Data och Informationsteknik / Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Erland Holmström Göteborg 23 jan 2015

Laboration 1 i Datastrukturer IT2 (Kan ge 3+3+3p)

Senaste inl.: Uppg 1+2 i LV2, Uppg 3 LV3 se exakt tid i Fire.

OBS: Ni lämnar in U1+2 i LV2 och sedan ALLA uppgifterna i en tar fil i LV3.

Anvisningar:

Döp filerna till Uppg1.java, Uppg2.pdf och Uppg3.java och tara enligt instruktionerna. Läs också i slutet av uppgifterna om saker att tänka på.

Alla program skall vara "välskrivna" (förhoppningsvis vet ni vad det innebär vid det här laget :-). Naturligtvis skall ni testköra era program och *testexemplen ingår i en "lösning" och skall också lämnas in*. Tala om i kommentarer vad varje test testar.

Som vanligt för laborationer så står alla som gör en laboration för alla delar i laborationen och skall kunna redogöra för varje del. När ni gör en laboration får ni naturligtvis samarbeta och diskutera hur man skall lösa en uppgift MEN när ni lämnar in laborationen är varje person personligen ansvarig för varje del i laborationen. Alla i labgruppen skall självständigt kunna redogöra för varje lösning. Det duger inte att säga att ni delat upp uppgifterna mellan er!

Om det står tex "40R" vid en uppgift så betyder det att programmet, inklusive eventuell main, kan skrivas på ca 40 rader. Det är ingen tävling men om du har många fler rader i ditt program så är det troligen inte välskrivet (och det är ju ett skäl till retur). Observera dock att jag undviker parenteser ensamma på en rad tex i början av metoder (men inte i slutet)

Inlämning:

Uppg 1 och 3: Java klasserna i .java filer (dvs textfiler) döpta enligt Uppg1.java, Uppg3.java.

Uppg 2: En pdf fil med analys och resultat döpt till Uppg2.pdf.

Uppg 1+2 lämnas in tillsammans som en tar-fil i LV2.

Uppg 1+2+3 lämnas in tillsammans som en tar-fil i LV3. Filerna med U1+2 skall vara identisk med den ni lämnade in i LV2.

Uppgift 0:

Förövning: Registrera er i Fire. Läs instruktionerna på hemsidan även om ni registrerat er i Fire förut! Läser och följer ni instruktioner dåligt så förlorar ni onödiga poäng på labbarna. (Många av ”reglerna” handlar om att underlätta rättningen och förmågan att ge bra handledning.)

Tips: Några av de strukturella saker vi kommer att anmärka på:

Allmänt:

- När ni registrerar er i Fire så läs instruktionerna ordentligt för hur man lämnar in på hemsidan. Det är inte säkert att det är likadant som på andra kurser.
- Tänk på att skriva era personummer som YYMMDD-XXXX
- Namn och grupp skall finnas först i ALLA filer, även källtext.
- Använd helst max 80-100 tecken på en rad i filerna.

Inga onödiga filer får skickas in tex tilde filer eller .class file

- zip fungerar dåligt med fire, det är tar som gäller och det skall göras precis som det står.

Tänk på att skapa rätt biblioteksstruktur. Annars kan inte våra testprogram hitta rätt i era filer.

- Ni måste ta bort eventuella paketdeklarationer som eclips skapar och lägger in först i era filer.

Uppgift 1:

Övar enkel rekursion. Ca 45R (Om du vill mjukstarta med rekursion föreslår jag att du börjar med tex binomialkoefficienterna, se övningslapp 1) Detta är repetition från ettan.

From Wikipedia:

”In mathematics, a square root of a number x is a number y such that $y^2 = x$, or, in other words, a number y whose square (the result of multiplying the number by itself, or $y \times y$) is x. For example, 4 is a square root of 16 because $4^2 = 16$.

Every non-negative real number x has a unique non-negative square root, called the principal square root, which is denoted by $\sqrt{x}$, where $\sqrt{}$ is called radical sign. For example, the principal square root of 9 is 3, denoted $\sqrt{9} = 3$, because $3^2 = 3 \times 3 = 9$ and 3 is non-negative.”

Roten ur ett positivt tal ligger alltså mellan 0 och talet självt. Antag nu att talet är >= 1 för enkelhets skull (och roten mellan 1 och talet), vi skall ju bara öva rekursion. Man kan beräkna roten ur ett tal>=1 med godtycklig noggrannhet genom att använda intervallhalvering (binärsökning).

Man börjar med att kvadrera mittpunkten mellan 1 och talet och ser om resultatet är större eller mindre än talet. Då vet vi i vilken halva vi skall fortsätta söka. Exempel: om vi skall beräkna $\sqrt{2}$ så kvadrerar vi 1.5 och ser att det blir för stort. Alltså skall vi fortsätta söka i intervallet 1..1.5.

Du skall nu skriva en funktion (en funktion = en Java metod)

```
static double binarySqrt(double sqr, double eps)
```

som beräknar roten ur det givna talet sqr med noggrannheten eps (låt eps vara 10^{-6}).

BinarySqrt's uppgift är bara att göra rimlighetskontroll av indata (en wrapper funktion) och anropa den interna *rekursiva* hjälpfunktionen (som du också skriver)

```
static double help(double sqr, double eps, double low, double high)
```

som förutom sqr och eps även innehåller nedre och övre gräns för sökintervallet och som gör det mesta av jobbet. Jämför din funktions resultat genom att anropa den inbyggda funktionen Math.sqrt.

(Det är ju fusk att använda Math.sqrt för att kontrollera om det fungerar men vi kan leva med det här, vi övar ju på rekursion.)

Vilka testfall behövs för att övertyga om att metoden fungerar (även om inte Math.sqrt fanns)? Tänk på att den skall klara alla riktigt typade argument (tex sqrt(-2, 1000)) på något vettigt sätt.

Note om exeptions: Exceptions är kanonbra men man måste vara noga med när dom är lämpliga. Om man tex inte vet vad man skall göra vid en felsituation så är oftast en exception enda möjligheten. Men man skall normalt använda dom för exceptionella händelser som är svåra att testa på och svåra att hantera. Man skall också tänka igenom vilken exception man skall fånga och var det är lämpligt att göra det. Att ha en ”catch (Exception e)” är *aldrig* lämpligt (utom möjligen för halvfärdiga program). Try satsen i en exception skall dessutom omsluta så lite kod som möjligt.

Tips: Några av de saker vi kommer att anmärka på:

- Vilka av metoderna ovan skall vara publika/privata?

- slutvillkoret ”if ((high - low) > eps) {...” kan ge långsam konvergens, skriv om.

- ”if((x - y) > 0) {...” ni menar if(x > y) och det är bättre att skriva det då.
(Det är annorlunda om man jämför på likhet (se nedan) generellt är likhet mellan flyttal ingen bra test, men här är det olikhet).

- ”if(something==0) {...” det inträffar i princip aldrig, i alla fall har du då rekurserat alldeles för länge. (här bör man använda tekniken ovan, att subtrahera och jämföra med noll.)

Några saker att tänka på:

- *Felhantering är svårt*! Tänk igenom alternativen. Felhantering typ utskrift och sen fortsätta som vanligt är aldrig bra.
- *Exceptions* skall användas med försiktighet och kunskap. Fånga aldrig ”Exception” utan det du fångar skall vara så långt ner i exception hierarkin som möjligt. Man kastar och fångar aldrig på samma ställe. `IndexOutOfBoundsException` eller `NoSuchElementException` kan vara lämplig att kasta för många fel här.
- Tänk också på att ”try-catch” skall omsluta så lite kod som möjligt, inte hela programmet. Och att det inte är troligt att du skall ha några try-catch block i din klass (men väl i testklassen eller main om du lägger den i samma klass).
- Kod skall vara logisk, strukturerad och vacker! Tänk på det.

  Och skall inte avslöja att man har bristande förståelse. Metoden empty kan tex skrivas så här: (med risk för att den här metoden är för kort för att poängen skall framgå)

```
private boolean empty() {
  if (size == 0)
    return true;
  return false;
}
```

här är man otydlig med strukturen och riskerar att indenteringen avgör hur läsaren tyder koden. Det är också risker med att inte använda block ({}) om man behöver lägga till satser. Dessutom verkar man inte ha förstått hur booleaner fungerar. Man skulle kunna fixa strukturen genom att skriva vad man egentligen menar så här:

```
private boolean empty() {
  if (size == 0) {
    return true;
  } else {
    return false;
  }
}
```

Här är strukturen tydlig, man ser att det är ”aningen eller” och lägger man till satser så är det blocken som avgör hur det tolkas inte indenteringen. Men har man förstått booleaner? Så här bör metoden skrivas:

```
private boolean empty() {
  return (size == 0);
}
```

Uppgift 2:

Övar *komplexitetsberäkningar*.
Vi går igenom komplexitet på föreläsning 3. Vänta med denna till efter det.
Här är 3 metoder som gör samma sak men på lite olika sätt.

```
public final class MaxSumTest {
  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;
    }
  ...
```

a) Vad gör dom? 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. Version2 ”inser” man när man förstått version 1 ordentligt. Version 3 är inte helt lätt att komma på.

b) Analysera deras komplexitet. Analysera alla 3 dels med handviftning och dels med en matematiskt korrekt uppskattning (dvs sätt upp summorna och lös dem, motivera vad du gör).

   Välj *en* av dem och gör en pedantisk analys för övnings skull.

   Glöm inte motivera vad du gör. Största bristen med studentsvar är avsaknaden av motiveringar till vad ni gör. Det matematiska underlaget för dina beräkningar är bra att redovisa här.
   (se OH om komplexitet för en förklaring av pedantisk analys, matematiskt korrekt uppskattning och handviftning (rough estimate))

c) Det finns ett testprogram, `MaxSumTest.java`, på hemsidan som du kan köra som kör programmen ovan (finns i `MaxSum.java`) och mäter tidsåtgången.

   Gör en tabell och rita en graf dels med mätta värden och dels med beräknade värden (använd O(.) resultaten).

   Vad förväntar ni er att få för relationer? Ge ett enkelt matematisk samband?

   Jämför nu med dina teoretiska resultat. Hur väl stämmer verkligheten med teorin?

   (Jag (och två av 2011 års studenter, Ludwig Kjellström & Daniel Olausson) har skrivit programmet men du måste sätta dig in i hur det fungerar. Fältens storlek är satta rätt lågt för att det skall gå snabbt när du kör det första gången *men du skall öka storleken så du får en bra tabell* utan att det tar halva dan att köra).

Uppgift 3:

se separat fil