Övningsuppgifter

Till varje föreläsning hör ett antal uppgifter i detta häfte och ibland uppgifter som hör till LABA. Förutom dessa bör man programmera när man går igenom föreläsningarna och när man läser boken. På kurswebben hittar du gamla tentor. Dessa innehåller många bra uppgifter du också kan använda som träning. Där hittar du också lösningar till dessa.

Uppgifter föreslagna från vår kursbok markeras med avsnitt och uppgiftstyp.

Krysstal

De uppgifter som är <u>understrukna</u> är krysstal och ska lämnas in och diskuteras för bonus. Observera att inte alla krysstal som hör till en föreläsning ska lämnas in till denna. För att få lämna in ett krysstal måste man ha gjort hela uppgiften. Försök att lösa uppgiften själv även när det finns lösningar i föreläsningen, i boken eller på nätet. Om ni tar hjälp av någon sådan källa är det viktigt att ni förstår koden och kan svara på frågor av typen vad gör den här delen av koden, eller varför får den här variabeln det här värdet, eller vad händer om vi bara kör den här delen av koden. Om man vid test inte kan svara på dylika frågor får man inte poängen. Det ger inget att bara skriva av pseudokod utan att förstå den. Det är inte en bra ide eller ok att använda chat gpt eller copilot eller liknande program.

Påminner också om att man måste ha lyssnat igenom föreläsningen och formulerat en fråga samt närvara och aktivt deltaga i föreläsningen för att få poäng.

Till alla uppgifter gäller att testkod skall skrivas även om det inte anges. Det är inte ok att skriva testkod i en datastruktursklass. Om vi tex skapar en stack så skall implementationen av stacken vara i en egen klass. Denna skall inte innehålla en main. Skapa istället en egen main-klass.

Föreläsning 1

NB 0

Implementera toString, remove, indexOf och set till vår NArrayList från föreläsningen. Försök själv först utan att använda boken.

2nd ed: Avsnitt 2.1: Self Check 1, Programming 1, 2 (s. 69) alternativt
4th ed: Avsnitt 2.2: Self Check 1, Programming 1, 2 (s. 70)

NB 1
På sidan 70 (2nd ed.), 71 (4th ed.) diskuteras en Phone Directory applikation. Där diskuteras hur man kan använda indexOf för att hitta telefonnumret till ett visst namn. På canvassidan kan du hitta ett Phone Directory projekt där man först får lägga in namn och telefonnummer och sedan kan söka upp telefonnummer. Det enda som saknas är metoden equals i DirectoryEntry-klassen. Lägg till denna så att applikationen fungerar.

2nd ed: Avsnitt 2.3: Self Check 1 alternativt
4th ed: Avsnitt 2.4: Self Check 1

NB 2 (inlämnas till föreläsning 2)
När vi använder de färdiga klasserna för lista för att lagra t.ex. heltal tappar vi lite i effektivitet. Om detta är viktigt får man skapa en egen lista specifikt för heltal. Skapa en klass IntList som implementerar en lista som lagrar heltal mha en array. Den skall alltså inte använda någon av klasserna i JCF utan endast en array. Den ska implementera metoderna:

- IntList(int initialCapacity)
- add(int element)
- add(int index, int element)
- get(int index)
- indexOf(int element)
- remove(int index)
- set(int index, int element)
- size()

och kommer precis som vår lista behöva en del hjälpmetoder såsom reallocate. Skriv också en main-klass som testar alla metoder. Fundera över vilka testfall som behövs. **Observera här och för framtiden att testerna inte får skrivas i själva datastrukturen.**

2nd ed: Avsnitt 2.4: Self Check 1, 2, 3 alternativt
4th ed: Avsnitt 2.1: Self Check 1, 2, 3

Föreläsning 2

Avsnitt 2.5: Self Check 1, 2, 3

NB 3
I den här uppgiften ska du träna på att programmera en länkad lista genom att hårdkoda några noder och länka dessa. Skapa en klass Node:

```
public class Node{

    public String data;

    public Node next;

}
```

Skriv en main-klass där du direkt i main utan hjälp av funktioner skriver kod som skapar en länkad lista med datat: ”Gilgamesh” -> ”löper” -> ”på” -> ”stäppen” mha klassen Node. Skriv sedan en while-loop som går igenom listan och skriver ut innehållet i denna till standard out. Avsluta med att lägga till kod som tar bort ”på” innan listan skrivs ut (hårdkoda detta). Fortsätt gärna att ”leka” med din ihopsatta lista. Gör två listor och sätt ihop dem. Klipp av listan halvvägs. Skriv ut listan med en while-loop för att se att resultatet blev som du tänkt dig.

Observera att det är 2 uppgifter i LABA-uppgifterna som hör till föreläsning 2 så gör gärna dessa nu.

Föreläsning 3

2nd ed: Avsnitt 3.1: Self Check 1, 2, 3 alternativt
4th ed: Avsnitt 4.1: Self Check 1, 2, 3

NB 4
Skriv en metod som tar en String som in-parameter och returnerar true om parenteserna är balanserade och false annars. Den ska ignorera andra tecken än parenteser.

Exempel på strängar som ska returnera true: ”(()()(()())((()))())()”, ”(((){}[])[]{()}){}”, ”[{}([]{})]”

Exempel på strängar som ska returnera false: ”({)}”, ”())”, ”[[]”, ”{[]([(){]}}”

Tips: Använd en stack ☺

NB 5
Implementera en stack med en ArrayList som inre datastruktur (ärv inte). Klassen skall implementera interfacet från föreläsningen. Skriv också en main som testar att klassens alla funktioner verkar fungera.

NB 7

Lägg till funktionen size(), peek(n), och flush() till LinkedStack från föreläsningen.

size() returnerar antal element på stacken (använd inte en medlemsvariabel utan gå igenom listan vid varje anrop)

peek(n) returnerar det n:te elementet på stacken utan att påverka denna. 0 motsvarar toppen på stacken.

flush() som tömmer stacken och returnerar det sista elementet på stacken

Skriv också en main som testar funktionerna.

OBS att peek(n) bryter mot en stacks interface. Den är här med endast som övning.

NB 7.1

Gör klart räknaren som kan evaluera postfix-uttryck från föreläsningen. Försök själv innan du tittar i boken.

2nd ed: Chapter 3 Review: Programming Projects 7 (s. 192) alternativt
4th ed: Chapter 4 Review: Programming Projects 4 (s. 209) (inlämnas till föreläsning 4)

NB 7.2 (inlämnas till föreläsning 4)

Skriv en räknare som kan beräkna uttryck i infix-notation med parenteser. Stöd finns i boken i avsnitt 3.4 (2nd) / 4.4 (4th).

Föreläsning 4

NB 8

I en applikation där kön någon enstaka gång växer sig mycket större än normalt skulle det kunna vara intressant att en array-kö också kan krympa när så är möjligt. Utgå från vår kö på föreläsningen och lägg funktionalitet som halverar storleken på arrayen om den bara använder en fjärdedel av den nuvarande arrayen. Tänk igenom noga vilka fall du behöver testa.

NB 8.1

På canvas hittar du en kö. Du ska endast skriva test-kod till denna. Försök att tänka ut vilka olika fel som skulle kunna uppstå i en cirkulär kö och se till att du testar för dessa. Detta är alltså inte ”black box”-testning utan den sorts testning vi som programmerare gör iterativt medans vi programmerar. Du ska inte bara testa för de fel du faktiskt ser i koden utan för de saker i koden som skulle kunna gå fel. När du hittar ett fel med din test-kod så rättar du felet och fortsätter testningen. Observera att allt som skiljer sig från koden på föreläsningen inte är fel. Rätta bara de saker som orsakar fel. Bland annat så pekar `rear` på första lediga istället för sista platsen i kön. Det ska vara så även efter du rättat koden. Du bör lägga till en `toString` till kön så att du får ett fönster in i klassen. Det behöver man nästan alltid när man testar icke black box.

På redovisningen ska du ha din fullständiga test-kod och ursprungskön. Du ska sedan köra koden och visa vad den testar och hur den visar på felen.

Detta med testning av koden medans vi programmerar är en mycket viktig färdighet att vara bra på och brister är ofta orsaken till att man tappar poäng på tentamen. Gör detta till alla uppgifter du löser även om du inte alltid behöver vara så noggrann som du förhoppningsvis var till denna uppgift.

NB 9

Implementera en kö med en enkellänkad lista med en pekare till varje ända av listan så att alla operationer blir O(1). Det räcker om du implementerar enqueue (offer), dequeue (poll) och size. Använd bara bokens exempel om du kör fast.

NB 10

Implementera en Dequeue som en dubbellänkad lista (obs använd ej klasser från Java’s API utan gör en egen dubbellänkad lista i klassen). En dequeue är en kö där man kan sätta in och ta bort element i båda ändar. Den behöver inte implementera något interface och ska endast ha följande operationer: pollFirst, pollLast, offerFirst, offerLast och empty. Se avsnitt 4.4 om du undrar över vad de ska göra. Skriv också testkod.

Föreläsning 5

När du skriver rekursiva funktioner sträva efter att inte använda medlemsvariabler (globala variabler i C). Funktioner ska få alla data de behöver som in-parametrar och returnera ut-data. Endast om man har goda skäl ska man frångå detta.

NB 11

Skriv en rekursiv static metod som hittar största värdet i en array av ints.

NB12

Skriv en rekursiv static metod som beräknar x^n där n är ett positivt heltal. Skriv en iterativ metod som löser samma uppgift.

NB13

Skriv en metod som beräknar roten ur ett tal större eller lika med 1 med tre decimalers noggrannhet med hjälp av den rekursiva algoritmen nedan. *e* styr noggrannheten och a kan sättas till 1 initialt.

$$\mathrm{ROT}(n,a,e) = \begin{cases} a & \text{om } |a^2 - n| < e \\ \mathrm{ROT}(n, \frac{a^2+n}{2a}, e) & \text{annars} \end{cases}$$

NB 14 Myntmaskinen

En maskin har en display på vilken den visar uppnådd poäng. Från början visar displayen 1 poäng. Genom att stoppa mynt i maskinen kan poängen förändras:

- Genom en 10-krona multipliceras poängen på displayen med 3
- Genom en 5-krona adderas 4 till poängen på displayen.

Skriv ett program som tar emot den poäng som ska uppnås. Programmet ska sedan beräkna lägsta belopp som krävs för att uppnå poängen exakt. Observera att man inte kan nå målet för alla slutpoäng.

En exempelkörning:

```
Vilken poäng ska uppnås: 109
Poängen kan nås med 45 kronor
```

Tips 1: Använd rekursion för att pröva alla lösningar. Försök inte hitta villkor för vilket val som är bäst. Det är inte det vi vill träna på. Försök inte lösa uppgiften baklänges för det är svårare. Räkna istället upp poängen mot det givna målet.

Tips 2: Försök lösa denna uppgift med en wrapper som anropar en rekursiv metod som vid varje anrop returnerar billigaste kostnaden. Funktionshuvudet för den rekursiva funktionen bör kunna se ut enligt:

```
int cost(int points, int goal)
```

och anropet till denna ifrån wrapper'n blir då cost(1,goal). Den anropar sedan sig själv med högre och högre poäng och varje anrop returnerar kostnaden för att komma från points till goal. Det betyder att när den anropas med points==goal så returnerar den 0. Om du följer denna instruktion kanske du

tänker så klart jag gör på detta sett men faktum är att de flesta brukade lösa denna med en rekursiv metod med huvudet:

```
int cost(int points, int goal, int totalValue)
```

där metoden returnerar totalValue när points==goal. Detta är inte fel men mindre snyggt och den går inte heller att effektivisera med dynamisk programmering. Pröva gärna att göra även denna lösning så att du bättre förstår skillnaden. När du löser problem försök sträva efter att använda den första metoden. Den är oftast bäst.

NB 15

Skriv en rekursiv statisk metod som tar en textsträng med ett tal i binär form och returnerar motsvarande heltal. Använd en wrapper-metod om det behövs. Exempel: ”1011” ska returnera 11. Skriv också en metod som tar en int och returnerar en sträng med det binära talet. Även här behöver du antagligen en wrapper-metod.

NB15.1 (inlämnas till föreläsning 6)

En lek med kulor går till på följande sätt. Man börjar med ett visst antal röda, vita och blå kulor. I en låda ligger obegränsat med röda, vita och blå kulor. Man ska nu se till att växla sina kulor med kulorna i lådan så att man får lika många kulor av varje färg och man ska göra detta på så få växlingar som möjligt. Följande växlingar är tillåtna:

- 1 blå kula kan växlas mot 1 vit kula och 3 röda kulor.
- 1 vit kula kan växlas mot 2 blåa kulor och 4 röda kulor.
- 1 röd kula kan växlas mot 1 blå kula och 5 vita kulor.

Exempel:

- Om man börjar med 2 blåa, 5 vita och 0 röda kulor behövs 1 växling (byt en vit)
- Om man börjar med 1 blå, 0 vita och 1 röd kula behövs 2 växlingar
- Om man börjar med 2 blåa, 1 vit och 0 röda kulor behövs 9 växlingar
- Om man börjar med 3 blåa, 2 vita och 1 röd kula behövs 9 växlingar
- Om man börjar med 5 blåa, 2 vita och 1 röd kula behövs 3 växlingar
- Om man börjar med 5 blåa, 1 vita och 3 röd kula behövs 15 växlingar

Skriv en metod som tar antalet kulor i de tre färgerna och returnerar antalet växlingar som behövs.

Tips: Denna uppgift skiljer sig på en punkt från de vi tittat på hitintills. Om vi naivt testar de tre alternativen rekursivt riskerar vi att hamna i en oändlig loop där vi aldrig når målet. Den här typen av problem behöver ett maxdjup, ett maximalt antal växlingar vi gör innan vi ger upp på just det här försöket. Man behöver då veta hur många växlingar problemet maximalt kan behöva eller så får man testa sig fram. Mer om detta i nästa föreläsning. Du kan utgå ifrån att det alltid räcker med 15 växlingar när du löser detta problem.

Föreläsning 6

NB 16

Skriv en ny lösning till NB15.1 som löser problemet med bredden först istället för djupet först.

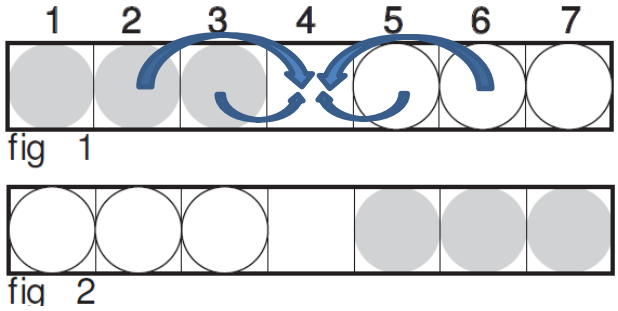
NB 17

Skriv om getNode till vår länkade lista så att den blir rekursiv. Du behöver inte hantera felaktiga index. Skriv också en rekursiv size som inte använder en medlemsvariabel utan beräknar size vid varje anrop. Tänk på att size ska fungera på en tom lista.

NB 18

Skriv ett program som hittar ut ur en labyrint med hjälp av backtracking. Utgå från skalet som finns på kth-social och använd backtracking för att skriva klart funktionen solve så att programmet presenterar en lösning till inläst labyrint. Prova programmet på några labyrinter som du skickar med din lösning. Enklast gör du detta genom att utgå ifrån startlabyrinten och prova att ändra väggar och att flytta målet.

NB 19



I ett enkelt brickspel ligger svarta och vita brickor enligt fig 1. Målet är att få brickorna enligt fig 2. Tillåtna drag är att flytta en bricka ett steg till en tom ruta eller hoppa över en bricka till en tom ruta. För att lösa problemet finns aldrig någon anledning att flytta en svart bricka åt vänster eller en vit bricka åt höger. Skriv ett program som hittar alla möjliga lösningar till problemet och skriver ut dragsekvensen för varje lösning. Lösningarna skrivs:

3->4, 5->3, 6->5, osv

NB 20 (inlämnas till föreläsning 7)

Skriv ett program som löser *n*-damer problemet och presenterar alla lösningar och antalet lösningar. *n*-damer-problemet är som åtta damer problemet där man får ange *n* där *n* är antalet damer som ska placeras på ett bräde med *n* rader och *n* kolumner. Observera att programmet ska visa hur alla lösningar ser ut.

Föreläsning 7

Utgå från föreläsningens implementering av ett binärt sökträd när du löser nedanstående uppgifter. Skriv också testkod så att du vet att de fungerar korrekt. Glöm ej att göra LABA-uppgiften på detta.

NB 21

Skriv en main som läser in sju ord från användaren och placerar dessa i ett BST. Kör programmet och skriv in 7 ord så att trädet får minsta möjliga höjd. Kör programmet och skriv in 7 ord så att trädet får maximal höjd.

NB 22

Skriv funktioner som traverserar trädet med preOrder och postOrder.

NB 22.1

För att kunna studera att en implementation av ett binärt sökträd faktiskt fungerar korrekt kan det vara bra att kunna skriva ut hur trädet faktiskt ser ut. Skriv metoden printTree som skriver ut ett träd till standard ut. Den ska skriva ut trädet med en nivå per rad. För varje nivå skriver den ut barnen till alla noder som fanns i nivån ovanför även om dessa är tomma (då skriver den null). På så sätt kan man entydigt se strukturen hos trädet. Trädet man får om man adderar H, B, N, A, E, C, F, D i given ordning skrivs då ut:

H
B N
A E null null
null null C F
null D null null
null null

Du får inte lägga till medlemsvariabler i det binära trädet's klass eller nod-klassen.

Tips: Använd en kö för att besöka noderna enligt bredden först. För att kunna göra radbryt när du byter nivå kan du också köa vilken nivå varje nod befinner sig på.

NB 23

Skriv funktionerna numberOfLeaves, numberOfNodes och height till vår implementation från föreläsningen. Obs de ska bestämma antalet utifrån trädet och inte använda privata medlemsvariabler.

NB 23.1

Utgå ifrån vårt binära sökträd från föreläsningen. Skriv om sökfunktionen så att den är iterativ istället för rekursiv. Skriv två funktioner maxRec och maxIt som returnerar det största värdet i ett binärt sökträd. Den ena rekursiv och den andra iterativ. Alla funktioner ska vara $O(\log n)$.

Föreläsning 8

NB 24

Skriv en toString-metod och en remove-metod till vår hashtabell med länkning från föreläsningen. Skriv toString-metoden med fokus på att den ska kunna hjälpa till vid testning av klassen. Den bör alltså skriva ut tabellen så att man kan se var i tabellen de olika värdena är lagrade. Skriv som alltid en main som testar din metod. På canvas hittar filer som utgår ifrån att du använder vår enkellänkade lista. Den bör anpassas så att den minimerar minnesbehovet hos varje instans och därmed inte ha någon variabel size. För att det ska fungera behöver du ha implementerat remove i iteratorn. Har du svårt att få till det med din enkellänkade lista kan du använda javas dubbellänkade lista.

NB 25

Lägg också till att den dubblar storleken på tabellen när antalet entries (key, value –par) överstiger 75% av tabellens storlek. Kom ihåg att du måste sätta in alla entries igen eftersom de kommer att få ett annat index. Skriv en main som testar din metod.

NB 26

För att göra denna behöver du ha gjort NB24. Skriv nu ordentlig testkod för vår hashtabell. Om ni gjort NB25 ska ni testa även denna. Fundera på vad den behöver testa. Inspiration kan du hämta på sid 394 i boken.

NB 26.1

Skriv en funktion som tar en array av strängar och returnerar antalet gånger som den vanligaste strängen förekommer i arrayen. Din lösning ska vara O(n). Helst ska du bara behöva gå igenom array:en en gång. Tips: Enklast är att använda sig av en HashMap där value är antalet gånger ordet förekommer. Kom också ihåg att det inte är bra att gå igenom en HashMap.

Ex:{”man”, ”gråter”,”när”,”man”,” tänker”,”när”,} ska returnera 2.

NB 26.2

Skriv en funktion som tar en array av strängar och returnerar antalet unika strängar. Din lösning ska vara O(n). Tips: Kanske ett hashset?

Ex:{”man”, ”gråter”,”när”,”man”,” tänker”,”när”,} ska returnera 4.

NB 26.3

Implementera en egen hashtabell som använder öppen adressering. Den ska minimum ha put, get och remove och göra rehash vid behov. Börja själv och använd boken som inspiration om du kör fast.

Föreläsning 9

NB 27

Skapa en klass Car som innehåller bilmärke, årsmodell och antal körda mil. Klassen ska implementera Comparable<Car> och därmed ha en metod compareTo som ska jämföra bilmärke som sträng mellan bilar.

a) Skriv ett program som läser in bilar från en textfil till en array och sorterar dessa med hjälp av Java API utifrån metoden compareTo. Skriv sedan ut dessa till en ny textfilen.

b) Skapa nu en klass CompareCar som implementerar Comparator<Car> och använd denna för att sortera bilarna efter årsmodell istället. Skriv även ut denna ordning men till en annan fil.

Vill här påminna om att ni ska kunna och kunna förklara er kod för att få ladda upp den. Detta gäller alltid men extra viktigt att påminna om för sorteringsalgoritmerna. Skriver ni in en sorteringsalgoritm utan att förstå den ska ni alltså inte lämna in den. Det kan bli så att ni blir ombedd att förklara er kod för mig.

NB 28

Implementera urvalssortering och instickssortering för en array av int. Du ska utgå från algoritmen på föreläsningen och inte använda någon kod eller pseudokod från nätet eller boken. Skriv också testkod.

NB 29a

Implementera Mergesort för en array av int. Du ska utgå från algoritmen på föreläsningen och inte använda någon kod eller pseudokod från nätet eller boken. Skriv också testkod.

NB29b

Implementera Shellsort för en array av int. Du ska utgå från algoritmen på föreläsningen och inte använda någon kod eller pseudokod från nätet eller boken. Skriv också testkod.

NB29c

Implementera Quicksort för en array av int. Utgå från algoritmerna i boken. Använd helst inte koden från boken men om du gör det se till att du förstår den.

Föreläsning 10

NB30

Implementera en generic min-heap: heap<E>. E skall implementera comparable och elementen ska sorteras efter compare-metoden. Den här implementeringen har alltså inte separata nycklar så som vår hash-tabell hade. Använd en array som inre datorstruktur. Konstruktorn ska tillåta att man sätter startstorleken. Om arrayen blir full skall storleken dubblas (däremot behöver den inte krympa). Skriv testkoden med strängar.

NB 29d

Implementera Heapsort för en array av int som är in-place. Du ska utgå från algoritmen på föreläsningen och inte använda någon kod eller pseudokod från nätet eller boken. Skriv också testkod.

Föreläsning 11

NB 32

Skriv en klass som använder en grannlista (förbindelselista, adjacency list) för att representera en viktad graf.

Konstruktorn tar ett filnamn på en textfil och skapar utifrån denna en grannlista. Du väljer själv hur du organiserar textfilen för att representera grafen. Det ska dock vara relativt enkelt att skriva en textfil för att representera en godtycklig graf.

Skriv toString som skriver ut en nod per rad följt av noderna som denne har bågar till med vikter för grannlistan.

Skriv en metod som returnera närmsta vägen mellan två noder för grannlistan. Den ska använda Dijkstra. Observera att den kan effektiviseras genom att du avslutar så fort slutnoden valts.

Skapa en textfil som representerar grafen från föreläsningen och testkör.

NB 33

Skriv en static metod som hittar det minsta uppspända trädet för en graf representerad med en grannmatris (förbindelsematris, adjacency matrix) med hjälp av Prims algoritm. Funktionen skall ta en grannmatris som input och returnera en int-array p, där p[v] anger den nod som noden v anslöts till enligt algoritmen. Använd samma knep som vi använde för Dijkstra så att din algoritm blir $O(n^2)$, där n är antalet noder. Tänk noga efter vad du behöver ändra.

Utgå ifrån filerna som finns på Canvas. Du kommer då att kunna se grafiskt vilket träd du hittar. Det är bra om du försöker förstå resten av koden. I alla fall bör du förstå hur p används för att få kanter (edges).

NB 33.2

Gör om uppgift nb33 men nu för en grannlista. Den ska då bli effektivare än vår lösning med matris om inte alla noder har bågar till varje annan nod. Obs här får du då skriva en egen main och klara dig utan grafik. Skriv ut resultatet i main på formatet:

Node A var startnod
Nod F anslöts till A
Nod D anslöts till F
Nod B anslöts till A
Nod C anslöts till B
Nod H anslöts till C
Nod E anslöts till C
Nod G anslöts till F
TotalVikt: 15

(obs att det kan finnas fler korrekta lösningar med avseende på hur man ansluter noderna)

Föreläsning 12

NB34

Lös växlingsproblemet med en girig algoritm. Skriv en metod som tar växlingssumman och en int-array med de olika tillgängliga valutorna I fallande ordning och returnerar en int-array med antalet av varje valuta. Ex. Man vill veta växeln för 789 kr I vår valuta. Man anropar då:

change(789,new int[] {1000,500,100,50,20,10,5,1})

som då returnerar: {0,1,2,1,1,1,1,4}.

NB35

Lös det obegränsade kappsäcksproblemet med en girig algoritm. Skriv ett primitivt användargränssnitt (ok med textbaserat) där man får ange storlek på kappsäcken och varornas vikt och värde. Programmet ska sedan presentera vilket totalt värde man fick med sig i kappsäcken och hur många av varje vara man packade.

NB36

Skriv en schemaläggningsfunktion enligt föreläsningen.

NB37 (egyptian fraction)

Ett rationellt tal mellan noll och ett kan alltid skrivas som en summa av bråk med täljaren 1 och olika nämnare. Några exempel:

$$\frac{7}{11}=\frac{1}{2}+\frac{1}{8}+\frac{1}{88},\ \frac{3}{11}=\frac{1}{4}+\frac{1}{44},\ \frac{4}{5}=\frac{1}{2}+\frac{1}{4}+\frac{1}{20}$$

En sådan summa kan hittas med en girig algoritm. Skriv ett program som läser in ett bråk och skriver ut summan enligt ovan. Använd en enkel girig algoritm. Observera att inte ens en long alltid kommer att räcka för att representera nämnaren så använd helst Biginteger. Vår giriga algoritm kommer inte alltid hitta den bästa lösningen. T.ex får vi:

$$\frac{5}{121}=\frac{1}{25}+\frac{1}{757}+\frac{1}{763309}+\frac{1}{873960180913}+\frac{1}{1527612795642093418846225}$$

trots att:

$$\frac{5}{121}=\frac{1}{33}+\frac{1}{121}+\frac{1}{363}$$

NB38

Skriv en algoritm som löser handelseresande-problemet med en girig algoritm. Den ska spara både faktisk väg och sträckans längd. Utgå ifrån skalet i Canvas så att vi får en grafisk representation. Använd helst ett BitSet för att representera vilka städer vi besökt. Du behöver bara skriva själva algoritmen i TravelingSalesMan-klassen. Obs i kommentaren står hur data är organiserat.

NB39

Vid en fabrik finns ett antal aktiviteter som behöver utföras. Varje aktivitet har en starttid och en sluttid som ska anges som en double. Under denna tid måste en person jobba med aktiviteten. Skriv en funktion som tar en array av aktiviteter och returnerar det minsta antalet personer som behövs. Använd en girig algoritm som du försöker få så effektiv som möjligt. För poäng ska den bli O($n$log$n$) där n är antalet aktiviteter, vilket går om du använder en heap. Försök utan att titta på tipset först.

Tips. Sortera aktiviteterna efter starttid. Använd en heap för att lagra pågående aktiviteter sorterade efter sluttid. För varje aktivitet a (i ordning) ta bort alla pågående som har sluttid före a's starttid från heapen. Lägg sedan in a i heapen. Det största antalet medlemmar heapen har är antalet personer som behövs men håll räkning på detta själv så du inte tappar effektivitet.

NB40

Givet n punkter på den reella tallinjen. Hitta det minsta antal intervall med längden 2,0 som tillsammans täcker alla punkter. Designa och implementera en girig algoritm. Låt programmet slumpa fram ett antal punkter. Försök verifiera att din algoritm fungerar med ett resonemang. Skriv även ner ditt resonemang i din redovisning.

Föreläsning 13

NB41

Skriv ett program som slumpar n stycken heltal mellan -1000 och 1000. Användaren anger n. Programmet beräknar sedan den maximala delsekvenssumman med hjälp av ett funktionsanrop av en funktion som använder en söndra och härska algoritm. Den skriver sedan ut vad summan blev och hur många anrop till funktionen som gjordes.

NB42

Skriv en funktion som tar en array med punkter (använd en klass för att representera en 2D-punkt med float för koordinaterna) och returnerar det kortaste avståndet mellan två punkter. Funktionen ska använda en effektiv söndra och härska algoritm enligt föreläsningen (dvs inte jämföra onödiga punkter över mittlinjen). Skriv också en funktion som löser problemet med genom att jämföra varje punkt med varje annan punkt och som då blir $O(n^2)$. Skriv nu en main som slumpar fram n punkter som ligger i intervallet: $-1<x<1$ och $-1<y<1$ med lika stor sannolikhet för alla värden. Anropa dina funktioner med de framslumpade punkterna och kontrollera att du får samma resultat.

NB43

Skriv ett program som löser det generella växlingsproblemet. Användaren får först ange hur många olika valutor det finns och sedan valören på dessa. Därefter får denne ange vilket belopp som skall växlas och programmet svarar med minsta antalet mynt och sedlar som behövs. Lägg till att programmet också skriver ut växeln. Observera att denna algoritm blir mycket ineffektiv och inte klarar annat än mycket små problem.

NB44

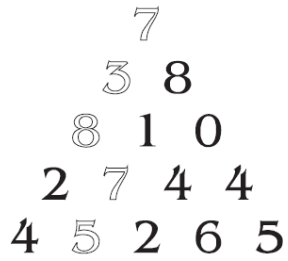
Skriv en funktion som löser Skyline-problemet. Det räcker att du testar den med sex hårdkodade fall varav minst två ska vara med minst 10 hus. Skriv om du vill ett program som slumpar fram n st hus (n anges av användaren) och ritar upp dessa och deras skyline.

NB45

Av n element skall vi ta reda på om det finns mer än n/2 av något och i så fall vilket. Utgå från att elementen kan testas för likhet men inte rangordnas (equal men inte compare). Om du vill kan du skriva metoden för en array av heltal men kom i så fall ihåg att du inte får utnyttja att de går att sortera. Designa en söndra och härska algoritm som hittar ett sådant elementet i en array om ett sådant finns. Skriv sedan en funktion och en main med några hårdkodade exempel. Sträva efter att få algoritmen så effektiv som möjligt (O(nlogn) blir resultatet av en bra söndra och härska). Man kan hitta en lösning som är O(n) men då inte med söndra och härska.

Föreläsning 14

NB46



Skriv en funktion som beräknar högsta summan man kan få genom att lägga ihop talen längs valfri väg från toppen av en taltriangel till botten. I varje steg nedåt kan man välja vänster eller höger. I triangeln finns inga negativa tal.

a) Lös problemet med enkel rekursion

b) Förbättra effektiviteten genom att tabulera värden du räknat ut (dynamisk programmering – top-down). Jämför antalet anrop med a) för en 10-tal hög triangel.

c) Skriv en funktion som bygger upp lösningen nerifrån och upp (dynamisk programmering bottom up)

NB46.1

Effektivisera LABA uppgift 5 med tabulering (dynamisk programmering). Observera att om du passerar maxdjupet och returnerar ett ogiltigt värde ska du inte tabulera det. Det är inte ett resultat.

NB46.2

Effektivisera LABA uppgift 6 genom att aldrig köa samma lådtillstånd 2 gånger. Använd ett HashSet för att åstadkomma detta.

NB46.3

Kommer ni ihåg myntmaskinen (NB14)? Lägg till tabulering. Dokumentera effektivitetsförbättringen för slutpoängen 109.

NB46.4

Kommer ni ihåg hissproblemet (Föreläsning 6). Den galna hissen ska nu få en knapp till: mult. Om mult = 3 och man står på våning 4 kommer man till våning 3 × 4 = 12 om möjligt. Ni ska nu lösa den med djupet först (maxdjup 30) men effektivisera genom att tabulera värden. Ta också reda på hur många anrop ni sparar med dynamisk programmering för exemplet: n=45, upp=7, ned=5, mult=3 destination=35. Obs eftersom denna uppgift kräver ett maxdjup blir den lite speciell. När vi nått ett maxdjup returnerar vi inte ett svar utan tex ett stort tal för att visa att vi inte nådde en lösning. Detta resultat är inget resultat och får inte sparas i tabellen. Frivilligt men bra tentaträning: Gör en version som returnerar bästa trycksekvensen.

NB47

Skriv en funktion som löser det generella myntväxlingsproblemet med hjälp av dynamisk programmering. För godkänt krävs en top-down lösning svarar både på antal mynt och vilka mynt eller en bottom-up lösning som svarar bara på antal mynt.

NB47.1

Utgå ifrån problem 14.1 och din rekursiva djupet först lösning. Effektivisera nu lösningen med tabulering. Tabellen skulle då kunna vara en tredimensionell array. Ett problem med detta val är att veta hur stora dimensioner vi behöver eftersom antalet kulor kan öka. Pröva här att istället använda en hash-tabell. Vi behöver då en klass för att representera de tre kulorna så att vi för varje konfiguration kan spara antalet byten i hash-tabellen. Förslag på klass finns på nedan.

```
private static class Marbles{
    private final int nrOfBlue, nrOfWhite, nrOfRed;

    public Marbles(int nrOfBlue, int nrOfWhite, int nrOfRed) {
        this.nrOfBlue = nrOfBlue;
        this.nrOfWhite = nrOfWhite;
        this.nrOfRed = nrOfRed;
    }
    public boolean existBlue() {
        return nrOfBlue > 0;
    }

    public boolean existWhite() {
        return nrOfWhite > 0;
    }

    public boolean existRed() {
        return nrOfRed > 0;
    }

    public Marbles exchangeBlue() {
        if (!existBlue()) return null;
        return new Marbles(nrOfBlue - 1, nrOfWhite + 1, nrOfRed + 3);
    }

    public Marbles exchangeWhite() {
        if (!existWhite()) return null;
        return new Marbles(nrOfBlue+2, nrOfWhite-1, nrOfRed +4);
    }
    public Marbles exchangeRed(){
        if(!existRed()) return null;
        return new Marbles(nrOfBlue+1, nrOfWhite+5, nrOfRed-1);
    }

    public boolean done(){
        return nrOfBlue == nrOfRed && nrOfBlue == nrOfWhite;
    }

    @Override
    public boolean equals(Object o){
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Marbles marbles = (Marbles) o;
        return nrOfBlue == marbles.nrOfBlue && nrOfWhite == marbles.nrOfWhite && nrOfRed ==
marbles.nrOfRed;
    }

    @Override
    public int hashCode() {
        return Objects.hash(nrOfBlue,nrOfWhite,nrOfRed);
    }
}
```

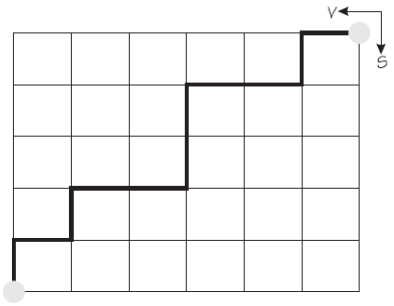
NB48

Implementera Floyd-Warshall. Testa med grann-matrisen vi använt på föreläsningen.

NB49

Implementera ett program som löser kappsäcksproblemet enligt föreläsningen. Helst bottom-up men det går bra med top-down också. Det räcker att den beräknar värdet. Skriv också ett enkelt gränssnitt.

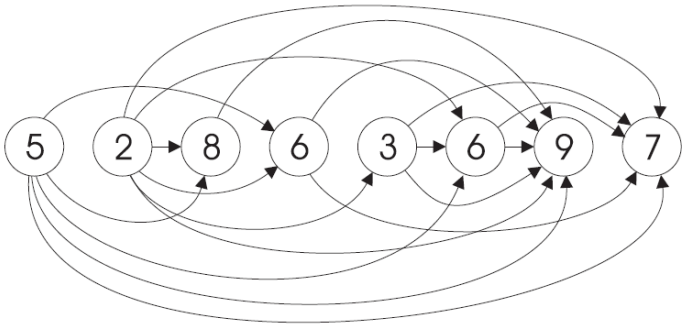
NB50



Hur många unika vägar finns det från övre högra hörnet till nedre vänstra hörnet om vi bara får gå väst och syd?

Förbättra vår lösning med hjälp av dynamisk programmering. Hur många funktionsanrop blir det för (12,6)? Jämför med vår gamla algoritm utan tabulering.

NB51



I en följd av heltal ska vi försöka hitta längden hos den längsta sekvensen stigande tal. Alla stigande sekvenser i talföljden 5, 2, 8, 6, 3, 6, 9, 7 är markerade ovan. Den längsta blir då 4 lång och är 2, 3, 6, 9 eller 2, 3, 6, 7. Skriv en $O(n^2)$ som löser problemet ovan.
Tips:
Börja längst till vänster. Varje nytt tal ger två alternativ:
Talet är större än föregående tal i vår sekvens:

1. Ta med talet 2. Hoppa över talet

Talet är inte större än föregående tal i vår sekvens:

1. Påbörja ny sekvens 2. Hoppa över talet

Det är faktiskt enklast att lösa detta problem med en iterativ bottom-up lösning.

NB 54

Utgå ifrån N!- lösningen av handelsresandeproblemet och lägg till kod så att du också får ut vilken väg algoritmen valt. Försök nu göra samma sak med den effektiva lösningen. Detta kräver nya klasser och en hel del jobb men är ett roligt problem om du har tid och intresse. När du är klar har du nog löst det mest ambitiösa algoritmiska problemet i kursen. Grattis!