<https://jovian.com/learn/data-structures-and-algorithms-in-python/lesson/lesson-1-binary-search-linked-lists-and-complexity>

Polynomials

Es una suma de términos con coeficientes que tienen un exponente positivo y entero además de que se multiplican con una variable.

Los coeficientes son los números positivos o negativos: -10 o +15.

10x7 -9x2 + 15x3 + 9, aquí hay 4 términos.

Monomio: 9 o 15x3

Binomio: 15x3 + 9

Trinomio: -9x2 + 15x3 + 9

Polinomio: 10x^­-9x2 + 15x3 + 9

El grado de un Polinomio: Es el exponente: 10x­7 (el grado de este polinomio es una séptima o 7).

Si te preguntan por el grado de todo el polinomio te están preguntando por el más alto: -9x2 + 15x4 + 9 (en este caso es un 4).

La forma estándar de escribir un polinomio con los grados en orden, mayor a menor: 10x7 -9x3 + 15x2 + 9 y el polinomio principal es el que este de primero en la lista: 10x7.

Los polinomios no pueden tener potencias negativas:

10x­-7 -9x2 + 15x3 + 9 10x­7 -9x2 + 15x3 + 9

10x1/2 -9x2 + 15x3 + 9

10xa -9x2 + 15x3 + 9

10√x-9x2 + 15x3 + 9

Matrices

Es una lista de datos.

A)

|  |  |  |
| --- | --- | --- |
| 1 | 0 | -7 |
| PI | 5 | 11 |

2 x 3 (2 filas x 3 columnas)

B)

|  |  |  |
| --- | --- | --- |
| 1 | 0 | -7 |
| 3 | 20 | 25 |
| PI | 5 | 11 |

3 x 3 (3 filas x 3 columnas)

La única forma de multiplicar dos matrices es si tiene el mismo número de columnas.

Si se puede definir: B x (otra matriz con 3 columnas)

No se puede definir:

A x B (diferentes columnas)

Pero si tenemos un (2x2) (1x2) tampoco se podría, porque el 1 en la fila. La forma correcta seria:

(1x2) (2x2), porque el numero de columnas de la primera matriz coincide con el numero de filas de la segunda matriz.

Por ejemplo (EL ORDEN IMPORTA AL MULTIPLICAR MATRICES)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A **(2X3)** | | | X | B **(3X2)** | |
| 0 | 3 | 5 | 3 | 4 |
| 5 | 5 | 2 | 3 | -2 |
|  | | | 4 | -2 |

La primera Fila de A por las dos Columnas de B. lo mismo con la segunda Fila de A, Se resuelven todas las columnas en conjunto, y se colocan en fila.

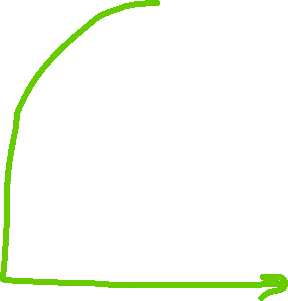
0x3 + 3x3 + 5x4 = 29 0x4 + 3(-2) + 5(-2) = -16



5x3 + 5x3 + 2x4 = 38 5x4 + 5(-2) + 2(-2) = 6



Representación en una Matriz



|  |  |  |  |
| --- | --- | --- | --- |
| F1 de A  Por C1 y C2 de B | | F2 de A  Por C1 y C2 de B | |
| 0x3 | 0x4 | 5x3 | 5x4 |
| 3x3 | 3(-2) | 5x3 | 5(-2) |
| 5x4 | 5(-2) | 2x4 | 2(-2) |

Resultado de la suma y multiplicación (Recordemos, el resultado de la F1 es en fila, lo mismo con la F2)

|  |  |
| --- | --- |
| 29 | -16 |
| 38 | 6 |

Una transposición de matriz o AT se refiere a que las Filas serán Columnas y las Columnas serán Filas.

|  |  |
| --- | --- |
| 3 | 4 |
| 3 | -2 |
| 4 | -2 |

Esto es igual al:

|  |  |  |
| --- | --- | --- |
| 3 | 3 | 4 |
| 4 | -2 | -2 |

Cuando multiplicas un entero por una matriz, estas multiplicando cada elemento de la matriz por ese numero para generar una matriz con las mismas filas y columnas, pero diferentes valores.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | X | 5 | 3 | = | 15 | 9 |
| -5 | -3 | -15 | -9 |

Para poder sumar dos matrices, estas deben tener las mismas dimensiones y se suma cada numero con el que le corresponde, ejemplo (1 + 5)



|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | -7 | 5 | + | 5 | 0 | 3 | = | 6 | -7 | 8 |
| 0 | 3 | -10 | 11 | -1 | 7 | 11 | 2 | -3 |

Con el signo negativo, debemos hacer uso de la ley de los signos.



|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | -7 | 5 | - | 5 | 0 | 3 | = | -4 | -7 | 2 |
| 0 | 3 | -10 | 11 | -1 | 7 | -11 | 4 | -17 |

Cuando las matrices, tienen diferentes dimensiones, son Undefined, o indefinidas.

Probability

Es la posibilidad de que pase algo, la ecuación es: P(H)

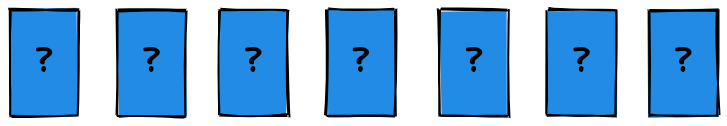
P es probabilidad y la H es: ***las posibilidades de que mi condición se cumpla*** / ***las posibilidades que hay en total***

Una moneda, y quiero que salga cara, su posibilidad es ½, o del 50% porque:

1 = porque solo hay una posibilidad de que se cumpla mi condición, solo hay una forma en la cual puede salir cara y es que la moneda aterrice en cara.

2 = porque solo hay dos posibilidades, que salga cara o cruz después de lanzarla.

PREGUNTA 1: Alicia tiene unas tarjetas con números escritos en ellas. Acomoda las cartas en orden decreciente y las coloca boca abajo en una secuencia sobre una mesa. Ella desafía a Bob a elegir la tarjeta que contiene un número dado dando la vuelta a la menor cantidad de tarjetas posible. Escribe una función para ayudar a Bob a localizar la tarjeta.



Antes de empezar a escribir debemos crear una estrategia de pasos de forma sistemática ya que, si empezamos sin plantearnos que haremos, gastaremos mas tiempo de lo normal.

Aquí hay una estrategia sistemática que aplicaremos para resolver los problemas:

1. Plantear claramente el problema. Identificar los formatos de entrada y salida.
2. Proponga algunos ejemplos de entradas y salidas. Trate de cubrir todos los casos extremos.
3. Llegar a una solución correcta para el problema. Dígalo en inglés sencillo.
4. Implementar la solución y probarla usando entradas de ejemplo. Corregir errores, si los hay.
5. Analizar la complejidad del algoritmo e identificar ineficiencias, si las hubiere.
6. Aplicar la técnica adecuada para superar la ineficiencia. Repita los pasos 3 a 6.

"Aplicar la técnica correcta" es donde el conocimiento de estructuras de datos y algoritmos comunes resulta útil.

APLICACIÓN:

1. Plantee el problema claramente. Identificar los formatos de entrada y salida.

A menudo encontrará problemas escritos detallados en desafíos de codificación y entrevistas. El primer paso es plantear el problema de forma clara y precisa en términos abstractos, es decir, que cualquier persona pueda entender.

Se vale hacer gráficos.

Diagrama, Esquemático

Descripción generada automáticamente

Por ejemplo, en este caso, las cartas se pueden representar como una lista (Sorted List) y darle la vuelta para saber que contiene esa carta, equivale a obtener el numero a través del índice.

Nota: Acostúmbrate a contar desde 0 no desde el 1.

Sabiendo esto podemos decir que:

*Plantear claramente el problema:*

Necesitamos un programa que acceda a los elementos (Números) de la lista que esta ordenada (Sort) decrecientemente, esto por el índice (Índex), también necesitamos minimizar la cantidad de veces que accedemos a los elementos de la lista.

*Identificar los formatos de entrada y salida.*

Input

Cards: Una lista de números ordenados en orden decreciente. P.ej. [13, 11, 10, 7, 4, 3, 1, 0]

Query: Un número, cuya posición en la matriz se va a determinar. P.ej. el número que buscamos es el 7

Output

Posición: La posición de la consulta en las tarjetas de lista. P.ej. buscamos el 7, entonces es el 3 en el caso anterior (contando desde 0)

Entonces, basados en esta información, podemos empezar a crear el código, en este caso una Def.

def located\_card (cards, query):

    pass

Con esta def cumplimos la parte del Input sobre recibir la lista de cartas (Cards) y el número que queremos buscar (Query)

Tips

1. Nombra tus funciones en base al problema, no con letras o cosas aleatorias, ya que esto es en equipo y se comparte con la empresa.
2. Nombra tus parámetros en base al Input que necesites para solucionar tu problema.
3. Puedes pedir ayuda a tu entrevistador si no sabes cómo enmarcarlo en términos abstractos.
4. Al declarar una variable, asegúrate de que tenga que ver con el problema.
5. También no pongas letras porque después de 30 minutos de código, puede que ni te acuerdes que hacía b.

Y con esto pasamos al paso 2 de la resolución de problemas.

2. Proponga algunos ejemplos de entradas y salidas. Trate de cubrir todos los casos extremos.

Antes de empezar a usar la función seria útil, empezar a realizar algunas pruebas y analizar posibles escenarios, ya que es normal que, al empezar a escribir código, haya fallos y situaciones no calculadas, en especial si es una entrevista de trabajo, donde hay mucho estrés.

Por ejemplo, definimos los inputs y el output que estaremos usando:

# la lista de cartas

***cards = [13, 11, 10, 7, 4, 3, 1, 0]***

# el número que queremos buscar

***query = 7***

# el valor que esperamos

***output = 3***

En este caso, lo que haremos para comprobar que todo esta bien es comprobar el resultado de la función (result = located\_card(cards, query)) con el output (output = 3).

Ya que es la única forma de ver si el resultado coincide y si es correcto.

En este caso, vamos a organizar los elementos en un DICT:

test = {

    "input”: {

        "cards": [13, 11, 10, 7, 4, 3, 1, 0],

        "query": 7

    },

    "output": 3

}

Esto es para comprobar que el resultado de la función es igual al output que tenemos en el Dict.

Nota: los \*\* van a pasar solo los datos del Dict que coincidan con los parámetros de la def, en este caso cards y Query, ya que son los parámetros que escribimos en la def.

located\_card(\*\*test['input']) == test['output']

La idea es que nuestra def pueda manejar cualquier cantidad de lista y buscar el Query sin problemas, por eso es normal que escribamos los posibles escenarios, en forma de enunciados y en código, por ejemplo:

1. El Query aparece en medio de la lista

test{

    'input': {

        'cards': [13, 11, 10, 7, 4, 3, 1, 0],

        'query': 7

    },

    'output': 3

}

1. Query es el primer elemento

test{

    'input': {

        'cards': [4, 2, 1, -1],

        'query': 4

    },

    'output': 0

}

1. Query es el último elemento

test{

    'input': {

        'cards': [3, -1, -9, -127],

        'query': -127

    },

    'output': 3

}

1. La lista contiene solo un elemento que es Query

test{

    'input': {

        'cards': [6],

        'query': 6

    },

    'output': 0

}

1. La lista no contiene el Query

test{

    'input': {

        'cards': [9, 7, 5, 2, -9],

        'query': 4

    },

    'output': -1

}

1. La lista está vacía

test{

    'input': {

        'cards': [],

        'query': 7

    },

    'output': -1

}

1. La lista contiene números que se repiten

test{

    'input': {

        'cards': [8, 8, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0],

        'query': 3

    },

    'output': 7

}

1. El Query aparece en más de 1 posición

test{

    'input': {

        'cards': [8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0],

        'query': 6

    },

    'output': 2

}

1. ¿Hay más posibles escenarios?

#?????

Es normal no pensar en todo esto cuando se nos plantea el problema, el caso de que la lista este vacía y otros, se llaman casos limites, ya que son cosas raras o extremas, si bien no son frecuentes, el programa debe tenerlos en cuenta.

Nota: es importante que, si tienes alguna duda, o quieres verificar algo, preguntes a tu entrevistador ya que esa duda puede ser un problema a futuro y un nuevo caso de prueba.

Los casos de pruebas nos servirán para evaluar nuestro Código, además de que a medida que vayamos avanzando en el Código, descubriremos más casos de prueba y podremos volver a este paso.

3. Llegar a una solución correcta para el problema. Dígalo en inglés sencillo.

Nuestro deber es encontrar la solución correcta que es la solución mas eficiente.

La solución simple u obvia, que es verificar todos los casos de pruebas, es solución de fuerza bruta.

En este problema Bob podría levantar todas las cartas una por una, hasta encontrar el Query (Lineal Search):

Pero es importante que antes de empezar a codificar, empecemos declarando el algoritmo, o una idea del paso a paso a seguir del Código, en este caso haremos una Lineal Search:

1. Creas la función con los parámetros de entrada ***(def located\_card (cards, query) :).***
2. Crear una var ***posición*** con el valor en 0, para el Index.
3. Crea un loop para iterar en los elementos de la lista, en este caso un ***While***.
4. Agrega un condicional ***(If)*** para verificar si el numero que esta iterando el Loop es igual al Query.
5. Si es True, Return a la ***posición***.
6. Si no, incrementa el valor de posición en 1 ***(position += 1),*** esto para que la ***posición*** recorra toda la lista.
7. Si no encuentra el número, que devuelva -1 o incluso False.
8. Llamar a la función, pasándole los parámetros (located\_card(cards, query)).

4. Implementar la solución y probarla usando entradas de ejemplo. Corregir errores, si los hay.

Una vez tenemos claros como vamos a proceder, seguiría implementar la solución o el paso a paso que hayamos seleccionado.

def located\_card (cards, query):

    # Crea una variable `posición` (Index) con el valor 0.

    position = 0

    # Set up a loop for repetition.

    while True:

        # Verifique si el número en el Index `posición` en `tarjeta` es igual a `Query`.

        if cards[position] == query:

        # Si es así, `posición` es la respuesta y se puede devolver desde la función.

            return position

        # Si no, incrementa el valor de `posición` en 1, para que el loop se vuelva a ejecutar

        position += 1

        # Check if we have reached the end of the array.

        if position == len(cards):

            # Si no se encontró el número, devuelva `-1`.

            return -1

located\_card(cards, query)

Aquí otra forma usando el loop For:

def located\_card(cards, querry):

    for i, b in enumerate(cards):

        if cards[i] == querry:

            print(f"el numero que buscas es: {b} y su posicion es: {i}")

            break

located\_card(cards, querry)

VERSION DONDE SOLO IMPRIME LA POSICION:

def located\_card(cards, querry):

    for i, b in enumerate(cards):

        if cards[i] == querry:

            return i

print(located\_card(cards, querry))

bien, entonces solo queda verificar si el resultado, es igual al que estábamos esperando:

Version While:

result = located\_card(test['input']['cards'], test['input']['querry'])

print(result == test['output'])

Resultado = True

Version For:

if result == output:

    print(f"genial, la posicion {result} coincide con lo que esperaba que era {output}")

else:

    print("intentalo con otro numero")

Si bien, puede que todo marche bien, con el caso idóneo, es necesario poner a funcionar los casos de prueba

Después de probar todas las versiones con el loop while, el test numero 6 tiene un error, debido a que la lista que le estamos pasando está vacía ():

test = {

    "input":{

        "cards": [],

        "querry": 7

    },

    "output": -1

}

Y cuando tratamos de acceder a la posición 0, que es por la que empieza el loop, nos salta el error de que esta fuera de rango, ya que no hay un elemento 0, porque la lista está vacía.

Observa como el loop While, solo se quedó en la posición 0, ya que solo pudo iterar hasta ahí.

cards: []

query: 7

position: 0

Traceback (most recent call last):

File "c:\Users\Admin\Desktop\FASTAPI\_3\main.py", line 41, in <module>

result = located\_card(test['input']['cards'], test['input']['querry'])

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

File "c:\Users\Admin\Desktop\FASTAPI\_3\main.py", line 35, in located\_card

if cards[position] == query:

~~~~~^^^^^^^^^^

IndexError: list index out of range

Entonces para solucionar esto:

Le diremos que compare la posición 0 con la cantidad de elementos, siempre y cuando sea menor a la cantidad de elementos de la lista.

Si la lista está vacía, no habrá elementos, por ende, esa condición no se cumple, y hará return al -1.

Haciendo que se compare con el output y retornando True.

test = {

    "input":{

        "cards": [],

        "querry": 7

    },

    "output": -1

}

def located\_card(cards, query):

    position = 0

    while position < len(cards):

        if cards[position] == query:

            return position

        position += 1

    return -1

result = located\_card(test['input']['cards'], test['input']['querry'])

print(result == test['output'])

Gracias a este error el Código ha logrado pasar todos los test.

En el test 8, hay varios números, el while solo comprobaba el 2do, y dejaba los demás atrás, por eso cuando lo comparaba con el output salía false, lo que hice fue hacer la condición final, parte del while:

def located\_card (cards, query, output):

    position = 0

    while position < len(cards):

        if cards[position] == query:

            if position == output:

                return True

        position += 1

    return -1

result = located\_card(test['input']['cards'], test['input']['querry'], test['output'])

print(result)

5. Analizar la complejidad del algoritmo e identificar ineficiencias, si las hubiere.

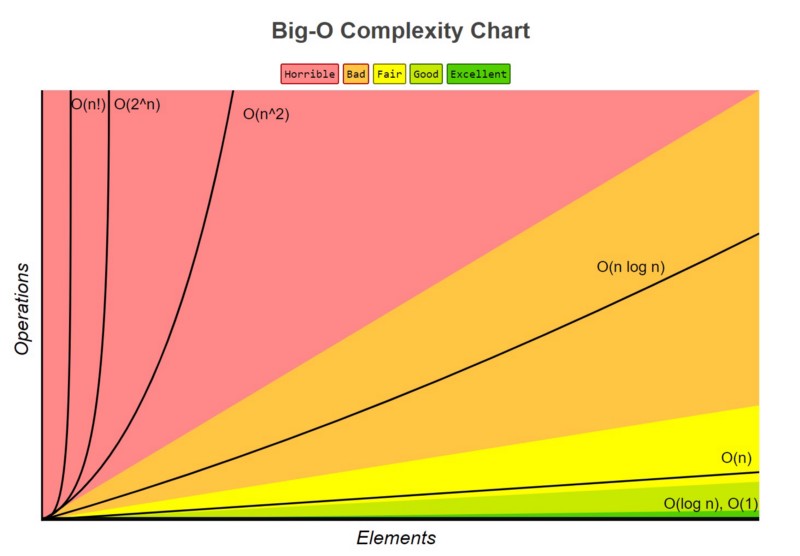
Recordemos que en el enunciado nos dijeron que debíamos de voltear la mínima cantidad de veces, en este caso, la forma de representar eso es con Big O.

Como para conocer donde esta la carta que buscamos (Querry) debemos ejecutar el Loop While, N número de veces, ósea cada iteración es N.

Por ejemplo, supongamos que el Loop solo itera un numero por minuto, le tomaría 25 minutos recorrer una lista con 25 elementos.

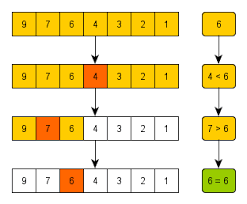
En cuando al espacio que ocupa (RAM), es algo constante, de ahí que en notación se le llame O, independientemente del tiempo que dure cada iteración, su espacio en el STACK será constante.

Por ende, el Big O de Search Lineal es 0(N) y su espacio es 0(1), en el ejemplo anterior seria, 0(25) – 0(1).



6. Aplicar la técnica adecuada para superar la ineficiencia. Repita los pasos 3 a 6.

Como la lista de cartas esta ordenada, podríamos aprovechar y utilizar la Binary Search.



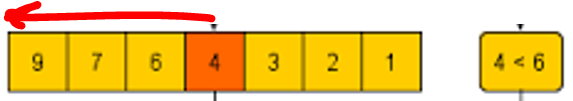
Si nos basamos en la idea de ejecutar menos pasos para conseguir el resultado, este tipo de algoritmo es más eficiente.

Así se aplicaría al problema de Bod:

* Buscamos el Numero del medio de la lista.



* Si el numero del medio coincide con el Querry, devuelve la posición.
* Si el numero del medio es menor al Querry, busca en la primera mitad.

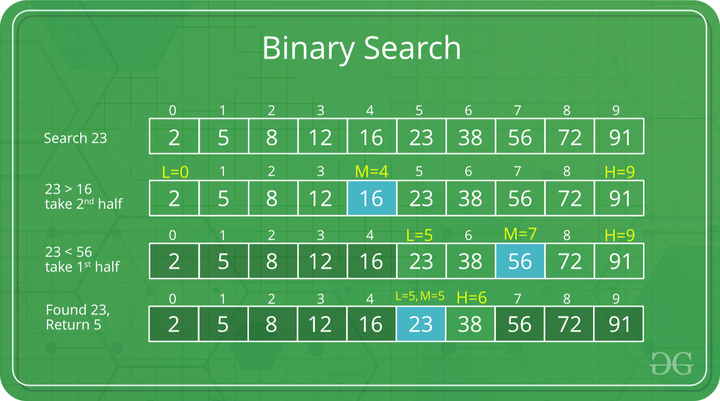


* Si el numero del medio es mayor al Querry, busca en la segunda mitad.

Diagrama

Descripción generada automáticamente

* Si el Query no está o no quedan más elementos, devuelve -1



Ejecución del Código:

Nota: al escribir una Def, se recomienda que esta no pase de 10 líneas de código, ya que es la cantidad de líneas en la que se puede retener información.

test = {

    "input":{

        "cards": [13, 11, 10, 7, 4, 3, 1, 0],

        "querry": 7},

    "output": 3}

def located\_card(cards, query):

    #esto es el inicio y el final de la búsqueda binaria, Lo = 0, y Hi = 7 (el len tira 8, pero el -1 lo deja en 7)

    lo, hi = 0, len(cards) - 1

    while lo <= hi:

        # Forma de buscar el Index del Medio

        mid = (lo + hi) // 2

        # forma de buscar el numero del medio a través del Index

        mid\_number = cards[mid]

        print("lo:", lo, ", hi:", hi, ", mid:", mid, ", mid\_number:", mid\_number)

        # si el numero del medio es igual al Querry

        if mid\_number == query:

            return mid

        # Si el numero del medio es menor al Query

        elif mid\_number < query:

            #Hi pasa a valer el numero del medio menos 1 y se ejecuta el while de nuevo

            hi = mid - 1

        #Si el numero del medio es mayor al Querry

        elif mid\_number > query:

            #Lo pasa a valer el número del medio más 1 y se ejecuta el while de nuevo

            lo = mid + 1

    # si el while no encuentra el numero se imprime esto

    return -1

result = located\_card(test['input']['cards'], test['input']['querry'])

print(result)

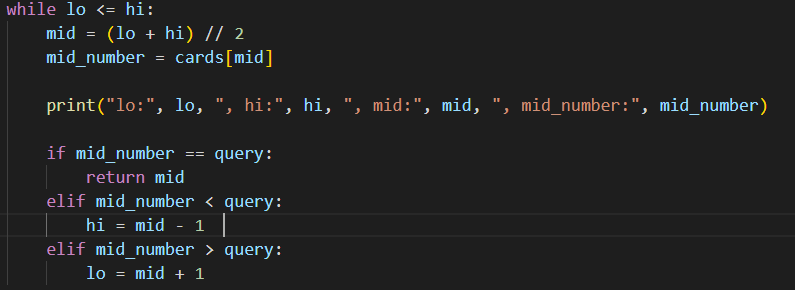
Toma en cuenta, que solo se parara el loop, hasta que se ejecute un Return, de lo contrario solo serán actualizaciones de valores para así seguir ejecutando el Código dentro del while.

Esta es la info:

Texto

Descripción generada automáticamente

El Código que se ejecuta es:



En este caso, buscábamos el 13:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Fíjate en la actualización de valores del Hi y el Mid en este caso.

En el ultimo test hay un error sobre si la lista tiene el Query repetido:

Texto

Descripción generada automáticamente con confianza media

La solución fue crear otra función donde se comprobase si el número del lado coincidía con el Query, y que retorne Left, que en el siguiente código significa que el algoritmo se moverá a la izquierda.

def test\_location(cards, query, mid):

    mid\_number = cards[mid]

    print("mid:", mid, ", mid\_number:", mid\_number)

    if mid\_number == query:

        if mid-1 >= 0 and cards[mid-1] == query:

            return 'left'

        else:

            return 'found'

    elif mid\_number < query:

        return 'left'

    else:

        return 'right'

def locate\_card(cards, query):

    lo, hi = 0, len(cards) - 1

    while lo <= hi:

        print("lo:", lo, ", hi:", hi)

        mid = (lo + hi) // 2

        result = test\_location(cards, query, mid)

        if result == 'found':

            return mid

        elif result == 'left':

            hi = mid - 1

        elif result == 'right':

            lo = mid + 1

    return -1

locate\_card(test["input"]["cards"], test["input"]["querry"])

De esta manera, se buscarán los números que coincidan, hasta llegar a la posición que quieras.

Su complejidad se refiere a una división de la matriz, ya que al comienzo contamos con una matriz completa de N, números ordenados.

Después en la siguiente iteración contamos con la mitad de N elementos, (N/2),

En la siguiente contamos con otra iteración (N/4) que podría ser (N/2^2)

Por ende, se diferencia de la búsqueda Lineal de O(N) ya que esta última, aumentaba por la cantidad de elementos de la lista, en esta, desciende, hasta encontrar el Query.

Por ejemplo:

cards = [13, 11, 10, 7, 4, 3, 1, 0]

Querry = 7

N se refiere al numero de elementos en la lista: en este caso, 8 (O (Log 8))

En la primera iteración esa lista se divide a la mitad, quedando 4 elementos. (O (Log 4))

En la segunda iteración esa lista se divide a la mitad, quedando 2 elementos. (O (Log 2))

En la tercera se reduce a 1 elemento. (O (Log N))

Si sacamos el (O (Log 8)), nos daremos cuenta de que es 3, la misma cantidad de pasos para conseguir el número 7.

La diferencia radical en cuando a la Binary Search y la Lineal Search, es el tiempo para conseguir un elemento.

|  |  |
| --- | --- |
| la Lineal Search | Binary Search |
| Result: 9999998  Passed: True  Execution Time: 1103.3 ms  Esta recorrió los: 10000000 elementos, uno por uno | Result: 9999998  Passed: True  Execution Time: 0.019 ms  Esta recorrió los: 10000000 elementos, Mitad a mitad |