DD1361 Programmeringsparadigm

Laborationer 2016

Innehåll

Labbkvitto 1
Allmänna Instruktioner 2
Labb F1: Uppvärmning i Haskell 3
Labb F2: Molekylärbiologi i Haskell 5
Labb F3: Evolutionära träd och I/O 9

Senast uppdaterad: 27 augusti 2016

Labbkvitto

DD1361 Programmeringsparadigm 2016

Underskrifter är **giltiga i 6 månader**. Det är *ditt ansvar* att kontrollera att dina labbresultat rapporterats in i rapp-systemet och att snarast kontakta kursledaren om så inte gjorts.

På kursen tillämpas CSC-skolans hederskodex. Jag intygar att jag har läst och förstått denna hederskodex, samt att jag har läst och förstått de allmänna labb-instruktionerna på nästa sida i labb-kompendiet.

. . . .

Signatur

Namnförtydligande

Labb	**Datum**	**Handledare**
F1		
F2		
F3		
L1		
L2		
L3		
S1		
S2		
S3		
S4		
X1		
X2		
Inet		

Fet stil indikerar vilka labbar som är obligatoriska.

Allmänna Instruktioner

1 Kattis-systemet

De flesta av labbarna använder sig av systemet Kattis för automatisk rättning av er kod. För att stifta en första bekantskap med systemet kan du kolla på hjälp-sidan för ditt favorit-språk.

2 Git

I kursen använder vi KTH:s Gitlab-installation för att arbeta med labbar och lämna in dem för redovisning.

För detaljerad information om detta se sidan om git på kurs-hemsidan.

3 Kod/dokumentations-krav

Utöver att er kod ska bli godkänd av Kattis krävs följande:

1. Det ska vara tydligt dokumenterat i kommentar högst upp i koden vilka som har skrivit koden. Detta gäller *alla* inskickningar ni gör till Kattis, och är inte något ni kan lägga till i slutet när ni väl fått er kod att bli godkänd av Kattis.
2. Själva koden ska vara ordentligt kommenterad. Syftet med olika funktioner/predikat som ni definierar ska förklaras.

4 Tidsfrister

Se sidan om laborationer på kurshemsidan för information om vilka tidsfrister som gäller för att göra färdigt de olika typerna av labbar.

5 Kösystem

På labb-passen använder vi kösystemet Stay A While, `http://queue.csc.kth.se/#/queue/Progp`.

Labb F1: Uppvärmning i Haskell

Problem-ID på Kattis: kth:progp:warmup

I denna labb ska du konstruera några enkla funktioner i Haskell. Alla funktioner du definierar i denna labb ska ligga i en modul som heter `F1`. I ditt git-repo för labben finns ett kod-skelett som du kan utgå ifrån, som innehåller trivial kodstubbar (som såklart inte gör rätt) för samtliga deluppgifter i labben.

1 Fibonacci-talen

Fibonacci-talen är en talföljd som definieras så här:

$$F(n) = \begin{cases} 0 & \text{om } n = 0 \\ 1 & \text{om } n = 1 \\ F(n-1) + F(n-2) & \text{om } n > 1 \end{cases}$$

Uppgift Skriv en funktion `fib` som tar ett heltal n och returnerar $F(n)$. Du behöver inte hantera negativa tal. Funktionen ska klara att beräkna $F(n)$ för n upp till 30 på en bråkdel av en sekund. Hur lång tid tar det att beräkna $F(35)$? $F(40)$?

Tips Lättast är att definiera funktionen med flera ekvationer (analogt med definitionen ovan).

Exempel

`fib(7)` ska returnera 13

`fib(17)` ska returnera 1597

2 Rövarspråket

I *rövarspråket* dubblerar man alla konsonanter och lägger ett "o" emellan, se exempel nedan. (För den här uppgiften ignorerar vi de specialfall som ibland tillämpas där t.ex. "x" behandlas som "ks".)

Uppgift Skriv en funktion `rovarsprak` som tar en sträng och returnerar en ny sträng där varje konsonant x har ersatts av strängen xox. Skriv också en funktion `karpsravor` som gör det omvända, dvs tar en sträng på rövarspråk och "avkodar" den.

Funktionerna behöver bara hantera strängar med gemener (inga mellanslag, siffror, stora bokstäver, eller andra tecken), och behöver inte hantera åäö. Funktionen `karpsravor` behöver bara fungera på strängar som verkligen tillhör rövarspråket, ingen felhantering behövs för felaktig indata.

Funktionerna ska gå i linjär tid och hantera strängar på upp till $100\,000$ tecken inom en bråkdel av en sekund.

Tips Ni vill antagligen skriva en funktion som avgör om ett givet tecken är vokal eller konsonant. Funktionen `elem` kan vara en praktisk byggsten för detta. I den här uppgiften anser vi "y" vara en vokal (som i svenskan).

Exempel

`rovarsprak("progp")` ska returnera `poprorogogpop`

`rovarsprak("cirkus")` ska returnera `cocirorkokusos`

`karpsravor("hohejoj")` ska returnera `hej`

`karpsravor("fofunonkoktotionon")` ska returnera `funktion`

3 Medellängd

Uppgift Skriv en funktion `medellangd` som tar en text (String) som indata och returnerar ett tal (Double) med medellängden på orden i texten.

Ett ord definierar vi som en sammanhängande delsträng av bokstäver ur alfabetet, stora eller små. Alla blanka tecken, kommatering, siffror, etc, är ord-delande.

Funktionen ska gå i linjär tid och hantera texter på upp till 100 000 tecken inom en bråkdel av en sekund.

Tips Funktionen `isAlpha :: Char -> Bool` returnerar sant på just de tecken som finns i alfabetet. För att komma åt `isAlpha` måste du importera modulen `Data.Char`.

En möjlig ansats är att först stycka upp texten i ord och sedan beräkna antal ord samt totala längden på orden.

Exempel

`medellangd("No, I am definitely not a pie!")` ska returnera 3.14285714...

`medellangd("w0w such t3xt...")` ska returnera 1.8

4 Listskyffling

Vi är intresserade av att kasta om elementen i en lista enligt följande: först tar vi varannat element (första, tredje, femte, etc). Vi upprepar sedan detta på elementen som återstår (dvs tar andra, sjätte, tionde, etc). Detta upprepas så länge det fortfarande finns element kvar. Om vi t.ex. börjar med listan (1, 2, 3, 4, 5, 6, 7, 8, 9) kommer vi i första vändan få (1, 3, 5, 7, 9), och elementen (2, 4, 6, 8) återstår. I andra vändan lägger vi till (2, 6), och bara (4, 8) återstår. I tredje vändan lägger vi bara till 4, och bara 8 återstår. I fjärde och sista vändan lägger vi slutligen till 8, och slutresultatet blir listan (1, 3, 5, 7, 9, 2, 6, 4, 8).

Uppgift Skriv en funktion `skyffla` som tar en lista som indata och returnerar en omkastad lista enligt beskrivningen ovan.

Funktionen ska fungera på alla typer av listor.

Funktionen ska kunna hantera listor på upp till 5 000 element inom en bråkdel av en sekund (var försiktig med "++"-operatorn!).

Exempel

`skyffla(["kasta", "ord", "om"])` ska returnera `["kasta", "om", "ord"]`

`skyffla([3.4, 2.3, 5, 185, 23])` ska returnera `[3.4, 5, 23, 2.3, 185]`

`skyffla([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])` ska returnera `[1, 3, 5, 7, 9, 11, 2, 6, 10, 4, 12, 8]`

`skyffla([1,2..5000])` ska returnera sitt svar inom en bråkdel av en sekund.

Labb F2: Molekylärbiologi i Haskell

Problem-ID på Kattis: kth:progp:f2

I denna labb ska du konstruera verktyg för att arbeta med molekylärbiologi i Haskell. Alla funktioner du definierar i denna labb ska ligga i en modul som heter `F2`.

1 Exempeldata och testning

För att hjälpa till på traven i testningen av din kod tillhandahålls en hjälpfil `molbio.hs` i ditt git-repo för labben. Den filen definierar en modul kallad Molbio som importerar din modul `F2`. Tanken är att du, om det passar dig, laddar molbio.hs i ghci och där kan testa din modul. Filen definierar följande dataset:

figur Ett mycket litet exempel på DNA-data, återfinns också i figur 1.

simple,sample Två små exempel på DNA-data.

foxp4 Sex proteiner från några ryggradsdjur.

fam1-fam5 Fem uppsättningar nukleära hormonreceptorer från ett flertal olika arter.

I filen finns också några funktioner för att köra snabba test av några av de olika funktioner du ska implementera i uppgifterna nedan. Mer information finner du i kommentarerna i filen.

2 Molekylära sekvenser

Det är främst två sorters molekyler som molekylärbiologer tittar på: DNA och proteiner. Båda har en linjär struktur som gör att man representerar dem som strängar, oftast benämnda "sekvenser" . DNA har välkänd struktur över fyra byggstenar, nukleotiderna `A`, `C`, `G` och `T`, och en DNA-sekvens kan därför se ut som t.ex. `ATTATCGGCTCT`. Proteinsekvenser är uppbyggda av 20 byggstenar, aminosyrorna, som brukar representeras med bokstäverna `ARNDCEQGHILKMFPSTWYV`.[1]

Längder på både DNA och proteiner kan variera starkt, men man måste kunna representera sekvenser som är från några tiotal symboler långa till över 10^4 symboler.

En vanlig operation på *par* av sekvenser är att beräkna deras *evolutionära avstånd*. Att bara räkna mutationer är också vanligt, men det måttet är inte proportionellt mot tiden, så därför används statistiska modeller för sekvensers evolution.

Enligt en känd och enkel modell som kallas *Jukes-Cantor* låter man avståndet $d_{a,b}$ mellan två DNA-sekvenser a och b (av samma längd) vara

$$d_{a,b} = -\frac{3}{4}\ln(1 - 4\alpha/3)$$

där α är andelen positioner där sekvenserna skiljer sig åt (det *normaliserade Hamming-avståndet* mellan sekvenserna). Formeln fungerar dock inte bra om sekvenserna skiljer sig åt mer än väntat, så om $\alpha > 0.74$ låter man $d_{a,b} = 3.3$.

Det finns en nästan likadan modell ("Poisson-modellen") för proteinsekvenser där man sätter avståndet till

$$d_{a,b} = -\frac{19}{20}\ln(1 - 20\alpha/19)$$

för $\alpha \leq 0.94$ och $d_{a,b} = 3.7$ annars. Parametrarna är alltså ändrade för att reflektera det större alfabetet hos proteinsekvenser.

[1] Borde inte aminosyrornas förkortningar ARNDCEQGHILKMFPSTWYV stå i bokstavsordning? Det gör de: A, R, och N representerar till exempel aminosyrorna Alanin, aRginin, och asparagiN.

Uppgifter

1. Skapa en datatyp `MolSeq` för molekylära sekvenser som anger sekvensnamn, sekvens (en sträng), och om det är DNA eller protein som sekvensen beskriver. Du behöver inte begränsa vilka bokstäver som får finnas i en DNA/protein-sträng.

2. Skriv en funktion `string2seq` med typsignaturen `String -> String -> MolSeq`. Dess första argument är ett namn och andra argument är en sekvens. Denna funktion ska automatiskt skilja på DNA och protein, genom att kontrollera om en sekvens bara innehåller A, C, G, samt T och då utgå ifrån att det är DNA.

3. Skriv tre funktioner `seqName`, `seqSequence`, `seqLength` som tar en `MolSeq` och returnerar namn, sekvens, respektive sekvenslängd. Du ska inte behöva duplicera din kod beroende på om det är DNA eller protein!

4. Implementera `seqDistance :: MolSeq -> MolSeq -> Double` som jämför två DNA-sekvenser eller två proteinsekvenser och returnerar deras evolutionära avstånd.

   Om man försöker jämföra DNA med protein ska det signaleras ett fel med hjälp av funktionen error.

   Du kan anta att de två sekvenserna har samma längd, och behöver inte hantera fallet att de har olika längd.

3 Profiler och sekvenser

Profiler används för att sammanfatta utseendet hos en mängd relaterade sekvenser. De är intressanta därför att man har funnit att om man vill söka efter likheter så är det bättre att söka med en profil, som sammanfattar liknande gener/proteiner, än att söka enskilda sekvenser. Vanligen används profiler för att sammanfatta viktiga delar av sekvenser, men i den här programmeringsövningen förenklar vi uppgiften till att arbeta med hela sekvenser.

En profil för en uppsättning DNA- eller protein-sekvenser är en matris $M = (m_{i,j})$ där element $m_{i,j}$ är frekvensen av bokstaven i på position j. Om alla sekvenser man studerar börjar med "A", då ska vi ha att $m_{A,0} = 1$. Om hälften av sekvenserna har "A" i position 1, och den andra hälften har "C", då ska vi ha $m_{A,1} = m_{C,1} = 0.5$. Figur 1 har ett exempel på hur man går från sekvenser till profil och exemplets data finns i molbio.hs.

```
ACATAA
AAGTCA
ACGTGC
AAGTTC
ACGTAA
```

$$\longrightarrow C = \begin{array}{c} \\ \text{A} \\ \text{C} \\ \text{G} \\ \text{T} \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 5 & 2 & 1 & 0 & 2 & 3 \\ 0 & 3 & 0 & 0 & 1 & 2 \\ 0 & 0 & 4 & 0 & 1 & 0 \\ 0 & 0 & 0 & 5 & 1 & 0 \end{pmatrix} \end{array} \longrightarrow M = \begin{array}{c} \\ \text{A} \\ \text{C} \\ \text{G} \\ \text{T} \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 1 & 0.4 & 0.2 & 0 & 0.4 & 0.6 \\ 0 & 0.6 & 0 & 0 & 0.2 & 0.4 \\ 0 & 0 & 0.8 & 0 & 0.2 & 0 \\ 0 & 0 & 0 & 1 & 0.2 & 0 \end{pmatrix} \end{array}$$

Figur 1: Ett exempel på hur fem DNA-sekvenser av längd sex omvandlas till en profil. Matrisen C räknar hur många gånger varje bokstav används i varje position. Matrisen M skapas från C genom att dela varje element i C med antalet sekvenser.

Det finns flera sätt man kan mäta avståndet (eller skillnaden) mellan två profiler. Ett sätt är att räkna ut den totala elementvisa skillnaden. Låt $M = (m_{i,j})$ och $M' = (m'_{i,j})$ vara två profiler över n positioner. Deras avstånd kan då skrivas

$$d(M, M') = \sum_{i \in \{\text{A,C,G,T}\}} \sum_{j=0}^{n-1} |m_{i,j} - m'_{i,j}|$$

```
nucleotides = "ACGT"
aminoacids = sort "ARNDCEQGHILKMFPSTWYVX"

makeProfileMatrix :: [MolSeq] -> ???
makeProfileMatrix [] = error "Empty␣sequence␣list"
makeProfileMatrix sl = res
  where
    t = seqType (head sl)
    defaults =
      if (t == DNA) then
        zip nucleotides (replicate (length nucleotides) 0) -- Rad (i)
      else
        zip aminoacids (replicate (length aminoacids) 0)    -- Rad (ii)
    strs = map seqSequence sl                               -- Rad (iii)
    tmp1 = map (map (\x -> ((head x), (length x))) . group . sort)
               (transpose strs)                             -- Rad (iv)
    equalFst a b = (fst a) == (fst b)
    res = map sort (map (\l -> unionBy equalFst l defaults) tmp1)
```

Figur 2: Hjälpkod för att konstruera profilmatris

Man summerar alltså över såväl alfabetet samt positionerna.

Om man skapar en profil för protein-sekvenser arbetar man med matriser som har 20 rader istället för 4, en rad för var och en av de tjugo aminosyrorna (`ARNDCEQGHILKMFPSTWYV`).

Uppgifter

1. Skapa en datatyp `Profile` för att lagra profiler. Datatypen ska lagra information om den profil som lagras med hjälp av matrisen M (enligt beskrivningen ovan), det är en profil för DNA eller protein, hur många sekvenser profilen är byggd ifrån, och ett namn på profilen.
2. Skriv en funktion `molseqs2profile :: String -> [MolSeq] -> Profile` som returnerar en profil från de givna sekvenserna med den givna strängen som namn. Som hjälp för att skapa profil-matrisen har du koden i figur 2. Vid redovisning ska du kunna förklara exakt hur den fungerar, speciellt raderna (i)-(iv). Skriv gärna kommentarer direkt in i koden inför redovisningen, för så här kryptiskt ska det ju inte se ut!
3. Skriv en funktion `profileName :: Profile -> String` som returnerar en profils namn, och en funktion `profileFrequency :: Profile -> Int -> Char -> Double` som tar en profil p, en heltalsposition i, och ett tecken c, och returnerar den relativa frekvensen för tecken c på position i i profilen p (med andra ord, värdet på elementet $m_{c,i}$ i profilens matris M).
4. Skriv `profileDistance :: Profile -> Profile -> Double`. Avståndet mellan två profiler M och M' mäts med hjälp av funktionen $d(M, M')$ beskriven ovan.

4 Generell beräkning av avståndsmatriser

Du har nu definierat två relaterade datatyper, `MolSeq` och `Profile`. De är i grunden olika, men en operation som att beräkna avståndet mellan två objekt, till till exempel, förenar dem även om de två implementationerna är olika. Eftersom vi har två skilda datatyper men med liknande funktioner, kan det vara praktiskt att skapa en typklass för att samla dem.

Vid studier av såväl molekylära sekvenser som profiler vill man ibland räkna ut alla parvisa avstånd och sammanfatta dessa i en *avståndsmatris*. Eftersom en typklass kan samla generella metoder kan man skriva en sådan funktion i typklassen istället för att implementera den särskilt för de två datatyperna.

En avståndsmatris kan representeras på många sätt, men i ett funktionellt språk är det ofta bra att ha en listrepresentation. Den representation du ska använda här är en lista av tripplar på formen (namn1, namn2, avstånd).

Uppgifter

1. Implementera typklassen `Evol` och låt `MolSeq` och `Profile` bli instanser av `Evol`. Alla instanser av `Evol` ska implementera en funktion `distance` som mäter avstånd mellan två `Evol`, och en funktion `name` som ger namnet på en `Evol`. Finns det någon mer funktion som man bör implementera i `Evol`?

2. Implementera funktionen `distanceMatrix` i `Evol` som tar en lista av någon typ som tillhör klassen `Evol`, och returnerar alla par av avstånd. Den här funktionen ska sedan automatiskt vara definierad för både listor av `MolSeq` och listor av `Profile`.

   Som nämndes ska avståndsmatrisen som returneras representeras som en lista av tripler på formen (namn1, namn2, avstånd). Denna ska komma i följande ordning: först kommer avstånden från det första elementet till alla andra. Sedan kommer avstånden från det andra elementet till alla andra utom det första (eftersom det redan angetts). Och så vidare. T.ex.: om vi har fyra `MolSeq`-objekt A, B, C, D och skickar in listan $[A, B, C, D]$, så ska `distanceMatrix` returnera listan

   $$[(A, A, \cdot), (A, B, \cdot), (A, C, \cdot), (A, D, \cdot), (B, B, \cdot), (B, C, \cdot), (B, D, \cdot), (C, C, \cdot), (C, D, \cdot), (D, D, \cdot)]$$

   (fast med samtliga “$\cdot$” utbytta mot avståndet mellan respektive objekt).

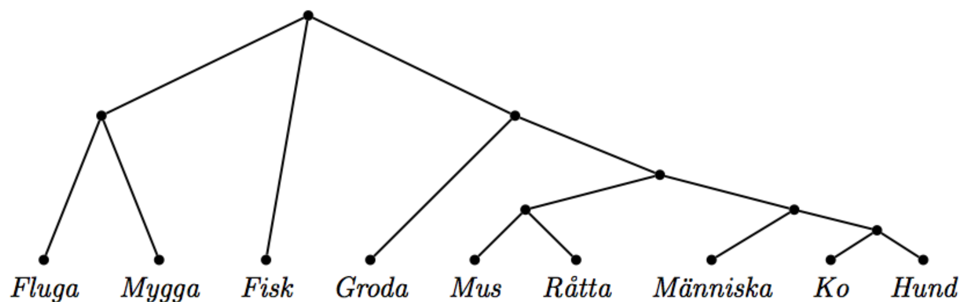
Labb F3: Evolutionära träd och I/O

Problem-ID på Kattis: kth:progp:f3

Det här är en laboration som dels testar din förmåga att arbeta funktionellt genom att överföra en abstrakt beskriven algoritm till ett funktionellt program, och dels låter dig skriva ett fullständigt Haskell-program med inläsning och utskrifter – operationer som har sido-effekter. Vi fortsätter arbeta med problem från molekylärbiologin och det är meningen att det du skriver ska bygga på laboration F2 – du kan antagligen med fördel använda din lösning från F2 utan ändringar, och skriva din lösning på denna labb i en ny modul F3 som importerar F2.

1 Bakgrund

Det finns många anledningar till att vara intresserad av hur arter och/eller gener har uppkommit, bland annat är det för många viktigt att helt enkelt förstå hur olika arter har uppstått. Inom medicin kan kunskap om geners utveckling ge kunskap om hur de fungerar och vilken funktion de har. En grundläggande fråga är då hur man rekonstruerar det evolutionära träd, en fylogeni, som gav upphov till sekvenserna? Figur 1 ger ett exempel på en fylogeni. Indata är alltså en mängd sekvenser, DNA eller protein, och utdata är ett träd där alla inre hörn har grad 3.



Figur 1: Exempelfylogeni för en hypotetisk gen funnen i fem arter. Lägg märke till att det här trädet inte är avsett att påstå något om var evolutionen började, dvs vilken punkt i trädet som är äldst. Man säger att trädet är orotat. (Med vår övriga kunskap om de olika arterna kan vi dock vara ganska säkra på att en eventuell rot skulle ligga på kanten mellan insekterna och fisken i den här specifika fylogenin).

2 Algoritmen Neighbor Joining

Den vanligaste och mest kända algoritmen för att återskapa träd givet avståndsdata är Neighbor Joining (NJ). Det är en iterativ algoritm som successivt väljer ut par av löv och slår ihop dem till ett nytt hörn. Vilket par man väljer är avgörande för att resultatet ska bli bra, och NJ garanterar faktiskt inte att det är det bästa trädet (i meningen: passar bäst med avstånden) som returneras: NJ är en girig heuristik.

Låt F_1 vara den mängd hörn som ska vara löv i det träd T vi bygger. Låt D_1 vara avståndsmatrisen över F_1. I varje iteration i av NJ kommer vi att välja ut två element a, b i F_i och kombinera ihop dessa till ett träd. Detta skapar en ny mängd F_{i+1} och avståndsmatris D_{i+1}.

Urvalsfunktionen S

Vi väljer de två element $a, b \in F_i$ som minimerar följande urvalsfunktionen S:

$$S_i(x, y) = (|F_i| - 2)D_i(x, y) - \sum_{z \in F_i} (D_i(x, z) + D_i(y, z))$$

Funktionen ser vid en första anblick en smula underlig ut, men man kan göra en tolkning av den. Den andra termen mäter hur långt bort övriga hörn ligger från x och y, och den första termen mäter hur nära

x och y ligger varandra, skalat med en faktor för att göra de två termerna jämförbara. Det S kan sägas välja ut är alltså de två hörn som ligger längst ifrån de andra.

Pseudokod

Vi använder oss av en förenklad version av NJ. Den som tittar på andra beskrivningar av NJ kommer att finna att steg 2d är lite hårigare än vad som ges här. Låt F_1 och D_1 vara indata.

1. $i \leftarrow 1$
2. Så länge $|F_i| > 3$:
   (a) Hitta det par $a, b \in F_i$ som minimerar $S_i(a, b)$
   (b) Skapa ett nytt träd T_i där träden a och b är barn till en nyskapad nod.
   (c) $F_{i+1} \leftarrow F_i \cup \{T_i\} \setminus \{a, b\}$ – Lägg till det nya trädet till F och ta bort de gamla
   (d) Skapa D_{i+1} från D_i enligt
   $$\begin{aligned} D_{i+1}(x,y) &= D_i(x,y) && \text{för } x, y \in F_{i+1} \setminus \{T_i\} \\ D_{i+1}(x,T_i) = D_{i+1}(T_i,x) &= \frac{D_i(x,a) + D_i(x,b)}{2} && \text{för } x \in F_{i+1} \setminus \{T_i\} \end{aligned}$$
   (e) $i \leftarrow i + 1$
3. Skapa ett nytt träd T där de tre kvarvarande träden i F_i är barn till en nyskapad nod
4. Returnera T

Exempelkörning

Antag att vi har de fem sekvenserna $a, b, \ldots, e$ som beskrivs i exempel-data 1 nedan. Använder vi vår kod från F2 för att beräkna avståndsmatrisen D_1 för dessa får vi:

$$D_1 \approx \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 0 & 0.304 & 0.824 & 0.520 & 0.824 \\ & 0 & 3.300 & 1.344 & 0.824 \\ & & 0 & 0.137 & 0.304 \\ & & & 0 & 0.137 \\ & & & & 0 \end{pmatrix} \end{array}.$$

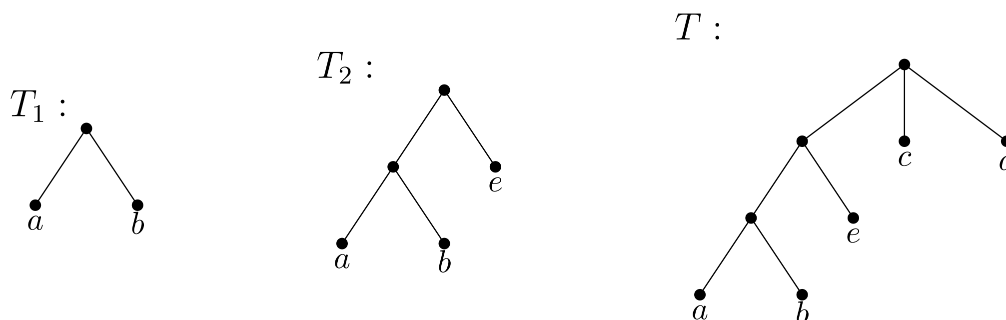
I algoritmens första iteration har vi "träden" $F_1 = \{a, b, c, d, e\}$, alla bestående av en enda nod. Urvalsfunktionen S_1 för första iterationen får följande värden:

$$S_1 \approx \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} b & c & d & e \end{array} \\ \begin{pmatrix} -7.331 & -4.565 & -3.049 & -2.089 \\ & -0.437 & -3.878 & -5.389 \\ & & -6.292 & -5.741 \\ & & & -3.816 \end{pmatrix} \end{array}.$$

Den minimeras alltså av paret (a, b), så vi skapar ett nytt träd T_1 som består av en ny nod med a och b som barn. Den nya avståndsmatrisen D_2 över de aktiva träden $F_2 = \{T_1, c, d, e\}$ och nya urvalsfunktionen S_2 kommer se ut som följer:

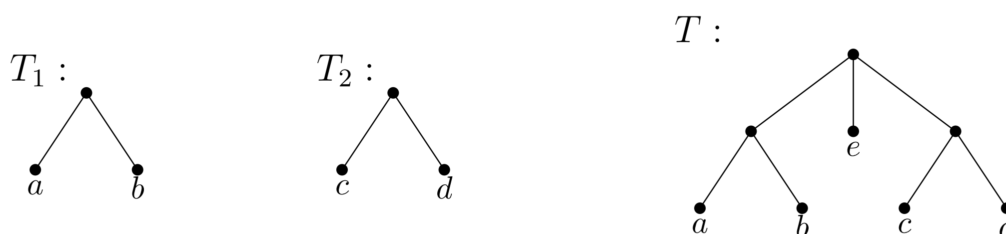
$$D_2 \approx \begin{array}{c} \\ T_1 \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{cccc} T_1 & c & d & e \end{array} \\ \begin{pmatrix} 0 & 2.062 & 0.932 & 0.824 \\ & 0 & 0.137 & 0.304 \\ & & 0 & 0.137 \\ & & & 0 \end{pmatrix} \end{array}, \quad S_2 \approx \begin{array}{c} \\ T_1 \\ c \\ d \end{array} \begin{array}{c} \begin{array}{ccc} c & d & e \end{array} \\ \begin{pmatrix} -2.197 & -3.159 & -3.435 \\ & -3.435 & -3.159 \\ & & -2.197 \end{pmatrix} \end{array}.$$

Urvalsfunktionen S_2 minimeras av endera paret (c, d) eller paret (T_1, e), och vi kan välja vilket som helst av dem. Låt oss säga att vi väljer det senare, dvs (T_1, e). Vi bildar då ett träd T_2 som består av en ny nod med T_1 och e som barn. Vi har nu bara tre aktiva träd kvar $F_3 = \{T_2, c, d\}$, och vi går till steg 3 i algoritmen och skapar trädet T bestående av en ny nod med T_2, c, och d som barn. Figur 2 illustrerar de olika träden som byggs upp under körningen.



Figur 2: Träden som byggs upp när algoritmen kör på första exempel-fallet

Hade vi i iteration 2 istället valt att para ihop (c, d) istället för (T_1, e) hade vi fått resultatet som visas i figur 3. Eftersom vi betraktar träden som orotade så är slutresultatet i själva verket samma träd som T från figur 2 även om det ritats annorlunda pga att vi kopplade ihop noderna i en annan ordning.



Figur 3: Alternativ körning på första exempel-fallet

3 Indata

Ditt program ska läsa en uppsättning DNA-sekvenser på standard input (till end of file). Vi kommer använda ett enkelt format där varje sekvens beskrivs av en rad med namnet på sekvensen följt av själva sekvensen, åtskiljda av mellanslag (varken namnet eller sekvensen kommer att innehålla några mellanslag).

Du kan anta följande om indata-strömmen:

- Alla sekvenser kommer att ha samma längd.
- Den innehåller minst 3 och högst 32 sekvenser.
- Varje sekvens är minst 1 och högst 1000 tecken lång.
- Namnen består bara av tecknen 'a'-'z', 'A'-'Z', och '0'-'9', och är mellan 1 och 15 tecken långa.
- Sekvenserna är DNA-sekvenser dvs består bara av tecknen ACGT.

4 Utdata

Ditt program ska skriva ut ett evolutionärt träd på standard output. Ett evolutionärt träd (eller fylogeni), skrivs ofta på ett format som kallas Newick. Det träd som visas i figur 2 kan då se ut så här:

```
(((a,b),e),c,d)
```

Så länge trädet saknar rot kan man skriva trädet på flera ekvivalenta sätt. Detta träd kan alltså även skrivas på följande sätt:

```
((a,b),e,(c,d))
(a,b,((c,d),e))
(b,a,((c,d),e))
```

och många fler. Ett löv i trädet representeras alltså av en sträng med lövets/sekvensens namn. Ett inre hörn i trädet representeras med hjälp av parenteser runt om två komma-åtskilda delträd. På den översta nivån använder vi parenteser runt tre delträd.

Alla ekvivalenta sätt att formatera trädet kommer att godkännas.

Tips

Börja med I/O-delen: skriva ett program som läser in indatasekvenserna, beräknar deras avståndsmatris (D_1) med hjälp av din lösning från F2, och sedan skriver ut avståndsmatrisen. Håll delarna av programmet som är inkapslade i `IO`-monaden minimala – du ska inte behöva gå in i din kod från F2 och ändra den!

I den här uppgiften är det bra att använda `ghc` för att kompilera Haskell-koden till ett körbart program (utöver att använda `ghci` för att testa olika funktioner i ert program). I terminalen i Unix kan man sedan använda omdirigering av standard input för att skicka en fil till programmet. Om ni t.ex. har sparat ned första exempel-fallet till filen `sample1.in` och kompilerat ert program till en körbar binär `Main` så ska ni i terminalen kunna skriva

```
> ./Main < sample1.in
```

för att köra ert program på det aktuella indatat.

Implementera sedan något sätt att representera träden som används i algoritmen, och utskrift av dessa. Modulerna Data.Either eller Data.Maybe kan vara behändiga här – kolla upp dessa!

Med kringarbetet avklarat kan man gripa sig an själva algoritmen. Du vill nog ha något sätt att representera aktuellt tillstånd i algoritmen (mängden F_i av träd vi har kvar att knyta ihop och avståndsmatrisen D_i), och en funktion som givet avståndsmatrisen D_i och två element $x, y \in F_i$ beräknar urvalsfunktionen $S_i(x, y)$.

Det är antagligen bra att följa den ovan givna algoritmens struktur ganska nära. Indata kommer vara relativt litet, högst 32 sekvenser, vilket innebär att din implementation inte behöver vara speciellt effektiv och du kan använda naiva lösningar till de olika delproblemen. Om du vill är det såklart tillåtet att göra en smartare, snabbare implementation – modulerna Data.Set och Data.Map kan vara en bra början till detta.

Sample Input 1

```
a AACCGGTT
b AACCGGGG
c AATTTTTT
d AACTTTTT
e AACTTTTG
```

Sample Output 1

```
(((a,b),e),c,d)
```

Sample Input 2

```
C CCCCCCCCCCCC
Go GGGGGGGGGGGG
Python GAGGCACCGGGG
Java ACACAACCCCCC
Prolog TTTCATCTTTTT
Haskell CGCGCGTTTGTT
```

Sample Output 2

```
((Prolog,Haskell),(Go,Python),(C,Java))
```

Sample Input 3

```
1 AAA
2 CCC
3 GGG
```

Sample Output 3

```
(1,2,3)
```