

A labb

Gustav Jenner gjenner@kth.se

Maj 2022

1 Uppgiftsbeskrivning

Denna uppgift gick ut på att lösa ett tilldelat problem. Det problem som löstes i denna rapport är kattis problemet vid namn "Equilibrium Mobile" som gick ut på att med minst antal förändringar finna jämvikter. I rapporten beskrivs de datastrukturer och algoritm som används för att lösa uppgiften. Dessutom analyseras tidskomplexiteten på lösningen. Det finns även två Appendix för koden för lösningen samt vilka tester som används för att titta så lösningen stämmer.

2 Kattis

ID	DATE	PROBLEM	STATUS	CPU	LANG
TEST CASES					
8956327	10:46:14	Equilibrium Mobile	✓ Accepted	0.24 s	Python 3
✓✓					

Figure 1: Kattis inlämning

3 Datastrukturer

De datastruktur som användes i lösningen var endast ett "dictionary/hashtable".

4 Algorithm

4.1 Observationer

För att lösa uppgiften behöver man komma fram till två faktum. Första faktumet är att alla noder vid samma djup måste vara samma vikt för att finna total jämvikt. Där djupet är antalet hakparenteser som vikten är i. Andra faktumet är att vikten för ett djup en nivå över måste vara dubbla vikten jämfört med vikten under. Genom dessa två observationer kan man komma fram till en effektiv algoritm för att lösa uppgiften. Vad detta innebär är att ett jämvikt träd kommer alltid innebära att alla vikter är en specifik vikt multiplicerat med multiplar av 2 baserat vilket djup de olika vikterna är.

4.2 Lösning

Den inmatade strängen är gjord på ett sådant sätt att varje "["-tecken innebär att man går ner ett djup och varje "]" innebär att man går upp ett djup. Djupet startar vid 0 och referas som "D". Algoritmen fungerar att läsa strängen från index 0 och uppdatera djupet när en ny hakparentes möts. När en "vikt" träffas på så läggs den till i ett dictionary/hashtable med nyckeln enligt ekvationen 1 och ett värde på 1. Om en vikt träffas på som genererar samma nyckel som tidigare sätts värdet till +1. Den nyckel som haft mest krockar hålls alltid reda på. Dessutom finns det en variabel som ökar varje gång en ny vikt hittas, detta är för att hålla koll på totala antalet vikter. När hela strängen har gått igenom subtraheras totala antalet vikter med antalet krockar som den vanligaste nyckeln hade.

$$Nyckel = vikt \times 2^D \quad (1)$$

4.3 Exempel

Om vi tittar på exemplet som ges i kattis strängen "[[3,7],6]" och använder den tidigare beskrivna algoritmen. Första vikten som stöttes på kommer vara "3" och vid det laget är $D = 2$ därför blir nyckeln $3 * 2^2 = 12$ (se ekvation ??). Största antalet krockar blir då 1 och totala antalet vikter ökar till 1. Efter det läggs "7" in som får nyckeln 28 och totala antalet vikter ökar med +1. Sista vikten är "6" och här är djupet istället 1 då ett "]"-tecken har lästs in; här blir nyckeln då $6 * 2^1 = 12$. Eftersom både "3" och "6" gav en nyckel på 12 är största antalet krockar 2. Det finns 3 vikter, $3 - 2 = 1$, vilket är svaret.

5 Tidskomplexitet

5.1 Teoretisk analys

Eftersom uppgiften endast kräver att man returnerar minst antalet förändringar och inte vilka förändringar det är krävs det bara att "vikträdet" gås igenom 1 gång. Vidare är tidskomplexiteten att lägga till och söka i en hashtable av $O(1)$. Om sträng Längden för viktträdet är "L" så blir tidskomplexiteten för lösningen $O(L)$. N är antalet viktträd som ska beräknas och de alla har längden L så blir tidskomplexiteten $O(N \times L)$. Om längderna varierar kan man säga att tidskomplexiteten blir $O(N \times \bar{L})$. Där \bar{L} är den genomsnittliga sträng längden.

5.2 Tester

Nedan är resultatet från tester där timeit modulen utnyttjades för att se tidsåtgången för algoritmen för olika parametrar. Ett förväntat resultat räknades ut baserad på tidigare tidskomplexitet, där resultatet för den förväntade kolumnen baserades på ett referensvärde. Observera timeit parametern "number" sätts lika med 100000. Resultatet verkar stämma bra med de förväntade värdena vilket stärker att tidsanalysen stämmer. Både för samma längder och varierande längder.

Test	N	L	förväntat(s)	resultat(s)
ref	10	29	-	27.52
1	5	29	13.76	14.16
2	10	48	55.04	49.66
3	10	38.5*	36.54	38.28

Table 1: tabell över resultat från tidstester, N syftar på antalet strängar, och L syftar på sträng längd. *Här varierades längden och ett genomsnitt togs fram.

6 Appendix1

```
import sys
#-----Algorithm-----
def expr(string, weightDict):#funktion som går igenom strängen och räknar djupet samt kallar
    string += "*" #slut markering
    index = 0
    while index in range(len(string)-1):
        flag = False #håller koll om loopen är på siffra
        num_string = ""
        if string[index] == "[":
            weightDict["depth"] += 1
        if string[index] == "]":
            weightDict["depth"] -= 1
        while string[index].isnumeric(): #loop för nummer
            flag = True
            num_string += string[index]
            index += 1 #för varje siffra ökas index med +1 (alltså för nummer)
        if not flag: #Om inte en siffra så ökar indexet med +1
            index += 1
        if num_string != "":
            weight(num_string, weightDict)

def weight(num_string, weightDict): #lägger in vikten på rätt plats i dictionaryet
    key = int(num_string) * 2 ** weightDict["depth"] # nyckel = vikt*2^D
    weightDict["tot_num"] += 1
    weightDict.setdefault(key, 0)
    weightDict[key] += 1
    if weightDict[key] >= weightDict["max_freq"]: #håller reda på högsta frekvensen
        weightDict["max_freq"] = weightDict[key]

def reset_dict(): #funktionen skapar och retunerar ett nytt dictionary med startvärden
    weightDict = {"max_freq": 0, #max frekvens
                  "depth": -1, #djup
                  "tot_num": 0 #totalt antal vikter
                  }
    return weightDict

def solve(string): #tar in jämvikts strängen och retunerar svaret
    weightDict = reset_dict() #återställer dictionaryet
    expr(string, weightDict)
    return weightDict["tot_num"] - weightDict["max_freq"]

def main():# main
```

```

n_tests = int(next(sys.stdin))
for _ in range(n_tests):
    string = next(sys.stdin)
    print(solve(string))

#-----Algoritm-----

#-----Tidstest-----
import timeit
def time():# funktion som läser av txt filen "Time_A" och retunerar alla rader i en lista
    line_list = []
    with open("Time_A", "r", encoding="utf-8") as file: #öppnar filen Time_A
        for line in file:
            line_list.append(line)
    return line_list
listA = time() #lista med alla strängar
def testA():#funktion som löser för alla strängar i "listA"
    for line in listA:
        solve(line)

print(timeit.timeit(stmt = lambda: testA(), number = 100000))
#-----Tidstest-----

```

7 Appendix2

```

import unittest
from A_labb import *

class test_Equilibrium(unittest.TestCase):
    def test_Number(self):#Testar om programmet kan läsa in nummer och inte bara siffor
        self.assertEqual(solve("[[8,5],16]",), 1)

    def test_multiple_strings(self):#Testar om programmet kan läsa in flera strängar i rad
        solve("[[1,2],[3,4]]")
        self.assertEqual(solve("[[1,2],[3,4]]"), 3)

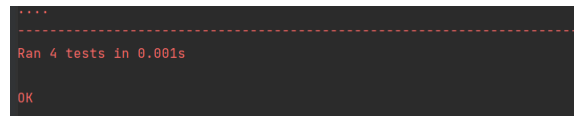
    def test_Solo_Weight(self):#Testar om programmet kan retunera rätt vid ensamma vikter
        self.assertEqual(solve("1"), 0)

    def test_Empty(self):#testar om det blir error när en tom sträng skickas in
        solve("")

if __name__ == "__main__":

```

```
unittest.main(argv=[""])
```

A terminal window with a dark background and red text. It shows the output of a unittest command: four dots, a dashed line, 'Ran 4 tests in 0.001s', and 'OK'.

```
....  
-----  
Ran 4 tests in 0.001s  
  
OK
```

Figure 2: Unittest körning, alla 4 tester klarades