Laboration 1

TDA416/DIT721, lp3 2016

Laborationen består av 2 delar.

1 Del 1 - rekursion

Uppgiften går ut på att skriva ett program som beräknar kvadratroten ur ett tal genom upprepade approximeringar. Om $y = \sqrt{x}$ så gäller ju $x = y^2$ (då $x \geq 0$). Så givet ett tal, x, kan man leta sig fram till kvadratroten ur x genom att prova olika tal, y_i, genom att kvadrera dem och jämföra resultatet med x. Använd detta och intervallhalvering / binärsökning för att implementera kvadratrotberäkning. Alltså, håll koll på ett talintervall (ett y_{min} och ett y_{max}) och prova talet mitt i intervallet, $y_{mitt} = (y_{min} + y_{max})/2$. Man kan inte räkna med att y^2_{mitt} någonsin blir exakt lika med x. Istället nöjer man sig med att skillnaden inte är större än en given feltolerans, ϵ. Är skillnaden större än så fortsätter man approximeringen genom att välja den ena (rätta) intervallhalvan och testa mittenvärdet där o.s.v.. Det finns tre fall ni ska hantera:

- $x < 0$: Reell kvadratrot saknas.
- $0 \leq x \leq 1$: Kvadratroten måste finnas i intervallet $[x, 1]$. Använd detta som startintervall.
- $x \geq 1$: kvadratroten måste finnas i intervallet $[1, x]$. Använd detta som startintervall.

Skriv en klass som har följande namn och metoder:

```
public class MySqrt {
    public static double mySqrtLoop(double x, double epsilon) {..}
    public static double mySqrtRecurse(double x, double epsilon) {..}
    public static void main(String [] args) {..}
}
```

Metoden `mySqrtLoop` ska implementera kvadratrotberäkningen som beskrivs ovan med hjälp av en loop-konstruktion.

Metoden `mySqrtRecurse` ska göra samma sak rekursivt och utan loop. För att göra det kan du implementera en hjälp-metod, som förutom huvudmetodens argument också har som indata två tal som definierar aktuellt intervall (y_{min} och y_{max}).

Argumentet `x` är värdet för vilket kvadratroten ska beräknas och `epsilon` är feltoleransen. Om `x` är icke-negativ ska metoderna returnera ett icke-negativt tal, y, sådant att $|y^2 - \texttt{x}| < \texttt{epsilon}$, d.v.s $y \approx \sqrt{\texttt{x}}$. Om `x` är negativ ska metoderna returnera $Double.NaN$. Metoderna behöver inte hantera fallet då `x` är `NaN`. Ni kan också utgå från att `epsilon` är ett positivt tal.

I `main`-metoden ska ni utföra 5 tester vardera av det två metoderna. Testindata ska variera på ett sånt sätt att det täcker olika fall väl. Feltoleransen kan sättas konstant till 10^{-6}.

Koden ska vara välstrukturerad, lättläst och kommenterad.

2 Del 2 - komplexitetsanalys

Här är 3 metoder som gör samma sak men på lite olika sätt.

```
static private int seqStart = 0;
static private int seqEnd = -1;
/**
* contiguous subsequence sum algorithm.
* seqStart and seqEnd represent the actual best sequence.
* Version 1
*/
public static int maxSubSum1( int[] a ) {
  int maxSum = 0;
  for( int i = 0; i < a.length; i++ )
    for( int j = i; j < a.length; j++ ) {
      int thisSum = 0;
      for( int k = i; k <= j; k++ ) {
        thisSum += a[k];
      }
      if( thisSum > maxSum ) {
      maxSum   = thisSum;
      seqStart = i;
      seqEnd   = j;
    }
  }
  return maxSum;
}

// Version 2
public static int maxSubSum2( int[] a ) {
  int maxSum = 0;
  for( int i = 0; i < a.length; i++ ) {
    int thisSum = 0;
    for( int j = i; j < a.length; j++ ) {
      thisSum += a[j];
      if( thisSum > maxSum ) {
        maxSum = thisSum;
        seqStart = i;
        seqEnd   = j;
      }
    }
  }
  return maxSum;
}
// Version 3
```

```
public static int maxSubSum3( int[] a ) {
  int maxSum  = 0;
  int thisSum = 0;
  for( int i = 0, j = 0; j < a.length; j++ ) {
    thisSum += a[j];
    if( thisSum > maxSum ) {
      maxSum = thisSum;
      seqStart = i;
      seqEnd   = j;
    }
    else if( thisSum < 0 ) {
      i = j + 1;
      thisSum = 0;
    }
  }
  return maxSum;
}
```

Fundera först på vad metoderna gör. Det är troligen lättast att förstå om man tittar på version 1 eller 2. Om man själv skriver dessa algoritmer så är version 1 naturligast att komma på först. Version 2 "inser" man när man förstått version 1 ordentligt. Version 3 är inte helt lätt att komma på.

Uppgiften är att analysera metodernas komplexitet. Analysera alla 3 dels med handviftning och dels med en matematiskt korrekt uppskattning (dvs sätt upp summorna och lös dem). Välj en av metoderna och gör en pedantisk analys. Glöm inte motivera vad ni gör. Se OH om komplexitet för en förklaring av pedantisk analys, matematiskt korrekt uppskattning och handviftning (rough estimate).

Det finns ett testprogram, `MaxSumTest.java`, på kurssidan som du kan använde. Det kör programmen ovan (finns i `MaxSum.java` på kurssidan) och mäter tidsåtgången. Gör en tabell och rita en graf dels med mätta värden och dels med beräknade värden (använd O(.)-resultaten). Rita också ut i graferna kurvor som motsvarar era teoretiska beräkningar av komplexiteten. Skala de teoretiska kurvorna i y-led så att de överlappar de experimentella så bra som möjligt. I `MaxSumTest.java` är fältens storlek satta rätt lågt för att det skall gå snabbt när du kör det första gången. Öka storleken så det blir stabila värden, utan att det tar halva dan att köra förstås.

Vad ska vi skicka in till Fire?

- För del 1: en java-fil, `MySqrt.java`.
- För del 2: en pdf-fil, `komplexitet.pdf`. Detta dokument ska överskådligt och lättläst presentera och motivera de olika (3+3+1) komplexitetsanalyserna, samt för varje metod presentera ett diagram innehållande kurvor för den experimentella och teoretiska tidsåtgången som funktion av indatafältets storlek.

Lösa filer, inga `.zip`, `.tar` eller dyligt.