1. (a) Истинное время работы increment в худшем случае составит k операций
   Это значение достигается когда массив a заполнен единицами.

   (b) Посмотрим сколько раз мы инвертировали бит с номером i.
   Нулевой бит инвертируется каждую операцию, первый бит - каждую вторую. Нетрудно заметить что i бит меняет значение только после 2^i операций increment. Итого получаем $n + \frac{n}{2} + \frac{n}{4} + ... \leq 2n$

   (c) Рассмотрим массив в котором на $k - 1$ месте стоит единица, а остальные значения - нули.
   Будем чередуя выполнять операции decrement и increment.
   Нетрудно заметить что каждая будет выполняться за k простых операций.
   Следовательно время работы в худшем случае — $\mathcal{O}(nk)$.

   (d) Заведём дополнительную переменную right, изначально равную -1, — самая правая единичка. После неё в массиве может храниться что угодно, но мы будем считать что там лежат нули.

      i. Когда цикл в increment дойдет до right $+1$ мы должны остановиться так как помним о том что все элементы после right равны нулю. Ещё нужно подвинуть right в случае когда $i >$ right, потому что самая правая единичка может подвинуться правее.

      ii. get(i) нужно изменить так что-бы при $i >$ right она возвращала 0.

      iii. Теперь setZero будет просто ствить right в значение -1

```
increment():
  i = 0
  while i < k and i <= right and a[i] = 1:
    a[i] = 0
    i++
  right = max(right, i)
  if i < k:
    a[i] = 1

get(i):
  if i > r:
    return 0;
  return a[i]

setZero():
  r = -1

```

2. Придумаем функцию потенциала:
   n — количество элементов в векторе
   c — фактический размер вектора
   $\frac{c}{4} \le n \le c$
   $$\Phi(n,c) = \begin{cases} c \le 2n, 2n - c, \\ 2n < c, 2c - n. \end{cases}$$
   $$\text{push} = \begin{cases} c \le 2n,\ 1 + \Phi(n+1,c) - \Phi(n,c) = \mathcal{O}(1), \\ 2n < c,\ 1 + \Phi(n+1,c) - \Phi(n,c) = \mathcal{O}(1), \\ n = c,\ n + \Phi(n+1,2c) - \Phi(n,c) = \\ = n + 2n + 2 - 2c - 2n + c = 3 + n - c = \mathcal{O}(1). \end{cases}$$
   $$\text{pop} = \begin{cases} c \le 2n,\ 1 + \Phi(n+1,c) - \Phi(n,c) = \mathcal{O}(1), \\ 2n < c,\ 1 + \Phi(n+1,c) - \Phi(n,c) = \mathcal{O}(1), \\ n = \frac{c}{4},\ c + \Phi(n-1,\frac{c}{2}) - \Phi(n,c) = \\ = c + c - n + 1 - 2c + n = 1 = \mathcal{O}(1). \end{cases} \Rightarrow$$
   $\Rightarrow$ все операции выполняются в среднем за $\mathcal{O}(1)$

3. Пусть у нас есть неинициализированный массив a. Заведём дополнительно массив b на n элементов и стек на массиве s. s будет соответствовать элементам, которым мы уже что-нибудь присвоили. В ячейке стека будем хранить позицию в массиве, куда мы присваивали.
   В a будем хранить последнее значение, либо мусор. Во b храним позицию в стеке нашего элемента, либо опять мусор.
   Как это работает?
   При запросе значения в ячейке i, будем смотреть на значение в b_i. Если $b_i \ge \text{size}(s)$, то очевидно в эту ячейку мы ничего не присваивали и нужно вернуть из $\text{get}(i)$ нуль.
   Посмотрим на случай, когда $b_i < \text{size}(s)$.
   Мы точно знаем что внутри стека нет мусора и каждый элемент соответствует инициализированному значению.
   Это значит что $s_{b_i} = i \iff i$ — была проинициализирована ранее.
   Осталось поддерживать эту структуру при добавлении элемента, это просто.
   В $\text{set}(i,x)$ есть два случая.
   a_i — проинициализирована, тогда нужно просто записать в a новое значение x.
   a_i — не проинициализирована, тогда нужно записать в b_i значение $\text{size}(s)$. Затем добавить в s вершину со значением i и присвоить элементу a_i значение x.

Вот код для понимания.

```
a, b, s = [], [], []
sSize = 0
init(n):
  a, b, s = [n], [n], [n]

isInit(i):
  if b[i] < sSize and s[b[i]] == i:
    return True
  else:
    return False

get(i):
  if isInit(i):
    return a[i]
  else:
    return 0

set(i, x):
  if isInit(i):
    a[i] = x
  else
    a[i] = x
    s[sSize] = i
    b[i] = sSize
    sSize++;

```

4. Будем внутри нашей памяти хранить односвязный список не занятых блоков. Заведём указатель head на последний элемент списка. В вершине списка будем хранить одно число — указатель на предыдущий элемент, или -1, если элемент является последним. Пусть для нас память представляет собой массив a длины n. Проинициализируем её так:
$\forall i \in \{0, 1, \ldots, n-1\},\ a_i = i - 1$
$\text{head} = n - 1$
Теперь очевидно. Когда нужно вернуть номер свободной ячейки, мы возвращаем номер head и удаляем последний элемент списка. Когда нам говорят сделать free некоторой ячейки, мы добавляем эту вершинку в наш список неиспользованных.

```
head = -1
mem = []
init(n):
  mem = [n]
  for i in range(0, n):
    mem[i] = i - 1
  head = n - 1

malloc():
  tmp = head
  head = mem[head]
  return tmp

free(p):
  mem[p] = head
  head = p
```

5. Найдем и докажем время работы push
   Пусть $\{a_1,\ a_2,\ \ldots, a_k\} = A$
   Введём функцию потенциала:
   $\Phi(A) = \sum_{i=0}^{k}(k-i)2^i$ Тогда рассмотрим что происходит при добавлении одного элемента в структуру.
   Пусть в структуре первые k' массивов не пусты.
   Тогда элемент, который мы вставляем в структуру, k' раз поучаствует в merge $\Rightarrow$ на это мы потратим k' операций.
   push работает за:
   $(1) k' + \Phi(A') - \Phi(A) = k' + (k-k'-1)2^{k'+1} - \sum_{i=0}^{k'}(k-i)2^i$
   Рассмотрим сумму поближе:
   $(2) \sum_{i=0}^{k'}(k-i)2^i = k2^{k'+1} - \sum_{i=0}^{k'} i2^i$
   Ещё ближе, теперь уже на сумму $\sum_{i=0}^{k} i2^i$.
   Докажем по индукции что она равна $2(k2^k - 2^k + 1)$
   $2(k2^k - 2^k + 1) + (k+1)2^{k+1} = k2^{k+1} - 2^{k+1} + 2 + 2^{k+1} + k2^{k+1} =$
   $= k2^{k+2} + 2^{k+2} - 2^{k+1} + 2 = 2((k+1)2^{k+1} - 2^{k+1} + 1)$
   Доказали, круто, вернёмся слегка назад к (2).
   $k2^{k'+1} - \sum_{i=0}^{k'} i2^i = k2^{k'+1} - k'2^{k'+1} - 2^{k'+1} + 2 =$
   $= 2^{k'+1}(k-k'-1) + 2$
   Окэй, посчитали (2), тогда можем почитать и (1).
   $k' + 2^{k'+1}(k-k'-1) - 2^{k'+1}(k-k'-1) + 2 = \mathcal{O}(k)$
   Значит push работает в среднем за $\mathcal{O}(k)$.
   С contains всё проще, потому что структуру данных эта функция не изменяет (разность потенциалов равна нулю). Значит просто считаем худший случай.
   На i-й итерации запускается бинарный поиск на массиве размера 2^i.
   contains работает за:
   $\log_2 2^0 + \log_2 2^1 + \log_2 2^2 + \cdots + \log_2 2^k = \frac{k(k+1)}{2} = \mathcal{O}(k^2)$