Lösningsförslag

Rekursion

U 1. a)
```
public static double power(double x, int n) {
      if (n == 0) {
          return 1;
      } else {
          return x * power(x, n - 1);
      }
  }
```

b)
```
public static double power(double x, int n) {
      if (n == 0) {
          return 1;
      } else {
          double r = power(x, n / 2);
          if (n % 2 == 0) {
              return r * r;
          } else {
              return x * r * r;
          }
      }
  }
```

Anm. Se till att `power(x, n / 2)` bara anropas en gång och att resultatet sparas i en lokal variabel som sedan används i multiplikationen. Gör man samma anrop flera gånger har man inte vunnit något.

c) Ändra namnet på metoden från deluppgift **??** till `recPower` och lägg till följande metod:

```
public static double power(double x, int n) {
    if (n < 0 ) {
        return 1.0/recPower(x, -n);
    } else {
        return recPower(x, n);
    }
}
```

U 2. 48412484

U 3. Ett basfall är ett enkelt fall av problemet som man kan lösa direkt utan att göra något rekursivt anrop. Om basfall saknas kommer det att göras rekursiva anrop i ”all oändlighet” (eller i praktiken tills minnet tar slut).

U 4.
```
/** Returnerar true om strängen s är en palindrom. */
public static boolean isPalindrome(String s) {
    return recPalindrome(s, 0, s.length() - 1);
}

private static boolean recPalindrome(String s, int first, int last) {
    if (last <= first) {
        return true;
```

```
        } else if (s.charAt(first) != s.charAt(last)) {
            return false;
        } else {
            return recPalindrome(s, first + 1, last - 1);
        }
    }
```

U 5.
```
    /** Beräknar hur mycket kapitalet capital vuxit efter years år om räntesatsen
        är interestRate. */
    public static double computeCapital(double capital, int years, double interestRate) {
        if (years == 0) {
            return capital;
        } else {
            return computeCapital(capital + (capital * interestRate / 100), years - 1,
                                                                        interestRate);
        }
    }
```

U 6. a)
```
    /** Beräkna nollstället för funktionen f i intervallet [low,high] med preci-
        sionen eps. Förutsätter att det finns ett nollställe i intervallet. */
    public static double getZero(double low, double high, double eps, Function f) {
        double mid = (low + high)/2;
        if (Math.abs((high - low)) < eps) {
            return mid;
        } else {
            if (f.evaluate(mid) * f.evaluate(low) > 0) {  // samma tecken
                return getZero(mid, high, eps, f);
            }  else {
                return getZero(low, mid, eps, f);
            }
        }
    }
```

b)
```
    public class MyFunction implements Function {
        public double evaluate(double x) {
            return Math.exp(-x) - 1 + Math.cos(x);
        }
    }
```

c)
```
    double zero = Bisection.getZero(0, 1.6, 0.0001, new MyFunction());
```

Deluppgift b och c kan lösas på ett enklare sätt. Som sista argument till metoden `getZero` skickar man med referensen till ett objekt av någon klass (här `MyFunction`) som implementerar interfacet `Function`. Sedan kan man anropa `evaluate` inuti metoden `getZero`. Det känns onödigt att skapa och skicka med referensen till ett objekt när det i själva verket bara är funktionen `evaluate` eller rent av `Math.exp(-x) - 1 + Math.cos(x)` man vill skicka med.

Fr.o.m Java 8 kan man lösa det med lambdauttryck (inte så krångligt som det låter!). Interfacet `Funktion` har bara en enda abstrakt metod (`evaluate`). Sådana interface kallas funktionella interface. Om en parameter har en typ som är ett funktionellt interface kan vi som argument skicka med ett lambdauttryck:

```
double zero = Bisection.getZero(0, 1.6, 0.0001,  x -> Math.exp(-x) - 1 + Math.cos(x));
```

`x` motsvarar parametern till `evaluate` och efter `->` finns det uttryck som beräknas i funktionen. Kompilatorn ”fattar” att det är metoden `evaluate` det handlar om eftersom den är den enda abstrakta metoden i interfacet. Vid exekveringen kommer det att skapas ett anonymt objekt av rätt typ vars referens skickas med som argument till `getZero`. Men vi behöver inte själva skriva klassen som implementerar `Funktion`.

Inte heller interfacet Function behöver vi skriva. Det finns ett färdigt interface i Java: `Function<T,R>`, där `T` ska ersättas av typen för funktionens parameter och `R` med resultatets typ. Metoden `getZero` kan då skrivas så här:

```
/** Beräkna nollstället för funktionen f i intervallet [low,high] med preci-
    sionen eps. Förutsätter att det finns ett nollställe i intervallet. */
public static double getZero(double low, double high, double eps,
                                                    Function<Double, Double> f) {
    double mid = (low + high)/2;
    if (Math.abs((high - low)) < eps) {
        return mid;
    } else {
        if (f.apply(mid) * f.apply(low) > 0) {  // samma tecken
            return getZero(mid, high, eps, f);
        }  else {
            return getZero(low, mid, eps, f);
        }
    }
}
```

Metoden `apply` beräknar funktionen med det angivna argumentet.

U 7. Felet är att vi i rekursionen går ända fram till `null`. Det sista anropet av den rekursiva metoden har `node = null`. Inuti metoden får `node` ett nytt värde (en nod med innehållet `x`). Men i den anropande metoden (vars `node.next == null`) blir det ingen förändring. Den nya noden länkas alltså aldrig in i listan, som förblir tom.

Ett sätt att korrigera felet är att i rekursionen stanna ett steg tidigare enligt följande:

```
public void add(E x) {
    if (first == null) {
        first = new ListNode<E>(x);
    } else {
        add(first, x);
    }
}
private void add(ListNode<E> node, E x) {
    if (node.next == null) {
        node.next = new ListNode<E>(x);
    } else {
        add(node.next, x);
    }
}
```

U 8.
a) Denna metod är korrekt och väl utformad. Har ett basfall.
b) Metoden är korrekt, men har ett onödigt basfall (n==1).
c) Metoden är inte korrekt. Basfallet är fel. Funktionen kommer att hamna i evig loop om den anropas för n=0, då svaret borde bli 1.
d) Helt fel. Saknar basfall. Hamnar i evig loop vilket värde vi än anropar med.

U 9. Förslaget a) är korrekt. Förslaget b) klarar ej tom lista. Dessutom saknar b) fallet att man inte hittar elementet. Om b) anropas för ett fall där elementet ej finns i listan kommer det att ge ett exekveringsfel. Det kommer då att göras rekursiva anrop tills den lista som undersöks är tom, varvid exekveringsfelet kommer.

U 10. Värsta fall inträffar om vektorn är sorterad i avtagande ordning. Då kommer jämförelsen `v[n-1] > findMax(v, n-1)` att bli `false` i varje rekursiv upplaga och därför kommer det rekursiva anropet `findMax(v, n-1)` i den sista `else`-grenen alltid att utföras. Vi får då två rekursiva anrop, ett i jämförelsen och ett i den sista `else`-grenen. `findMax(v, n)` kommer alltså att göra två anrop av `findMax(v, n-1)`. Var och en av `findMax(v, n-1)` kommer att göra två anrop av `findMax(v, n-2)`, d.v.s. totalt fyra anrop av denna. Det blir totalt $1 + 2 + 4 + ...2^i + ...2^n$ rekursiva anrop d.v.s. $O(2^n)$.

Man gör lösningen mycket effektivare genom att undvika två rekursiva anrop:

```
public static int findMax(int[] v, int n) {
    if (n==1) {
        return v[0];
    } else {
        int  temp = findMax(v, n-1);
        if (v[n-1] > temp) {
            return v[n-1];
        } else {
            return temp;
        }
    }
}
```

Nu blir det bara ett rekursivt anrop i varje upplaga. Antalet upplagor är n och varje upplaga gör, förutom det rekursiva anropet, ett konstant arbete. Totalt blir det $O(n)$.

U 11. Dynamisk programmering innebär att man i en tabell håller reda på vilka instanser av problemet man redan löst. När man behöver lösningen till en viss instans kontrollerar man först i tabellen om den redan beräknats. I så fall hämtas lösningen där, dvs man gör inget rekursivt anrop.

U 12. Den andra är både enklare att läsa och effektivare. Den första lösningen får tidskomplexitet $O(n^2)$ och den andra $O(n)$.