8.Частотный анализ и сортировка подсчётом. Алгоритмическая сложность и применимость.

Частотный анализ - определение частоты (количества) встречающихся в строке/массиве/списке/файле/тексте элементов.

```
a = [5, 4, 5, 6, 6, 6, 100, 0, 2]

def count_sort(a):
    count=[0]*(max(a)+1)
    for i in range (len(a)):
        count[a[i]]+=1
    a[:]=[]
    for i in range(len(count)):
        a+=[i]*count[i]
    return a

print(count_sort(a))
```

```
[0, 2, 4, 5, 5, 6, 6, 6, 100]
```

```
%%time
(count_sort(a))
```

```
CPU times: user 76 µs, sys: 1e+03 ns, total: 77 µs
Wall time: 82 µs

[0, 2, 4, 5, 5, 6, 6, 6, 100]
```

Алгоритмическая сложность - O(n)

Применимость - в том случае, когда точно известно, максимальное число в сортируемой структуре данных и это число укладывается в массив достаточно маленького размера.

9.Поразрядная сортировка. Алгоритмическая сложность и применимость.

Переводим числа в двоичную систему счисления. Сортируем по самому старшему биту, то есть сначала будут идти числа с 0 в старшем бите, а потом с единицей. Внутри этих групп проводим аналогичные сортировки, начиная со следующего бита. Продолжаем, пока не дойдем до самого младшего бита.

```
m = list(map(int, input().split()))

for i in range(len(bin(max(m))) - 2):
    m = list(filter(lambda x: not (x & (1 << i)), m)) + list(filter(lambda x: x & (1 << i), m))

print(m)
```

Сложность $O(N * K)$, где K - битовая длина самого большого элемента

Применимость: работает только для целых чисел.

11.Рекурсия. Крайний и рекуррентный случай, ход рекурсии. Генерация комбинаторных объектов.

Рекурсия - способ, заключающийся в сведении задачи к подзадаче, которая аналогична самой задаче, но проще её.

Случай, когда рекурсия останавливается, называется базовым или крайним.

Случай, когда крайний случай (условие) не выполняется (то есть функция снова вызывает сама себя), называется рекуррентным.

Прямой и обратный ход рекурсии

Действия, выполняемые функцией до входа на следующий уровень рекурсии, называются выполняющимися на прямом ходу рекурсии, а действия, выполняемые по возврату с более глубокого уровня к текущему, – выполняющимися на обратном ходу рекурсии.

Генерация всех чисел в k-ой системе счисления длины n

```
def gen_num(n, k, prefix):
    if n==0:
        print(prefix)
    else:
        for i in range(k):
            gen_num(n-1,k,prefix+str(i))


gen_num(3, 10, "")
```

Генерация всех перестановок

```
def perm(prefix, original):
    if len(prefix)==len(original):
        print(prefix)
    else:
        for i in original:
            if i not in prefix:
                perm(prefix+i, original)
perm("", "12345")
```

12.Динамичное программирование сверху. Чистые функции. Кеширование

Чистая функция - функция, при одном и том же наборе входных данных выдаёт один и тот же набор выходных данных

Кэширование - сохранение выходных данных для определённого набора входных данных. Кэширование наиболее эффективно и "хорошо" работает для чистых функций

Динамическое программирование сверху - разбиваем текущую задачу на несколько маленьких. Зная ответ на маленькие задачи мы можем вычислить ответ для текущей. Аналогично маленькие задачи решаются через еще более маленькие.

Динамическое программирование чисел Фиббоначи через кэширование

```
from functools import lru_cache

@lru_cache
def fib (x):
    if x==0: return 0
    if x==1: return 1
    if x>=2: return fib(x-1)+fib(x-2)

print(fib(int(input())))
```

10.Двоичный поиск в массиве/списке. Алгоритмическая сложность и применимость

Прежде всего нам нужен уже отсортированный массив. Идея следующая: делим массив пополам и переходим к той части, в которой может находиться элемент. Далее аналогично делим эту часть пополам, пока не найдем элемент или поймем, что его нет.

```
def binary_search(a, key):
    left=-1
    right=len(a)
    while right>left+1:
        middle=(right+left)//2
        if a[middle]>key:
            right=middle
        else:
            left=middle
    if left!=-1 and a[left]==key:
        return left

a = list(map(int, input().split()))
key = int(input())

print(binary_search(a, key))
```

Сложность: $O(log_2(n))$

Применимость: работает только на отсортированном массиве целых чисел.

4.Постановка задачи сортировки. Когда задача некорректна. Сортировка обезьяны и сортировка дурака.

Определение. На множестве U задан линейный порядок, если есть такой бинарный предикат f, удовлетворяющая следующим свойствам:

1. (иррефлексивность) $\forall x \in U \ f(x, x) = False$
2. (транзитивность) $\forall x, y, z \in U \ f(x, y) \text{ and } f(y, z) \Longrightarrow f(x, z)$
3. (антисимметричность) $\forall x, y \in U \ f(x, y) = \text{not } f(y, x)$

Определение. Пусть дан массив a длины N из элементов из множества U, при этом на элементах U введен линейный порядок, тогда задачей сортировки называют поиск такой перестановки $\sigma \in S_N$, что $a_{\sigma(1)} < \ldots < a_{\sigma(N)}$.

Для сортировки необходимо, чтобы структура данных была изменяема, однородна (чтобы элменты были схожего типа и мы могли бы их сравнивать) и в ней мог быть порядок. Если какое-то из условий не выполняется, то задача сортировки некорректна.

Пример некоректной задачи сортировки [Камень, Ножницы, Бумага]

Сортировка обезьяны - переставлять элементы структуры данных случайным образом и проверять их на соблюдение необходимого порядка.

Утверждение (б/д). Среднее время работы такой сортировки составит $O(N * N!)$.

```
def is_sorted(arr):
    for i in range(len(arr) - 1):
        if arr[i] > arr[i + 1]:
            return False
    return True

def bogosort(arr):
    while is_sorted(arr) == False:
        random.shuffle(arr)
```

```
%%time

bogosort([1, 3, 4, 6, 2, 5, 8, 10, 9])
```

```
CPU times: user 763 ms, sys: 9.29 ms, total: 772 ms
Wall time: 772 ms
```

```
%%time

[1, 3, 4, 6, 2, 5, 8, 10, 9].sort()
```

```
CPU times: user 1e+03 ns, sys: 1 µs, total: 2 µs
Wall time: 3.1 µs
```

Сортировка дурака - идем с самого начала до нахождения первой инверсии двух послежовательных элементов. Если такая есть, то устраняем ее и возвращаемся в начало.

Докажем корректность такого алгоритма. Рассмотрим число инверсий. Каждая итерация внешнего while уменьшает их количество хотя бы на 1. И в случае, если инверсий нет, то алгоритм заканчивает работу.

Время работы. Суммарно число инверсий в массиве не превосходит $\frac{n(n-1)}{2}$, откуда число инверсий в изначальном массиве длины N составит $O(N^2)$. А, так как на поиск каждой инверсии мы тратим $O(N)$ действий, итоговое время работы: $O(N^3)$.

```
def stupid_sort(arr):
    cur_idx = 1
    while cur_idx != len(arr) - 1:
        if arr[cur_idx - 1] > arr[cur_idx]:
            arr[cur_idx - 1], arr[cur_idx] = arr[cur_idx], arr[cur_idx - 1]
            cur_idx = 1
        else:
            cur_idx += 1
    return arr
```

```
%%time

stupid_sort([1, 3, 4, 6, 2, 5, 8, 10, 9])
```

```
CPU times: user 5 µs, sys: 0 ns, total: 5 µs
Wall time: 7.39 µs

[1, 2, 3, 4, 5, 6, 8, 10, 9]
```

1.Примитивный тест простоты и решето Эратосфена. Что экономнее и когда?

Примитивный тест простоты

```
def is_prime(x):
    if x == 2:
        return True
    if x < 2:
        return False
    for divider in range(2, x):
        if x % divider == 0:
            return False
        if divider * divider > x:
            return True
```

```
pprint(list(map(lambda x: (x, is_prime(x)), [i for i in range(10)])))
```

```
[(0, False),
 (1, False),
 (2, True),
 (3, True),
 (4, False),
 (5, True),
 (6, False),
 (7, True),
 (8, False),
 (9, False)]
```

Решето Эратосфена. Алгоритм для проверки всех чисел до N на простоту.

Работает за окололинейное время (стороного порядка $N \log \log N$ действий) и требует линейную память (массив длины N).

Наивный же алгоритм будет совершать порядка $N\sqrt{N}$ действий и также потребует линейной памяти на хранение результата.

По сути алгоритм считает изначально каждое число простым, а далее идет по ним и вычеркивает все числа, делящиеся на данное.

```
def eratosphene_sieve(n):
    is_prime = [True] * (n + 1)
    is_prime[0] = is_prime[1] = False

    divider = 2
    while divider * divider <= n:
        if is_prime[divider]:
            current_number = 2 * divider
            while current_number < n:
                is_prime[current_number] = False
                current_number += divider
        divider += 1
    return is_prime
```

```
pprint(list(zip(range(10), eratosphene_sieve(10))))
```

```
[(0, False),
 (1, False),
 (2, True),
 (3, True),
 (4, False),
 (5, True),
 (6, False),
 (7, True),
 (8, False),
 (9, False)]
```

2.Алгоритм Евклида. Факторизация числа. Сложность по времени вычислений.

Определение. Функция $f(n) = O(g(n))$, если $\exists C, N > 0: \ \forall n > N \ f(n) < C * g(n)$

Алгоритм Евклида - алгоритм нахождения НОД (наибольшего общего делителя). Работает за $O(\log \min(a, b))$ времени и константное количество дополнительной памяти.

```
def gcd(a, b):
    while a != 0 and b != 0:
        if a > b:
            a = a % b
        else:
            b = b % a
    return max(a, b)
```

```
n = 5
pprint([[(i, j, gcd(i, j)) for i in range(2, n)] for j in range(2, n)])
```

```
[[(2, 2, 2), (3, 2, 1), (4, 2, 2)],
 [(2, 3, 1), (3, 3, 3), (4, 3, 1)],
 [(2, 4, 2), (3, 4, 1), (4, 4, 4)]]
```

```
x = 72

n = 2
while n * n <= x:
    if x % n == 0:
        print(n)
        x = x // n
    else:
        n += 1
if x != 1:
    print(x)
```

```
2
2
2
3
3
```

7.Сортировка Тони Хоара. Чем хороша и плоха?

```
def quick_sort(a):
    if len(a)<=1:
        return a
    else:
        pivot=random.choice(a)
        less=[x for x in a if x < pivot]
        equal=[pivot]*a.count(pivot)
        greater=[x for x in a if x > pivot]
        return quick_sort(less)+equal+quick_sort(greater)
```

```
%%time

a = quick_sort(m)
```

```
CPU times: user 1.35 ms, sys: 40 µs, total: 1.39 ms
Wall time: 1.39 ms
```

Достоинства:

1. Сложность $O(N \log N)$.

Недостатки:

1. Требует выделение дополнительной памяти на стек рекурсии (существует реализация Partition без привлечения допмамяти).
2. В худшем случае время работы $O(N^2)$.

3.Однопроходные алгоритмы редукции последовательности. Инициализация переменных и итерирование.

Редукция последовательности - выполнение определённого алгоритма, на вход которому подают последовательность, а на выходе получают некоторое интегральное число (сумма, произведение или количество всех членов последовательности)

```
def max_from_nonnegative_array(array):
    result = 0 # инициализация
    for elem in array: # итерирование
        result = max(result, elem)
    return result
```

5.Три квадратичные универсальные сортировки. Сравнить и выбрать лучшую, обосновать свой выбор.

```
m = [random.randint(1, 100000) for _ in range(1000)]
```

****Сортировка пузырьком****

На каждой итерации поднимаем элемент, пока он больше следующего. Например, после первой итерации в конце будет самый большой элемент массива. Работает за $O(N^2)$

```
%%time

a = m.copy()

for i in range (len(a)-1):
    for j in range (len(a)-1-i):
        if a[j]>a[j+1]:
            a[j],a[j+1]=a[j+1],a[j]
```

```
CPU times: user 66.7 ms, sys: 1.4 ms, total: 68.1 ms
Wall time: 67.2 ms
```

Сортировка выбором

Ищем в массиве элемент, который должено стоять на i-ой похиции. Например, на первой итерации мы поставим минимальный элемент в начало массива, на второй минимальный элемент из оставшегося массива и так далее. Работает за $O(N^2)$

```
%%time

a = m.copy()

for i in range (len(a) - 1):
    imin=i
    for j in range (i + 1,len(a)):
        if a[j] < a[imin]:
            imin=j
    a[imin],a[i]=a[i],a[imin]
```

```
CPU times: user 33.1 ms, sys: 909 µs, total: 34 ms
Wall time: 33.5 ms
```

Сортировка вставками

Ищем на каком мвесте олжен стоять текущий элемент. Вставляем его на это место в новом массиве. Работает за $O(N^2)$

```
%%time

a = m.copy()

for i in range (1,len(a)):
    vrem=a[i]
    j=i-1
    while j>=0 and a[j]>vrem:
        a[j+1]=a[j]
        j-=1
    a[j+1]=vrem
```

```
CPU times: user 34.8 ms, sys: 1.22 ms, total: 36 ms
Wall time: 35.8 ms
```

Самая эффективная в python - выбором, так как делает много дешевых операций сравнения и очень мало дорогих операций замены

6.Алгоритм слияния упорядоченных списков. Сортировка слиянием. Чем хороша и чем плоха.

Определение. Пусть даны два отсортированных списка, b, тогда процедура *Merge(a, b)* вернет третий список c, который состоит из элементов a и b, при этом c будет отсортирован.

Процедура *Merge(A, B)* требует $O(|A| + |B|)$ времени и дополнительной памяти.

Устроена следующим образом. Заводим два счетчика $i = 0, j = 0$. i указывает на начало a, а j на начало b. Далее сравниваем a_i и b_j и пишем в результат меньший из них, сдвигая соответствующий счетчик вправо.

```
def merge(a, b):
    c = []
    i = 0
    j = 0
    while i < len(a) and j < len(b):
        if a[i] < b[j]:
            c.append(a[i])
            i += 1
        else:
            c.append(b[j])
            j += 1
    c += a[i:] + b[j:]
    return c
```

```
a = [1, 2, 5]
b = [0, 0, 0, 1, 2, 5, 6, 6]
print(merge(a, b))
```

```
[0, 0, 0, 1, 1, 2, 2, 5, 5, 6, 6]
```

Алгоритм. Сортировка слиянием:

1. Разбей массив на два пополам
2. Рекурсивно отсортируй
3. Сделай Merge от половинок

Время работы. Пусть $T(N)$ - время работы на массиве длины N. Тогда заметим, что мы рекурсивно вызываемся от двух половинок и делаем слияние за $O(N)$ времени. Откуда $T(N) = 2T(N/2) + c * N$.

Решение рекурренты.
$T(N) = 2T(N/2) + O(N) = 4T(N/4) + c * N + 2 * c * N/2 = 4T(N/4) + 2cN = \ldots = 2^k T(N/2^k) + k * c * N = \log_2 N * c * N = O(N \log N)$

```
def merge_sort(a):
    if len(a) <= 1:
        return a
    else:
        l = a[:len(a) // 2]
        r = a[len(a) // 2:]
    return merge(merge_sort(l), merge_sort(r))
```

```
%%time

sort = merge_sort(m)
```

```
CPU times: user 1.48 ms, sys: 14 µs, total: 1.5 ms
Wall time: 1.52 ms
```

Достоинства:

1. Сложность по времени.
2. Стабильна.

Недостатки:

1. На «почти отсортированных» массивах работает столь же долго, как на произвольных.
2. Требует $O(N)$ дополнительной памяти при слиянии.