Branch: master

Find file Copy path

03MAIR-Algoritmos-de-Optimizacion-2019 / SEMINARIO / Luis Fauré Navarro-SEMINARIO1.ipynb

lfauren Add files via upload f8350a7 a minute ago

1 contributor

561 lines (560 sloc) 21.2 KB

Raw Blame History

Open in Colab

Algoritmos de optimización - Seminario

Nombre y Apellidos: Luis Fauré Navarro
Url: https://github.com/lfauren/03MAIR-Algoritmos-de-Optimizacion-2019/tree/master/SEMINARIO
Problema:

> 1. Elección de grupos de población homogéneos
> 2. Organizar los horarios de partidos de La Liga
> 3. Combinar cifras y operaciones

Descripción del problema:(copiar enunciado)

1. Combinar cifras y operaciones. El problema consiste en analizar el siguiente problema y diseñar un algoritmo que lo resuelva. Disponemos de las 9 cifras del 1 al 9 y de los 4 signos básicos de las operaciones fundamentales: suma(+), resta(-), multiplicación(*) y división(/). Debemos combinarlos alternadamente sin repetir los números ni los signos. Debe analizarse el problema para encontrar todos los valores enteros posibles planteando las siguientes cuestiones: ####
   *¿Qué valor máximo y mínimo a priori se pueden obtener según las condiciones? Aplicando lógica el Mínimo = 4/2+1-9*8 = -69 y el Máximo = 9*8/1+7-2 = 77*
   #### ¿Es posible encontrar todos los valores enteros posibles entre dicho mínimo y máximo? A priori no se puede saber, porque uno podría hacer la resta del máximo con el mínimo, pero puede que se salten, además hay que considerar si es con repetición o sin repetición de los valores.
   Usando el programa sale el Mínimo = -69 y el Máximo = 77, efectivamente. La cantidad de valores entre el intervalo [-69, 77] es 147 y no se saltan, pero si consideramos que los valores se repiten el valor asciende de 147 a 90000.

(*) La respuesta es obligatoria

(*)¿Cuantas posibilidades hay sin tener en cuenta las restricciones?

(n!/(n-r)!)*m! Siendo n=digitos, r=cuantos dígitos se van a considerar, m=operaciones*
*(9!/(9-5)!)*4! = 362880

¿Cuantas posibilidades hay teniendo en cuenta todas las restricciones.

Hay dos casos los valores flotantes y los valores enteros. Lo que había que descartar eran los valores flotantes y cabe señalar que tuve un problema, ya que todos los resultados daban flotantes, incluso los enteros, por ejemplo 5.0 ya que habia una división en todos los casos, se hicieron una serie de operaciones para quitarle a los enteros el cero y así, transformarlos en enteros. Habían 147 enteros sin repetir los valores y 90000 enteros con repetición de los números. No existe un número uniforme de repetición para cada número, es decir, 147*n = 90000, no existe un n.

In [1]:
```
def factorial(n):
    if n <1:
        return 1
    else:
        return n*factorial(n-1)

num_posibilidades = (factorial(9)/factorial(9-5))*factorial(4)
print (num_posibilidades)
```

```
362880.0
```

Modelo para el espacio de soluciones
(*) ¿Cual es la estructura de datos que mejor se adapta al problema? Argumentalo.(Es posible que hayas elegido una al principio y veas la necesidad de cambiar, arguentalo)

Las mejores estructuras de datos en este caso fueron las listas de string para luego convertir con la función eval a una expresión numérica, si los números hubieran sido del 1 al 1000, fallaba el algoritmo, pero como eran dígitos se podía usar la función permutations del módulo itertools. También hubiera servido letras, pero ahí no se hubiera podido calcular la expresión.

Según el modelo para el espacio de soluciones
(*)¿Cual es la función objetivo?

expresion = eval(i[0]+j[0]+i[1]+j[1]+i[2]+j[2]+i[3]+j[3]+i[4]) Donde los "i" son números y los "j" son operaciones aritméticas, i[xyznm] donde xyznm toma 5 dígitos del 1 al 9 y j[abcd] donde abcd toma las 4 operaciones aritméticas, todas en string.

(*)¿Es un problema de maximización o minimización?

Es un problema de minimización, porque se buscan los valores enteros desde el menor -69 al mayor 77, osea, 147 valores de un total de 362880 posibilidades numéricas de los cuales se descartan flotantes y repetidos.

Diseña un algoritmo para resolver el problema por fuerza bruta

Este algoritmo funciona desde la función principal valores, la función val entrega los resultados. En la función valores toma dos listas. En una lista guarda los valores enteros repetidos y en la otra guarda los valores enteros no repetidos.

In [13]:
```
from itertools import permutations
from time import time

#Función para calcular el tiempo de ejecución
def calcular_tiempo(f):
    def wrapper(*args, **kwargs):
        inicio = time()
        resultado = f(*args, **kwargs)
        tiempo = time() - inicio
        print("Tiempo de ejecución para algoritmo: "+str(tiempo))
        return resultado
    return wrapper

def valores():
    lista1,lista2=[],[]
    for i in permutations('123456789', 5):
        for j in permutations('+-*/', 4):
            expresion = eval(i[0]+j[0]+i[1]+j[1]+i[2]+j[2]+i[3]+j[3]+i[4])
            if abs(expresion) - abs(int(expresion)) == 0:
                lista1.append(int(expresion))
    lista1 = sorted(lista1)  # todos los valores enteros repetidos
    for k in lista1:         # filtra los valores repetidos
        if k not in lista2:
            lista2.append(k)
    return lista2

@calcular_tiempo
def val(a):
    lista=valores()
    print('Valores no repetidos',lista)
    minimo=lista[0]
    maximo=lista[-1]
    distancia=len(lista)
    print('máximo:',maximo,'mínimo:',minimo,'distancia(inclusive):',distancia)
val(2)
```

```
Valores no repetidos [-69, -68, -67, -66, -65, -64, -63, -62, -61, -60, -59, -58, -57, -56, -55,
-54, -53, -52, -51, -50, -49, -48, -47, -46, -45, -44, -43, -42, -41, -40, -39, -38, -37, -36, -
35, -34, -33, -32, -31, -30, -29, -28, -27, -26, -25, -24, -23, -22, -21, -20, -19, -18, -17, -1
6, -15, -14, -13, -12, -11, -10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8,
9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 3
3, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 5
7, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77]
máximo: 77 mínimo: -69 distancia(inclusive): 147
Tiempo de ejecución para algoritmo: 3.2348897457122803
```

Calcula la complejidad del algoritmo por fuerza bruta

Este cálculo está hecho sobre la función principal valores. Para calcular la complejidad del algoritmo hay que usar lo que se llama variación o variación sin repetición, si de n números escojo 5, me queda $\frac{n!}{(n-5)!}$ y si le anido un ciclo for multiplico en este caso por 4!(las 4 operaciones).

$$\frac{n!}{(n-5)!} \cdot 4! = \frac{9!}{(9-5)!} \cdot 4! = \frac{9!}{4!} \cdot 4! = 9! = n!, (9+4+3) \cdot n! + 2n + 2 + 1 + 1 = 16n! + 2n + 2 = O(n!)$$

(*)Diseña un algoritmo que mejore la complejidad del algortimo por fuerza bruta. Argumenta porque crees que mejora el algoritmo por fuerza bruta

El algoritmo mejora porque tiene menos código, trabaja con una sola lista, el anterior con dos. Quite un for, es decir, una "n" menos. Si bien es cierto que el algoritmo no muestra tiempos de ejecución, tal vez con más o menos datos si se compara con el anterior debería demorar menos.
En este programa la función valores trabaja con una sola lista donde guarda los elementos [-69, 77], la lista contiene 147 elementos.

In [12]:
```python
from itertools import permutations
from time import time

#Función para calcular el tiempo de ejecución
def calcular_tiempo(f):
    def wrapper(*args, **kwargs):
        inicio = time()
        resultado = f(*args, **kwargs)
        tiempo = time() - inicio
        print("Tiempo de ejecución para algoritmo: "+str(tiempo))
        return resultado
```

```
    return wrapper

def valores():
    lista1=[]
    for i in permutations('123456789', 5):
        for j in permutations('+-*/', 4):
            expresion = eval(i[0]+j[0]+i[1]+j[1]+i[2]+j[2]+i[3]+j[3]+i[4])
            exp=int(expresion)
            if abs(expresion) - abs(exp) == 0 and exp not in lista1:
                lista1.append(exp)
    lista1 = sorted(lista1)
    return lista1

@calcular_tiempo
def val(a):
    lista=valores()
    print('Valores no repetidos',lista)
    minimo=lista[0]
    maximo=lista[-1]
    distancia=len(lista)
    print('máximo:',maximo,'mínimo:',minimo,'distancia(inclusive):',distancia)
val(2)
```

```
Valores no repetidos [-69, -68, -67, -66, -65, -64, -63, -62, -61, -60, -59, -58, -57, -56, -55,
-54, -53, -52, -51, -50, -49, -48, -47, -46, -45, -44, -43, -42, -41, -40, -39, -38, -37, -36, -
35, -34, -33, -32, -31, -30, -29, -28, -27, -26, -25, -24, -23, -22, -21, -20, -19, -18, -17, -1
6, -15, -14, -13, -12, -11, -10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8,
9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 3
3, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 5
7, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77]
máximo: 77 mínimo: -69 distancia(inclusive): 147
Tiempo de ejecución para algoritmo: 3.3127963542938232
```

(*)Calcula la complejidad del algoritmo

Este cálculo está hecho sobre la función principal valores. Igual que el cálculo de complejidad del ejercicio anterior para calcular la complejidad del algoritmo hay que usar lo que se llama variación y si le anido un ciclo for multiplico en este caso por 4!(las 4 operaciones).

$$\frac{n!}{(n-5)!} \cdot 4! = \frac{9!}{(9-5)!} \cdot 4! = \frac{9!}{4!} \cdot 4! = 9! = n!$$

$$(9+2+2)\cdot n!+1+1+1=13n!+3=O(n!)$$

Según el problema (y tenga sentido), diseña un juego de datos de entrada aleatorios

La entrada de datos estaría dada por "random.choice" que admite strings. Lo otro interesante de esta función es el "if not n in l2". Lo ocupo en un for y un if del primer programa principal en la función valores. Todo esto está en la bibliografía de internet.

In [18]:
```
import random

def aleatorio():
    expresion = 2.3
    while (abs(expresion) - abs(int(expresion))) != 0:
        l1,l2=[],[]
        while len(l1) < 5:
            n = random.choice('123456789')
            if not n in l1:
                l1.append(n)
            while len(l2) < 4:
                n = random.choice('+-*/')
                if not n in l2:
                    l2.append(str(n))
        cadena = l1[0]+l2[0]+l1[1]+l2[1]+l1[2]+l2[2]+l1[3]+l2[3]+l1[4]
        expresion = eval(l1[0]+l2[0]+l1[1]+l2[1]+l1[2]+l2[2]+l1[3]+l2[3]+l1[4])
    return cadena,expresion

cadena, expresion = aleatorio()
print(cadena,'=',int(expresion))
```

```
5+3/1*7-9 = 17
```

Aplica el algoritmo al juego de datos generado

Este programa fusiona el programa anterior con el programa mejorado en su complejidad, aunque es un poco distinto, porque utiliza ciclos while. Este programa calcula 100 valores que realiza la función valores y desde ahí discrimina eligiendo los valores que se imprimen al final. Además, define una función aleatorio, la cual calcula un valor.

In [22]:
```
from itertools import permutations
from time import time
import random
```

```
#Función para calcular el tiempo de ejecución
def calcular_tiempo(f):
    def wrapper(*args, **kwargs):
        inicio = time()
        resultado = f(*args, **kwargs)
        tiempo = time() - inicio
        print("Tiempo de ejecución para algoritmo: "+str(tiempo))
        return resultado
    return wrapper

def aleatorio():
    expresion = 2.3
    while (abs(expresion) - abs(int(expresion))) != 0:
        l1,l2=[],[]
        while len(l1) < 5:
            n = random.choice('123456789')
            if not n in l1:
                l1.append(n)
            while len(l2) < 4:
                n = random.choice('+-*/')
                if not n in l2:
                    l2.append(str(n))
        expresion = eval(l1[0]+l2[0]+l1[1]+l2[1]+l1[2]+l2[2]+l1[3]+l2[3]+l1[4])
    return expresion

def valores():
    lista1=[]
    for i in range(100):
        expresion = int(aleatorio())
        if expresion not in lista1:
            lista1.append(expresion)
    lista1 = sorted(lista1)
    return lista1

@calcular_tiempo
def val(a):
    lista=valores()
    print('Valores no repetidos',lista)
    minimo=lista[0]
```

```
    maximo=lista[-1]
    distancia=len(lista)
    print('máximo:',maximo,'mínimo:',minimo,'distancia(inclusive):',distancia)
val(2)
```

```
Valores no repetidos [-63, -60, -49, -42, -36, -31, -29, -26, -25, -18, -17, -11, -10, -9, -5, -
4, -3, -2, 0, 1, 3, 4, 5, 6, 7, 8, 9, 11, 12, 14, 15, 16, 17, 18, 19, 20, 22, 23, 24, 25, 26, 2
7, 28, 29, 31, 33, 36, 38, 40, 41, 45, 47, 49, 52, 53, 55, 57, 74, 75]
máximo: 75 mínimo: -63 distancia(inclusive): 59
Tiempo de ejecución para algoritmo: 0.01564621925354004
```

Enumera las referencias que has utilizado(si ha sido necesario) para llevar a cabo el trabajo

https://docs.python.org/3/library/itertools.html
https://es.wikipedia.org/wiki/Variaci%C3%B3n_(combinatoria)
https://python-para-impacientes.blogspot.com/2015/09/el-modulo-random.html
https://blog.elcodiguero.com/python/eliminar-objetos-repetidos-de-una-lista.html
Diapositivas de la asignatura: VIU-03MAIR-Sesion 05- VC3.pdf
Diapositivas de la asignatura: VIU-03MAIR-Sesion 07- VC4.pdf

Describe brevemente las lineas de como crees que es posible avanzar en el estudio del problema. Ten en cuenta incluso posibles variaciones del problema y/o variaciones al alza del tamaño

Sería interesante agregar la operación al cuadrado o al cubo. También que no sean dígitos los números, que sean números de dos cifras o más. Ya no sería un problema O(n^2).
Cabe mencionar que tuve problemas con la recursividad, implementé muchas funciones recursivas, pero el error típico que salía era "RecursionError: maximum recursion depth exceeded while calling a Python object" o algo semejante y claramente las recursiones eran demasiadas y no se podía implementar "dividir y vencer", además tuve dificultades con permutations de itertools, no había forma de hacerlas recursivas, ya que era una función (la que yo creaba) más la función de caja negra digamos, permutations, con un for funcionaban bien dentro del programa principal.
Por lo que mencionaba eran demasiadas permutations. Más arriba se menciona la cantidad de repeticiones o factoriales que se ejecutaron. Por eso pienso que si el problema se quiere simplificar hay que disminuir los factoriales y si se quiere complicar que se tome en cuenta lo que dije al principio, pero con menos repeticiones para que las funciones se puedan ejecutar.

In [ ]: