TDA416 VT16 Grupp 4
Emil Axelsson, 951114-2854, emilaxe@student.chalmers.se
Michelle Tran Luu, 940124-0024, luum@student.chalmers.se

Datastrukturer och Algoritmer

Laboration 2

Tidskomplexitet för deluppgift 2a:

I laboration 2a finns en nästlad loop (se första kodstycket) och en loop inuti metoden removePoint (se andra kodstycket).

```
while(copy.length/2 > k){

  // copy.length-5 since point R's Y value is in i+5
  for(int i = 0; i < copy.length-5; i = i + 2){

    //Gets the points for L,P,R
    double [] L = {copy[i], copy[i+1]};
    double [] P = {copy[i+2], copy[i+3]};
    double [] R = {copy[i+4], copy[i+5]};


    l1 = Math.sqrt(Math.pow(L[0]-P[0], 2) + Math.pow(L[1]-P[1], 2));
    l2 = Math.sqrt(Math.pow(P[0]-R[0], 2) + Math.pow(P[1]-R[1], 2));
    l3 = Math.sqrt(Math.pow(L[0]-R[0], 2) + Math.pow(L[1]-R[1], 2));

    // calculates the value
    currentValue = Math.abs(l1+l2-l3);

    if(lowestValue > currentValue || i == 0){
      lowestValue = currentValue;

      // position of the point P with lowest value
      lowValPosition = i+2;
    }
  }

  //
  copy = removePoint(copy, lowValPosition);
}
-------------------------------------------------------------
for(int i = 0; i < array.length; i++){

  if(!(i == position || i == (position + 1))){

    newArray[pos] = array[i];
    pos++;
```

```
  }

}
```

Den yttre loopen kommer iterera n gånger och den inre kommer iterera (n-5) gånger. Sedan kommer metoden removePoint anropas och i denna metoden finns det en loop som itereras n gånger. Detta ger oss således en tidskomplexitet på $O(n^2)$.

$$n * ((n-5)+n) = n^2 - 5 * n = O(n^2)$$

Tidskomplexitet för deluppgift 2b:

Deluppgift 2b startar med

```
for (int i = 4; i < poly.length; i = i + 2) {
  double[] point = {poly[i], poly[i + 1]};

  currentNode = list.insertAfter(point, node);
  priorityQueue.add(node);
  node = currentNode;
}
```

Denna loopas (n - 4) gånger. Inuti loopen anropas PriorityQueue's metod add, som enligt Javas API har komplexiteten log(n). Komplexiten blir då n*log(n)

```
while (priorityQueue.size()+2 > k) {
  DLList.Node head = priorityQueue.remove();
  list.remove(head);
  if (head.getPrev().getPrev() != null) {
     priorityQueue.remove(head.getPrev());
     priorityQueue.add(head.getPrev());
  }
  if (head.getNext().getNext() != null) {
     priorityQueue.remove(head.getNext());
     priorityQueue.add(head.getNext());
  }
}
```

Nästa loop itereras (n + 2) gånger, Den kallar sedan på remove från PriorityQueue som enligt Java API har komplexiteten log(n). För att sedan uppdatera kön tar vi bort de element bredvid det elementet som nyligen tagits bort, och lägger till det igen. Metoden add(Object) har som sagt komplexiteten log(n), och metoden remove(Object) har linjär komplexitet, dvs. n. Loopen har alltså totala komplexiteten:

Emil Axelsson, 951114-2854, emilaxe@student.chalmers.se
Michelle Tran Luu, 940124-0024, luum@student.chalmers.se

$$n(\ n + log(n) + n + \ log(n)\ + n\ + log(n))\ =\ O(n^2)$$

Den sista loopen innehåller endast konstanta operationer. Komplexiteten är då n

```
while (curr != null) {
  polyArray[i] = curr.elt[0];
  polyArray[i + 1] = curr.elt[1];
  i = i + 2;
  curr = curr.getNext();

}
```

Den totala komplexiteten beräknas då

$$n * log(n)\ +\ n^2\ +\ n\ =\ O(n^2)$$

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

LinkedList ger inte en förbättrad komplexitet eftersom vid borttagning av en nod från listan så behövs prioritetskön uppdateras. För att prioritetskön ska uppdateras krävs det att noderna till vänster och höger om själva noden tas bort och sedan läggas tillbaka. Detta kan förbättras genom att använda oss av binary heap. Eftersom Binary heap har komplexitet O(log(n)) för remove(object).

En annan anledning är att man inte har tillgång till en individuell nod utan man måste iterera igenom hela listan för att hitta just den noden. Man kan lösa detta problemet genom att markera