Lösningsförslag: Övningar i rekursion, stackar, köer och abstrakta datatyper

Förstå koncept:

1) Rekursion inom programmering är en metod som anropar sig själv. Mer formellt är det någonting som definieras i termer av sig självt.

2) I fibonaccisekvensen är varje nytt tal summan av de två föregående. Formeln är: Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2), där n är det tal i sekvensen som vi vill beräkna.

3) Stacken är det minne i programmet där primitiva datatyper och activation frames för metodanrop sparas. Den fungerar som ungefär som en stapel: man lägger på saker och när man vill ta bort dem igen måste man börja med det man sist la på. Det är därför man säger att den fungerar enligt minnesregeln LIFO: Last In, First Out.

4) Den är beräkningstung eftersom varje nytt rekursivt anrop producerar två nya anrop i gengäld. Varje gång n ökar med 1 fördubblas tiden. Man säger att den har exponentiell tillväxt, med tidskomplexiteten $O(2^n)$.

5) Programmet kommer att börja skriva ut negativa värden när fibonaccialgoritmen anropas med tal större än 161 eftersom storleken på dessa tal kommer att överskrida kapaciteten för vad man kan lagra i en long, som är den största primitiva datatypen i Java.

En signed long (alla datatyper är signed i Java) som representerar positiva tal kan hålla 63 bit. När talet blir större än så får vi en overflow som aktiverar den 64:e biten, som är signbiten. Om signbiten blir 1 får vi en negativ representation av det binära numret i stället.

6) En binär sökalgoritm är ett exempel på divide-and-conquer och har tidskomplexiteten O(logn). Varje gång den anropar sig själv halveras datamängden eftersom vi kan utesluta halva listan (vi vet ju om talet är större eller mindre än mittvärdet).

7) Tidskomplexiteten för binär sökalgoritm är O(logn), eftersom den **halverar** datamängden för varje nytt metodanrop.

8)
O(logn) : Mängden n halveras för varje anrop.
O(n*logn) : Tiden ökar linjärt med antalet n, multiplicerat med logn.
O(n): Tiden ökar linjärt: när antalet n dubblas dubblas också körtiden.

9) En naiv algoritm är den enklaste lösningen på ett problem, men väldigt sällan den bästa.

10) En int sparas på stacken i Java.

11) När vi instansierar ett objekt sparas det i heapminnet. En referens till objektet sparas på stacken, och det är referensen som skickas med som argument i parametrarna till en metod. Det är därför vi t.ex. kan skapa en void-metod som tar in en array och sorterar den utan att sedan returnera en array igen: både metoden och kodblocket där arrayen deklarerades håller samma referens.

12) Dynamisk programmering är en optimeringsteknik där man kan använda antingen en top-down-approach (memoisering) eller en bottom-up-approach (tabulering) för att optimisera rekursiva och iterativa algoritmer.

13) LIFO står för Last In, First Out och beskriver hur en stack fungerar.

14) När vi skickar någonting som värde till en metod kopieras värdet. Det innebär att om du skickar en int-variabel till en metod och ändrar värdet på den inuti metoden så kommer inte ursprungsvariabeln att ändra värde. Alla primitiva datatyper skickas automatiskt som värde i Java.

När vi däremot skickar någonting som referens skickar vi, lite förenklat, en pekare till någonting som ligger lagrat på heapen. Eftersom referensen pekar mot *samma* minnesblock som ursprungsvariabeln kan vi manipulera det här minnet direkt. Alla objekt skickas automatiskt som referenser i Java.

15) Primitiv rekursion innebär att det går att skriva om algoritmen så att rekursiva anropen ersätts med for-loopar i stället.

16) En stack är LIFO (Last in, first out) medan en kö fungerar enligt principen FIFO (First in, first out).

17) En ADT, eller Abstact Data Type, är en sorts datastruktur som inte definieras utifrån en specifik implementation utan utifrån ett beteende. Alla datastrukturer som beter sig som en stack kan exempelvis sägas vara stackar.

18) T.ex. stack och kö

19) Generiska typer är en sorts templates som innebär att vi inte behöver definiera på förhand vad datastrukturen ska laga för sorts data. Detta berättar vi i stället vid instansiering, t.ex. när vi skapar en ArrayList och säger att den ska vara av typen Integer.

20) Kontrollflöden är strukturer som manipulerar programflödet på något vis. Exempelvis if-satser (selektion), loopar (iteration) och självanropande metoder (rekursion).

Förstå kod:

1)

a) – sum blir 30 när värdet skrivs ut.
– Algoritmen får tidskomplexiteten O(n).

b) – sum blir 1000 när värdet skrivs ut.
– Tidskomplexiteten blir $O(n^3)$: O(n) * O(n) * O(n), dvs 10 * 10 * 10.

c) – sum blir 30 när värdet skrivs ut.
– Koden får tidskomplexiteten O(n): det rekursiva metodanropet har ersatt for-looparna från exemplet i a).
– increment() får tidskomplexiteten O(n) eftersom den körs n gånger.
– Metoden tar in ett värde, n. Den har ett basfall som kontrollerar om n är 0, och i så fall returnerar 0. Annars inkrementerar den en klassvariabel kallad för sum och anropar sig själv igen med n-1 som nytt n-värde.

d) – sum blir 1000 när värdet skrivs ut.
– Tidskomplexiteten blir $O(n^3)$.
– Här är det viktigt att förstå att även om vi bara har två nästade for-loopar så kommer metodanropet i den inre loopen att fungera likadant som ännu en loop. Det finns därför i praktiken ingen skillnad mellan den här koden och den i b).

2) Se Rekursivt_fibonacciträd.pdf.

3)

a) Kolumnerna följer inte samma ordning eftersom saker bara tas av stacken igen när alla rekursiva anrop returnerat till den punkten. De första stackframes som läggs på kommer därför också att vara bland de sista att försvinna.

b) Den första halvan innan stacken återvänder till A, dvs:

```
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
```

(man kan inkludera A i båda ändarna också, beroende på hur man tolkar frågan!)

c) Om den senaste operationen var att poppa D innehåller stacken A, B, C. Nästa operation kommer vara att pusha G så att stacken innehåller A, B, C, G.

Metodanropet C kommer så småningom att returnera 2 till B.

d) Den andra anropskedjan fib(n-2) kommer alltid att bli lite kortare eftersom vi anropar den med ett lägre nummer än fib(n-1). Det kommer inte behövas lika många anrop för att räkna ut det numret.

e) Tidskomplexiteten för en rekursiv fibonaccialgoritm är mycket riktigt något lägre än $O(2^n)$ i praktiken. Detta beror på att inte varje nytt metodanrop genererar två nya anrop: de anrop som når 0 eller 1 först kommer att stanna och sedan returnera sina värden. Det uppochnedvända trädet kommer alltid att bli ojämnt i kronan eftersom den högra anropskedjan startar med ett lägre värde och därför aldrig går lika djupt som den vänstra. Varje nytt fibonaccianrop som görs genererar i genomsnitt ~1.6 nya anrop.

Faktum är att förhållandet mellan anropen exakt speglar förhållandet mellan fibonaccinumren själva: 1.618, också känt som **det gyllene snittet**, som betecknas φ. Det gyllene snittet är en av världens mest kända proportioner och förekommer överallt i naturen, från mönster i plantor till förhållanden mellan lederna på våra fingrar, mellan hand-underarm-överarm, och-så-vidare. Renässansmålare som Leonardo Da Vinci populariserade dess användning inom bildkonsten (i t.ex. Den vitruvianska mannen och Mona Lisa).

Anledningen till att vi ändå uttrycker tidskomplexiteten som $O(2^n)$ i stället för t.ex. $O(\varphi^2)$ eller $O(1.618^2)$ är för att det dels är lättare att förstå, men även för att ”Big O”-notationen är intresserad av den övre gränsen, dit algoritmen är på väg. Den är inte menad att vara ett exakt värde utan att beskiva en tendens. Tidskomplexiteten är fortfarande exponentiell för en rekursiv fibonaccialgoritm, och för tydlighetens skull betecknar vi den likadant för alla exponentiella algoritmer.

4)
a) Skillnaden är att vi bara gör ett nytt rekursivt anrop i varje metodanrop i stället för två. Därmed undviker vi den exponentiella tillväxten som snabbt förgrenar sig till ett sorts rekursivt träd.

b) Tidskomplexiteten blir $O(n)$ eftersom den inre loopen itererar över alla värden baklänges, från n till $n<=1$.

c) Nej, vi kan inte optimera kodens tidskomplexitet eftersom vi måste multiplicera in varje tal i kedjan med de andra.

Överkurstillägg:

Däremot kan vi optimera dess platskomplexitet, dvs mängden minnesresurser som den använder för att beräkna talet, genom att använda en iterativ lösning i stället:

```
public long factorial(int n)
{
   long result = 1;
   for(int i = 2; i <= n; i++)
   {
      result *= i;
   }
}
```

Skillnaden mellan koden ovan och den rekursiva varianten är att vi bara använder en fixerad mängd minne (variabeln result och loopvariabeln i) oavsett hur stort n är. Inget extra minne kommer att användas oavsett om n är 5 eller 5000. Den rekursiva varianten kommer som jämförelse att generera 5000 activation frames på callstacken om n är 5000.

Man säger att tidskomplexiteten är O(n) för den iterativa versionen medan platskomplexiteten (engelska: space complexity) är konstant, O(1). Platskomplexitet är främst någonting man talar om i system där RAM-minnet är begränsat (t.ex. Internet of Things-saker), eller då datamängderna är så pass stora att den extra overhead som en mellanliggande datastruktur resulterar i blir helt oacceptabel. MergeSort skapar exempelvis nya tillfälliga arrayer varje gång den körs, vilket kostar minne.