Сортировки

Филипп Воронов

Филипп Воронов

Teamlead, Поиск Mail.ru

Аккаунты в соц.сетях

@Филипп Воронов

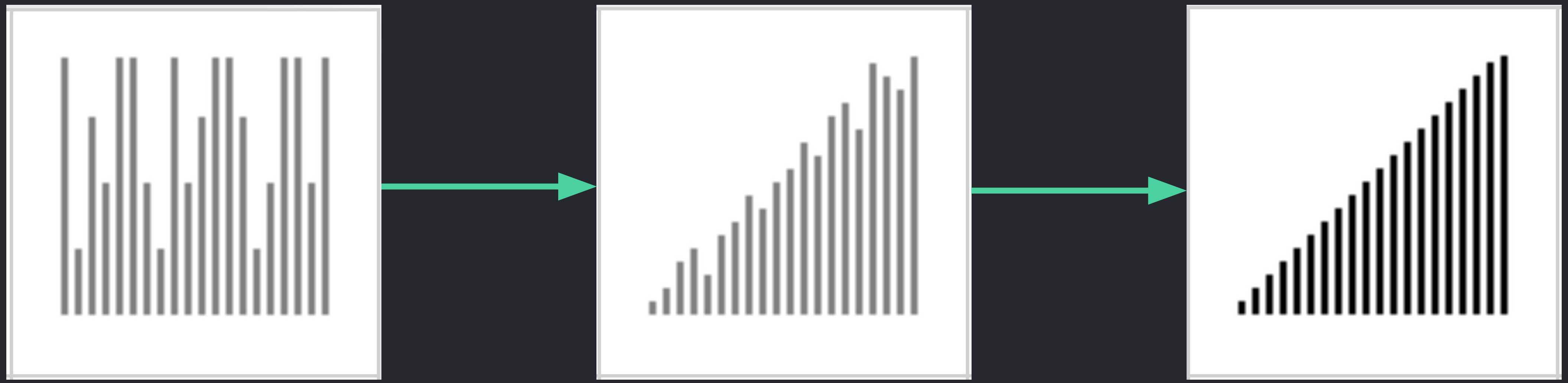
План занятия

1. Что такое сортировка?
2. Сортировка пузырьком
3. Сортировка слиянием
4. «Быстрая» сортировка
5. Линейные сортировки
6. Какую сортировку выбрать?
7. Итоги

Что такое сортировка?

Что такое сортировка?

Сортировка — алгоритм, который упорядочивает элементы массива в порядке возрастания или убывания



1 2 3

Пример сортировки

- Что у нас есть?

Массив из восьми чисел:

53, 65, 11, 89, 93, 54, 30, 41

Исходный массив

53 65 11 89 93 54 30 41

Пример сортировки

- Что у нас есть?

Массив из восьми чисел:

53, 65, 11, 89, 93, 54, 30, 41

- Что сделать?

Отсортировать массив
от меньшего к большему:

11, 30, … …. …. …. …. …. 93

Исходный массив

53 65 11 89 93 54 30 41

?

Пример сортировки

- Что у нас есть?

Массив из восьми чисел:

53, 65, 11, 89, 93, 54, 30, 41

- Что сделать?

Отсортировать массив
от меньшего к большему:

11, 30, … …. …. …. …. …. 93

Исходный массив

53 65 11 89 93 54 30 41

Сортировка

Отсортированный массив

11 30 41 53 54 65 89 93

Сортировка пузырьком

Сортировка пузырьком

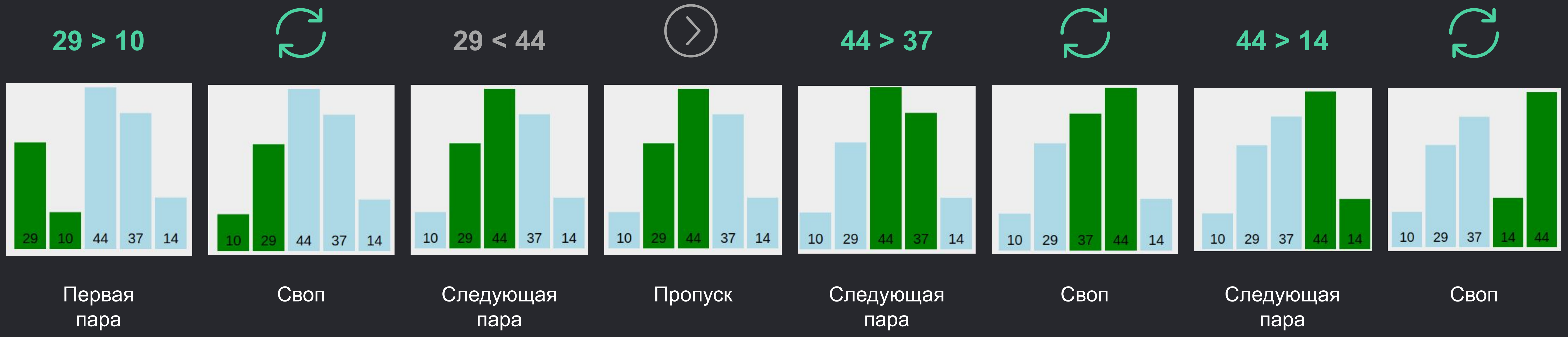
Динамический пример сортировки пузырьком:

[Bubble Sort — Sorting Algorithm Animations]

[Смотреть другую визуализацию >>]
На сайте вверху нажать на BUB

Сортировка пузырьком

Сортировка пузырьком



В последней ячейке массива «всплыл» самый большой элемент — 44

Сортировка пузырьком

Всплывание пузырька:

> for i от 0 до n-1 — задаём цикл по индексам массива (for)
> if arr[i] > arr[i+1] — проверяем, что текущий элемент больше, чем следующий (if)

swap arr[i] arr[i+1] — меняем местами два элемента (swap)

Алгосложность

Обычный цикл по массиву с константной итерацией, стало быть время $O(n)$ и дополнительная память $O(1)$

Сортировка пузырьком

Проведём операцию всплывания n-1 раз, каждый раз будет всплывать самый большой из невсплывших элементов в конец к всплывшим элементам

```
for i от 0 до n-1
  for j от 0 до n-i-1
    if arr[j] > arr[j+1]
      swap arr[j] arr[j+1]
```

Визуализация -> BUBBLE SORT

Сортировка пузырьком

```
for i от 0 до n-1
  for j от 0 до n-i-1
    if arr[j] > arr[j+1]
      swap arr[j] arr[j+1]
```

Визуализация -> BUBBLE SORT

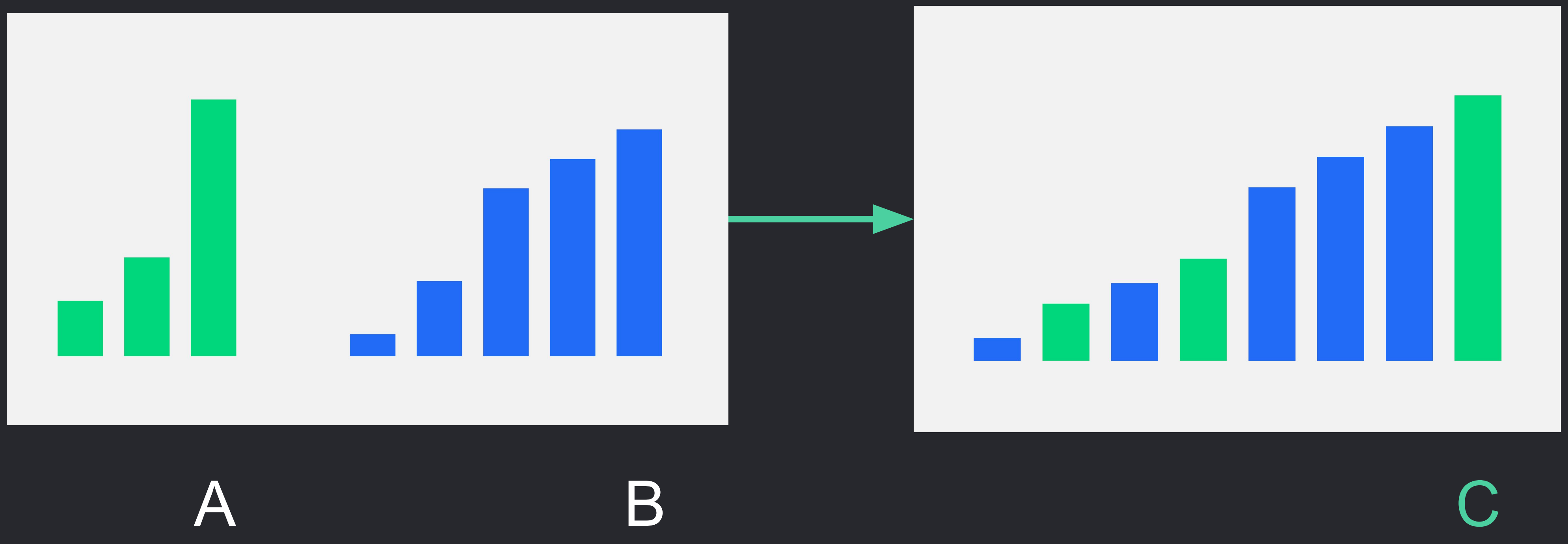
Алгосложность. Теперь у нас двойной цикл, мы порядка n раз делаем операцию всплывания за $O(n)$. Поэтому общее время $O(n^2)$ и дополнительная память $O(1)$. Сортировки за квадратичное время называются квадратичными и для больших n считаются слишком медленными

Сортировка слиянием

Операция слияния

Слияние — операция, соединяющая в один массив два и более отсортированных массива.

Например, мы взяли набор упорядоченных, но неодинаковых по количеству столбцов A и B. Путём их сложения и упорядочивания получили C



Операция слияния

```
merge(A, B):
  C = [длина(A) + длина(B) нулей]
  ia = 0
  ib = 0
  ic = 0

while ia < длина(A) или ib < длина(B)
  if ia = длина(A)  // A закончился
    C[ic] = B[ib]
    ib += 1
else if ib = длина(B)  // B закончился
    C[ic] = A[ia]
    ia += 1
  else
    if A[ia] <= B[ib]
      C[ic] = A[ia]
      ia += 1
    else
      C[ic] = B[ib]
      ib += 1
      ic += 1
  return C
```

Счётчики в соответствующих массивах

Операция слияния

```
merge(A, B):
  C = [длина(A) + длина(B) нулей]
  ia = 0
  ib = 0
  ic = 0

while ia < длина(A) или ib < длина(B)
  if ia = длина(A)  // A закончился
    C[ic] = B[ib]
    ib += 1
else if ib = длина(B)  // B закончился
    C[ic] = A[ia]
    ia += 1
  else
    if A[ia] <= B[ib]
      C[ic] = A[ia]
      ia += 1
    else
      C[ic] = B[ib]
      ib += 1
      ic += 1
  return C
```

Счётчики в соответствующих массивах

Пока хотя бы один из счётчиков для исходных массивов не прошёл массив до конца

Операция слияния

```
merge(A, B):
  C = [длина(A) + длина(B) нулей]
  ia = 0
  ib = 0
  ic = 0

while ia < длина(A) или ib < длина(B)
  if ia = длина(A)  // A закончился
    C[ic] = B[ib]
    ib += 1
else if ib = длина(B)  // B закончился
    C[ic] = A[ia]
    ia += 1
  else
    if A[ia] <= B[ib]
      C[ic] = A[ia]
      ia += 1
    else
      C[ic] = B[ib]
      ib += 1
      ic += 1
  return C
```

Счётчики в соответствующих массивах

Пока хотя бы один из счётчиков для исходных массивов не прошёл массив до конца

Если счётчик уже пробежал все элементы в A, то берём следующий из B в качестве следующего элемента для C

Операция слияния

```
merge(A, B):
  C = [длина(A) + длина(B) нулей]
  ia = 0
  ib = 0
  ic = 0

while ia < длина(A) или ib < длина
(B)
  if ia = длина(A)  // A закончился
    C[ic] = B[ib]
    ib += 1
else if ib = длина(B)  // B
закончился
    C[ic] = A[ia]
    ia += 1
  else
    if A[ia] <= B[ib]
      C[ic] = A[ia]
      ia += 1
    else
      C[ic] = B[ib]
      ib += 1
      ic += 1
  return C
```

Счётчики в соответствующих массивах

Пока хотя бы один из счётчиков для исходных массивов не прошёл массив до конца

Если счётчик уже пробежал все элементы в A, то берём следующий из B в качестве следующего элемента для C

Если счётчик уже пробежал все элементы в B, то берём следующий из A в качестве следующего элемента для C

Операция слияния

```
merge(A, B):
  C = [длина(A) + длина(B) нулей]
  ia = 0
  ib = 0
  ic = 0

while ia < длина(A) или ib < длина
(B)
  if ia = длина(A)  // A закончился
    C[ic] = B[ib]
    ib += 1
else if ib = длина(B)  // B
закончился
    C[ic] = A[ia]
    ia += 1
  else
    if A[ia] <= B[ib]
      C[ic] = A[ia]
      ia += 1
    else
      C[ic] = B[ib]
      ib += 1
      ic += 1
  return C
```

Счётчики в соответствующих массивах

Пока хотя бы один из счётчиков для исходных массивов не прошёл массив до конца

Если счётчик уже пробежал все элементы в A, то берём следующий из B в качестве следующего элемента для C

Если счётчик уже пробежал все элементы в B, то берём следующий из A в качестве следующего элемента для C

Иначе, берём минимальный из тех, на которые указывают счётчики в качестве следующего элемента для C

Операция слияния. Алгосложность

Из памяти мы потратили только на исходные массивы:

- на массив ответа
- на локальные переменные

Операция слияния. Алгосложность

Затраты по времени:

- итерация цикла константна и не зависит от длины исходных массивов

- количество итераций = сумме длин исходных массивов, т. к. на каждой итерации один из счётчиков ia или ib увеличивается на 1

Если общее количество элементов в массивах длина(A) + длина(B) = n, то время работы линейное — O(n)

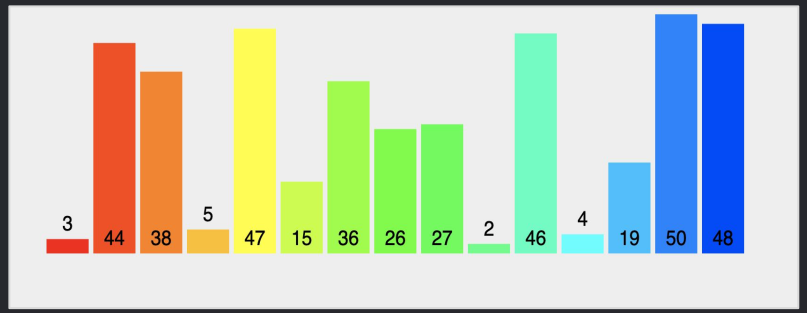
Сортировка слияния

Рекурсивная сортировка слиянием — сортировка, в которой мы:

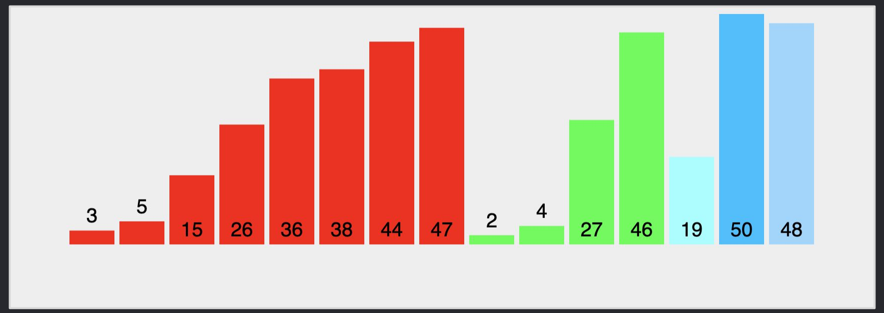
a) разбиваем массив пополам

b) рекурсивно запускаем сортировку половинок

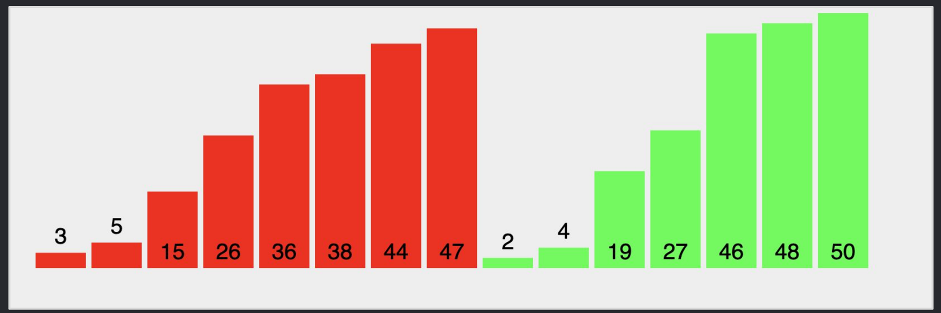
c) отдаём их слияние как ответ



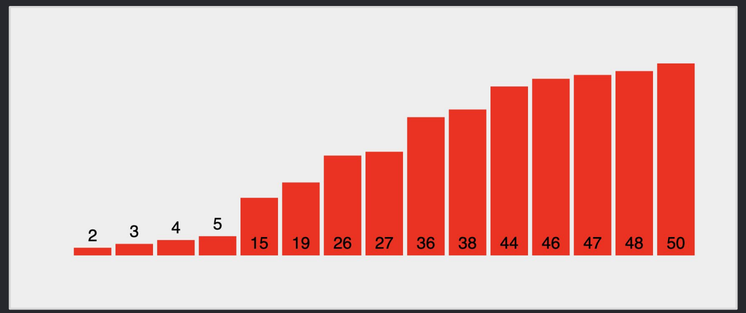
1



2



3



4

Сортировка слиянием

Динамический пример сортировки слиянием:

[Merge Sort — Sorting Algorithm Animations]

[Смотреть другую визуализацию >>]

На сайте вверху нажать на MER

Сортировка слиянием

```
merge_sort(arr):
  if длина(arr) = 1
    return arr

  left, right = разбить arr на
две половинки

left_sorted = merge_sort(left)
  right_sorted =
merge_sort(right)

  return merge(left_sorted,
right_sorted)
```

Если просят отсортировать массив из одного элемента,
то ничего не делаем, всё уже ОК

Сортировка слиянием

```
merge_sort(arr):
  if длина(arr) = 1
    return arr

  left, right = разбить arr на две половинки

  left_sorted = merge_sort(left)
  right_sorted = merge_sort(right)

  return merge(left_sorted, right_sorted)
```

Если просят отсортировать массив из одного элемента, то ничего не делаем, всё уже ОК

Иначе, разбиваем массив пополам на два подмассива

Сортировка слиянием

```
merge_sort(arr):
  if длина(arr) = 1
    return arr

  left, right = разбить arr на
две половинки

left_sorted = merge_sort(left)
  right_sorted =
merge_sort(right)

  return merge(left_sorted,
right_sorted)
```

Если просят отсортировать массив из одного элемента, то ничего не делаем, всё уже ОК

Иначе, разбиваем массив пополам на два подмассива

Рекурсивно запускаемся для обеих половинок, после чего они будут отсортированными

Сортировка слиянием

```
merge_sort(arr):
  if длина(arr) = 1
    return arr

  left, right = разбить arr на
две половинки

left_sorted = merge_sort(left)
  right_sorted =
merge_sort(right)

  return merge(left_sorted,
right_sorted)
```

Если просят отсортировать массив из одного элемента, то ничего не делаем, всё уже ОК

Иначе, разбиваем массив пополам на два подмассива

Рекурсивно запускаемся для обеих половинок, после чего они будут отсортированными

Запускаем операцию слияния на двух отсортированных половинках, получая в ответ отсортированное их объединение

Сортировка слиянием

Алгосложность одного вызова

Оценим алгосложность одного вызова функции merge_sort без учёта рекурсивных вызовов. Она не константная, а линейная, из-за операции merge в конце. Если w — размер массива (или его части) на котором был вызов, то время его работы (без учёта рекурсивных вызовов) O(w)

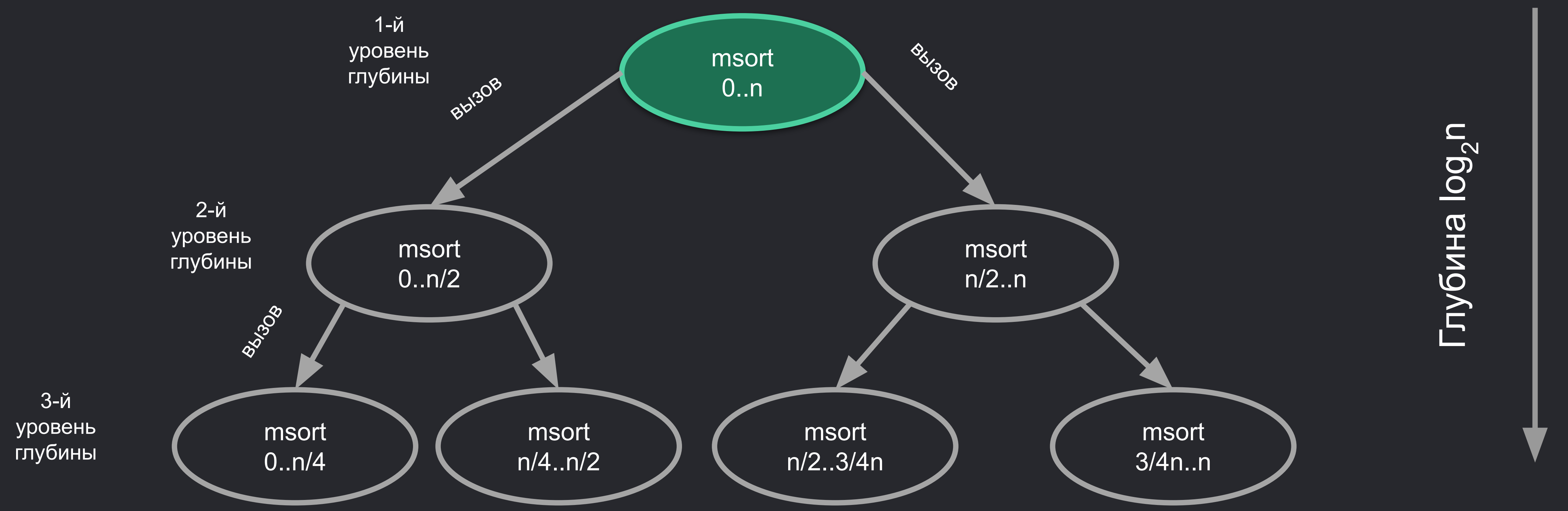


График рекурсивных вызовов merge_sort.

Сортировка слиянием

Алгосложность обработки одного уровня глубины рекурсии
Рассмотрим k-й уровень глубины рекурсии. На нём 2^k вызовов, размер каждого куска массива, на котором производится вызов на этом уровне ~ $n / 2^k$.
Итого: время работы всех вызовов (без учёта рекурсивных вызовов) на k-м уровне глубины это количество вызовов • время работы каждого вызова: $O(2^k \times n / 2^k) = O(n)$, т. е. линейна и не зависит от уровня глубины!

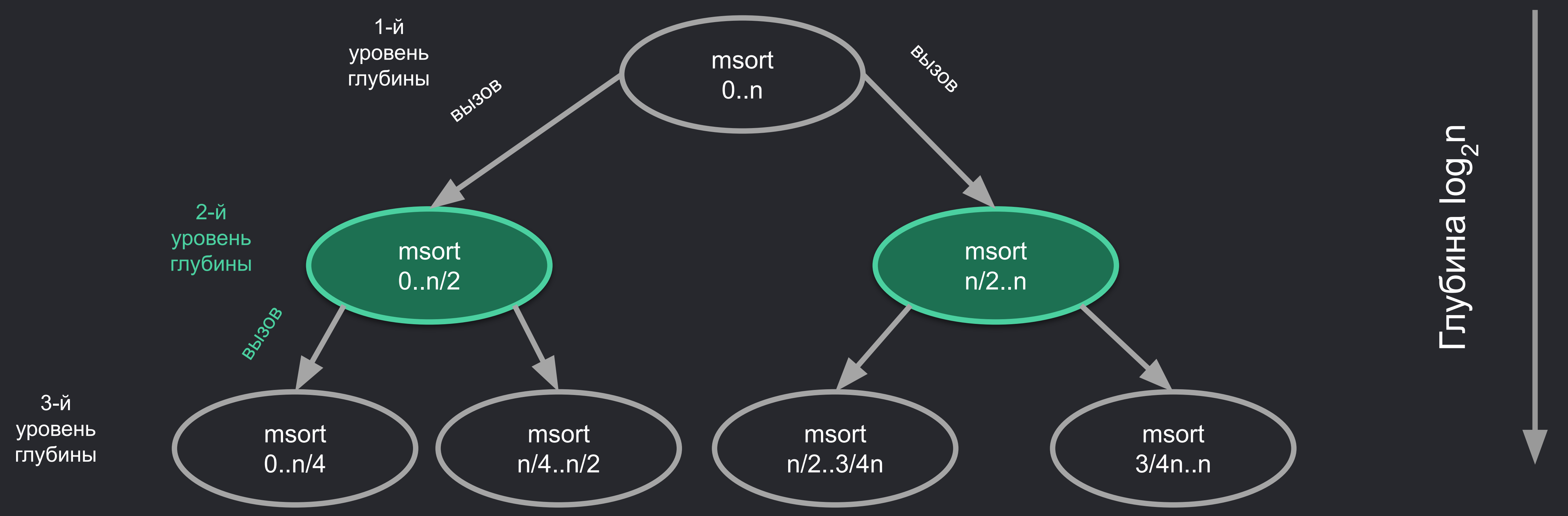


График рекурсивных вызовов merge_sort.

Сортировка слиянием

Алгосложность всего алгоритма

Просуммируем время работы всех уровней. Каждый уровень обрабатывается за $O(n)$, количество уровней — $\log_2 n$, поэтому общее время работы $O(n \log_2 n)$. Дополнительная память — $O(n)$, для операции merge. Сравним с пузырьком: на 1 млрд. элементов будет работать порядка минуты вместо 3,5 лет!

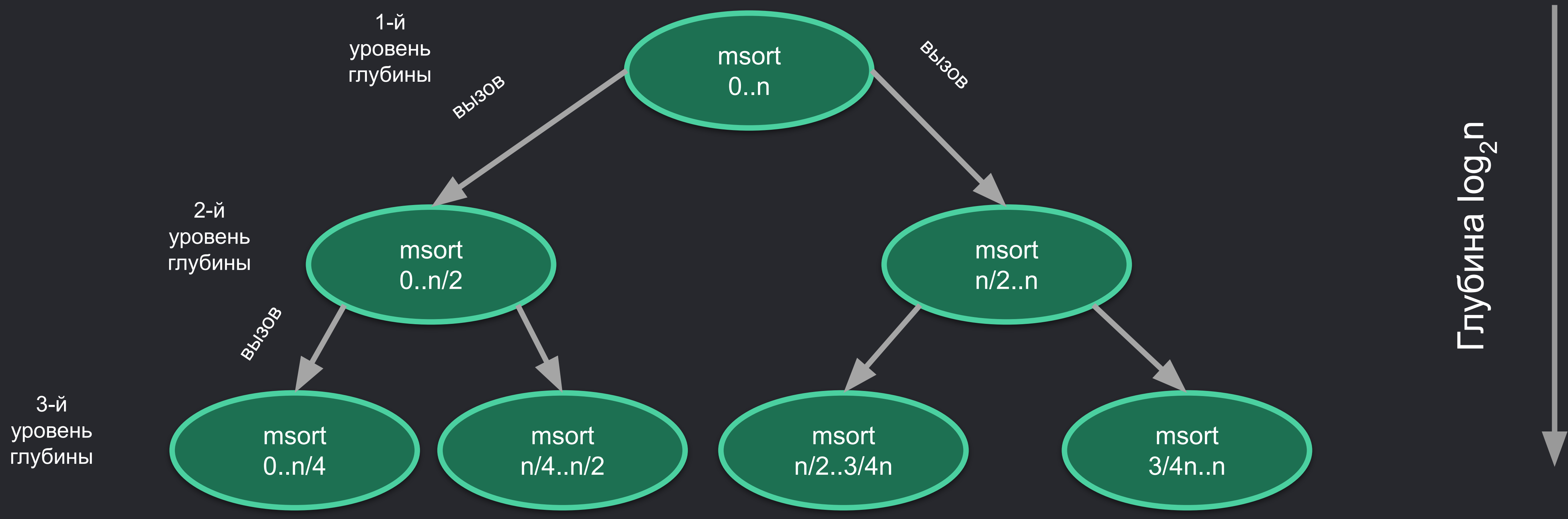


График рекурсивных вызовов merge_sort.

«Быстрая» сортировка

Пивотирование. Опорный элемент

Массив А:

27 92 54 24 76 45 14 81 72 31 81 13 17 70 48 20 58 35 57 97

Пивотирование. Опорный элемент

Массив A:

A[pi] — случайная опора (англ. pivot)

27 92 54 24 76 45 14 81 72 31 81 13 17 70 48 20 58 35 57 97

Пивотирование. Опорный элемент

Массив A:

A[pi] — случайная опора (англ. pivot)

27 92 54 24 76 45 14 81 72 31 81 13 17 70 48 20 58 35 57 97

Случайное пивотирование относительно 72:

<= 72 | >= 72

27 57 54 24 35 45 14 58 20 31 48 13 17 70 | 81 72 81 76 92 97

Пивотирование. Алгосложность

27 92 54 24 76 45 14 81 72 31 81 13 17 70 48 20 58 35 57 97

Заводим два счётчика:

- left идёт слева
- right идёт справа

Увеличиваем left, пока элементы меньше пивота, уменьшаем right, пока элементы больше пивота.

Если left и right ещё не встретились, поменяем местами эти элементы и увеличим left и уменьшим right, иначе вернём left как границ пивотирования. Слева от границы все элементы не больше от пивота, справа — не меньше пивота.

Время $O(n)$, дополнительная память $O(1)$

Пивотирование. Алгосложность

27	92	54	24	76	45	14	81	72	31	81	13	17	70	48	20	58	35	57	97

left right

27	92	54	24	76	45	14	81	72	31	81	13	17	70	48	20	58	35	57	97

27	57	54	24	76	45	14	81	72	31	81	13	17	70	48	20	58	35	92	97

left right

27	57	54	24	76	45	14	81	72	31	81	13	17	70	48	20	58	35	92	97

27	92	54	24	35	45	14	81	72	31	81	13	17	70	48	20	58	76	57	97

left right

Пивотирование. Алгосложность

27	57	54	24	35	45	14	81	72	31	81	13	17	70	48	20	58	76	92	97

27	57	54	24	35	45	14	58	72	31	81	13	17	70	48	20	81	76	92	97

left right

27	57	54	24	35	45	14	58	72	31	81	13	17	70	48	20	81	76	92	97

27	57	54	24	35	45	14	58	72	31	20	13	17	70	48	81	81	76	92	97

left right

27	57	54	24	35	45	14	58	72	31	20	13	17	70	48	81	81	76	92	97

Пивотирование. Алгосложность

```
pivoting(A, pi):
  left = 0
  right = длина(A)-1
  pivot = A[pi]
  while true
    while A[left] < pivot
      left += 1
    while A[right] > pivot
      right -= 1
    if left >= right
      return left
    swap A[left] A[right]
    left += 1
    right -= 1
```

Заводим два счётчика: left идёт слева, right идёт справа

Увеличиваем left пока элементы меньше пивота, уменьшаем right пока элементы больше пивота

Если left и right ещё не встретились, то поменяем местами эти элементы, увеличим left и уменьшим right, иначе вернём left как границу пивотирования

Пивотирование. Опорный элемент

Слева от границы все элементы не больше от пивота, справа — не меньше пивота. Время O(n), дополнительная память O(1)

<= 72 | >= 72

27 57 54 24 35 45 14 58 20 31 48 13 17 70 | 81 72 81 76 92 97

Быстрая сортировка. Алгосложность.

Быстрая сортировка. Решим задачу рекурсивно. Если нас просят отсортировать массив из 1 элемента, то ничего не делаем. Такой всегда можно считать уже отсортированным. Иначе делаем случайное пивотирование и запускаемся рекурсивно от половинок

27 57 54 24 35 45 14 58 20 31 | 48 13 17 70 81 72 81 76 92 97

14 57 54 24 35 45 27 58 20 31 | 48 13 17 70 81 72 81 76 92 97

Быстрая сортировка. Алгосложность.

Сортировка на месте

Мы можем передавать части массива в рекурсивные вызовы, не создавая новые массивы для частей. Передаём исходный массив и граничные индексы для каждой части

Так мы избежим необходимости выделять дополнительную память, ведь пивотирование её тоже не использует.

Итого, в отличие от сортировки слиянием, доппамять у нас $O(1)$, такие сортировки называются сортировками на месте

Быстрая сортировка. Алгосложность.

```
quick_sort(A):
  if длина(A) = 1
    return A
  pi = случайный индекс в A
  border = pivoting(A, pi)
  quick_sort(A до border)
  quick_sort(A после border)
  return A
```

Выбирая опору случайным образом, в среднем на каждом шаге массив будет делиться пополам

Выбирая опору случайным образом, в среднем на каждом шаге массив будет делиться пополам

Временная сложность алгоритма $O(n \log_2 n)$

Визуализация: https://visualgo.net/en/sorting?slide=1 (раздел QUI)

Линейные сортировки. Count sort

А можно быстрее?

Ограниченность сортировок на сравнениях

Скорость

Сортировка слиянием и быстрая сортировка требуют лишь умения сравнивать два произвольных элемента.

→ $O(n \log_2 n)$

Такие сортировки, не использующие дополнительные знания об элементах (например, о диапазоне допустимых числовых значений у элементов), называются сортировками на сравнениях

А можно быстрее?

Линейные сортировки

Когда нам известна дополнительная информация об элементах, иногда можно улучшить асимптотику и даже довести её до линейной → O(n)

Count sort или сортировка подсчётом

Допустим, мы знаем, что значения в массиве — числа, и каждое влезает в маленький диапазон от 0 до K

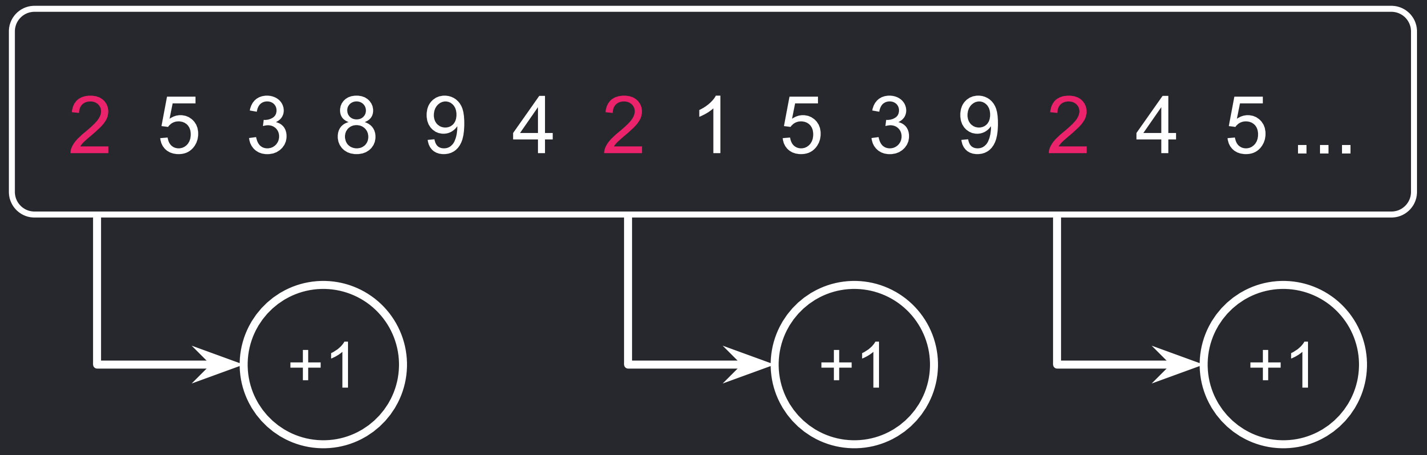
Массив A:

2 5 3 8 9 4 2 1 5 3 9 2 4 5 ...

Count sort или сортировка подсчётом

Заведём массив COUNTS длиною K и, проходясь по исходному массиву, будем считать, сколько какого значения нам встретилось

Массив A:

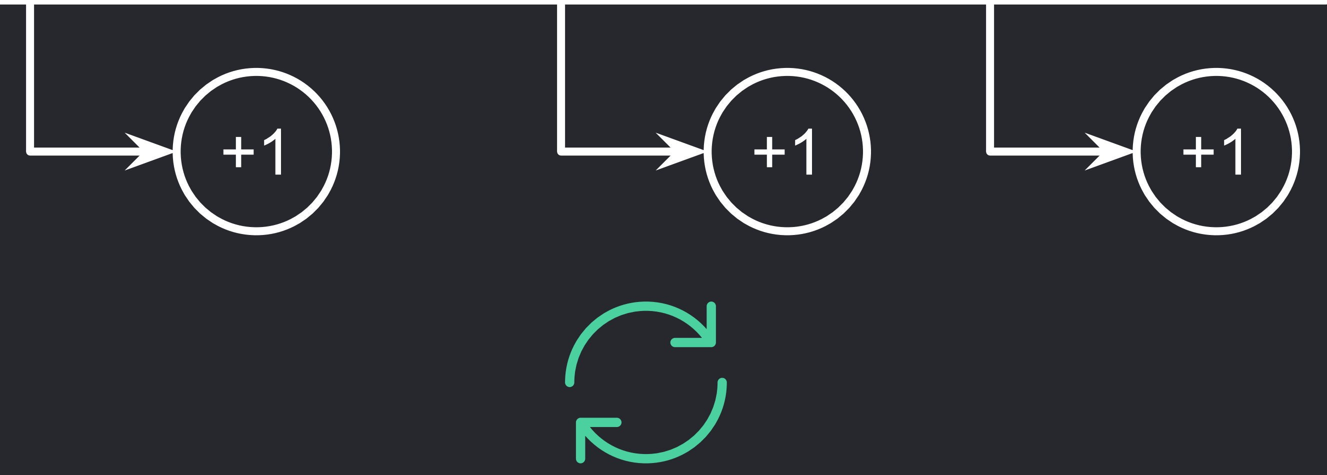


Count sort или сортировка подсчётом

Заведём массив COUNTS длиною K и, проходясь по исходному массиву, будем считать, сколько какого значения нам встретилось

Массив A:

2 5 3 8 9 4 2 1 5 3 9 2 4 5 ...



Массив A:

1 1 2 2 2 3 4 4 4 4 5 5 5 5 ...

Count sort. Алгосложность

Сортировка подсчётом:

```
count_sort(A, K):
  counts = [K нулей]
  for i от 0 до длина(A)
    counts[A[i]] += 1

  c = 0
  for i от 0 до длина(A)
    while counts[c] = 0
      c += 1
    A[i] = c
    counts[c] -= 1

  return A
```

Первый цикл пробегается по исходному массиву за O(n)

Количество итераций второго цикла не больше, чем n + K.

Т. к. K мы считаем маленьким числом, меньшим n, то можем считать, что второй цикл тоже за O(n)

Какую сортировку выбрать?

Дополнительные свойства сортировок

Устойчивость. Сортировка называется устойчивой, если равные, с точки зрения сравнений, элементы после сортировки остаются в том же порядке:

Сортировка слияния устойчивая, быстрая сортировка — нет

Адаптивность. Если алгоритм сортировки работает быстрее на массивах, близких к упорядоченным, то такой алгоритм называется адаптивным.

Сильно неотсортированный:
54, 1, 33, 4, 80, 72, 13 — 50 шагов

Почти отсортированный:
1, 13, 4, 54, 33, 72, 80 — 6 шагов

Дополнительные свойства сортировок

Адаптивность. Если алгоритм сортировки работает быстрее на массивах, близких к упорядоченным, то такой алгоритм называется адаптивным.

Сильно неотсортированный:
54, 1, 33, 4, 80, 72, 13 — 50 шагов

Почти отсортированный:
1, 13, 4, 54, 33, 72, 80 — 6 шагов

Какая сортировка лучше?

	Сортировка			
	Пузырьком	Слиянием	Быстрая	Подсчётом
Время	$O(n^2)$	O(n log n)	O(n log n)	O(n + k), где k — диапазон значений чисел
Дополнительная память	O(1)	O(n)	O(1)	O(k)
Устойчивость	+	+	—	Понятие не имеет смысла