Динамическое программирование

Семинар

Последовательность из 0 и 1

Требуется подсчитать количество последовательностей длины N состоящей из 0 и 1 в которых нет стоящих подряд двух единиц

N = 3

0 0 0
0 0 1
0 1 0
1 0 0
1 0 1
~~0 1 1~~
~~1 1 0~~

Последовательность из 0 и 1

- Попробуем найти рекуррентное соотношение
- Определяем базовые случаи: для **n = 0**, количество последовательностей равно **1** (пустая последовательность), для **n = 1**, количество последовательностей равно **2** (0 или 1).
- Создаем массив dp размером, где **dp[i]** будет хранить количество последовательностей длины i.

1: 0, 1
2: 00, 01, 10
3: 000, 001, 100, 010, 101

Последовательность из 0 и 1

Требуется подсчитать количество последовательностей длины N состоящей из 0 и 1 в которых нет стоящих подряд двух единиц

1: 0, 1
2: 00, 01, 10
3: 000, 001, 100, 010, 101
...
dp[i] = dp[i-1] + dp[i-2]

```
f (n) {
  // определяем базовые случаи
  dp = [1, 2]

  for i = 2; i <= n; n++ {
     dp.append(dp[i - 1] + dp[i - 2])
  }
```

Последовательность без трех единиц подряд

Определите количество последовательностей из нулей и единиц длины, в которых никакие три единицы не стоят рядом.

000 101 011 110 ~~111~~

Последовательность без трех единиц подряд

- Если последний элемент — 0, то перед ним может стоять последовательность любой длины n-1 без трех единиц подряд
- Если последний элемент — 1, рассмотрим два случая:
  - Если предпоследний элемент также 1, то элемент перед рассматриваемой парой единиц не может быть 1 (чтобы не было трех единиц подряд), и перед этим элементом может быть любая последовательность длины n-3.
  - Если предпоследний элемент - 0, перед ним может быть любая последовательность длины n-2.

Таким образом, мы можем выразить количество последовательностей длины n через количество последовательностей более коротких длин:

```
function count_sequences(n) {
    if n == 0 || n == 1 || n == 2 {
        return n
    }
    dp = [1, 2, 4]
    ...
}
```

Последовательность без трех единиц подряд

- $dp[0] = 1$(пустая последовательность)
- $dp[1] = 2(0, 1)$
- $dp[2] = 4(00, 01, 10, 11)$
- $dp[3] = 7(000, 001, 010, 011, 100, 101, 110)$
- $dp[4] = 13(0000, 0001, 0010, 0011, 0100, 0101, 0110, 1000, 1001, 1010, 1011, 1100, 1101)$

```
function count_sequences(n) {
    if n == 0 || n == 1 || n == 2 {
        return n
    }

    dp = [1, 2, 4]

    for i = 3; i <= n; i++ {
        ...
    }

    return dp[n]
}
```

Последовательность без трех единиц подряд

dp[n] = dp[n-1] + dp[n-2] + dp[n-3]

```
function count_sequences(n) {
    if n == 0 || n == 1 || n == 2 {
        return n
    }

    dp = [1, 2, 4]

    for i = 3; i <= n; i++ {
        dp.append(dp[i - 1] + dp[i - 2] + dp[i - 3])
    }

    return dp[n]
}
```

Наибольшая возрастающая последовательность

Дан массив не отсортированных чисел. Необходимо найти максимально длинную возрастающую последовательность и вернуть ее длину

[3, 2, 8, 9, 5, 10] - ответ 3 так как максимально длинная последовательность

2, 8, 9

[1, 2, 7, 9, 0, 10] - ответ 4 так как максимально длинная последовательность

1, 2, 7, 9

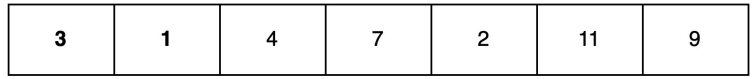
[8, 8, 8, 8] - ответ 1

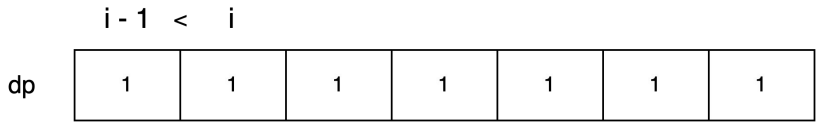
Наибольшая возрастающая последовательность

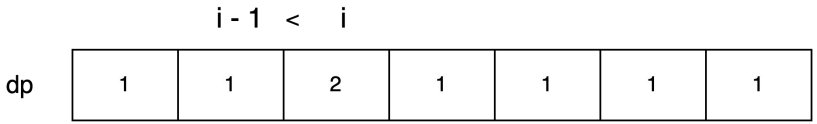
```
function findLIS(nums) {
    // если длина массива
    // равен 0 или 1 то возвращаем
    // создаем дополнительный массив dp
    // размером nums и заполняем 1

    return // длину максимальной последовательности
}
```

Наибольшая возрастающая последовательность

```
function findLIS(nums) {
    if len(nums) == 0 {
        return 0
    }
    if len(nums) == 1 {
        return 1
    }
    dp = [1] * len(nums)
    // в цикле проверяем равенство
    // предыдущего элемента и текущего

    return max(dp)
}
```

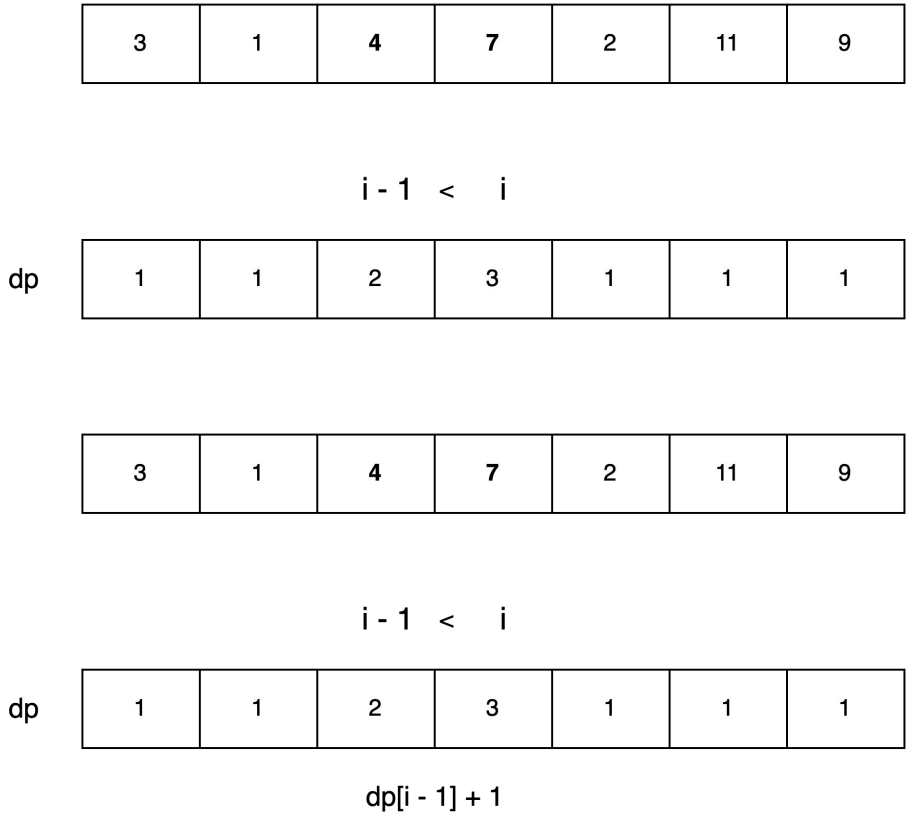






Наибольшая возрастающая последовательность

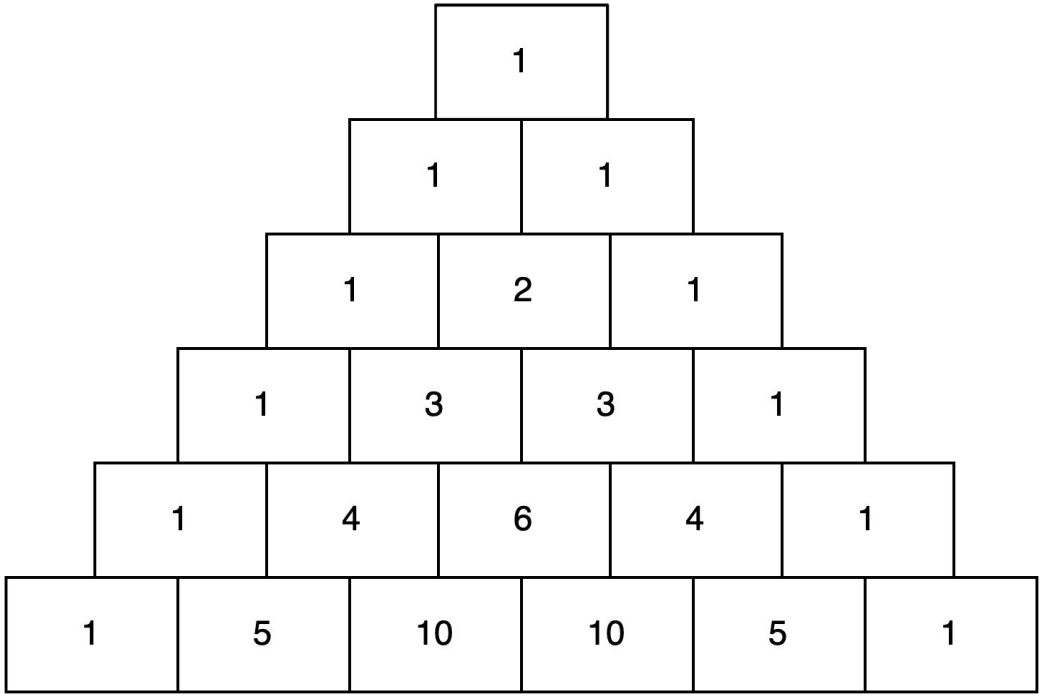
```
function findLIS(nums) {
    if len(nums) == 0 {
        return 0
    }
    if len(nums) == 1 {
        return 1
    }
    dp = [1] * len(nums)
    for i = 1; i < len(nums); i++ {
        if nums[i - 1] < nums[i] {
            dp[i] = dp[i - 1] + 1
        }
    }

    return max(dp)
}
```



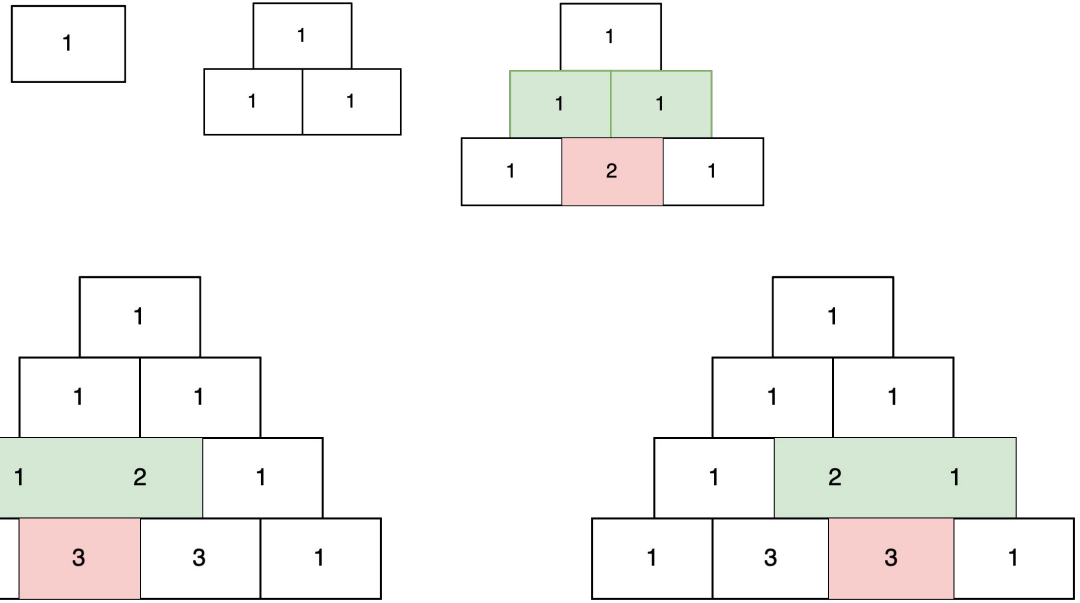
Треугольник Паскаля

Дано некоторое число n. Необходимо создать треугольник Паскаля состоящего из n строк

Треугольник Паскаля

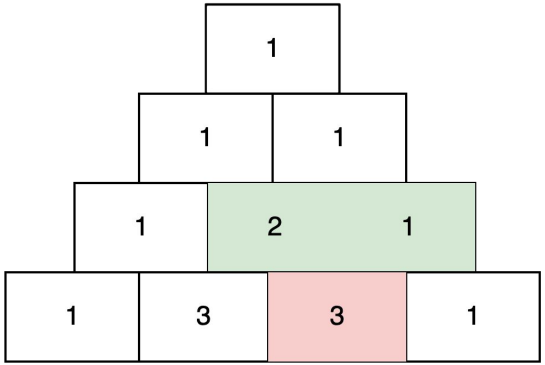


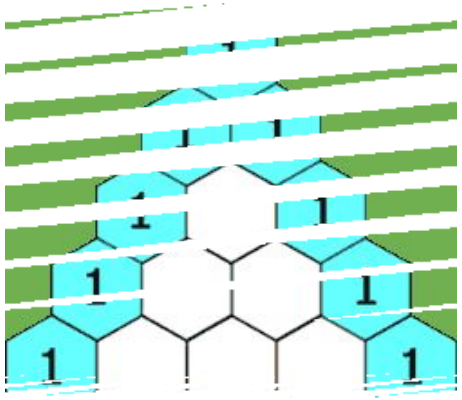
Треугольник Паскаля

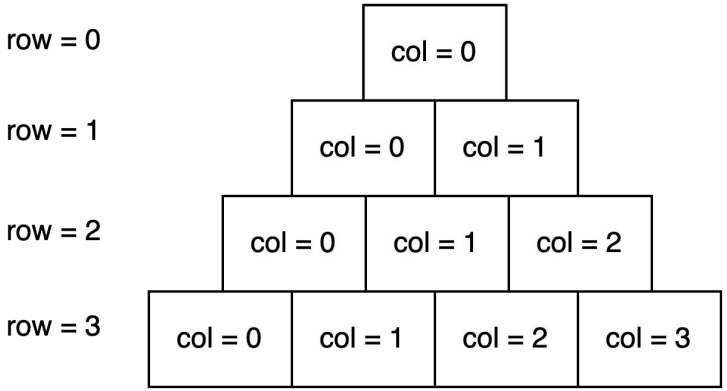


Треугольник Паскаля

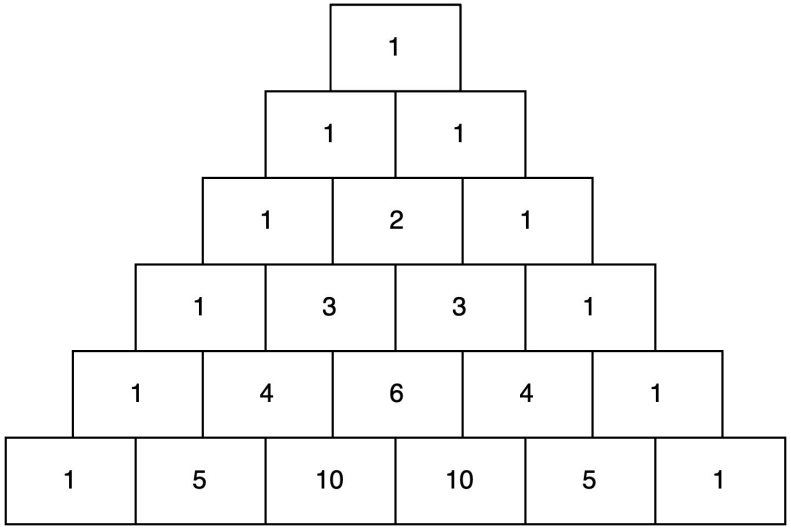
- Пробуем найти соотношение между ячейками
- pascal_triangle[row - 1][col - 1] + pascal_triangle[row - 1][col]





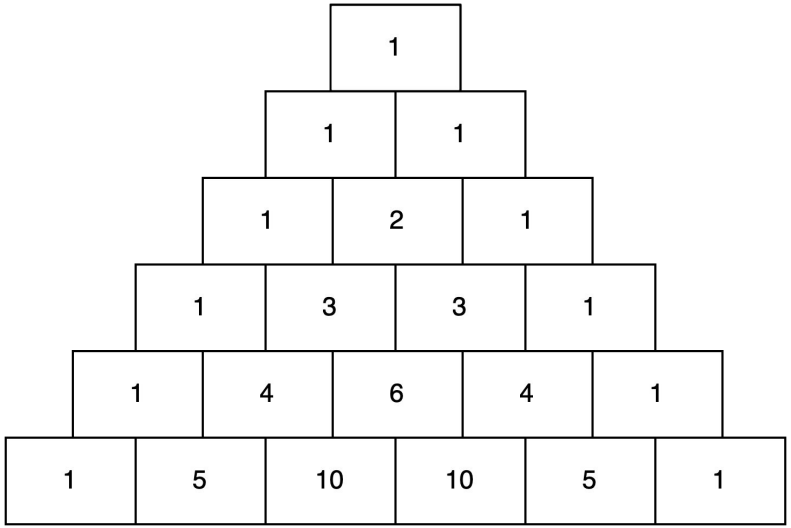


Треугольник Паскаля




```
n = 6
function pascal_triangle(row, col) {
    // если это вершина треугольника
    // или элемент его стороны
    // возвращаем 1
    // вычисляем наше соотношение
}

// выводим полученный массив на экран
dp = []
for row =0; row < n; row++{
    // создаем массив для хранения строк
    // создаем цикл по столбцам
    // для каждой ячейки
    // вычисляем ее значение используя
    // pascal_triangle. Добавляем созданную
    // строку треугольника в dp
}
```

Треугольник Паскаля




```
n = 6
function pascal_triangle(row, col) {
   // либо это вершина треугольника
   // либо элемент его стороны
   if col == 0 or row == col {
      return 1
   } else {
      // наше соотношение
   }
}
```

```
// выводим полученный массив на экран
dp = []
for row in range(n) {
   currentRow = []
   for col = 0; col <= row; col++ {
      // добавляем в строку ячейку
   }
   dp.append(currentRow)
}
```

Треугольник Паскаля

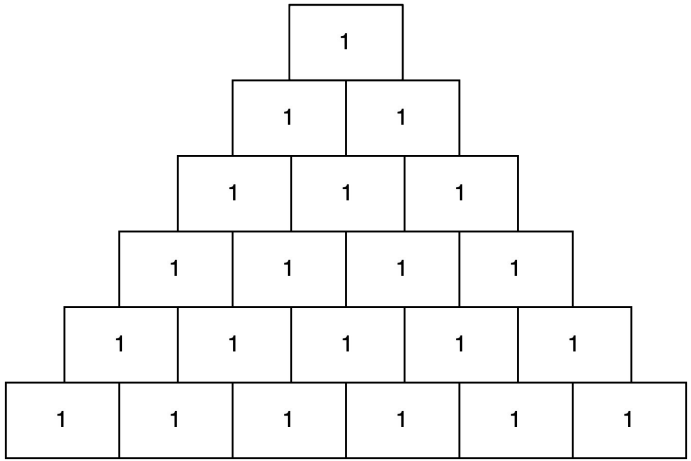
Массив **dp** для 6 строк:

```
[
        [1],
        [1, 1],
        [1, 2, 1],
        [1, 3, 3, 1],
        [1, 4, 6, 4, 1],
        [1, 5, 10, 10, 5, 1]
]
```

```
n = 6
function pascal_triangle(row, col) {
   // либо это вершина треугольника
   // либо элемент его стороны
   if col == 0 or row == col {
      return 1
   } else {
      return pascal_triangle(row - 1, col - 1) +
                pascal_triangle(row - 1, col)
   }
}


// выводим полученный массив на экран
dp = []
for row in range(n) {
   currentRow = []
   for col in range(row + 1) {
      currentRow.append(pascal_triangle(row, col))
   }
   dp.append(currentRow)
}
```

Треугольник Паскаля Итерационный подход

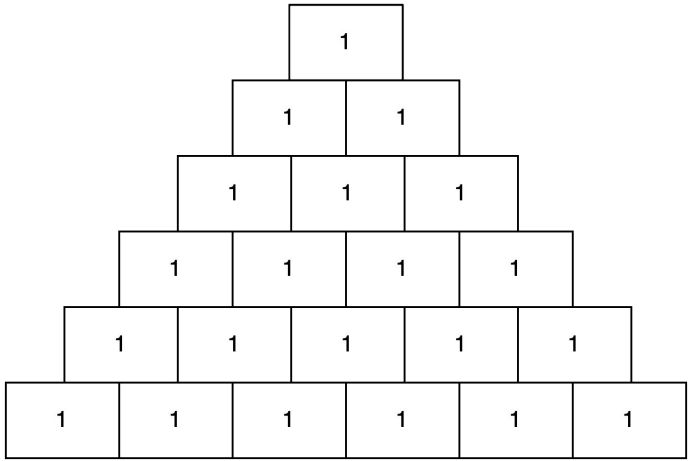
- Проинициализируем двумерный массив и заполним его единицами
- Создадим цикл для строк
- Для каждой строки будем в цикле по столбцам менять ячейки



[[1], [1, 1], [1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1]]

Треугольник Паскаля Итерационный подход

- Проинициализируем двумерный массив и заполним его единицами
- Создадим цикл для строк
- Для каждой строки будем в цикле по столбцам меняем ячейки



[[1], [1, 1], [1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1]]

Треугольник Паскаля Итерационный подход

```
// задаем количество строк в треугольнике
n = 6

// создаем двумерный массив
// для хранения треугольника Паскаля
dp = []
for i = 1; i <= n; i++ {
   tmp = []
   for j = 1; j <= i; j++ {
      tmp.append(1)
   }
   dp.append(tmp)
}

// заполняем массив значениями
// считаем строки и столбцы начиная с единицы
for row = 1; row < n; row++ {
   // для каждой строки будем
   // в цикле по столбцам менять ячейки
}
```

Треугольник Паскаля Итерационный подход

Массив **dp** для 6 строк:

```
[
    [1],
    [1, 1],
    [1, 1, 1],
    [1, 1, 1, 1],
    [1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1]
]
```

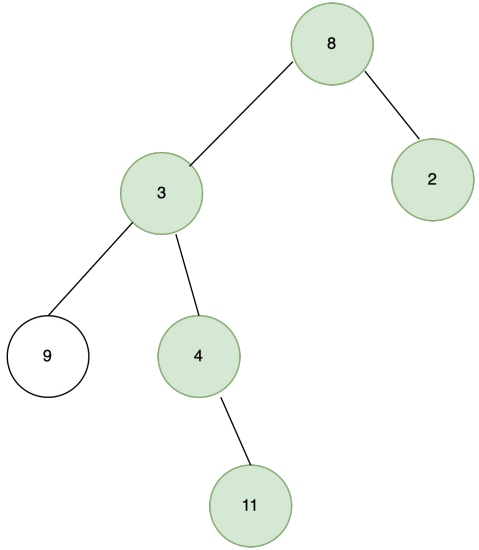
```
// задаем количество строк в треугольнике
n = 6

// создаем двумерный массив
// для хранения треугольника Паскаля
dp = []
for i = 1; i <= n; i++ {
   tmp = []
   for j = 1; j <= i; j++ {
      tmp.append(1)
   }
   dp.append(tmp)
}

// заполняем массив значениями
for row = 1; row < n; row++ {
   for col = 1; col < row; col++ {
      dp[row][col] = dp[row-1][col-1] + dp[row-1][col]
   }
}
```

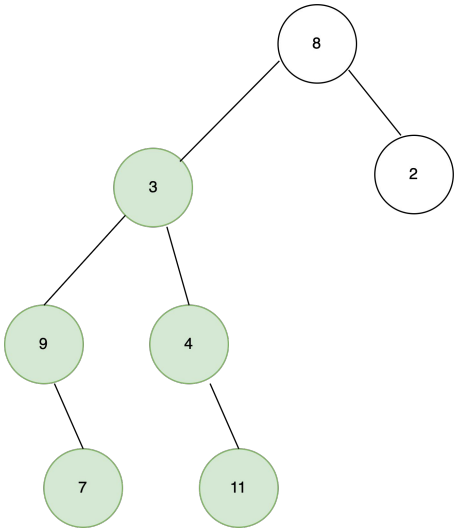
Динамическое программирование на деревьях

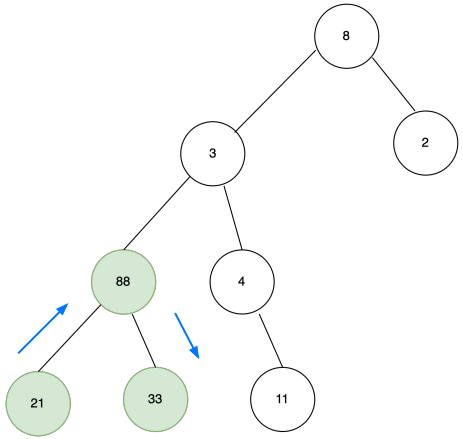
Дано бинарное дерево, необходимо вернуть максимальный путь с максимальной суммой, которая может получиться при проходе дерева по одному пути.

Путь с максимальной суммой

- Один узел учитывается только один раз
- Пути не должны пересекаться. То есть в результате по одной вершине в финальном пути можно пройтись только один раз







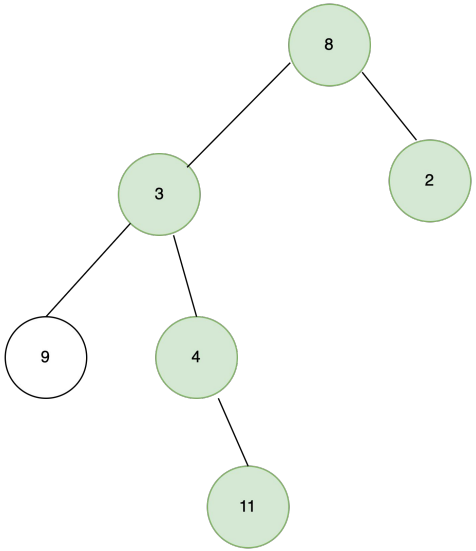

```
type Result struct {
        PathSum int
        Path    []int
}
```

Путь с максимальной суммой

- Задача сводится к обходу в глубину
- Как и раньше начинаем с базовых кейсов
- Теперь нам надо запоминать сумму узлов
- Переписывать путь как только найдем максимальное значение

```
function find_max_path_sum(root) {
  function dfs(node) {
    if not node {
      return 0, []
    }

    // при обходе в глубину теперь запоминаем
    // сумму узлов и путь

    // если сумма левого поддерева больше правого
    // то запоминаем путь через него и наоборот
    // в качестве значения максимальной суммы запоминаем
    // max(left, right) + текущая нода

    // обновляем максимальный путь, если сумма
    // его текущих узлов больше

    return // максимальную сумму для данного поддерева, путь
  }

  max_sum = MIN_INT
  max_path = []
  dfs(root)
  return max_path
}
```

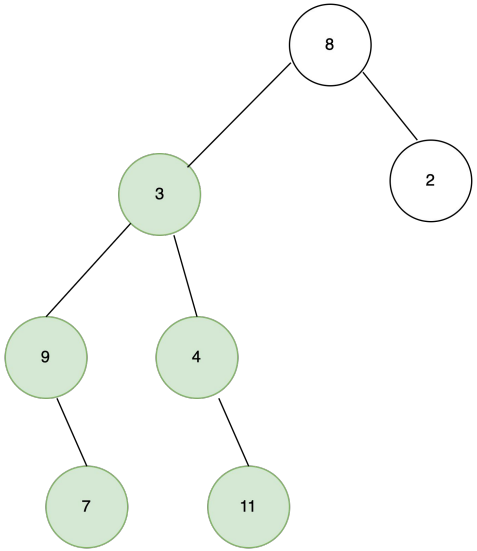
Путь с максимальной суммой



[11, 4, 3, 8, 2]

```
function find_max_path_sum(root) {
  function dfs(node) {
    if not node {
      return 0, []
    }

    left_sum, left_path = dfs(node.left)
    right_sum, right_path = dfs(node.right)

    // если сумма левого поддерева больше правого
    // то запоминаем путь через него и наоборот
    // в значении максимальной суммы запоминаем
    // max(left, right) + текущая нода

    // обновляем максимальный путь, если сумма
    // его текущих узлов больше

    return current_max_sum, current_max_path
  }

  max_sum = MIN_INT
  max_path = []
  dfs(root)
  return max_path
}
```

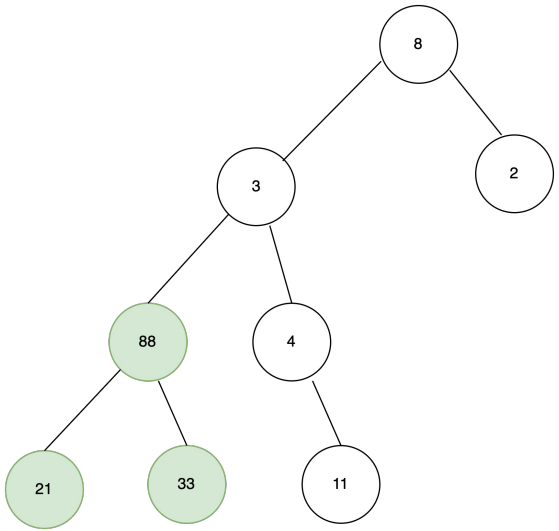
Путь с максимальной суммой



[7, 9, 3, 4, 11]

```
function find_max_path_sum(root) {
  function dfs(node) {
    if not node {
      return 0, []
    }

    left_sum, left_path = dfs(node.left)
    right_sum, right_path = dfs(node.right)

    if left_sum > right_sum {
      current_max_path = left_path + [node.val]
    } else {
      current_max_path = right_path + [node.val]
    }
    current_max_sum = max(left_sum, right_sum) + node.val

    // обновляем максимальный путь, если сумма
    // его текущих узлов больше

    return current_max_sum, current_max_path
  }

  max_sum = MIN_INT
  max_path = []
  dfs(root)
  return max_path
}
```

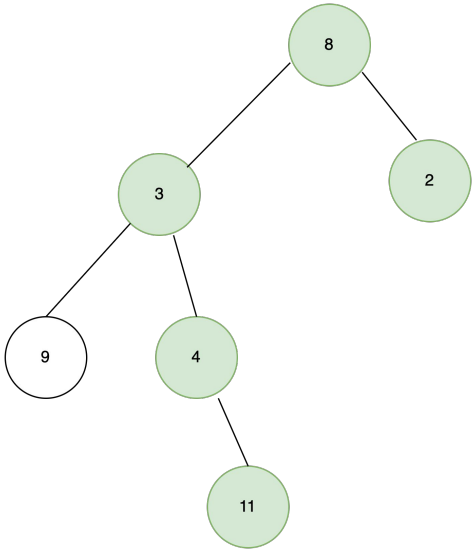
Путь с максимальной суммой



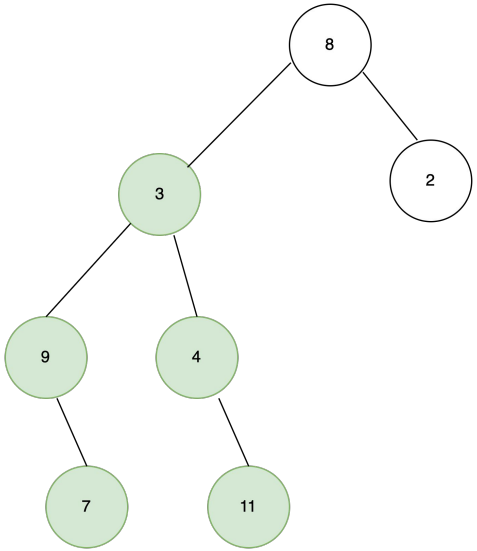
[21, 88, 33]

```
function find_max_path_sum(root) {
  function dfs(node) {
    if not node {
      return 0, []
    }

    left_sum, left_path = dfs(node.left)
    right_sum, right_path = dfs(node.right)

    if left_sum > right_sum {
      current_max_path = left_path + [node.val]
    } else {
      current_max_path = right_path + [node.val]
    }
    current_max_sum = max(left_sum, right_sum) + node.val

    // обновляем максимальный путь, если сумма
    // его текущих узлов больше
    if left_sum + node.val + right_sum > max_sum {
      max_sum = left_sum + node.val + right_sum
      max_path = left_path + [node.val] + right_path
    }

    return current_max_sum, current_max_path
  }

  max_sum = MIN_INT
  max_path = []
  dfs(root)
  return max_path
}
```

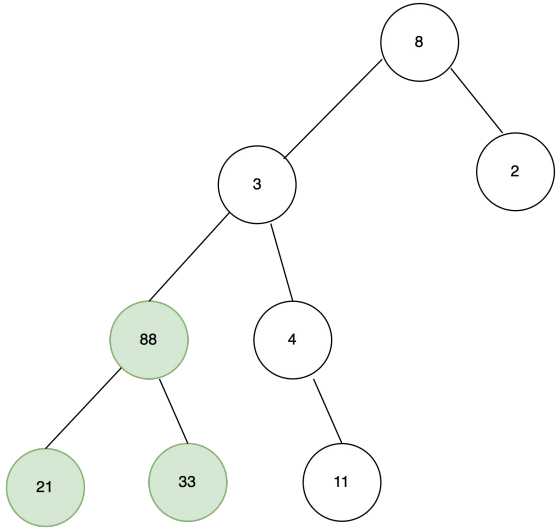
Путь с максимальной суммой



28



37



142

Максимальная выгода

Дан массив целых чисел. Каждое число - стоимость акции. Нам нужно купить максимально дешево, а продать максимально дорого. Сделать это надо за O(n)

Максимальная выгода

[8, 9, 3, 7, 4, 16, 12] - максимальная выгода 13. Купили за 3, продали за 16

[1, 2, 3, 4, 5, 6, 7] - максимальная выгода 6. Купили за 1, продали за 7

[8, 7, 6, 5, 4, 3, 2] - максимальная выгода 0.

Максимальная выгода

Как не надо делать: O(n^2)

	8	9	3	7	4	16	12
8		1	-5	-1	-4	8	4
9			-6	-2	-5	7	-3
3				4	1	**13**	9
7					-3	9	5
4						12	8
16							
12							

Максимальная выгода

```
function maxProfit(prices) {
    // считаем в начале нашу выгоду равную нулю
    // какую акцию возьмем для сравнения с предыдущими?
    // в цикле на каждой итерации обновляем profit и
    // минимально возможную цену

    return profit
}
```

[8, 9, 3, 7, 4, 16, 12] - максимальная выгода 13. Купили за 3, продали за 16

Максимальная выгода

```
function maxProfit(prices) {
    profit = 0
    min_price = prices[0]
    // в цикле на каждой итерации обновляем profit и
    // минимально возможную цену

    return profit
}
```

[8, 9, 3, 7, 4, 16, 12] - максимальная выгода 13. Купили за 3, продали за 16

Максимальная выгода

```
function maxProfit(prices) {
    profit = 0
    min_price = prices[0]
    for currentPriceIndex = 1; currentPriceIndex < length(prices); currentPriceIndex++ {
        profit = max(profit, prices[currentPriceIndex] - min_price)
        min_price = min(prices[currentPriceIndex], min_price)
    }

    return profit
}
```

[8, 9, 3, 7, 4, 16, 12] - максимальная выгода 13. Купили за 3, продали за 16

Всем спасибо

и хороших выходных:))