LINKÖPINGS UNIVERSITET

TDDI16: Datastrukturer och algoritmer

Lab 3: Ordkedjor

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2018

2018-05-14

1 Upplägg

Första delen av instruktionen, avsnitt 2 till 6, innehåller en fullständig beskrivning av problemet utan några konkreta lösningsidéer. Känner ni att ni har en bra idé redan efter att ha läst första delen kan ni börja programmera redan då.

Resten av instruktionen innehåller tips och lösningsidéer om ni känner att ni har kört fast. Om ni kör fast är det därför en bra idé att läsa igenom resten av instruktionen! Titta extra noga på avsnitt 9 om frågorna känns oklara, avsnitt 10 för lösningsidéer och avsnitt 11 för tips om implementationsdetaljer.

2 Vad är en ordkedja?

På korsordssidan i tidningar finns det ibland problem som kallas för *ordkedjor*. Dessa uppgifter brukar gå ut på att hitta ett antal ord som bildar en kedja mellan ett startord och ett slutord. För att orden ska bilda en kedja måste det gå att bilda nästa ord i kedjan genom att byta ut *en* bokstav. Exempelvis kan vi hitta en ordkedja mellan orden `aula` och `labb`:

`aula` → `gula` → `gala` → `gama` → `jama` → `jamb` → `jabb` → `labb`

Notera att det skiljer exakt en bokstav mellan varje ord i kedjan. Alla ord på vägen måste dessutom vara giltiga ord, annars skulle vi enkelt kunna lösa alla problem genom att bara byta ut en bokstav i taget.

En ordkedja är alltså en sekvens av ord där det skiljer en bokstav mellan varje ord i kedjan. Eftersom det ofta finns flera möjliga kedjor mellan två ord kommer vi i den här labben vara intresserade av den *kortaste* ordkedjan mellan två ord. Det innebär helt enkelt att vi vill hitta den kedjan som involverar så få ord som möjligt.

3 Uppgiften

Uppgiften är att skriva ett program som hittar ordkedjor i en ordlista. Programmet ska kunna svara på två typer av frågor:

1. Vilken är den kortaste ordkedjan från x till y?
2. Vilken är den längsta av alla kortaste ordkedjor som slutar med ordet x? (förklaras i avsnitt 9)

Programmet ska klara att svara på alla frågorna i var och en av testfilerna `swedish1.txt`, `swedish1.txt`, `english1.txt` och `english2.txt` på under ungefär 10 sekunder (det går att göra på under 2 sekunder utan större problem). Tänk också på vad som händer med din lösning om du ger den en ordlista som innehåller nästan alla kombinationer av fyra bokstäver (likt ordlistan `full.txt`). Fungerar din lösning då? Testerna i `full1.txt` ska gå att köra på under ungefär en minut.

4 Indata

Programmet ska läsa både ordlistan och frågorna från *standard in* (exempelvis med `cin`). Först ska ordlistan i sin helhet matas in, ett ord per rad. Ordlistan avslutas av en rad som innehåller tecknet `#`. Du kan anta att orden i ordlistan endast består av tecknen `a` till `z` och att endast gemener används, och att de inmatade orden finns i ordlistan. Du kan dessutom anta att orden alltid är 4 tecken långa om du vill.

Resterande rader i indatan beskriver de frågor som programmet ska svara på. Indatan avslutas med *end of file* (`Ctrl+D` i terminalen). Frågorna är en av de som beskrevs i avsnitt 3 representerade enligt nedan:

1. Om raden innehåller två ord separerade med ett blanksteg ska programmet hitta den kortaste ordkedjan från det första ordet till det andra ordet.
2. Om raden innehåller ett ord ska programmet hitta den längsta av alla kortaste ordkedjorna som slutar med ordet x.

Se avsnitt 6 för exempel på in- och utdata, samt avsnitt 8 för testningstips.

5 Utdata

För varje fråga i indatan ska programmet mata ut en eller två rader som svarar på frågan. Hur svaret ser ut beror på vilken fråga som fanns i indatan:

1. Om frågan bestod av två ord: Skriv ut en rad med första och sista orden i ordkedjan (samma ord som i frågan) följt av ett kolon och längden av ordkedjan. På nästa rad ska programmet skriva ut hela ordkedjan, inklusive orden i frågan. Orden i kedjan ska separeras av en pil (`->`). Om det finns fler än en ordkedja som är kortast spelar det ingen roll vilken av dem ditt program väljer.

   Om ingen ordkedja hittades ska i stället en rad skrivas ut, även denna med orden i frågan följt av ett kolon. I stället för antalet ord i ordkedjan ska i stället texten `ingen lösning` skrivas ut.

   Exempel för indata `aula jama`:

   ```
   aula jama: 5 ord
   aula -> gula -> gala -> gama -> jama
   ```

2. Om frågan bestod av ett ord: Skriv ut en rad med det sista ordet i den funna ordkedjan (samma ord som i frågan) följt av ett kolon och längden av ordkedjan. På nästa rad ska programmet skriva ut hela ordkedjan, inklusive ordet i frågan. Ordet i frågan ska vara det sista ordet som skrivs ut, och alla ord ska även här separeras av en pil (`->`). Om det finns fler än en möjlig ordkedja spelar det ingen roll vilken av dem ditt program väljer.

   Exempel för indata `aula`

   ```
   aula: 5 ord
   jama -> gama -> gala -> gula -> aula
   ```

Se avsnitt 6 för exempel på in- och utdata, samt avsnitt 8 för testningstips.

6 Exempel

Indata

```
aula
gula
gala
gama
jama
sten
#
aula jama
aula
sten aula
```

Utdata

```
aula jama: 5 ord
aula -> gula -> gala -> gama -> jama
aula: 5 ord
jama -> gama -> gala -> gula -> aula
sten aula: ingen lösning
```

Ytterligare testfall finns tillsammans med den givna koden. Tänk också på att det är viktigt att göra egna testfall för att hitta eventuella randfall i just er implementation!

7 Given kod

Filen `wordchain.cpp` innehåller kod som läser in och skriver ut data på det givna formatet. Den givna koden är bara tänkt som en hjälp att komma igång. Det är alltså helt okej att skriva hela lösningen från början om ni känner att det är enklare, eller om ni tycker att den givna koden är strukturerad på ett sådant sätt att ni inte kan implementera en effektiv lösning.

Det minimala som behöver göras i den givna koden är att implementera funktionerna `find_shortest` och `find_longest`. De funktionerna anropas av `read_questions` för att hitta ett svar till de två frågorna som beskrevs i avsnitt 3. Svaret på frågorna returneras i form av en ordkedja representerad i form av en `vector<string>`. Den givna koden kommer också att skicka med ordlistan till dessa funktioner i form av typen `Dictionary`. Från början är `Dictionary` ett alias för typen `vector<string>`. Dock kan det vara värt att fundera på hur ni vill representera ordlistan och ändra `Dictionary` därefter. Beroende på vilken typ ni väljer kan ni behöva modifiera koden i `main` för att omvandla ordlistan till den typ ni har valt innan den skickas in till `read_questions`.

Notera: Den givna koden är skriven för att vara lätt att läsa och lätt att förstå, inte för att vara så effektiv som möjligt. Om ni är intresserade av att placera er högt på topplistan i Kattis (se avsnitt 12) kan det vara värt att fundera på vad den givna koden gör ”i onödan” och förbättra den därefter, alternativt skriva en helt egen lösning med lärdomarna från den givna koden.

8 Tips för testning

I och med att all indata matas in från *standard in* är det ansträngande att själv skriva in ordlistan varje gång man kör programmet. Därför kan det vara lämpligt att göra något av följande för att förenkla testningen (`program` är namnet på ert program):

- Om ni vill skriva in testfallen manuellt:

  ```
  cat ordlista.txt - | program
  ```

Förklaring: Vi använder `cat` för att konkatenera filen `ordlista.txt` med *standard in* för `cat`, vilket gör att den indata vi själva skriver kommer programmet se efter ordlistan. Vi antar att filen `ordlista.txt` avslutas med ett `#` på en egen rad.

- Om ni har en fil med ordlistan och en fil med testfall (smidigt om ni har olika testfall för samma ordlista):

  ```
  cat ordlista.txt testfall.txt | program
  ```

  Förklaring: Vi använder återigen `cat` för att slå samman två filer. Den här gången läser vi inget från *standard in*, utan allt kommer från två filer. Återigen antar vi att `ordlista.txt` avslutas med `#`.

  Det är så här de testfallen som finns i mappen `tests/` fungerar. Filerna `swedish1.txt` och `swedish2.txt` ska användas tillsammans med `swedish.txt` och `english1.txt` och `english2.txt` ska användas tillsammans med `english.txt`.

- Om ni har en fil med både ordlistan och testfall (några av de givna testfallen):

  ```
  program < testfall.in
  ```

- För att ta tid på din lösning kan du använda kommandot `time`. Detta kör en kommandorad och skriver ut hur lång tid allt tog i slutändan:

  ```
  time program < testfall.in
  ```

  eller

  ```
  time cat ordlista.txt testfall.txt | program
  ```

  I slutändan skrivs något i stil med följande ut:

  ```
  real 0m1.086s
  user 0m1.087s
  sys 0m0.000s
  ```

  Det som är intressant är raden `real`, som anger hur lång tid körningen tog totalt.

För att enkelt testa ifall ditt program ger samma utdata som ett exempel kan du använda programmet `diff`:

```
program < testfall.in | diff - testfall.ans
```

Förklaring: Här använder vi en *pipe* för att ta utdatan från vårt program och skicka in det till `diff`, som jämför den med filen `testfall.ans`. Vi anger `-` som den första filen för att instruera `diff` att läsa den ena filen från *standard in*. Om `diff` inte säger något alls var utdatan identisk med den givna utdatan, annars kommer `diff` att säga vad som skiljde i ett format som påminner om hur Git visar skillnader mellan filer.

9 Vad är den ”längsta av alla kortaste vägar”?

Avsnitt 3 beskriver två frågor som programmet ska kunna svara på. Fråga nummer 1 bör inte vara så konstig, men fråga 2 kan vara värd att förtydliga. För att enklare förstå vad som menas kan man tänka: ”Vilken är den längsta ordkedjan som slutar med ordet x?” Den formuleringen är dock **inte** ekvivalent med fråga 2 ovan eftersom vi alltid kan förlänga en ordkedja genom att upprepa ord. Exempelvis skulle vi kunna förlänga ordkedjan:

$$\texttt{aula} \rightarrow \texttt{gula} \rightarrow \texttt{gala} \rightarrow \texttt{gama} \rightarrow \texttt{jama}$$

genom att upprepa `aula` och `gula`:

$$\texttt{aula} \to \texttt{gula} \to \texttt{aula} \to \texttt{gula} \to \texttt{gala} \to \texttt{gama} \to \texttt{jama}$$

Det här är fortfarande en giltig ordkedja eftersom det bara skiljer en bokstav mellan ett ord och nästa. Vi kan till och med repetera samma ord flera gånger för att ordkedjan ska bli godtyckligt lång:

$$\texttt{aula} \to \texttt{gula} \to \texttt{aula} \to \texttt{gula} \to \texttt{aula} \to \texttt{gula} \to \cdots \to \texttt{gala} \to \texttt{gama} \to \texttt{jama}$$

Det var nog inte den här typen av ordkedjor vi tänkte på när vi frågade efter den längsta ordkedjan. I och med att vi kan gå fram och tillbaka mellan ord på det här viset är det inte intressant att prata om en längsta ordkedja (det finns ingen unik längsta ordkedja, vi kan oftast konstruera flera ordkedjor av oändlig längd).

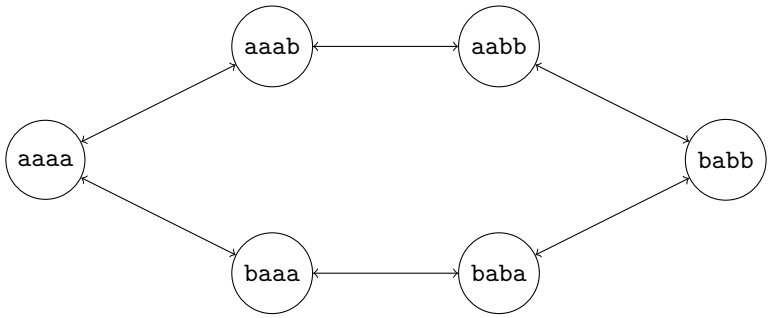
För att undvika det här problemet kan vi formulera om vår fråga så att repetition inte kan hända. Ett alternativ som dessutom bättre speglar det vi egentligen var ute efter från början är att bara undersöka de kortaste ordkedjorna. Vi kan tänka oss att vi hittar den kortaste ordkedjan från alla ord i ordlistan till ordet x, och sedan väljer vi den längsta. Det här är precis det som menas med ”den längsta av alla kortaste ordkedjor” i fråga 2. Man skulle också kunna se det som att vi hittar det ord som ligger ”längst ifrån” x, och den kortaste ordkedjan därifrån till x.

Det här är bara ett sätt att se problemet på. Se avsnitt 10 för ett annat sätt att se problemet. Om problemet känns klart kan du titta på avsnitt 11 för att få lite idéer som kan vara användbara när du implementerar din lösning.

10 Lösning med grafer

Ett sätt att hitta en effektiv lösning till problemet är att representera det i form av en graf. I det här fallet känns det naturligt att representera orden i ordlistan som noder i grafen. I det här problemet så är det viktigt att veta vilka ord som kan vara bredvid varandra i en ordkedja, så det känns naturligt att representera den här relationen mellan ord som bågar i grafen. Vi har alltså en båge mellan två noder ifall orden i noderna är lika så när som på en bokstav.

För att bättre se hur vi kan använda en graf för att hitta ordkedjor i en ordlista, antag att vi har följande ord i ordlistan: `aaaa`, `aaab`, `aabb`, `baaa`, `baba` och `babb`. Då kan vi bilda följande graf:



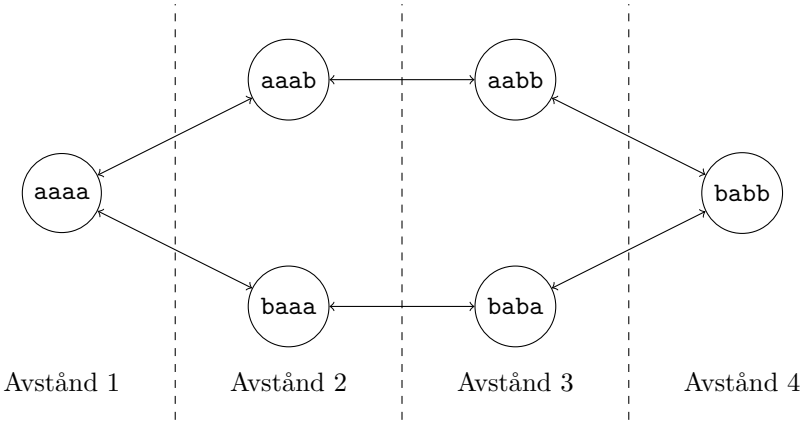
Här kan vi ganska enkelt se vilka ord som kan vara bredvid varandra i en ordkedja. Exempelvis så ser vi att ordet `aaaa` kan vara bredvid `aaab` och `baaa` och inga andra. Med hjälp av grafen blir det också enkelt att hitta ordkedjor! Om vi vill hitta den kortaste ordkedjan mellan orden `aaaa` och `aabb` kan vi helt enkelt hitta den kortaste vägen mellan de två noderna som motsvarar orden i grafen. I det här fallet kan vi enkelt se att lösningen blir: `aaaa` → `aaab` → `aabb`.

I och med att vi har lyckats representera vårt problem som en graf blir det inte bara enklare för oss människor att hitta lösningar på problemet, det finns också ett stort antal algoritmer som vi kan använda för att bearbeta grafer. Vi såg tidigare att frågan ”Vad är den kortaste ordkedjan mellan x och y?” kan formuleras som ”Vad

är den kortaste vägen mellan x och y?", vilket innebär att vi kan lösa problemet genom att implementera en lämplig grafalgoritm. Det finns ett antal algoritmer som hittar den kortaste vägen mellan två noder, exempelvis *bredden-först* och *Dijkstra*. I och med att alla bågar är lika dyra så är en bredden-först-sökning lämplig.

Fundera på om du verkligen behöver bygga upp hela grafen innan du börjar söka i den. De flesta grafsökningsalgoritmer behöver inte känna till hela grafen på en gång. De nöjer sig med att ha ett sätt att ta reda på alla grannar till en specifik nod. Det gör att man oftast klarar sig med att bara implementera en funktion som hittar grannarna snarare än att konstruera en hel graf i minnet.

Vi kan också använda en liknande idé för att hitta en lösning till den andra frågan i avsnitt 3: "Vilken är den längsta av alla kortaste ordkedjor som slutar i x?". Om vi tittar närmare på grafen vi ritade tidigare så kan vi se att noderna är ordnade efter deras avstånd till `aaaa`:



Här kan vi snabbt se hur lång den kortaste vägen från `aaaa` till alla andra noder är. Med den här informationen kan vi snabbt se att den längsta av de alla möjliga kortaste vägar som slutar i `aaaa` är noden `babb`, som är 4 ord bort, och att kortaste vägen är antingen `babb` → `baba` → `baaa` → `aaaa` eller `babb` → `aabb` → `aaab` → `aaaa`. I fall som detta kan ditt program svara med vilken som helst av ordkedjorna. I de givna testfallen finns det alltid en unik längsta kortaste ordkedja. Med det här synsättet kan vi formulera frågan som: "Vilket ord är längst bort ifrån x?" Den här frågan kan vi också enkelt och snabbt svara på med hjälp av en bredden-först-sökning, men exakt hur är upp till er att fundera på!

11 Att tänka på

- Hur representerar ni problemet? Fundera på om ni verkligen behöver lagra hela grafen för att kunna söka i den, eller om ni kan generera de bitar av grafen som behövs när de behövs. Även de bitar som har genererats kanske inte behöver lagras, utan kan genereras på nytt om de skulle behövas igen.
- Fundera på hur ni hittar alla grannar till ett ord. Det finns i allmänhet två sätt att göra detta på:
    1. Jämför det nuvarande ordet med vart och ett av orden i ordlistan och se vilka som skiljer med exakt en bokstav. Exempelvis börjar vi med `aula` och jämför det med vart och ett av orden i ordlistan. Vi börjar med `aber`, och ser att det skiljer för mycket, sedan `acne`, vilket också skiljer för mycket. Efter ett tag hittar vi `gula`, vilket bara skiljer med en bokstav. På detta sättet fortsätter vi genom hela ordlistan.
    2. Utgå från det nuvarande ordet och generera alla möjliga grannar genom att byta ut varje tecken mot tecknen `a` till `z` och sedan se vilka av dessa som faktiskt fanns i ordlistan. Exempelvis utgår

vi från `aula` och genererar orden `bula`, `cula`, ... `aala`, `abla`, ... och så vidare, och ser vilka av dessa som finns i ordlistan.

Vilket av sätten är bäst? Ordlistorna `swedish.txt` och `english.txt` har båda ungefär 2500 ord. Hur många ord måste de två alternativen undersöka för den här storleken av ordlista, och vilket alternativ är därför bäst för den här storleken på ordlistor (dvs. relativt stora ordlistor)?

- Fundera på vilken data ni lagrar och hur ni vill använda den datan. Använd den informationen för att välja en lämplig datastruktur som är bra på de viktiga operationerna.
- Hur många sökningar behöver göras för att hitta ett svar till fråga nummer 2: ”Vilken är den längsta av alla kortaste ordkedjorna som slutar i ord x?” Räcker det med en?
- Vad har er lösning för tidskomplexitet? Finns det några speciella fall då er lösning är bättre eller sämre?

12 Topplista i Kattis

Notera: Det är helt frivilligt att försöka klara testerna i Kattis för att komma med på topplistan.

Om du vill se hur snabb din lösning är i förhållande till andra som har löst samma problem kan du ladda upp din kod i Kattis: `https://liu.kattis.com/problems/liu.dalg.wordchain`. Problembeskrivningen säger att du ska ladda upp Javakod, men detta problem går utmärkt att lösa i C++ också, även om beskrivningen inte säger det.

Kattis är ett onlinesystem som används för att bedömma lösningar till problem av den här typen. Ofta ska programmen man laddar upp läsa indata från *standard in*, beräkna någon lösning till ett problem, och sedan skriva resultate till *standard out*. Kattis kommer sedan titta på utdatan och jämföra den med det förväntade resultatet (lite liknande vad `diff` gör) för att se om det var rätt. Det är därför viktigt att följa problemspecifikationen till punkt och pricka! Programmet får inte heller köra längre än den givna tiden (20 sekunder i detta fall) och förbruka mer minne än vad som anges (1 GB). Om allt går väl kommer Kattis att ge din lösning godkänt, och om den är tillräckligt snabb så får du komma med i topplistan!

Om du tycker problem av den här typen är roliga finns det många fler problem i Kattis (de allra flesta kan lösas i C++). IDA anordnar också programmeringstävling som heter IMPA. Den pågår hela tiden under terminerna, och brukar ge presentkort till de som placerar sig bäst i varje deltävling. IMPA har också ett handikappssystem som gör att de som just har börjat får mer poäng för enklare uppgifter, så att alla ska ha en chans. Titta på `http://www.ida.liu.se/impa/` för mer information.

Utöver detta så anordnas det varje höst en nationell programmeringstävling som heter NCPC där deltagarna ungefär 6 timmar på sig att lösa så många uppgifter som möjligt. Uppgifterna kommer ha liknande karaktär som denna, och en onlinedomare (i stil med Kattis) brukar användas. Placerar man sig bra i den tävlingen kan man gå vidare till internationell nivå. Att ha placerat sig bra i tävlingar som denna brukar företag se som en bra merit.

12.1 Skillnader i Kattis

De testfall som Kattis använder utgår från en problembeskrivning som skiljer sig lite från det som beskrivs här. Först och främst måste ditt program hantera tecknen `å`, `ä`, `ö` och `é` utöver `a-z`, vilket är lite krångligt i C++. Problemet beror på att C++ representerar strängar som UTF-8, vilket innebär att vissa tecken (exempelvis å, ä, ö och é) representeras som två eller flera bytes i minnet, vilket gör att följande inte kommer fungera som man tror:

```
string x{"ö"};
cout << x.size() << endl; // Skriver ut 2, inte 1.
```

```
x[0] = 'a';
cout << x << endl;
```

I filen `kattis.h` finns en speciell strängtyp implementerad som löser problemet. Mer detaljer finns i filen.

Om ni vill testa att ni hanterar svenska tecken korrekt kan ni använda de testfall som finns i kursen TDDC91: `https://www.ida.liu.se/~TDDC91/current/info/code/lab4/testfall/`