

# Лекция 1

## Амортизационный анализ (amortized analysis)

**Курносов Михаил Георгиевич**

E-mail: [mkurnosov@gmail.com](mailto:mkurnosov@gmail.com)

WWW: [www.mkurnosov.net](http://www.mkurnosov.net)

Курс «Структуры и алгоритмы обработки данных»

Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)

Осенний семестр, 2015

# Анализ вычислительной сложности алгоритмов

1. Определяем *параметры* алгоритма, от которых зависит время его выполнения

$n, m$

2. Выражаем количество операций, выполняемых алгоритмом, как *функцию от его параметров* (для худшего, среднего или лучшего случаев)

$$T(n, m) = 2n^2 + 4\log_2 m$$

3. Строим *асимптотическую оценку* вычислительной сложности алгоритма – переходим к асимптотическим обозначениям:  
 $O, \Theta, \Omega$

$$\begin{aligned} T(n, m) &= O(2n^2 + 4\log_2 m) \\ &= O(\max\{n^2, \log_2 m\}) \end{aligned}$$

# Анализ сортировки выбором (худший случай)

```
function SelectionSort(v[1..n])
  for i = 1 to n - 1 do
    min = i
    for j = i + 1 to n do
      if v[j] < v[min] then
        min = j
      end if
    end for
    if min != i then
      temp = v[i]
      v[i] = v[min]
      v[min] = temp
    end if
  end for
end function
```

# Анализ сортировки выбором (худший случай)

```
function SelectionSort(v[1..n])
  for i = 1 to n - 1 do
    min = i
    for j = i + 1 to n do
      if v[j] < v[min] then
        min = j
      end if
    end for
    if min != i then
      temp = v[i]
      v[i] = v[min]
      v[min] = temp
    end if
  end for
end function
```

1 операция

# Анализ сортировки выбором (худший случай)

```
function SelectionSort(v[1..n])
```

```
  for i = 1 to n - 1 do
```

```
    min = i
```

1 операция

```
    for j = i + 1 to n do
```

```
      if v[j] < v[min] then
```

```
        min = j
```

```
      end if
```

2 операции

(худший случай – worst case)

```
    end for
```

```
    if min != i then
```

```
      temp = v[i]
```

```
      v[i] = v[min]
```

```
      v[min] = temp
```

```
    end if
```

```
  end for
```

```
end function
```

# Анализ сортировки выбором (худший случай)

```
function SelectionSort(v[1..n])
```

```
  for i = 1 to n - 1 do
```

```
    min = i
```

1 операция

```
    for j = i + 1 to n do
```

```
      if v[j] < v[min] then
```

```
        min = j
```

```
      end if
```

2 операции  
(худший случай – worst case)

```
    end for
```

```
    if min != i then
```

```
      temp = v[i]
```

```
      v[i] = v[min]
```

```
      v[min] = temp
```

```
    end if
```

4 операции

```
  end for
```

```
end function
```

# Анализ сортировки выбором (худший случай)

```
function SelectionSort(v[1..n])
```

```
  for i = 1 to n - 1 do
```

```
    min = i
```

1 операция

```
    for j = i + 1 to n do
```

```
      if v[j] < v[min] then
```

```
        min = j
```

```
      end if
```

2 операции

(худший случай – worst case)

```
    end for
```

```
    if min != i then
```

```
      temp = v[i]
```

```
      v[i] = v[min]
```

```
      v[min] = temp
```

```
    end if
```

4 операции

```
  end for
```

```
end function
```

$i = 1, j = 2 \text{ to } n, n - 1$  итер.

$i = 2, j = 3 \text{ to } n, n - 2$  итер.

$i = 3, j = 4 \text{ to } n, n - 3$  итер.

...

$i = n - 1, j = n \text{ to } n, 1$  итер.

# Анализ сортировки выбором (худший случай)

```
function SelectionSort(v[1..n])
```

```
  for i = 1 to n - 1 do
```

```
    min = i
```

1 операция

```
    for j = i + 1 to n do
```

```
      if v[j] < v[min] then
```

```
        min = j
```

```
      end if
```

2 операции  
(худший случай – worst case)

```
    end for
```

```
    if min != i then
```

```
      temp = v[i]
```

```
      v[i] = v[min]
```

```
      v[min] = temp
```

```
    end if
```

4 операции

```
  end for
```

```
end function
```

$$T(n) = (n - 1) + 4(n - 1) + 2((n - 1) + (n - 2) + \dots + 1) = ?$$



# Анализ сортировки выбором (худший случай)

```
function SelectionSort(v[1..n])
```

```
  for i = 1 to n - 1 do
```

```
    min = i
```

1 операция

```
    for j = i + 1 to n do
```

```
      if v[j] < v[min] then
```

```
        min = j
```

```
      end if
```

2 операции  
(худший случай – worst case)

```
    end for
```

```
    if min != i then
```

```
      temp = v[i]
```

```
      v[i] = v[min]
```

```
      v[min] = temp
```

```
    end if
```

4 операции

```
  end for
```

```
end function
```

Сумма членов  
арифметической прогрессии

$$T(n) = (n - 1) + 4 (n - 1) + 2((n - 1) + (n - 2) + \dots + 1) = ?$$

# Анализ сортировки выбором (худший случай)

```
function SelectionSort(v[1..n])
```

```
  for i = 1 to n - 1 do
```

```
    min = i
```

1 операция

```
    for j = i + 1 to n do
```

```
      if v[j] < v[min] then
```

```
        min = j
```

```
      end if
```

2 операции  
(худший случай – worst case)

```
    end for
```

```
    if min != i then
```

```
      temp = v[i]
```

```
      v[i] = v[min]
```

```
      v[min] = temp
```

```
    end if
```

4 операции

```
  end for
```

```
end function
```

$$T(n) = 5n - 5 + 2((n^2 - n) / 2) = n^2 + 4n - 5$$

# Анализ сортировки выбором (худший случай)

```
function SelectionSort(v[1..n])
```

```
  for i = 1 to n - 1 do
```

```
    min = i
```

1 операция

```
    for j = i + 1 to n do
```

```
      if v[j] < v[min] then
```

```
        min = j
```

```
      end if
```

2 операции

(худший случай – worst case)

```
    end for
```

```
    if min != i then
```

```
      temp = v[i]
```

```
      v[i] = v[min]
```

```
      v[min] = temp
```

```
    end if
```

4 операции

```
  end for
```

```
end function
```

$$T(n) = n^2 + 4n - 5 = O(n^2)$$

# Бинарный счетчик (binary counter)

- Счетчик имеет длину  $L$  бит (может принимать  $2^L$  значений)
- Поддерживает операцию Increment, которая увеличивает его значение на единицу
- Начальное значение счетчика – 0
- Пример 5 разрядного счетчика:

Разряд	4	3	2	1	0
Вес	16	8	4	2	1
Значение	0	0	0	0	0

# Бинарный счетчик (binary counter)

Increment: 0 → 1

Разряд	4	3	2	1	0
Вес	16	8	4	2	1
Значение	0	0	0	0	1

Increment: 1 → 2

Разряд	4	3	2	1	0
Вес	16	8	4	2	1
Значение	0	0	0	1	0

Increment: 2 → 3

Разряд	4	3	2	1	0
Вес	16	8	4	2	1
Значение	0	0	0	1	1

# Бинарный счетчик (binary counter)

Increment: 3 → 4

Разряд	4	3	2	1	0
Вес	16	8	4	2	1
Значение	0	0	1	0	0

Increment: 4 → 5

Разряд	4	3	2	1	0
Вес	16	8	4	2	1
Значение	0	0	1	0	1

Increment: 5 → 6

Разряд	4	3	2	1	0
Вес	16	8	4	2	1
Значение	0	0	1	1	0

# Бинарный счетчик (binary counter)

```
function Increment(A)
    i = 0
    while i < L and A[i] = 1 do
        A[i] = 0
        i = i + 1
    end while
    if i < L then
        A[i] = 1
    end if
end function
```

Заменяем начальную  
последовательность  
единиц нулями  
(перенос в старший  
разряд)

Разряд	3	2	1	0
Вес	8	4	2	1
Значение	1	0	1	1

= 11<sub>10</sub>

# Бинарный счетчик (binary counter)

```
function Increment(A)
```

```
    i = 0
```

```
    while i < L and A[i] = 1 do
```

```
        A[i] = 0
```

```
        i = i + 1
```

```
    end while
```

```
    if i < L then
```

```
        A[i] = 1
```

```
    end if
```

```
end function
```

- При каждом вызове функции Increment время ее работы разное
- Время зависит от внутреннего состояния – значений  $A[1..L]$

Разряд	3	2	1	0
Вес	8	4	2	1
Значение	1	0	1	1

= 11<sub>10</sub>



# Бинарный счетчик (binary counter)

```
function Increment(A)
```

```
    i = 0
```

```
    while i < L and A[i] = 1 do
```

```
        A[i] = 0
```

```
        i = i + 1
```

```
    end while
```

```
    if i < L then
```

```
        A[i] = 1
```

```
    end if
```

```
end function
```

- При каждом вызове функции Increment время ее работы разное
- Время зависит от внутреннего состояния – значений  $A[1..L]$

Вычислительная сложность функции Increment?

# Бинарный счетчик (binary counter)

- В худшем случае массив  $A[1..L]$  состоит только из единиц, для выполнения операции Increment требуется время  $O(L)$
- Это пессимистическая оценка!
- При каждом вызове функции Increment время ее работы разное

**Как анализировать вычислительную сложность алгоритмов  
время, выполнения которых зависит  
от их внутреннего состояния?**

# Амортизационный анализ

- **Амортизационный анализ (amortized analysis)** – метод анализа алгоритмов, позволяющий осуществлять оценку времени выполнения последовательности из  $n$  операций над некоторой структурой данных
- Время выполнения усредняется по всем  $n$  операциям, и оценивается *среднее время выполнения одной операции в худшем случае*

# Амортизационный анализ

- Некоторые операции структуры данных могут иметь высокую вычислительную сложность, другие низкую
- Например, некоторая операция может подготавливать структуру данных для быстрого выполнения других операций
- Такие «тяжелые» операции выполняются редко и могут оказывать незначительное влияние на суммарное время выполнения последовательности из  $n$  операций

# Амортизационный анализ

- Амортизационный анализ возник из группового анализа (aggregate analysis)
- Введен в практику Робертом Тарьяном (Robert Tarjan) в 1985 году:

Tarjan R. Amortized Computational Complexity // SIAM. J. on Algebraic and Discrete Methods, 6(2), 1985.  
— P. 306–318.

# Методы амортизационного анализа

- Групповой анализ (aggregate analysis)
- Метод бухгалтерского учета (accounting method)
- Метод потенциалов (potential method)

**Все методы позволяют получить одну и ту же  
оценку, но разными способами**

**[CLRS ed., С. 487]**

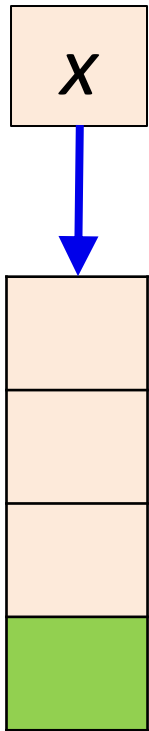
# Групповой анализ (aggregate analysis)

- **Групповой анализ (aggregate analysis)** – метод амортизационного анализа, позволяющий оценивать верхнюю границу времени  $T(n)$  выполнения последовательности из  $n$  операций в худшем случае
- **Амортизированная стоимость (amortized cost, учетная стоимость)** выполнения одной операции определяется как

$$T(n) / n$$

- Амортизированная стоимость операции – это оценка *сверху среднего времени выполнения операции в худшем случае*

# Стековые операции (Last In – First Out)



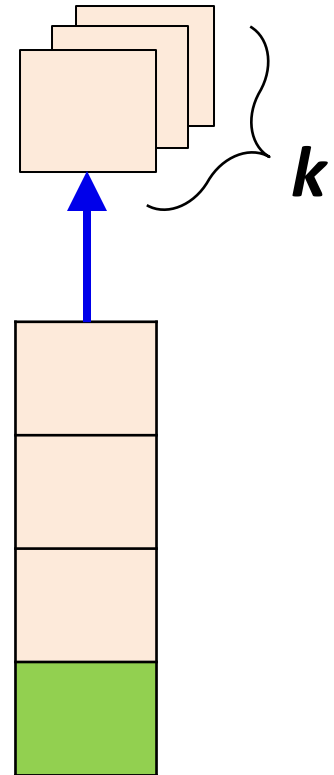
**Push( $S, x$ )**

$$T_{Push} = O(1)$$



**Pop( $S$ )**

$$T_{Pop} = O(1)$$



**MultiPop( $S, k$ )**

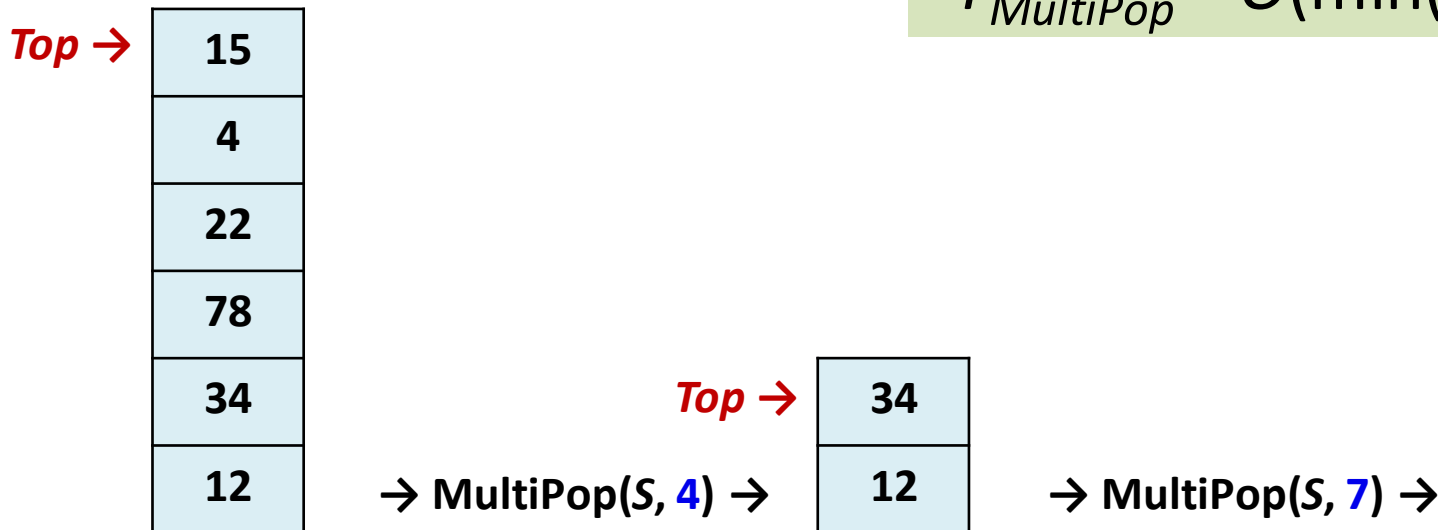
$$T_{MultiPop} = O(\min(|S|, k))$$



# Стековые операции (Last In – First Out)

```
function MultiPop(S, k)
  while StackEmpty(S) = False and k > 0 do
    StackPop(S)
    k = k - 1
  end while
end function
```

$$T_{MultiPop} = O(\min(|S|, k))$$



# Групповой анализ стековых операций v1.0

- Методом группового анализа оценим верхнюю границу времени  $T(n)$  выполнения произвольной последовательности из  $n$  стековых операций (Push, Pop, MultiPop)
  1. Стоимость операции Pop равна  $O(1)$
  2. Стоимость операции Push равна  $O(1)$
  3. Стоимость операции MultiPop в худшем случае  $O(n)$ , так как в ходе выполнения  $n$  операций в стеке не может находиться более  $n$  объектов
- В худшем случае последовательность из  $n$  операций может содержать только операции MultiPop
- Тогда, суммарное время  $T(n)$  выполнения  $n$  операций есть  $O(n^2)$ , а амортизированная стоимость одной операции

$$O(n^2) / n = O(n)$$

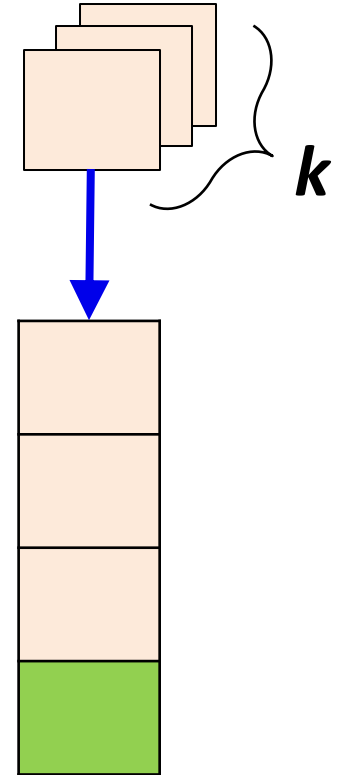
# Групповой анализ стековых операций v2.0

- Получим более точную оценку сверху времени  $T(n)$  выполнения произвольной последовательности из  $n$  стековых операций
  1. Количество операций Pop (включая вызовы из MultiPop) не превышает количества операций Push. В свою очередь, число операций Push не превышает  $n$  (операция MultiPop реализована на базе Pop)
  2. Таким образом для выполнения произвольной последовательности из  $n$  операций Push, Pop, MultiPop требуется время  $O(n)$
- Суммарное время выполнения  $n$  операций в худшем случае есть  $O(n)$ , тогда *амортизированная стоимость* (средняя стоимость) одной операции над стеком есть

$$O(n) / n = O(1)$$

# Групповой анализ стековых операций: **вопрос**

Останется ли справедливой оценка амортизированной стоимости стековых операций, равная  $O(1)$ , если включить в множество стековых операций операцию  $\text{MultiPush}(S, k)$ , помещающую в стек  $k$  элементов?



**$\text{MultiPush}(S, k)$**

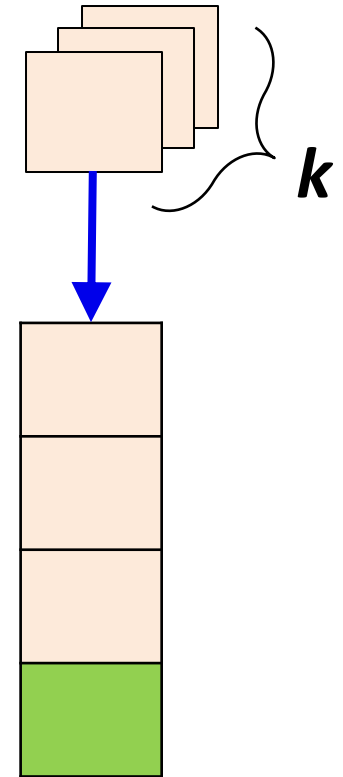
$$T_{\text{MultiPush}} = O(k)$$

# Групповой анализ стековых операций: **ответ**

**Ответ – нет**

Оценка амортизированной стоимости одной стековой операций станет  $O(k)$

- В последовательности из  $n$  стековых операций может быть  $n$  операций MultiPush, что требует времени  $O(nk)$
- Тогда амортизированная стоимость (средняя стоимость) одной стековой операции  $T(n) / n = O(nk) / n = O(k)$



**MultiPush( $S, k$ )**

# Бинарный счетчик (binary counter)

- Счетчик имеет длину  $L$  бит (может принимать  $2^L$  значений)
- Поддерживает операцию Increment
- Пример 5 битный счетчик
  - Значение счетчика 1, битовая последовательность: **00001**
  - Значение счетчика 3, битовая последовательность: **00011**
  - Значение счетчика 4, битовая последовательность: **00100**
  - Значение счетчика 5, битовая последовательность: **00101**
  - Значение счетчика 10, битовая последовательность: **01010**

Разряд	4	3	2	1	0
Вес	16	8	4	2	1
Значение	<b>0</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>0</b>

# Бинарный счетчик (binary counter)

```
function Increment(A)
```

```
    i = 0
```

```
    while i < L and A[i] = 1 do
```

```
        A[i] = 0
```

```
        i = i + 1
```

```
    end while
```

```
    if i < L then
```

```
        A[i] = 1
```

```
    end if
```

```
end function
```

- При каждом вызове функции Increment время ее работы разное
- Время работы зависит от внутреннего состояния – значений A

Разряд	3	2	1	0
Вес	8	4	2	1
Значение	1	0	1	1

# Бинарный счетчик (binary counter)

Знач.	A[7]	...	A[4]	A[3]	A[2]	A[1]	A[0]	Стоимость (# операций)	Суммарная стоимость
1	0	...	0	0	0	0	1	1	1
2	0		0	0	0	1	0	2	3
3	0		0	0	0	1	1	1	4
4	0		0	0	1	0	0	3	7
5	0		0	0	1	0	1	1	8
6	0		0	0	1	1	0	2	10
7	0		0	0	1	1	1	1	11
8	0		0	1	0	0	0	4	15
9	0		0	1	0	0	1	1	16
10	0		0	1	0	1	0	2	18
11	0		0	1	0	1	1	1	19
12	0		0	1	1	0	0	3	22
13	0		0	1	1	0	1	1	23
14	0		0	1	1	1	0	2	25
15	0		0	1	1	1	1	1	26
16	0		1	0	0	0	0	5	31



# Бинарный счетчик (binary counter)

Знач.	A[7]	...	A[4]	A[3]	A[2]	A[1]	A[0]	Стоимость (# операций)	Суммарная стоимость
1	0		0	0	0	0	1	1	1
2	0		0	0	0	1	0	2	3
3	0		0	0	0	1	1	1	4
4	0		0	0	1	0	0	2	7

**Какова амортизированная стоимость выполнения функции Increment (среднее время в худшем случае)?**

11	0		0	1	0	1	1	1	19
12	0		0	1	1	0	0	3	22
13	0		0	1	1	0	1	1	23
14	0		0	1	1	1	0	2	25
15	0		0	1	1	1	1	1	26
16	0		1	0	0	0	0	5	31

# Бинарный счетчик (binary counter)

Знач.	A[7]	...	A[4]	A[3]	A[2]	A[1]	A[0]	Стоимость (# операций)	Суммарная стоимость
1	0		0	0	0	0	1	1	1

- Оценим сверху время  $T(n)$  выполнения  $n$  операций Increment
- Тогда амортизированная стоимость операции Increment (среднее время выполнения) будет

$$T(n) / n$$

10	0		0	1	0	1	0	2	18
11	0		0	1	0	1	1	1	19
12	0		0	1	1	0	0	3	22
13	0		0	1	1	0	1	1	23
14	0		0	1	1	1	0	2	25
15	0		0	1	1	1	1	1	26
16	0		1	0	0	0	0	5	31

# Бинарный счетчик (binary counter)

Знач.	A[7]	...	A[4]	A[3]	A[2]	A[1]	A[0]	Стоимость	Суммарная стоимость
1	0	...	0	0	0	0	1	1	1
2	0		0	0	0	1	0	1	3
3	0		0	0	1	0	0	1	4
4	0		0	1	0	0	0	1	7
5	0		0	1	0	1	0	1	8
$n$	0		0	1	1	0	0	1	$T(n)$
7	0		0	0	1	1	1	1	11
8	0	...	0	1	0	0	0	4	15
9	0		0	1	0	0	1	1	16
10	0		0	1	0	1	0	2	18
11	0		0	1	0	1	1	1	19
12	0		0	1	1	0	0	3	22
13	0		0	1	1	0	1	1	23
14	0		0	1	1	1	0	2	25
15	0		0	1	1	1	1	1	26
16	0		1	0	0	0	0	5	31

Можно заметить,  
что время выполнения  $n$  операций  
 $2n \geq T(n)$

# Бинарный счетчик (binary counter)

Знач.	A[7]	...	A[4]	A[3]	A[2]	A[1]	A[0]	Стоимость	Суммарная стоимость
1	0		0	0	0	0	1	1	1
2	0		0	0	0	1	0	2	3
3	0		0	0	0	1	1	1	4
4								3	7
5								1	8
6								2	10
7								1	11
8								4	15
9								1	16
10								2	18
11								1	19
12								3	22
13								1	23
14								2	25
15								1	26
16								5	31

▪ Бит 0 изменяется  $n$  раз  
(при каждом вызове Increment)

▪ Бит 1:  $\lfloor n/2 \rfloor$  раз

▪ Бит 2:  $\lfloor n/4 \rfloor$  раз

▪ ...

▪ Бит  $L-1$  изменяется  $\lfloor n/2^{L-1} \rfloor$  раз

$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \dots + \frac{n}{2^{L-1}} =$$

$$= n + n = 2n$$

# Бинарный счетчик (binary counter)

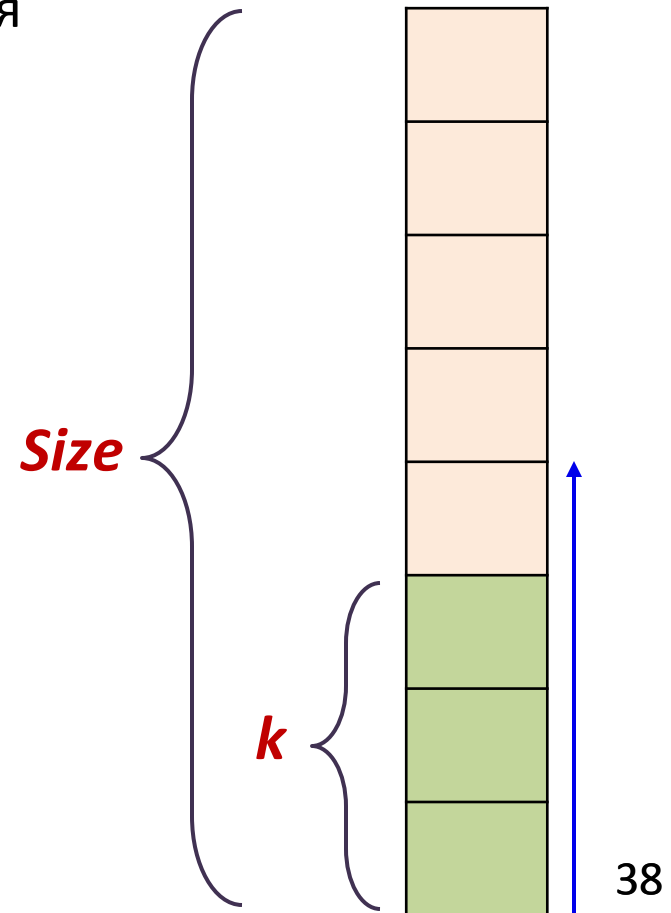
Знач.	A[7]	...	A[4]	A[3]	A[2]	A[1]	A[0]	Стоимость	Суммарная стоимость
1	0		0	0	0	0	1	1	1
2	0		0	0	0	1	0	2	3
3	0		0	0	0	1	1	1	4

Средняя (амортизированная) стоимость  
одной операции Increment  
 $O(n) / n = O(1)$

11	0		0	1	0	1	1	1	19
12	0		0	1	1	0	0	3	22
13	0		0	1	1	0	1	1	23
14	0		0	1	1	1	0	2	25
15	0		0	1	1	1	1	1	26
16	0		1	0	0	0	0	5	31

# Динамические таблицы (dynamic tables)

- Динамическая таблица (Dynamic table, dynamic array, growable array) – это массив поддерживающий вставку и удаление элементов и динамически изменяющий свой размер до необходимого значения
- Поддерживаемые операции
  - $Insert(T, x)$
  - $Delete(T, x)$
  - **Size** – количество свободных ячеек
  - $k$  – количество элементов добавленных в массив



# Динамические таблицы (dynamic tables)

- При выполнении операции Insert размер массива увеличивается
- Как увеличивать размер массива?
- **Аддитивная схема** – текущий размер массива увеличивается на  $k$  ячеек (по арифметической прогрессии)
- **Мультипликативная схема** – текущий размер массива увеличивается в  $k$  раз (по геометрической прогрессии)
- **Примеры реализации:**
  - **C++:** `std::vector`
  - **Java:** `ArrayList`
  - **.NET 2.0:** `List<>`
  - **Python:** `list`

# Динамические таблицы (dynamic tables)

```
function Insert(x)
  if Size = 0 then
    Size = 1
    Table = AllocateMemory(1)
    k = 0
  else if k = Size then
    Size = Size * 2
    NewTable = AllocateMemory(Size)
    for i = 0 to k - 1 do
      NewTable[i] = Table[i]
    end for
    FreeMemory(Table)
    Table = NewTable
  end if
  Table[k] = x
  k = k + 1
end function
```



# Динамические таблицы (dynamic tables)

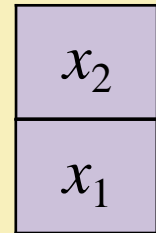
```
function Insert(x)
  if Size = 0 then
    Size = 1
    Table = AllocateMemory(1)
    k = 0
  else if k = Size then
    Size = Size * 2
    NewTable = AllocateMemory(Size)
    for i = 0 to k - 1 do
      NewTable[i] = Table[i]
    end for
    FreeMemory(Table)
    Table = NewTable
  end if
  Table[k] = x
  k = k + 1
end function
```



$x_1$

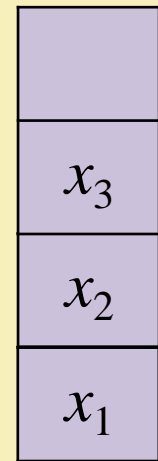
# Динамические таблицы (dynamic tables)

```
function Insert(x)
  if Size = 0 then
    Size = 1
    Table = AllocateMemory(1)
    k = 0
  else if k = Size then
    Size = Size * 2
    NewTable = AllocateMemory(Size)
    for i = 0 to k - 1 do
      NewTable[i] = Table[i]
    end for
    FreeMemory(Table)
    Table = NewTable
  end if
  Table[k] = x
  k = k + 1
end function
```



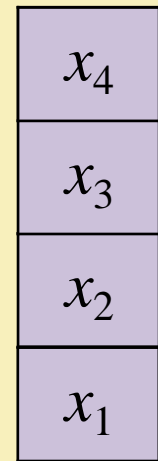
# Динамические таблицы (dynamic tables)

```
function Insert(x)
  if Size = 0 then
    Size = 1
    Table = AllocateMemory(1)
    k = 0
  else if k = Size then
    Size = Size * 2
    NewTable = AllocateMemory(Size)
    for i = 0 to k - 1 do
      NewTable[i] = Table[i]
    end for
    FreeMemory(Table)
    Table = NewTable
  end if
  Table[k] = x
  k = k + 1
end function
```



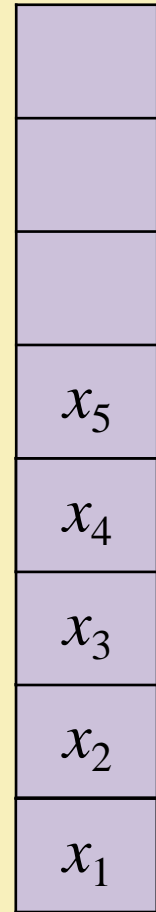
# Динамические таблицы (dynamic tables)

```
function Insert(x)
  if Size = 0 then
    Size = 1
    Table = AllocateMemory(1)
    k = 0
  else if k = Size then
    Size = Size * 2
    NewTable = AllocateMemory(Size)
    for i = 0 to k - 1 do
      NewTable[i] = Table[i]
    end for
    FreeMemory(Table)
    Table = NewTable
  end if
  Table[k] = x
  k = k + 1
end function
```



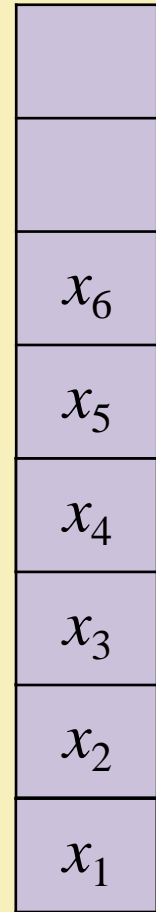
# Динамические таблицы (dynamic tables)

```
function Insert(x)
  if Size = 0 then
    Size = 1
    Table = AllocateMemory(1)
    k = 0
  else if k = Size then
    Size = Size * 2
    NewTable = AllocateMemory(Size)
    for i = 0 to k - 1 do
      NewTable[i] = Table[i]
    end for
    FreeMemory(Table)
    Table = NewTable
  end if
  Table[k] = x
  k = k + 1
end function
```



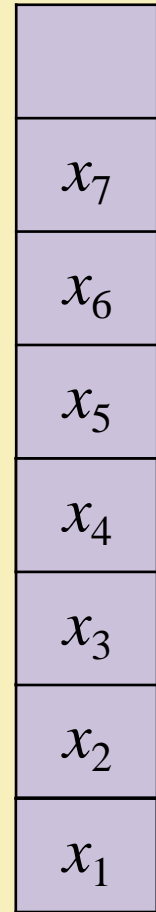
# Динамические таблицы (dynamic tables)

```
function Insert(x)
  if Size = 0 then
    Size = 1
    Table = AllocateMemory(1)
    k = 0
  else if k = Size then
    Size = Size * 2
    NewTable = AllocateMemory(Size)
    for i = 0 to k - 1 do
      NewTable[i] = Table[i]
    end for
    FreeMemory(Table)
    Table = NewTable
  end if
  Table[k] = x
  k = k + 1
end function
```



# Динамические таблицы (dynamic tables)

```
function Insert(x)
  if Size = 0 then
    Size = 1
    Table = AllocateMemory(1)
    k = 0
  else if k = Size then
    Size = Size * 2
    NewTable = AllocateMemory(Size)
    for i = 0 to k - 1 do
      NewTable[i] = Table[i]
    end for
    FreeMemory(Table)
    Table = NewTable
  end if
  Table[k] = x
  k = k + 1
end function
```

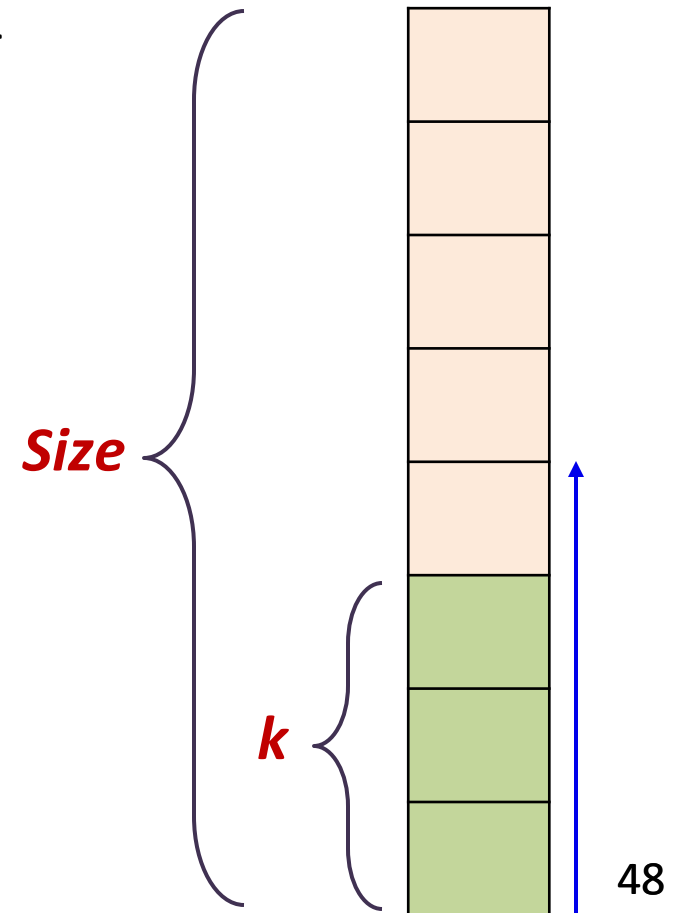


# Динамические таблицы (dynamic tables)

Проведем амортизационный анализ времени  $T(n)$  выполнения последовательности из  $n$  операций Insert

- Время работы операции Insert зависит от состояния таблицы: количества  $k$  элементов и её размера  $Size$
- Будем учитывать только операции записи элементов в таблицу

**Table[k] = x**





# Амортизационный анализ операции Insert

```
function Insert(x)
  if Size = 0 then
    Size = 1
    Table = AllocateMemory(1)
    k = 0
  else if k = Size then
    Size = Size * 2
    NewTable = AllocateMemory(Size)
    for i = 0 to k - 1 do
      NewTable[i] = Table[i]
    end for
    FreeMemory(Table)
    Table = NewTable
  end if
  Table[k] = x
  k = k + 1
end function
```

- Первый вызов Insert – **1 операция**
- Второй вызов – **2 оп.** (Copy 1 + Write 1)
- Третий – **3 оп.** (Copy 2 + Write 1)
- Четвертый – **1 оп.**
- Пятый – **5 оп.** (Copy 4 + Write 1)
- ...

# Амортизационный анализ операции Insert

- Обозначим через  $c_i$  количество операций, выполняемых на  $i$ -ом вызове Insert

$$c_i = \begin{cases} i, & \text{если } i - 1 \text{ степень } 2, \\ 1, & \text{иначе.} \end{cases}$$

- Тогда оценка сверху времени  $T(n)$  выполнения  $n$  операций Insert есть

$$T(n) = c_1 + c_2 + \dots + c_n \leq n + 2^0 + 2^1 + 2^2 + \dots + 2^{\lfloor \log_2 n \rfloor}$$

$$T(n) < n + 2n = 3n$$

# Амортизационный анализ операции Insert

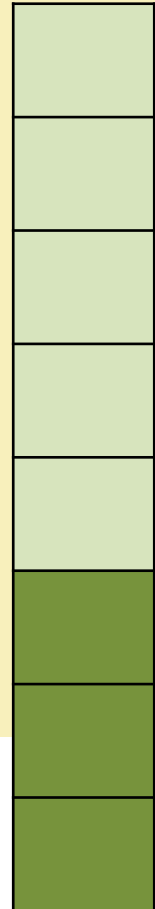
- Оценка сверху амортизированной сложности одной операции Insert есть

$$T_{Insert} = \frac{T(n)}{n} = \frac{3n}{n} = 3 = \mathbf{O(1)}$$

**Среднее время выполнения (вычислительная сложность)  
одной операции Insert в худшем случае есть  $O(1)$**

# Обработчик с накопителем

```
function AddToBuffer(value)
    Count = Count + 1
    Buffer[Count] = value
    if Count = H then
        for i = 1 to H do
            Packet[i] = Buffer[i]
        end for
        Count = 0
    end if
end function
```



# Обработчик с накопителем

- Построить оценку сверху времени  $T(n)$  выполнения  $n$  операций `AddToBuffer`
- Оценить амортизированную сложность функции `AddToBuffer`

# Амортизационный анализ операции AddToBuffer

- Обозначим через  $c_i$  количество операций, выполняемых на  $i$ -ом вызове AddToBuffer

$$c_i = \begin{cases} H + 3, & \text{если } i \% H = 0, \\ 2, & \text{иначе.} \end{cases}$$

- Тогда оценка сверху времени  $T(n)$  выполнения  $n$  операций AddToBuffer есть

$$\begin{aligned} T(n) &= c_1 + c_2 + \dots + c_n < 2n + \frac{n(3 + H)}{H} < 2n + 3n + H \\ &= 5n + H \end{aligned}$$

$$T(n) = O(5n + H) = O(n + H)$$

# Амортизационный анализ операции AddToBuffer

- Оценка сверху времени  $T(n)$  выполнения  $n$  операций AddToBuffer

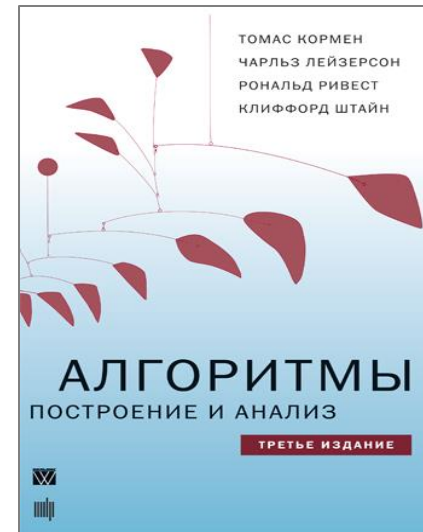
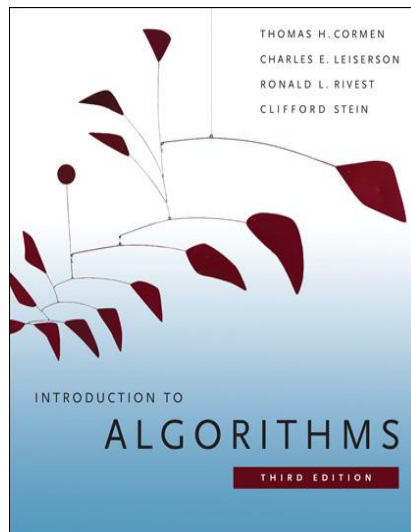
$$T(n) = O(5n + H) = O(n + H)$$

- Амортизированная стоимость (сложность) одной операции AddToBuffer

$$\frac{T(n)}{n} = \frac{5n + H}{n} = O(1)$$

# Задание

- Прочитать в [CLRS 3ed., С. 487]:
  - 17.2 Метод бухгалтерского учета
  - 17.3 Метод потенциалов



Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К.  
**Алгоритмы: построение и анализ.**  
– 3-е изд. – М.: Вильямс, 2013. – 1328 с.