## Lösungsidee

Zunächst werden alle Punkte eingelesen und in einer zweidimensionalen Liste gespeichert. Felder sind hierfür ungünstig, da man nicht weiß, wie viele Reihen und Lücken es gibt. Jede Lücke (von mir in der Implementierung Punkt genannt) einer Reihe wird in einer Liste gespeichert. Koordinaten mit dem x-Wert Null werden nicht in der Liste berücksichtigt, da Max und Minnie bereits bei Null starten. Die y-Koordinaten des Starts der Hunde wird gespeichert.
Jeder Punkt hat Koordinaten und einen Wahrheitswert, ob die Lücke groß ist.
Nun wird in der ersten Reihe für jede Lücke berechnet, wie lange Max braucht und dies im jeweiligen Punkt gespeichert. Die Berechnung funktioniert mit dem Satz des Pythagoras. Für die zweite Reihe wiederholt sich das von jeder großen Lücke der ersten Reihe aus. Der schnellste Wert wird gespeichert. Für alle weiteren Reihen geschieht das Gleiche. Nun weiß man für jede Lücke, wie schnell Max dort sein könnte.
Für Minnie wird nun rekursiv für jede Lücke berechnet, ob sie dort schneller ist als Max. Wenn das der Fall ist, wird für alle Lücken in der nächsten Reihe untersucht, ob sie wieder schneller ist. Es muss nicht extra berechnet werden, ob Max Minnie zwischen zwei Buhnenreihen fangen könnte, weil er ihn in diesem Fall dank seiner Schnelligkeit sicher auch bei einer Lücke fangen kann.
Hierbei werden die Zeit aus den vorherigen Reihen verwendet und dazu die neue Zeit addiert. Wenn sie irgendwann ein letzte Lücke schneller erreicht als Max, kann sie sicher entkommen und dieser Weg wird ausgegeben.

## Beispiele

Dies sind die Lösungen zu den Beispielen von „buhnenrennen1.txt“ bis „buhnenrennen14.txt“

(70/45.9)(140/42.2)(210/49.4)
(70/14.1)(140/28.0)(210/67.9)(280/64.5)(350/56.55)(420/50.3)(490/97.85)
(70/55.85)(140/76.8)(210/100.95)(280/167.65)(350/141.7)(420/101.65)(490/145.1)
(70/15.65)(140/16.55)(210/25.55)(280/34.4)(350/40.05)(420/58.3)(490/53.5)
(70/28.25)(140/63.2)(210/42.05)(280/34.95)(350/12.85)(420/28.1)(490/27.5)
(70/119.4)(140/119.25)(210/121.45)(280/123.45)(350/106.95)(420/141.45)(490/118.9)(560/118.1)(630/84.6)(700/116.5)(770/126.05)(840/123.0)(910/131.1)(980/91.1)(1050/71.55)(1120/11.45)(1190/26.1)
(70/173.5)(140/149.4)(210/191.9)(280/187.7)(350/256.5)(420/269.55)(490/175.25)(560/184.5)(630/152.3)(700/91.4)(770/71.0)(840/111.75)(910/137.95)(980/131.55)(1050/162.85)(1120/173.95)(1190/146.6)(1260/120.5)(1330/134.95)(1400/176.75)
(70/254.0)(140/260.0)(210/225.3)(280/209.0)(350/189.8)(420/85.65)(490/69.95)(560/57.0)(630/56.35)(700/66.8)(770/52.5)(840/50.2)(910/48.3)(980/33.5)(1050/114.3)(1120/35.75)(1190/126.55)(1260/103.75)(1330/129.55)(1400/142.75)(1470/121.75)(1540/244.4)(1610/175.6)
(70/206.3)(140/231.85)(210/287.5)(280/300.05)(350/311.75)(420/277.05)(490/303.55)(560/277.45)(630/310.25)(700/321.15)(770/338.15)(840/302.45)(910/297.5)(980/277.45)(1050/243.25)(1120/231.7)(1190/225.65)(1260/213.75)(1330/186.4)(1400/151.75)(1470/192.5)(1540/165.7)(1610/192.95)(1680/194.25)(1750/285.65)(1820/347.85)
(70/287.65)(140/317.4)(210/318.15)(280/268.25)(350/206.5)(420/165.25)(490/282.25)(560/270.75)(630/247.15)(700/151.05)(770/264.65)(840/248.0)(910/166.45)(980/34.2)(1050/30.15)(1120/64.4)(1190/109.35)(1260/79.2)(1330/57.4)(1400/157.85)(1470/60.7)(1540/79.25)(1610/121.25)(1680/121.8)(1750/184.7)(1820/186.85)(1890/163.85)(1960/112.3)(2030/12.35)
(70/66.35)(140/71.9)(210/91.2)(280/85.95)(350/141.25)(420/105.85)(490/35.45)(560/23.4)(630/71.85)(700/94.7)(770/57.45)(840/14.4)(910/13.6)(980/41.85)(1050/64.9)(1120/71.6)(1190/51.7)(1260/90.0)(1330/192.6)(1400/116.7)(1470/130.75)(1540/108.75)(1610/70.85)(1680/109.15)(1750/78.55)(1820/43.8)(1890/53.95)(1960/46.15)(2030/54.65)(2100/120.7)(2170/128.85)(2240/101.1)
(70/158.45)(140/178.4)(210/173.15)(280/383.5)(350/437.6)(420/327.0)(490/197.5)(560/155.8)(630/81.55)(700/160.2)(770/93.25)(840/94.2)(910/71.0)(980/61.05)(1050/12.95)(1120/21.15)(1190/19.1)(1260/75.15)(1330/90.5)(1400/129.7)(1470/257.45)(1540/294.95)(1610/198.6)(1680/285.2)(1750/261.4)(1820/206.6)(1890/213.85)(1960/150.5)(2030/116.65)(2100/99.95)(2170/108.8)(2240/186.6)(2310/183.8)(2380/200.2)(2450/249.3)
(70/55.4)(140/83.9)(210/12.75)(280/133.65)(350/105.2)(420/174.85)(490/358.85)(560/414.35)(630/449.45)(700/427.75)(770/440.0)(840/442.7)(910/455.4)(980/415.3)(1050/395.0)(1120/391.4)(1190/422.0)(1260/431.5)(1330/415.85)(1400/469.45)(1470/462.0)(1540/443.75)(1610/461.15)(1680/473.2)(1750/328.0)(1820/380.35)(1890/369.15)(1960/257.1)(2030/241.55)(2100/233.0)(2170/349.65)(2240/338.25)(2310/347.35)(2380/382.6)(2450/335.15)(2520/349.05)(2590/339.0)(2660/266.75)
(70/217.6)(140/128.85)(210/12.6)(280/44.6)(350/137.05)(420/458.05)(490/438.15)(560/361.05)(630/386.8)(700/417.7)(770/476.7)(840/451.85)(910/404.55)(980/437.0)(1050/447.35)(1120/415.4)(1190/405.9)(1260/400.75)(1330/416.6)(1400/380.6)(1470/227.75)(1540/154.75)(1610/176.85)(1680/173.65)(1750/151.1)(1820/131.75)(1890/147.45)(1960/280.25)(2030/226.4)(2100/236.15)(2170/236.65)(2240/141.6)(2310/104.15)(2380/102.85)(2450/78.6)(2520/105.8)(2590/30.85)(2660/92.5)(2730/107.4)(2800/86.0)(2870/39.55)

## Quelltext

Berechnung der Maxzeiten:

```
   private void calculateMaxTimes()
   {
//Berechnung der Zeiten vom Start zu den Lücken der ersten Reihe
      for(int i=0; i<points.get(0).size(); i++){
         points.get(0).get(i).setMaxTime(calculateDistance(maxStart,points.get(0).get(i).getHeight())/(30/3.6));
      }
//Berechnung für alle anderen Punkte wie oben beschrieben
      for(int i=1; i<points.size(); i++){
         for(int j=0; j<points.get(i-1).size(); j++){
//Wenn Maxi durch die Lücke passt
            if(points.get(i-1).get(j).isBig()){
               for(int k=0; k<points.get(i).size(); k++){
//Berechnung der Zeit: Distanz / (30/3,6), da km/h vorzugsweise in m/s umgewandelt wird. Auch wenn es für die Lösung keinen Unterschied macht, kann man so ggf. die Zeit in Sekunden auslesen
                  double time = points.get(i-1).get(j).getMaxTime()+
                     calculateDistance(points.get(i-1).get(j).getHeight(),points.get(i).get(k).getHeight())/(30/3.6);
//Wenn der neue Weg zu dem Punkt schneller ist
```

```
                if(points.get(i).get(k).getMaxTime()==0 || points.get(i).get(k).getMaxTime()>time)
                points.get(i).get(k).setMaxTime(time);
            }
        }
    }
  }
}
```

Berechnung der Distanz zwischen zwei Punkten und benachbarten Reihen:

```
  private double calculateDistance(double height1, double height2){
    return Math.sqrt(Math.pow(70,2)+Math.pow(height2-height1,2));
  }
```

Berechnung eines sicheren Fluchtplan für Minnie:

```
  private String getPossibleMinnieWay()
  {
//Berechnung der Zeit zu den ersten Lücken
    return calculateMinnieTime(0,minnieStart,0);
  }

  private String calculateMinnieTime(int row, double positionHeight, double timeSoFar)
  {
    for(int i=0; i<points.get(row).size(); i++)
    {
//Damit die Zeiten für jedes Loch in einer Reihe nicht ggf. addiert werden
      double minnieTime = timeSoFar;
//Berechnung der Zeit (siehe Max)
      minnieTime += calculateDistance(positionHeight, points.get(row).get(i).getHeight())/(20/3.6);
//Wenn Minnie dort schneller ist als Max, für die nächste Reihe von diesem Punkt mit dieser Zeit überprüfen oder, falls letzte Reihe Weg in Form von Koordinaten zurückgeben
      if(minnieTime<points.get(row).get(i).getMaxTime()){
        if(points.size()>row+1)
        {
          String s = calculateMinnieTime(row+1,points.get(row).get(i).getHeight(),minnieTime);
          if(!s.equals("No way found!")){
            return "("+(row+1)*70+"/"+points.get(row).get(i).getHeight()+")"+s;
          }
        }
        else
        {
          return "("+(row+1)*70+"/"+points.get(row).get(i).getHeight()+")";
        }
      }
    }
    return "No way found!";
  }
```