Beginselen van Programmeren: Oefenzitting 11

Intuïtieve complexiteit

Als we de (tijds-)efficiëntie van een algoritme onderzoeken, zijn we vaak geïnteresseerd in het slechtste geval (d.i. worst-case complexity). We analyseren dus de algoritmen op inputs die de algoritmen het slechtst doen presteren. We bekijken dan hoe deze slechtste-geval uitvoeringstijd groeit in functie van de grootte van de input.

De code *test_suite.py* die op Toledo staat, zullen we gebruiken om de prestaties van algoritmen te plotten. In deze file hoef je geen code aan te passen.

1. Bij binair zoeken verwachten we een tijdscomplexiteit die logaritmisch verloopt: O(log(n)). Op Toledo vind je het bestand *binair.py* terug dat een implementatie van lineair zoeken en een naïef binair zoekalgoritme *binairZoekenNaief* bevat. De functie *binairZoekenEfficient* is nog niet geïmplementeerd. De functie *setup* is een functie die een input genereert van gegeven lengte die het slechtste geval in een zoekalgoritme naar boven brengt. Het slechtste geval hier is gedefinieerd als het zoeken naar een getal (getal 1) in een lijst waar dat getal niet voorkomt (een lijst met enkel nullen).
    a. Run *binair.py* en observeer dat het algoritme *binairZoekenNaief* een andere tijdscomplexiteit heeft dan verwacht. Welke complexiteit heeft het wel? Vergelijk de resultaten met de functie *linearZoeken*.

    b. Identificeer het probleem en implementeer dan *binairZoekenEfficient* (denk even terug aan een oefening uit een voorgaande les) die wel de verwachte tijdscomplexiteit heeft.

    c. (**UOVT**) Implementeer een niet-recursieve versie van binair zoeken (m.a.w. een iteratieve versie).

2. In het bestand *sorteren.py* (Toledo) kan je drie geïmplementeerde sorteeralgoritmen terugvinden: *selectionSort*, *quickSort* en *mergeSort*. Ook vind je *setupRandom* terug die een willekeurige lijst van reële getallen in [0, 1] genereert. Gelijkwaardig aan vorige oefening wordt het resultaat van deze setup aan de sorteeralgoritmen doorgegeven als ze worden getest. Omdat de te sorteren lijst willekeurig is, worden de resultaten de gemiddelde-gevalscomplexiteit (average case complexity) genoemd.

    a. Run *sorteren.py* en beschouw de resulaten. Kan je zien dat *selectionSort* gemiddeld $O(n^2)$ is en *mergeSort* gemiddeld $O(n * \log(n))$ is? Wat is de gemiddelde complexiteit van *quickSort*?

b. Bestudeer de implementatie van *quickSort* en implementeer *setupSlechtsteGeval* zodat *quickSort* zoveel mogelijk werk verricht (worst-case complexity analysis). Hoe ziet de plot er nu uit? Wat is dus in het slechtste geval de complexiteit van de drie algoritmen?

3. Gegeven is het onderstaande stukje code. Bepaal de complexiteit van de functie *main*. Hoe kan je de complexiteit hier verbeteren?

```
from random import *

def f(lengte):
     s = [0] * lengte
     for i in range(0, len(s)):
          m = randint(0, lengte)
          j = i
          p = j
          while j > 0:
               if m < s[j-1]:
                    s[j] = s[j-1]
                    p = j-1
               j -= 1
          s[p] = m
     print(s)

def main():
     invoer = int(input("Invoer? "))
     f(invoer)

main()
```