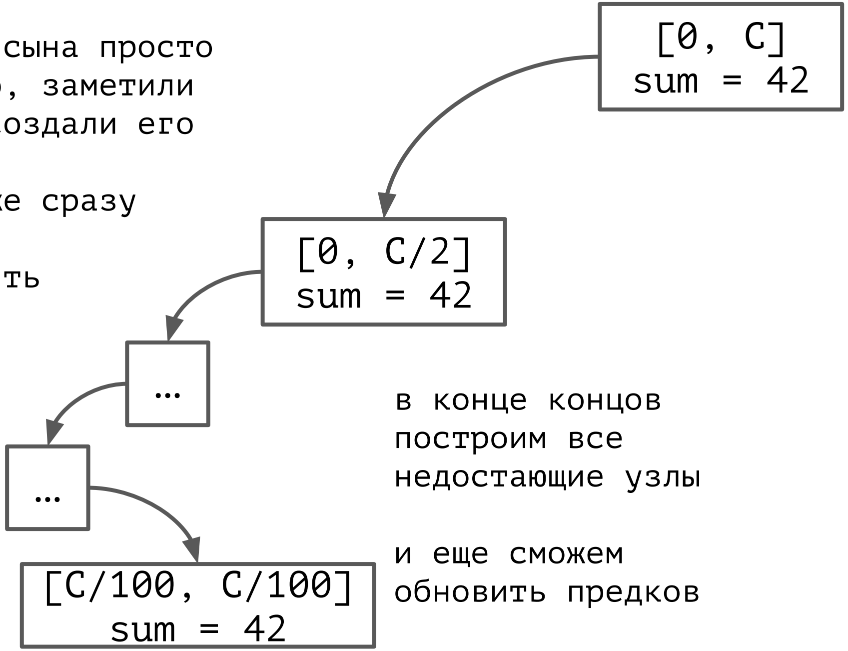
в конце концов построим все недостающие узлы

и еще сможем обновить предков

Задача: индексы от 0 до C. Изначально по каждому индексы "записан" 0. Поддержать update и getSum

левого сына просто не было, заметили это и создали его

sum тоже сразу можем выставить



update(C/100, 42)

getSum(C/2 + 1, 2*C / 3)

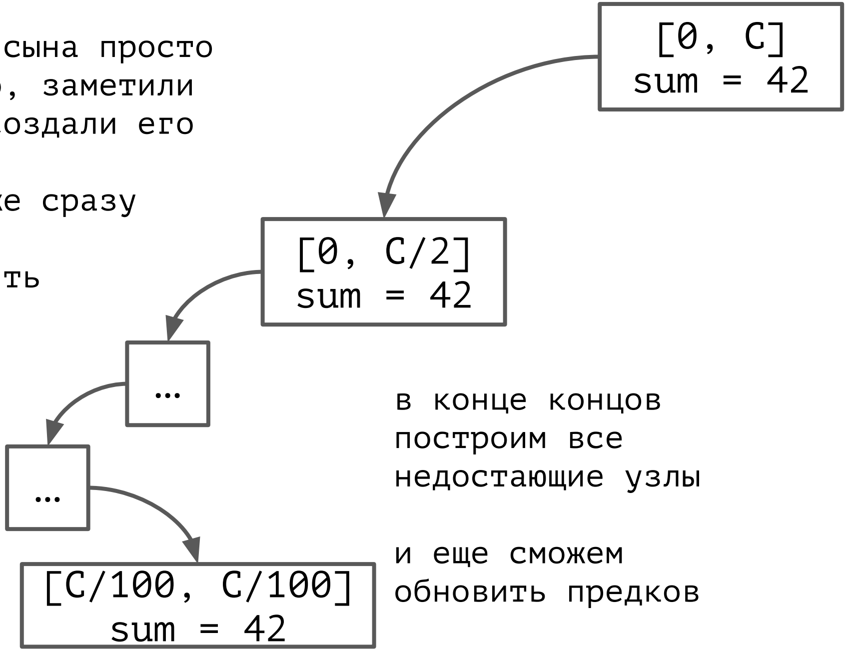
в конце концов построим все недостающие узлы

и еще сможем обновить предков

Задача: индексы от 0 до C. Изначально по каждому индексы "записан" 0. Поддержать update и getSum

левого сына просто не было, заметили это и создали его

sum тоже сразу можем выставить



в конце концов построим все недостающие узлы

и еще сможем обновить предков

update(C/100, 42)

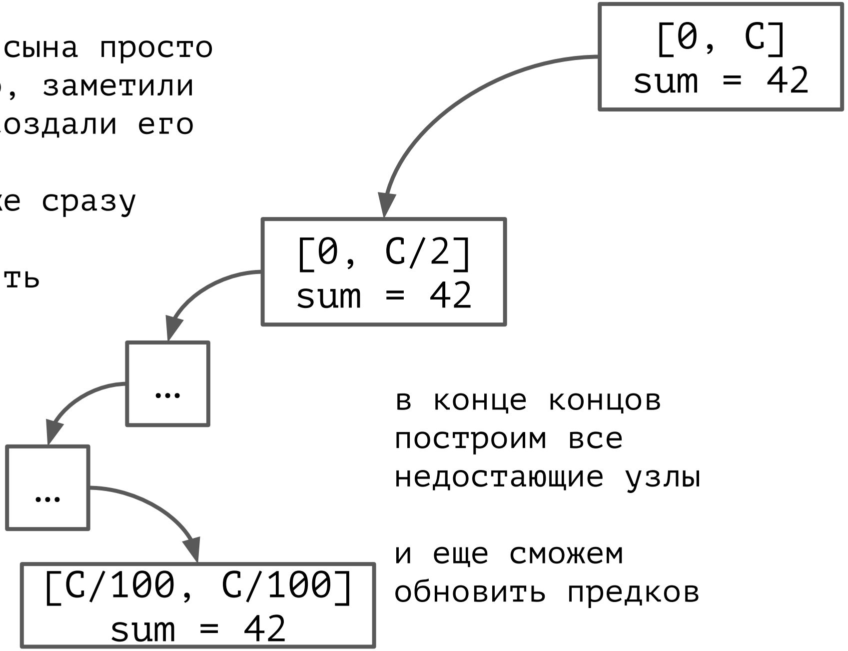
getSum(C/2 + 1, 2*C / 3)

При попытке спустится в правого сына поймем, что его нет! А сколько значит там сумма?

Задача: индексы от 0 до C. Изначально по каждому индексы "записан" 0. Поддержать update и getSum

левого сына просто не было, заметили это и создали его

sum тоже сразу можем выставить



в конце концов построим все недостающие узлы

и еще сможем обновить предков

update(C/100, 42)

getSum(C/2 + 1, 2*C / 3)

При попытке спустится в правого сына поймем, что его нет! А сколько значит там сумма?

0, ведь никто не делал update => сразу возвращаемся

Динамическое дерево отрезков

Задача: индексы от 0 до C. Изначально по каждому индексы "записан" 0. Поддержать update и getSum

Сложность?

Динамическое дерево отрезков

Задача: индексы от 0 до C. Изначально по каждому индексы "записан" 0. Поддержать update и getSum

Сложность?

Операции будут O(logC), худшие случаи, конечно, ужасные.

Динамическое дерево отрезков

Задача: индексы от 0 до C. Изначально по каждому индексы "записан" 0. Поддержать update и getSum

Сложность?

Операции будут O(logC), худшие случаи, конечно, ужасные. Каждый update может потратить O(logC) памяти.

Динамическое дерево отрезков

Задача: индексы от 0 до C. Изначально по каждому индексы "записан" 0. Поддержать update и getSum

Сложность?

Операции будут O(logC), худшие случаи, конечно, ужасные. Каждый update может потратить O(logC) памяти.

При определенном характере запросов имеет смысл: например, когда мало update-ов (но они могут обновлять очень далекие элементы).

Динамическое дерево отрезков

Задача: индексы от 0 до C. Изначально по каждому индексы "записан" 0. Поддержать update и getSum

Замечание по реализации:

Больше хранить просто массив и обращаться к индексам 2*v и 2*v + 1 в поисках детей не выйдет (их же нет).

Динамическое дерево отрезков

Задача: индексы от 0 до C. Изначально по каждому индексы "записан" 0. Поддержать update и getSum

Замечание по реализации:

Больше хранить просто массив и обращаться к индексам 2*v и 2*v + 1 в поисках детей не выйдет (их же нет).

Можем хранить динамический массив (и добавлять их), но все равно каждая вершина должна знать индексы своих детей (их придется хранить).

Дерево отрезков: многомерный случай

Дерево отрезков: многомерный случай

Задача: пусть есть 2D массив фиксированного размера n*m. Пусть для простоты n и m - это степень двойки.

```
a_11, a_12, a_13, …, a_1m
a_21, a_22, a_23, …, a_2m
.........................
a_n1, a_n2, a_n3, …, a_nm
```

Дерево отрезков: многомерный случай

Задача: пусть есть 2D массив фиксированного размера n*m. Пусть для простоты n и m - это степень двойки.

```
a_11, a_12, a_13, …, a_1m
a_21, a_22, a_23, …, a_2m
.........................
a_n1, a_n2, a_n3, …, a_nm
```

Реализовать:

1. update(x, y, val) - обновляем значение элемента a_xy

2. sum(lx, rx, ly, ry) - найти сумму элементов на прямоугольнике с заданными границами

Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

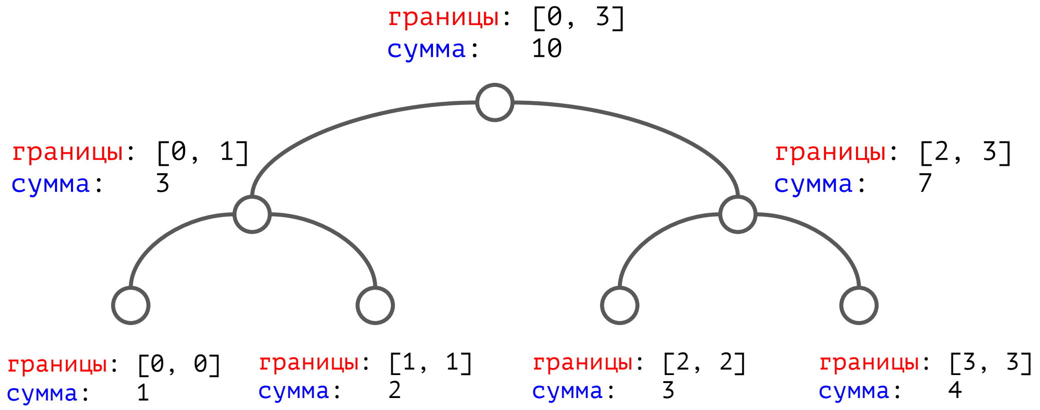
Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

Сначала построим дерево отрезков для каждой из строчек (т.е. по координате x)

Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2



Сначала построим дерево отрезков для каждой из строчек (т.е. по координате x)

Дерево отрезков: многомерный случай

```
1 2 3 4          10 3  7  1 2 3 4
5 6 7 8
9 8 7 6
5 4 3 2
```

Сначала построим дерево отрезков для каждой из строчек (будем сразу хранить массивом)

Дерево отрезков: многомерный случай

1	2	3	4	←	10	3	7	1	2	3	4
5	6	7	8	←	26	11	15	5	6	7	8
9	8	7	6								
5	4	3	2								

Сначала построим дерево отрезков для каждой из строчек (будем сразу хранить массивом)

Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

10	3	7	1	2	3	4
26	11	15	5	6	7	8
30	17	13	9	8	7	6

Сначала построим дерево отрезков для каждой из строчек (будем сразу хранить массивом)

Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

10	3	7	1	2	3	4
26	11	15	5	6	7	8
30	17	13	9	8	7	6
14	9	5	5	4	3	2

Сначала построим дерево отрезков для каждой из строчек (будем сразу хранить массивом)

Дерево отрезков: многомерный случай

1	2	3	4	←	10	3	7	1	2	3	4
5	6	7	8	←	26	11	15	5	6	7	8
9	8	7	6	←	30	17	13	9	8	7	6
5	4	3	2	←	14	9	5	5	4	3	2

Теперь сами полученные элементы тоже соберем в дерево отрезков

Сначала построим дерево отрезков для каждой из строчек (будем сразу хранить массивом)

Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

10	3	7	1	2	3	4
26	11	15	5	6	7	8
30	17	13	9	8	7	6
14	9	5	5	4	3	2

Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

10	3	7	1	2	3	4
26	11	15	5	6	7	8
30	17	13	9	8	7	6
14	9	5	5	4	3	2

10 3 7 1 2 3 4	26 11 15 5 6 7 8	30 17 13 9 8 7 6	14 9 5 5 4 3 2

Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

10	3	7	1	2	3	4
26	11	15	5	6	7	8
30	17	13	9	8	7	6
14	9	5	5	4	3	2

36 14 21 6 8 10 12

44 26 18 14 12 10 8

10 3 7 1 2 3 4

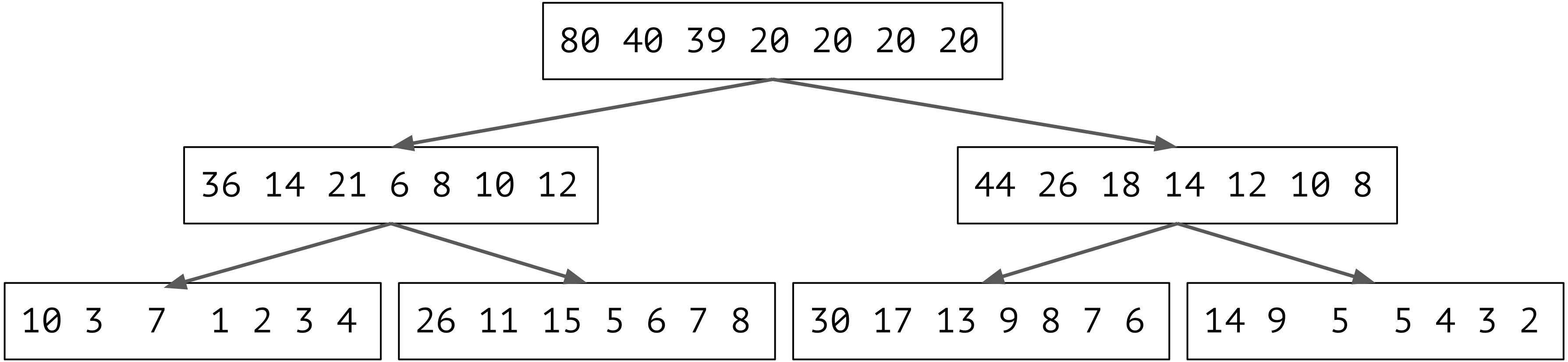
26 11 15 5 6 7 8

30 17 13 9 8 7 6

14 9 5 5 4 3 2

Дерево отрезков: многомерный случай

1 2 3 4	10 3 7 1 2 3 4
5 6 7 8	26 11 15 5 6 7 8
9 8 7 6	30 17 13 9 8 7 6
5 4 3 2	14 9 5 5 4 3 2



Дерево отрезков: многомерный случай

1	2	3	4	←	10	3	7	1	2	3	4
5	6	7	8	←	26	11	15	5	6	7	8
9	8	7	6	←	30	17	13	9	8	7	6
5	4	3	2	←	14	9	5	5	4	3	2

Получили дерево отрезков по координате y

80 40 39 20 20 20 20

36 14 21 6 8 10 12

44 26 18 14 12 10 8

10 3 7 1 2 3 4

26 11 15 5 6 7 8

30 17 13 9 8 7 6

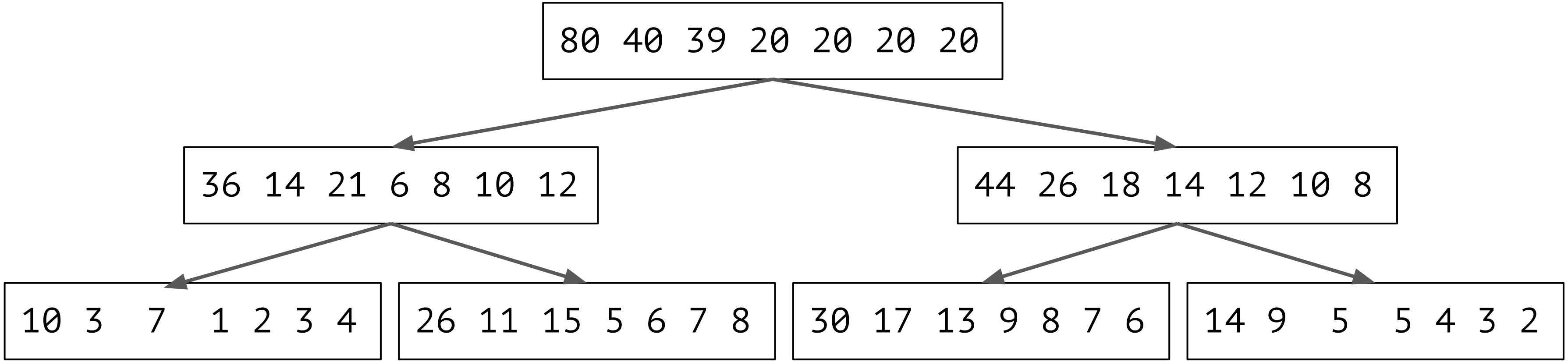
14 9 5 5 4 3 2

Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

10	3	7	1	2	3	4
26	11	15	5	6	7	8
30	17	13	9	8	7	6
14	9	5	5	4	3	2

sum(2, 3, 1, 2)?

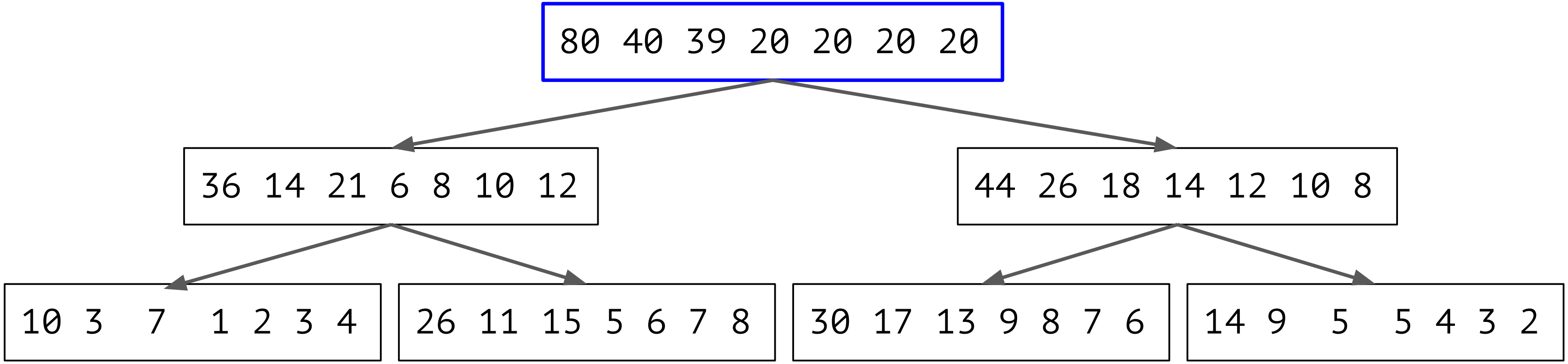


Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

10	3	7	1	2	3	4
26	11	15	5	6	7	8
30	17	13	9	8	7	6
14	9	5	5	4	3	2

sum(2, 3, 1, 2)?

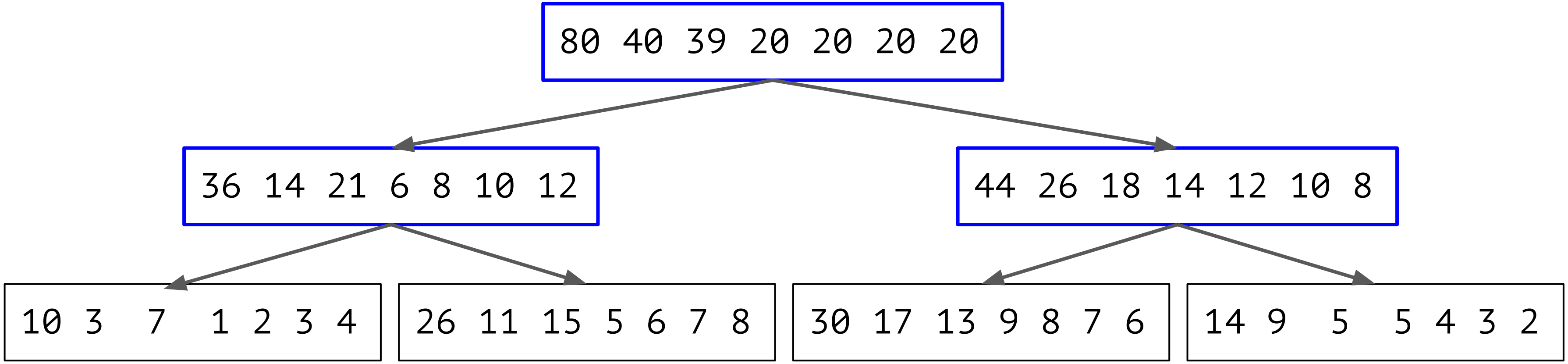


Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

10	3	7	1	2	3	4
26	11	15	5	6	7	8
30	17	13	9	8	7	6
14	9	5	5	4	3	2

sum(2, 3, 1, 2)?

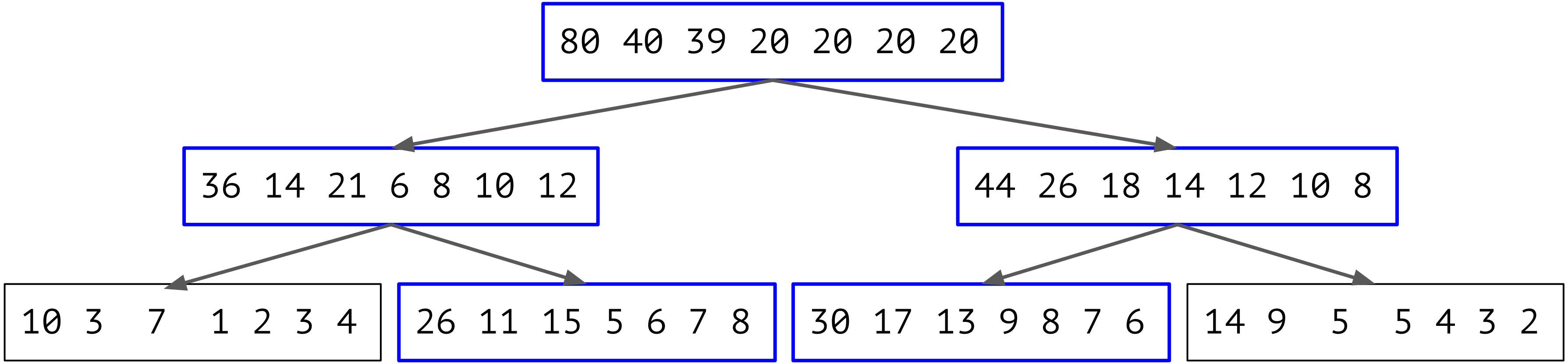


Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

10	3	7	1	2	3	4
26	11	15	5	6	7	8
30	17	13	9	8	7	6
14	9	5	5	4	3	2

sum(2, 3, 1, 2)?

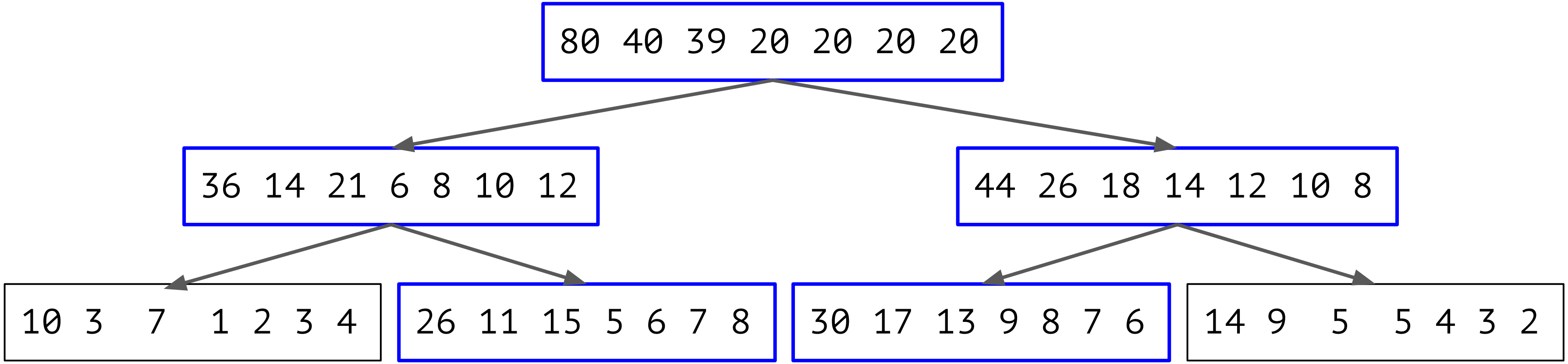


Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

10	3	7	1	2	3	4
26	11	15	5	6	7	8
30	17	13	9	8	7	6
14	9	5	5	4	3	2

sum(2, 3, 1, 2)?



Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

10	3	7	1	2	3	4
26	11	15	5	6	7	8
30	17	13	9	8	7	6
14	9	5	5	4	3	2

sum(2, 3, 1, 2)?

80 40 39 20 20 20 20

36 14 21 6 8 10 12

44 26 18 14 12 10 8

10 3 7 1 2 3 4

26 11 **15** 5 6 7 8

30 17 **13** 9 8 7 6

14 9 5 5 4 3 2

Дерево отрезков: многомерный случай

1	2	3	4		10	3	7	1	2	3	4
5	6	7	8		26	11	15	5	6	7	8
9	8	7	6		30	17	13	9	8	7	6
5	4	3	2		14	9	5	5	4	3	2

sum(2, 3, 1, 2) =
15 + 13 = 28

80 40 39 20 20 20 20

36 14 21 6 8 10 12 | 44 26 18 14 12 10 8

10 3 7 1 2 3 4 | 26 11 **15** 5 6 7 8 | 30 17 **13** 9 8 7 6 | 14 9 5 5 4 3 2

Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

80	40	39	20	20	20	20
36	14	21	6	8	10	12
44	26	18	14	12	10	8
10	3	7	1	2	3	4
26	11	15	5	6	7	8
30	17	13	9	8	7	6
14	9	5	5	4	3	2

Само двумерное дерево тоже представляем, как массив (правда, теперь двумерный)

Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

80	40	39	20	20	20	20
36	14	21	6	8	10	12
44	26	18	14	12	10	8
10	3	7	1	2	3	4
26	11	15	5	6	7	8
30	17	13	9	8	7	6
14	9	5	5	4	3	2

Само двумерное дерево тоже представляем, как массив (правда, теперь двумерный)

Правила как раньше: у вершины v дети 2*v и 2*v + 1

Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

80	40	39	20	20	20	20
36	14	21	6	8	10	12
44	26	18	14	12	10	8
10	3	7	1	2	3	4
26	11	15	5	6	7	8
30	17	13	9	8	7	6
14	9	5	5	4	3	2

Само двумерное дерево тоже представляем, как массив (правда, теперь двумерный)

Правила как раньше: у вершины v дети 2*v и 2*v + 1

Памяти потратим?

Дерево отрезков: многомерный случай

1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

80	40	39	20	20	20	20
36	14	21	6	8	10	12
44	26	18	14	12	10	8
10	3	7	1	2	3	4
26	11	15	5	6	7	8
30	17	13	9	8	7	6
14	9	5	5	4	3	2

Само двумерное дерево тоже представляем, как массив (правда, теперь двумерный)

Правила как раньше: у вершины v дети 2*v и 2*v + 1

Памяти потратим: 4*n*4*m = 16nm (если не степени двойки)

Дерево отрезков: детали реализации

```
t = [0] * (4*N)
build(a, 1, 0, N - 1)


def getSum(v, tl, tr, l, r: int) -> int:
  if l == tl and r == tr:
    return t[v]

  tm = (tl + tr) >> 1
  res = 0

  if l <= tm:
      res += getSum(v*2, tl, tm, l, min(r, tm))

  if r >= tm + 1:
      res += getSum(v*2 + 1, tm + 1, tr, max(l, tm + 1), r)

  return res
```

Дерево отрезков: многомерный случай

```
def getSumY(vx, vy, tly, try, ly, ry) -> int:
  if tly == ly && ry == try:
    return t[vx][vy]

  tmy = (tly + try) >> 1
  res = 0

  if ly <= tmy:
    res += getSumY(vx, vy*2, tmy, ly, min(ry, tmy))

  if ry >= tmy + 1:
    res += getSumY(vx, vy*2 + 1, tmy + 1, try, max(ly, tmy + 1), ry)

  return res
```

Дерево отрезков: многомерный случай

```
def getSumY(vx, vy, tly, try, ly, ry) -> int:
  ...


def getSumX(vx, tlx, trx, lx, rx, ly, ry) -> int:
  if tlx == lx && rx == trx:
    return getSumY(vx, 1, 0, m - 1, ly, ry);

  tmx = (tlx + trx) >> 1
  res = 0

  if lx <= tmx:
    res += getSumX(vx*2, tlx, tmx, lx, min(rx, tmx), ly, ry)

  if rx >= tmx + 1:
    res += getSumX(vx*2 + 1, tmx + 1, trx, max(lx, tmx + 1), rx, ly, ry)

  return res
```

Дерево отрезков: многомерный случай

```
def getSumY(vx, vy, tly, try, ly, ry) -> int:
  ...


def getSumX(vx, tlx, trx, lx, rx, ly, ry) -> int:
  if tlx == lx && rx == trx:
    return getSumY(vx, 1, 0, m - 1, ly, ry);

  tmx = (tlx + trx) >> 1
  res = 0

  if lx <= tmx:
    res += getSumX(vx*2, tlx, tmx, lx, min(rx, tmx), ly, ry)

  if rx >= tmx + 1:
    res += getSumX(vx*2 + 1, tmx + 1, trx, max(lx, tmx + 1), rx, ly, ry)

  return res
```

Сложность:
O(logn*logm)

Дерево отрезков: многомерный случай

- Дерево отрезков легко обобщается на многомерный (не только двумерный!) случай,

Дерево отрезков: многомерный случай

- Дерево отрезков легко обобщается на многомерный (не только двумерный!) случай,

- Принцип всегда одинаковый: храним в каждом элемент дерева отрезков новое дерево отрезков

- Сложность операций будет O(logP1*logP2*...*logPk), где P1,P2,...Pk - соответствующие размерности

Дерево отрезков: многомерный случай

- Дерево отрезков легко обобщается на многомерный (не только двумерный!) случай,

- Принцип всегда одинаковый: храним в каждом элемент дерева отрезков новое дерево отрезков

- Сложность операций будет O(logP1*logP2*...*logPk), где P1,P2,...Pk - соответствующие размерности

- В наивной реализации память будет O(4^k*P1*...Pk), но существуют алгоритмы сжатия.

Takeaways

- Видите требование "батч" операций над отрезками => вспоминайте деревья отрезков
- Очень много вариаций базовой идеи!
- Трейдоффы память/скорость запросов
- Легкое обобщение на многомерный случай

https://cp-algorithms.com/data_structures/segment_tree.html