Profilering och optimering av programmet Mnemonics

Denna uppgift fokuserar på profilering och optimering av ett existerande program. Använd profileringsverktyget i Netbeans för att identifiera så kallade ”hot spots” i programmet som bör optimeras (och i viss mån även hur), och för att utvärdera optimeringarna.

Uppgift

1. Skriv ett Hello World-program i Java, kompilera och kör det. Fixa eventuella kompileringsfel eller varningar.
2. Kör Hello World-programmet i Netbeans profileringsverktyg och studera resultatet. Hur många objekt skapas under körning av programmet, hur många metodanrop görs, etc.? Studera vad som faktiskt händer i ett Java-program under körning och tänk igenom och försök förstå profileringsverktygets output!
3. Skriv ett enkelt program som räknar ut n:te talet i en Fibonacciserie med hjälp av en rekursiv algoritm som du sparar i `FibRec.java`. Testa programmet för några n med hjälp av `time`. (En enkel mätning av Javas uppstartstid kan f.ö. fås genom att dra av tiden för Hello World.)
4. Kör det rekursiva Fibonacciprogrammet i profileringsverktyget och studera resultatet. Gör en utskrift eller en skärmdump av det; du kommer att behöva gå tillbaka till det senare.
5. Kopiera det rekursiva Fibonacciprogrammet till `FibIter.java` och modifiera det så att det blir iterativt istället för rekursivt. Testa programmet för några n med hjälp av `time` och jämför tiden med det rekursiva programmet. Försök förklara skillnaderna!
6. Kör det iterativa Fibonacciprogrammet i profileringsverktyget och jämför resultatet med det från punkt 5 ovan. Jämför för samma värde på n.
7. Ändra typerna för Fibonaccifunktionen i `FibRec.java` och `FibIter.java` från primitiva datatyper till motsvarande objekttyp (t.ex. från `long` till `Long`) i filerna `FibRecBoxed.java` och `FibIterBoxed.java`. Jämför tiderna och profileringsresultat igen.
8. Börja med att kompilera programmet i paketet `remember` (t.ex. med `java -d classes remember/*.java`) och testkör det. Fixa eventuella kompileringsfel eller varningar.

   Programmet tar som indata två sökvägar till filer, en radorienterad lista av telefonnummer och ett radorienterat ordlexikon. Programmet går igenom listan av telefonnummer och för varje nummer skriver den ut vilka ”minnesord” som kunde bildas utifrån numret. Exempellexikon och nummerlistor medföljer koden.

   En utskrift från det körande programmet kan se ut så här:

   ```
   tobias$ java -cp classes remember.Mnemonics dictionary.txt numbers.txt
   562482: Mix Tor
   562482: mir Tor
   4824: Torf
   4824: fort
   107835: je Bo da
   ```

   Uppgiften går ut på att optimera programmet utan att ändra dess funktioner. Använd profileraren för att ta reda på vilka metoder som tar mest tid att köra, är det några få ”långsamma” metoder, eller många anrop till en ”snabb” metod? Försök att jaga upp programmets hastighet, mät med hjälp av `time`, och se förändringarna i körtid hos programmet med profileraren.

   Glöm inte att ta en kopia på orginalprogrammet så att du enkelt kan mäta dess körtid igen senare!
9. De största prestandavinsterna fås i regel genom att man inser att man kan minska problemrymden för programmet eller hitta ett bättre angreppssätt att lösa problemet. Pröva t.ex. att ändra funktionen `couldMatch` i `Dictionary.java` så här:

   ```
   public Dictionary couldMatch(String match) {
       final HashSet<String> _ = new HashSet<String>();
   ```

```
    for (String word : words) {
        if (word.startsWith(match)) _.add(word);
    }

    return (_.size() > 0) ? new Dictionary(_) : null;
}
```

(OBS! Denna ändring kräver ytterligare småändringar för att fungera.)

Nu minskar sökrymden i programmet för varje rekursivt steg (i det generalla fallet). Hur påverkas programmets körtid? Är vinsten större än den ökade overheaden det innebär att skapa nya lexikon? Använd profileraren för att besvara frågan! (Titta speciellt på minnesåtgången – hur har den förändrats?)