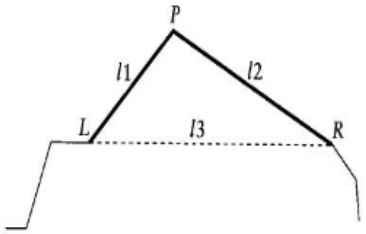
Laboration 2

TDA416/DIT721, lp3 2016

Uppgiften går ut på att förenkla 2-dimensionella konturer enligt följande princip. Idén är tagen från programmeringsuppgift 7, kursboken (Koffman&Wolfgang), kapitel 2. En kontur representeras av en punklista som utgörs av lista av talpar, där varje talpar motsvarar x- och y-koordinaterna för en punkt på konturen. Punkterna är ordnade i listan så att de följer konturen från en start- till en slut-punkt. Ert program ska förenkla en kontur genom att ta bort de minst viktiga punkterna.

Vi kan ge varje punkt, P, ett värdemått på hur viktig den är för konturen genom att titta på dess grannar L och R. Vi beräknar distanserna $L - P$, $P - R$ och $L - R$ och kallar dem $l1$, $l2$ och $l3$. Definiera sedan värdemåttet som $l1 + l2 - l3$. Värdemåttet är alltid icke-negativt och är det 0 så märks det inte alls om man tar bort P.



Programmet ska, givet en kontur med n punkter ta bort dem med minst värdemått så att endast k punkter kvarstår. Dessa k punkter är beräkningens resultat. Det är inte de $n - k$ stycken minst värdefulla punkterna i den ursprungliga punktlistan som ska bort. Man ska ta bort en punkt i taget, den minst värdefulla, sedan räkna om värdemåttet för punkterna, och ta bort den punkt som nu har minst värde o.s.v..

Vi gör en förenkling; listan med punkter är inte sluten d.v.s. det finns en första och en sista punkt (första och sista punkterna i listan) och dessa kan inte tas bort.

Metoden som gör detta ska ha följande signatur:

```
public static double[] simplifyShape(double[] poly, int k)
```

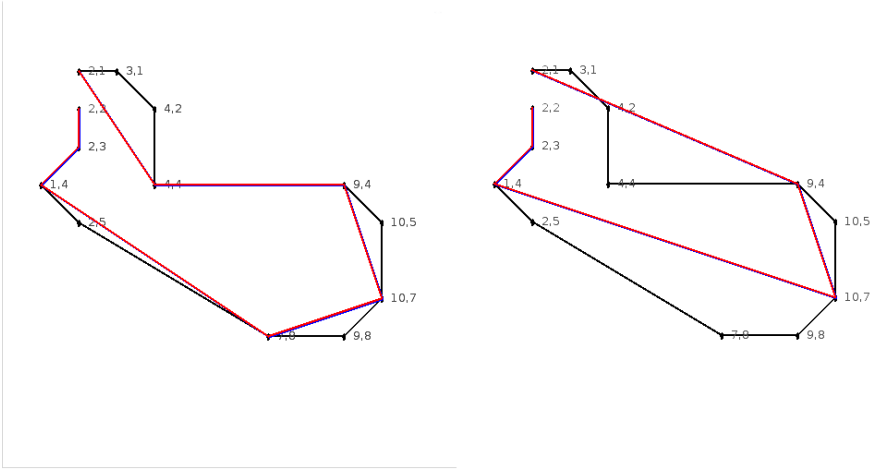
`poly` motsvarar en punktlista med n punkter och innehåller $2 \cdot n$ element. `poly[0]` och `poly[1]` är x- och y-koordinater för första punkten, `poly[2]` och

`poly[3]` x- och y-koordinater för andra punkten o.s.v.. `poly.length` är minst 4 (d.v.s. listan har minst två punkter). `poly` ska lämnas oförändrad av metoden. `k` är antalet punkter som ska finnas i slutändan. `k` är minst 2 och mest n. Metoden returnerar en ny array som innehåller de punkter som inte tas bort av algoritmen. Representationen i denna array är densamma som i `poly`. Den returnerade arrayen ska alltså ha längden $2 \cdot$`k` och ska förhålla sig till `poly` enligt beskrivningen ovan.

Ni ska implementera an lösning till detta problem på två olika sätt. Till laborationen hör några filer som ni ska använda er av. `Lab2.java` implementerar ett program, som anropas t.ex. så här:

```
java Lab2 -k8 -w12 -h12 < fig1.txt
```

`-k8` säger att det ska finnas 8 punkter kvar. `-w12 -h12` innebär att ytan $0 < x < 12$ och $0 < y < 12$ visualiseras. Tänk på att era båda implementeringar (röd och blå linje) bör sammanfalla. Programmet anropar era implementeringar som antas ligga i klasserna `Lab2a` och `Lab2b`. Det använder också `DrawGraph.java` för att visualisera konturerna före och efter förenklingen. `fig1.txt` är en exempelkontur som medföljer. Det finns en kontur till, `fig2.txt` som är liknande. I denna har positionerna för punkterna valts så att aldrig två punkter ska ha samma värdemått. Detta för att korrekta implementeringar inte ska kunna ge olika resultat. Följande figur visar resultatet för `fig2.txt` med $k = 8$ resp. $k = 6$.



Uppgift a

Implementera `simplifyShape` genom att representera aktuell punktlista med en **array** enligt följande algoritm:

```
while antalet punkter är större än k
  beräkna värdemåttet av varje intern punkt (ej ändpunkterna)
  tag bort den minst betydelsefulla
end while
```

Kom ihåg att aldrig ta bort ändpunkterna. Beräkna tidskomplexiteten för er implementering. Denna ska uttryckas som en funktion av enbart n (ni kan räkna med att `k` är minimal, vilket utgör det värsta fallet).

Uppgift b

Algoritmen ovan gör onödigt arbete. När en punkt plockats bort räknas värdet för alla resterande punkter om, fast det bara ändras för den borttagna punktens närmaste två grannar. Utgå nu istället från följande algorithm:

```
beräkna värdemåttet för varje intern punkt
while antalet punkter är större än k
  tag bort den minst betydelsefulla
  beräkna om den borttagna punktens närmsta grannars värdemått
end while
```

För att dra nytta av denna optimering ska ni använda en dubbellänkad lista istället för array för att internt representera aktuell punktlista. Att använda Java API:s klass LinkedList är ingen bra idé för den ger inte användaren tillgång till noder.

Förklara varför Java API:s LinkedList inte ger en förbättrad komplexitet jämfört med uppgift a.

Implementera istället en egen dubbellänkad lista som har den rätta funktionaliteten så att tillgång till ett element och dess grannar går på konstant tid genom att användaren har tillgång till noderna. För detta använd det gränssnitt som definieras i medföljande filen `DLList.java`, d.v.s. fyll i de ofärdiga bitarna i denna fil. Använd sedan `DLList` för att implementera algoritmen. Observera att det gränssnitt som är definierat i `DLList.java` inte ska ändras, t.ex. genom att låta generiska typen `E` utöka `Comparable`. Tillgång till jämförare för elementen som stoppas i prioritetskön ska ordnas på annat sätt.

Om man letar igenom hela listan för att hitta den med minst värdemått tar detta linjär tid i varje iterering och inget (Ordo-mässigt sett) har blivit bättre. För att undvika detta ska ni också använda en prioritetskö för att hålla koll på aktuell punkt med minst värdemått. Till detta ska ni använda java API:s implementering, `PriorityQueue`. I Oracles dokumentation om klassen (i översta avsnittet, konstruktorerna) kan ni läsa om olika metoders komplexitet. Viktigt är att prioritetskön måste innehålla noder till den länkade listan för att ni snabbt ska hitta rätta element att ta bort och modifiera grannarna för. För att använda `PriorityQueue` måste ni implementera `Comparable` eller `Comparator` för den elementtyp ni använder i kön så att den placerar punkter med minst värdemått först.

Tänk på att javas `PriorityQueue` inte har något sätt att uppdatera värdet av ett element i listan. Det skulle behövas för när en punkt tagits bort ändras värdemåttet för dess grannar. Man får lösa detta på något annat sätt. Ett sätt är att ta bort kö-elementet med det gamla värdemåttet (d.v.s. använda `remove`) och lägga in ett nytt element.

Beräkna tidskomplexiteten även för denna implementering. Om ni får samma komplexitet som i uppgift a, vad är det för del av algoritmen som kan ses som orsaken till detta? Hur skulle man kunna komma runt det? Beskriv vad ni

kommer fram till i text-dokumentet. Om ni hittar en möjlig förbättring är det frivilligt att implementera den i er kod.

Vad ska vi skicka in till Fire?

- `Lab2a.java` och `Lab2b.java` som innehåller implementeringarna av `simplifyShape` enligt uppgift a resp. b.
- `DLList.java` som `Lab2b.java` använder sig av. Alla deklarerade metoder ska vara implementerade, inte bara de som ni använder i `Lab2b.java`.
- `lab2.pdf` där ni anger och motiverar (i vanligt språk räcker,d.v.s. handviftning) tidskomplexiteten för de två algoritmerna, samt svaret på de saker ni uppmanas förklara i uppgifterna.

Lösa filer, inga `.zip`, `.tar` eller dyligt.