Övningar i rekursion, stackar, köer och abstrakta datatyper

Förstå koncept:

1) Vad är rekursion?

2) Vad innebär divide-and-Conquer?

3) Hur fungerar fibonaccisekvensen? (Själva talsekvensen, inte någon algoritmisk implementation för att beräkna fibonaccinummer!)

4) Hur fungerar stackminnet i Java?

5) Varför är en rekursiv fibonaccialgoritm så beräkningstung? Vad har den för tidskomplexitet?

6) När man anropar en rekursiv fibonaccimetod med höga nummer kommer resultatet plötsligt att börja bli negativt. Varför?

7) Vad är tidskomplexiteten för en binär sökalgoritm? Varför?

8) Vad är skillnaden mellan O(n*logn), O(logn) och O(n)?

9) Vad är en naiv algoritm för någonting?

10) När vi deklarerar en int, var någonstans i minnet sparas den?

11) Beskriv vad som händer när vi instansierar ett objekt och sen skickar in det som argument i en metod. Var sparas objektet? Vad skickas till metoden?

12) Vad är dynamisk programming för någonting? Varför är det bra att känna till?

13) Vad menas med LIFO? Vilken datastruktur är det förknippat med?

14) Vad är skillnaden mellan att skicka någonting som värde och att skicka någonting som referens?

15) Vad innebär primitiv rekursion?

16) Vad är det för skillnad mellan en stack och en kö?

17) Vad är en ADT för något?

18) Ge två exempel på ADT:er.

19) Vad är en generisk typ (Generics) för någonting? Varför är det bra att känna till?

20) Vad är kontrollflöden?

Förstå kod:

1) Studera följande algoritmer:

a)

```java
public static void main (String[] args)
{
    int n = 10;
    int sum = 0;

    for(int i = 0; i < n; i++)
        sum++;

    for(int i = 0; i < n; i++)
        sum++;

    for(int i = 0; i < n; i++)
        sum++;

    System.out.println(sum);
}
```

Vad blir sum när den skrivs ut?
Vilken tidskomplexitet får koden?

b)

```java
public static void main (String[] args)
{
    int n = 10;
    int sum = 0;

    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < n; j++)
        {
            for(int k = 0; k < n; k++)
            {
                sum++;
            }
        }
    }

    System.out.println(sum);
}
```

Vad blir sum när den skrivs ut?
Vilken tidskomplexitet får koden?

c)

```java
public class LoopExample
{
    static int n = 10;
    static int sum = 0;

    public static void main (String[] args)
    {
        increment(n);
        increment(n);
        increment(n);
        System.out.println(sum);
    }

    public static int increment(int n)
    {
        if (n == 0) return n;
        sum++;
        return increment( n: n-1);
    }
}
```

Vad blir sum när den skrivs ut?
Vilken tidskomplexitet får koden i main()?
Vilken tidskomplexitet får koden i increment()?
Förklara steg för steg vad increment() gör.

d)

```java
public class LoopExample
{
    static int n = 10;
    static int sum = 0;

    public static void main (String[] args)
    {
        for(int i = 0; i < n; i++)
        {
            for(int j = 0; j < n; j++)
            {
                increment(n);
            }
        }
        System.out.println(sum);
    }

    public static int increment(int n)
    {
        if (n == 0) return n;
        sum++;
        return increment( n: n-1);
    }
}
```

Vad blir sum när den skrivs ut?
Vad blir tidskomplexiteten för main()?
Förklara steg för steg vad som händer i main()-metoden.

2) En rekursiv fibonaccimetod ser ut så här:

```
public static long fib(int n)
{
   if (n <= 1) return n;
   return fib(n-1) + fib(n-2);
}
```

Ta ett papper och rita upp metodanropen som genereras om den här metoden anropas med `fib(5)`, samt vad vart och ett av dem returnerar. (Se till att ha gott om plats!)

3) Öppna programmet `RecursiveFibonacciStackCounter` och testa att köra det ett par gånger med olika nummer. Bekanta er med webbsidan det genererar. Fundera över vad som händer med stacken vid de olika operationerna.

a) Den första kolumnen visar vad som händer i programmet rad för rad. Den andra kolumnen visar i vilken ordning anropen pushas på och poppas från stacken. Den tredje kolumnen visar i vilken ordning anropen skapas. Varför är de två sista kolumnerna inte i samma ordning? Om vi kör programmet och beräknar det 6:e fibonaccitalet, varför får vi då t.ex. en rad som säger att stacken vid något tillfälle innehåller anropen `A, B, C, I`?

b) Vilka stackoperationer användes för att beräkna `fib(n-1)` om stackhistoriken ser ut så här?

```
A
A, B
A, B, C
A, B, C, D
A, B, C
A, B, C, E
A, B, C
A, B
A, B, F
A, B
A
A, G
A, G, H
A, G
A, G, I
A, G
A, G
A
```

c) Anta att en rekursiv fibonacci genererar följande anropsträd:

```
A: fib(5)
   B: fib(4)
      C: fib(3)
         D: fib(2)
            E: fib(1)
            F: fib(0)
         G: fib(1)
      H: fib(2)
         I: fib(1)
         J: fib(0)
   K: fib(3)
      L: fib(2)
         M: fib(1)
         N: fib(0)
      O: fib(1)
```

Om den senaste stackoperationen vi gjorde var att poppa `D`, vad innehåller i så fall stacken nu? Vad kommer nästa instruktion att vara? Vad innehåller stacken efter den? Vilket värde kommer `C` så småningom att returnera till `B`?

d) Kör programmet några gånger med olika värden. Notera hur den andra anropskedjan, som används för att beräkna `fib(n-2)`, alltid blir kortare än den första (utom vid de allra minsta värdena för `n`). Varför är det så?

e) Innebär det här att tidskomplexiteten verkligen är $O(2^n)$ för algoritmen, eller är den egentligen något annat? Motivera! Varför tror du att vi trots det betecknar den som $O(2^n)$ i så fall?

(Frågan ovan är överkurs och inget ni förväntas kunna, men fundera gärna på den en stund och kolla facit för svar! Den knyter an till vad vi diskuterade i början av första föreläsningen!)

4) Fakulteter (engelska: `factorial`) är en talföljd där varje tal, `n`, är lika med *produkten av alla tal från 1 upp till och med n självt*.

Exempel: $5_{fak} = 1 * 2 * 3 * 4 * 5 = 120$
Exempel: $7_{fak} = 1 * 2 * 3 * 4 * 5 * 6 * 7 = 5040$

Man kan beräkna fakulteter rekursivt så här:

```
public long factorial(int n)
{
   if (n <= 1) return 1;
```

```
   return n * factorial(n-1);
}
```

a) Koden är extremt lik den rekursiva fibonaccialgoritmen i uppgift 2. Finns det någon skillnad? Motivera!

b) Vilken tidskomplexitet får koden?

c) Finns det något sätt att optimera den här koden på? Varför/varför inte?