Алгоритмы поисков

1) Линейный поиск

Выводит индекс первого вхождения искомого элемента

```
[ ]
    def LinearSearch(lys, element):
        for i in range (len(lys)):
            if lys[i] == element:
                return i
        return -1

    print(LinearSearch([1,2,3,4,5,2,1], 2))
```

2) Бинарный поиск

Бинарный поиск работает по принципу «разделяй и властвуй». Он быстрее, чем линейный поиск, но требует, чтобы массив был отсортирован перед выполнением алгоритма.

```
[ ] def BinarySearch(lys, val):
        first = 0
        last = len(lys)-1
        index = -1
        while (first <= last) and (index == -1):
            mid = (first+last)//2
            if lys[mid] == val:
                index = mid
            else:
                if val<lys[mid]:
                    last = mid -1
                else:
                    first = mid +1
        return index

    BinarySearch([10,20,30,40,50], 20)
```

3) Jump Search

Jump Search похож на бинарный поиск тем, что он также работает с отсортированным массивом и использует аналогичный подход «разделяй и властвуй» для поиска по нему. Его можно классифицировать как усовершенствованный алгоритм линейного поиска, поскольку он зависит от линейного поиска для выполнения фактического сравнения при поиске значения

```
    import math
    def JumpSearch (lys, val):
        length = len(lys)
        jump = int(math.sqrt(length))
        left, right = 0, 0
        while left < length and lys[left] <= val:
            right = min(length - 1, left + jump)
            if lys[left] <= val and lys[right] >= val:
                break
            left += jump;
        if left >= length or lys[left] > val:
            return -1
        right = min(length - 1, right)
        i = left
        while i <= right and lys[i] <= val:
            if lys[i] == val:
                return i
            i += 1
        return -1
    print(JumpSearch([1,2,3,4,5,6,7,8,9], 5))
```

4) Поиск Фибоначчи

Поиск Фибоначчи — это еще один алгоритм «разделяй и властвуй», который имеет сходство как с бинарным поиском, так и с jump search. Он получил свое название потому, что использует числа Фибоначчи для вычисления размера блока или диапазона поиска на каждом шаге.

```
def FibonacciSearch(lys, val):
    fibM_minus_2 = 0
    fibM_minus_1 = 1
    fibM = fibM_minus_1 + fibM_minus_2
    while (fibM < len(lys)):
        fibM_minus_2 = fibM_minus_1
        fibM_minus_1 = fibM
        fibM = fibM_minus_1 + fibM_minus_2
    index = -1;
    while (fibM > 1):
        i = min(index + fibM_minus_2, (len(lys)-1))
        if (lys[i] < val):
            fibM = fibM_minus_1
            fibM_minus_1 = fibM_minus_2
            fibM_minus_2 = fibM - fibM_minus_1
            index = i
        elif (lys[i] > val):
            fibM = fibM_minus_2
            fibM_minus_1 = fibM_minus_1 - fibM_minus_2
            fibM_minus_2 = fibM - fibM_minus_1
        else :
            return i
    if(fibM_minus_1 and index < (len(lys)-1) and lys[index+1] == val):
        return index+1;
    return -1

print(FibonacciSearch([1,2,3,4,5,6,7,8,9,10,11], 6))
```

5) Экспоненциальный поиск

Определяется диапазон, в котором, скорее всего, будет находиться искомый элемент. В этом диапазоне используется двоичный поиск для нахождения индекса элемента.

```
def ExponentialSearch(lys, val):
    if lys[0] == val:
        return 0
    index = 1
    while index < len(lys) and lys[index] <= val:
        index = index * 2
    return BinarySearch( lys[:min(index, len(lys))], val)
```

Сложность алгоритмов:

1) Сложность: O(n)
2) Сложность: O(nlogn)
3) Сложность: O(√n)
4) Сложность: O(logn)
5) Сложность: O(nlogn)