

# Das nächste-Punktepaar-Problem in der Ebene

---

## - Finding the closest pair of points -

---

Markus Becker

24. Juni 2009

## 1 Einleitung

### 1.1 Das Problem

Gegeben sind eine Ebene (etwa  $\mathbb{R}^2$ ) und  $n$  beliebige Punkte darin, wobei  $n \in \mathbb{N}$ . Die Aufgabe ist es, (auf möglichst „schnelle“ Weise) das Punktepaar zu finden, welches am nächsten beisammen liegt.

### 1.2 Mein Ziel

Für dieses Problem stelle ich einen Algorithmus vor, der in den 70er Jahren von *M. Shamos & D. Hoey* entworfen wurde. Mein Ziel ist es, die Idee Schritt für Schritt herzuleiten und dabei direkt auf die Laufzeiten einzugehen. Somit ergibt sich am Ende recht schnell die Laufzeitanalyse, eine Zusammenfassung in Pseudocode sowie die Korrektheit des Algorithmus'. Man wird sehen, dass es sich um einen *Divide- & Conquer-Algorithmus* handelt und dass dadurch die typische  $O(n \cdot \log(n))$ -Laufzeit entsteht.

### 1.3 mathematische Formulierung, Notationen, Vereinbarungen

Sei  $P = \{p_1, \dots, p_n\}$  eine Menge von Punkten, wobei  $p_i = (x_i, y_i) \in \mathbb{R}^2 \forall i = 0, \dots, n; n \in \mathbb{N}$ . Für 2 Punkte  $p_i, p_j \in P$ , bezeichnen wir mit  $d(p_i, p_j)$  ihren Abstand, indem wir die euklidische Metrik  $d(x, y) := \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$  definieren für  $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \in \mathbb{R}^2$ .

Ziel: Finde  $p_i, p_j \in P$  so, dass  $d(p_i, p_j)$  minimal wird  $\forall p_i, p_j \in P$ .

Vereinbarung: oBdA haben keine 2 Punkte in  $P$  die selben x- oder y-Koordinaten. Dies erleichtert das Vorgehen bei der Erstellung der Listen in Abschnitt 3. (Man könnte das Problem recht einfach umgehen, aber darauf gehe ich hier nicht genauer ein.)

## 2 Zwei (naive) Lösungsideen

Im Folgenden stelle ich kurz zwei Ansätze vor, die jedoch bei Weitem nicht unsere angestrebten Ziele erfüllen werden, dafür aber erste Ideen für den endgültigen Algorithmus liefern.

## 2.1 Idee 1

Die sicherlich „naivste“ Idee ist Folgende: Bestimme die Distanzen zwischen allen  $n$  Punkten in  $P$  und nehme das Minimum. Die Korrektheit ist offensichtlich, leider auch die Laufzeit von  $O(n^2)$  wegen der  $\frac{n(n-1)}{2}$  Punktepaarvergleiche. Idee 1 sollte daher schnell verworfen werden.

## 2.2 Idee 2

Wie würde man dieses Problem im Ein-Dimensionalen angehen? Gegeben eine willkürliche Punktmenge  $P = \{a, b, c, \dots\}$  auf einer Linie. Man würde diese Punkte jetzt in  $O(n \cdot \log(n))$  sortieren, dann einmal durch die sortierte Liste gehen und sich die jeweiligen Abstände zum nächsten Punkt merken. Offensichtlich ist einer dieser Abstände das Minimum.

Im Zwei-Dimensionalen könnte man die Punkte nach ihren x- bzw. y-Koordinaten sortieren und vorgehen wie oben. Jedoch konstruiert man leicht Beispiele, die zeigen, dass wir so meist nicht das Paar mit minimalem Abstand finden.

Ganz so abwegig ist die Idee dennoch nicht, wie wir später sehen werden.

## 3 Entwurf des Algorithmus'

In diesem Abschnitt beschäftige ich mich mit dem Entwurf des oben angesprochenen Algorithmus'. Dazu sei im Folgenden immer  $P = \{p_1, \dots, p_n\}$  die Punktmenge mit Mächtigkeit  $n$  wie in Abschnitt 1 beschrieben.

Der Plan ist, die Divide-&-Conquer Strategie anzuwenden (3.1). Teile  $P$  auf in eine „linke“ Hälfte  $Q$  und in eine „rechte“ Hälfte  $R$  und finde die closest pairs rekursiv in  $Q$  und  $R$ . (vergl. Abb. 1). Mit dieser Information lässt sich eine globale Lösung für  $P$  in linearer Zeit finden (3.2). Ein Algorithmus dieser Art, hat eine Laufzeit von  $O(n \cdot \log(n))$ . (vergl.(5))

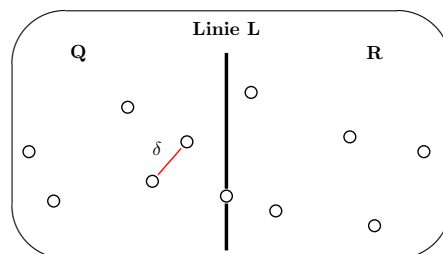
### 3.1 Die Rekursion

Ganz zu Beginn werden alle Punkte in  $P = \{p_1, \dots, p_n\}$  nach aufsteigenden x- bzw. y-Koordinaten sortiert. (z.B. mittels Merge-Sort in  $O(n \cdot \log(n))$ ). Es werden also Listen  $P_x := [p_{1_x}, \dots, p_{n_x}]$  und  $P_y := [p_{1_y}, \dots, p_{n_y}]$  erstellt. (Hier kommt es also auf die Reihenfolge der Punkte an!) Jeder Punkt in jeder der beiden Listen erhält dabei einen Vermerk über seine Position in beiden Listen.

$f_y : P_y \rightarrow \{1, \dots, n\} \subset \mathbb{N}$  bezeichne dabei (für theoretische Zwecke) die Funktion, die jedem Punkt der Liste  $P_y$ , seine Position in  $P_x$  zuweist. Also  $f_y(p_{i_y}) = j$ , weist beispielsweise dem  $i$ -ten Punkt der Liste  $P_y$ , die Position  $j$  in der Liste  $P_x$  zu. Analog könnte man  $f_x$  definieren.

Generell wird *jeder* rekursive Aufruf auf einer Teilmenge  $P' \subseteq P$  des Algorithmus' so beginnen, dass wieder 2 Listen  $P'_x, P'_y$  zur Verfügung stehen. Warum das gilt, sehen wir unten in Bem. 1.

Mit Hilfe dieser Listen können wir uns nun die *Rekursion* anschauen: Im ersten Rekursionsschritt spalten wir die Menge  $P$  in der Mitte bezüglich ihrer x-Koordinaten auf. Das heißt wir bestimmen  $\lceil \frac{n}{2} \rceil$ , betrachten  $P_x$  und definieren  $Q := \{p_{1_x}, \dots, p_{\lceil \frac{n}{2} \rceil_x}\}$  sowie  $R := \{p_{\lceil \frac{n}{2} \rceil_x + 1_x}, \dots, p_{n_x}\}$ . (vergl. Abb. 1). Alle folgenden Rekursionsschritte verlaufen natürlich analog.



**Abbildung 1:** Der 1. Rekursionsschritt: Die Linie L trennt P in Q und R. Auf jeder Seite wird rekursiv das closest pair gefunden.

**Bemerkung 1:** Mit einem einfachen Lauf durch die Listen  $P_x$  und  $P_y$ , können wir nun in linearer Zeit die folgenden vier Listen erstellen:  $Q_x$ , bestehend aus Punkten aus  $Q$ , nach aufsteigenden  $x$ -Koordinaten sortiert;  $Q_y$ , bestehend aus Punkten aus  $Q$ , nach aufsteigenden  $y$ -Koordinaten sortiert; sowie gleichermaßen die Listen  $R_x$  und  $R_y$ , wobei wie zu Beginn, jedem Eintrag in jeder Liste ein Vermerk über seine Position in beiden seiner zugehörigen Listen angehängen wird. Die Konstruktion dieser vier Listen funktioniert wie folgt:

Offensichtlich gilt:  $Q_x := [p_{1_x}, \dots, p_{\lceil \frac{n}{2} \rceil_x}]$  und  $R_x := [p_{\lceil \frac{n}{2} \rceil_x + 1_x}, \dots, p_{n_x}]$  wobei  $n = |P|$ .

Etwas aufwändiger wird es für  $Q_y$  und  $R_y$ :

Man läuft Punkt für Punkt durch die Liste  $P_y = [p_{1_y}, \dots, p_{n_y}]$  und schaut, welche Position der jeweilige Punkt  $p_{i_y}$  in der Liste  $P_x$  hat. Wenn er dort in der „linken“ Hälfte liegt, so kommt er in die Liste  $Q_y$ . Wenn er in der „rechten“ Hälfte liegt, so kommt er in  $R_y$ . Formal heißt das also, wir laufen für  $i = 0, \dots, n$  durch  $P_y$  und die Abfrage  $f_y(p_{i_y}) \leq \lceil \frac{n}{2} \rceil$  entscheidet für alle Punkte, ob sie in  $Q_y$  oder  $R_y$  angehängen werden. Falls sie wahr ist, so landet der Punkt in  $Q_y$ . Falls nicht, dann in  $R_y$ . Auch hier ist  $n$  die Mächtigkeit der gerade betrachteten (Teil-)menge.

Da wir also am Anfang einmal in  $O(n \cdot \log(n))$  die Listen  $P_x, P_y$  konstruiert hatten, bedarf es uns ab jetzt in jedem Rekursionsschritt wegen obiger Konstruktion nur noch einem Aufwand von  $O(n)$  um die 4 Listen  $Q_x, Q_y, R_x$  und  $R_y$  zu erstellen!

Unter Benutzung dieser vier Listen erhalten wir rekursiv ein closest pair in  $Q$  bzw.  $R$ , nennen wir es  $q_0^*$  und  $q_1^*$  sowie  $r_0^*$  und  $r_1^*$ .

Bis hierhin haben wir nur die generelle Divide-&-Conquer Strategie angewandt, aber noch nicht benutzt, dass es sich um das closest-pair-Problem handelt. Das heißt, bisher haben wir mehr oder weniger nur die naive *Idee 2* von oben benutzt. Das sollte die Frage aufwerfen, ob wir hier fertig sind.

Nein! Denn einfach nur das Minimum aller Teillösungen zu nehmen liefert nicht das gewünschte Ergebnis. Das führt uns zu dem Problem, wie man die Teillösungen geschickt zusammenfügt.

### 3.2 Kombiniere die Teillösungen

Wo liegt das Problem? - Die Distanzen, die wir bisher noch nicht betrachtet haben, sind jene zwischen Punkten in der „linken“ Hälfte und Punkten in der „rechten“ Hälfte. Bisher haben wir ein Paar mit minimaler Distanz in  $Q$  und ein Paar in  $R$  gefunden. Was aber, wenn gilt:

$$\exists q \in Q, r \in R : d(q, r) < \min\{d(q_0^*, q_1^*), d(r_0^*, r_1^*)\} \quad (\star)$$

In diesem Fall hätten wir ein Paar gefunden, das minimalere Distanz hat als die beiden vorigen. Wie können wir jetzt das „minimalste“ Paar  $(q, r) \in Q \times R$  in linearer Zeit finden?

Jedes  $q \in Q$  mit jedem  $r \in R$  zu vergleichen würde unsere verlangte Laufzeit natürlich sofort kaputt machen. Daher sollten wir versuchen einzuschränken, welche Kandidaten überhaupt in Frage kommen ( $\star$ ) zu erfüllen. Nur so kann man eine Strategie entwickeln um die Kombination der Teillösungen in  $O(n)$  zu erreichen und die Algorithmenlaufzeit bei  $O(n \cdot \log(n))$  zu halten.

Dazu sei  $\delta := \min\{d(q_0^*, q_1^*), d(r_0^*, r_1^*)\}$ ,  $x^*$  die  $x$ -Koordinate des Punktes in  $Q$  der am weitesten rechts liegt und  $L := \{(x, y) : x = x^*\}$ , d.h.  $L$  trennt  $Q$  vertikal von  $R$ . (vergl. Abb.1)

Die Frage ist jetzt: Gibt es solche  $q$  und  $r$  die ( $\star$ ) erfüllen? Falls nein, dann haben wir das nächste Paar bereits in einem unserer rekursiven Aufrufe gefunden. Falls ja, dann bilden eben diese  $q$  und  $r$  das closest pair in  $Q \cup R$ . Dazu ein kleines Lemma.

**Lemma 1:** Falls  $q \in Q$  und  $r \in R$  existieren mit  $d(q, r) < \delta$ , dann haben sowohl  $q$  als auch  $r$  höchstens Abstand  $\delta$  zur Linie  $L$ .

**Beweis:** Wir nehmen an, solche  $q$  und  $r$  existieren. Sei  $q = (q_x, q_y)$  und  $r = (r_x, r_y)$ . Aus der Definition von  $L$  wissen wir, dass  $q_x \leq x^* \leq r_x$  gilt. Damit gilt auch  $x^* - q_x \leq r_x - q_x$ . Wegen

$$d(q, r) = \sqrt{(r_x - q_x)^2 + \underbrace{(r_y - q_y)^2}_{\geq 0}} \text{ gilt: } r_x - q_x \leq d(q, r) < \delta,$$

also auch

$$x^* - q_x \leq r_x - q_x \leq d(q, r) < \delta \text{ und analog } r_x - x^* \leq r_x - q_x \leq d(q, r) < \delta.$$

Folglich haben die x-Koordinaten von  $q$  und  $r$  höchstens Abstand  $\delta$  zu  $x^*$  und somit liegen die Punkte  $q$  und  $r$  mit maximal Abstand  $\delta$  an der Linie  $L$ .  $\square$

Daraus folgt, dass sich die Suche solcher  $q$  und  $r$ , die  $(\star)$  erfüllen könnten, beschränkt auf die Suche von Punkten in einem  $2\delta$ -Streifen um  $L$ .

Sei  $S \subseteq P$  die Menge von Punkten, die in diesem Streifen liegen und  $S_y$  die Liste die aus Punkten in  $S$  besteht, geordnet nach aufsteigenden y-Koordinaten.

$S_y$  lässt sich mit einem einzigen Lauf durch  $P_y = [p_{1_y}, \dots, p_{n_y}]$  in  $O(n)$  konstruieren. Dazu bezeichne  $p_{i_{y_x}}$  die x-Koordinate des Punktes  $p_{i_y}$  aus der Liste  $P_y$ . Die Abfrage  $|p_{i_{y_x}} - x^*| \leq \delta$  genügt, um zu entscheiden ob  $p_{i_y}$  in  $S_y$  eingefügt wird oder nicht.

Mit *Lemma 1* folgt in anderen Worten:

**Korollar 1:** Sei  $\delta$  wie oben. Dann gilt:

$$\exists q \in Q, r \in R : d(q, r) < \delta \Leftrightarrow \exists s, s' \in S : d(s, s') < \delta$$

**Beweis:**

„ $\Rightarrow$ “ klar: Aus  $d(q, r) < \delta \Rightarrow q, r \in S$  nach *Lemma 1* und Def. von  $S$ , d.h. setze  $s := q, s' := r$ .

„ $\Leftarrow$ “ Seien  $s, s' \in S$  mit  $d(s, s') < \delta$ . Dann gilt sogar, dass  $s$  und  $s'$  *nicht* beide gleichzeitig in  $Q$  oder  $R$  liegen können. Falls dies so wäre, hätte die Rekursion bereits  $(s, s')$  als closest pair identifiziert. Da dies wegen  $d(s, s') \leq \delta$  nicht der Fall war, können wir  $q := s$  und  $r := s'$  setzen.  $\square$

Wenn wir also nach  $(\star)$  suchen wollen, so reicht es ab jetzt, wenn wir nach  $s, s' \in S$  suchen mit  $d(s, s') < \delta$ .

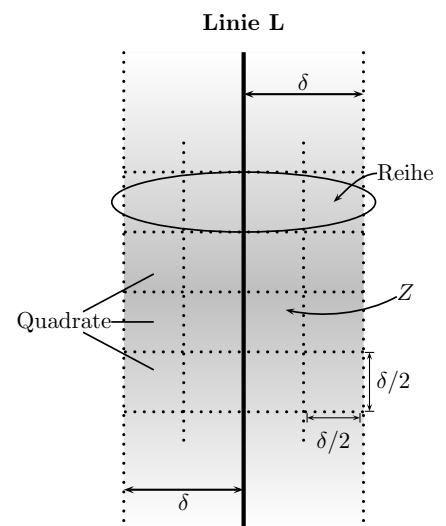
Folgendes wichtige Lemma erleichtert uns die Suche noch einmal erheblich, indem es die Anzahl der fraglichen  $s \in S$  einschränkt:

**Lemma 2:** Falls man  $s, s' \in S$  findet mit  $d(s, s') < \delta$ , so liegen  $s$  und  $s'$  in der sortierten Liste  $S_y$  maximal 15 Positionen auseinander.

Das heißt, bei der Kombination der Teillösungen, muss jedes  $s \in S_y$  nur noch mit maximal 15 folgenden Punkten in  $S_y$  verglichen werden. Das ist ein konstanter Wert!

**Beweis:** Sei  $Z \subset \mathbb{R}^2$  eine Teilmenge der Ebene, die alle Punkte mit Abstand  $\delta$  zu  $L$  enthält. Wir unterteilen  $Z$  in Quadrate mit Seitenlängen  $\delta/2$ . Dabei bestehe eine Reihe in  $Z$  aus vier Quadraten mit gleichen y-Koordinaten an ihren horizontalen Seiten. (vergl. Abb. 2)

In jedem solchen Quadrat kann maximal ein Punkt aus  $S$  liegen. Lägen zwei Punkte in einem Quadrat, so lägen sie beide in



**Abbildung 2:**  $Z$  wird in Quadrate unterteilt wie im Beweis von Lemma 2.

$Q$  oder beide in  $R$ , da alle Punkte in einem Quadrat auf der selben Seite von  $L$  liegen. Wenn 2 Punkte jedoch im selben Quadrat lägen, so hätten sie maximal Abstand  $\delta/\sqrt{2}$  zueinander. Das widerspricht der Definition von  $\delta$  als minimale Distanz zwischen Punktepaaren in  $Q$  oder  $R$ . Folglich enthält jedes Quadrat maximal einen Punkt aus  $S$ .

Betrachten wir nun  $s$  und  $s'$  mit  $d(s, s') < \delta$ . Nehmen wir an, sie lägen in der Liste  $S_y$  mindestens 16 Positionen auseinander. Da maximal ein Punkt pro Quadrat existieren kann, würden mindestens drei Reihen von  $Z$  zwischen  $s$  und  $s'$  liegen. Das heißt wiederum, dass  $s$  und  $s'$  mindestens Abstand  $3\delta/2$  hätten.  $\nexists$  zu  $d(s, s') < \delta$   $\square$

Man bemerke, dass 15 nur eine obere Schranke ist und reduziert werden kann. Für uns ist es momentan jedoch nur wichtig, dass man lediglich eine *konstante* Anzahl an Punktevergleichen in  $S$  braucht. Also unabhängig von  $|P| = n$ .

Damit ist es nun möglich, den Algorithmus mit seiner Laufzeit von  $O(n \cdot \log(n))$  vollständig aufzustellen.

## 4 Der Algorithmus

### 4.1 Zusammenfassung des Algorithmus'

Wir behalten alle oben eingeführten Notationen bei und fassen zusammen:

Gegeben sei die Menge  $P = \{p_1, \dots, p_n\}$ . Gesucht sind

$$p_k, p_l \in P \text{ mit } k \neq l : d(p_k, p_l) = \min_{1 \leq i, j \leq n} \{d(p_i, p_j) : i \neq j\}$$

Zu *Beginn* bestimmt der Algorithmus  $P_x$  und  $P_y$  durch sortieren in  $O(n \cdot \log(n))$  Zeit.

Jede *Rekursion* auf einer Teilmenge  $P' \subseteq P$  startet immer mit der Abfrage, ob  $|P'| \leq 3$  gilt. Falls ja, so bestimmen wir das kleinste Paar durch paarweise Abstandsbestimmung aller möglichen Kombinationen. Falls nein, so erfolgt der Divide-Schritt und wir teilen  $P'$  gemäß (3.1) so lange, bis wir irgendwann weniger als vier Punkte in einer Teilmenge haben.

Somit liefert die Rekursion irgendwann zwei minimale Paare  $q_0^*, q_1^* \in Q$  und  $r_0^*, r_1^* \in R$ , wobei  $Q \cup R = T$  für irgendeine Teilmenge  $T \subseteq P$  ist.

In (3.2) haben wir dann hergeleitet, wie wir mit Hilfe dieser beiden Paare, deren Distanzen und der speziellen Verwaltung der Listen, ein minimales Paar in ganz  $T$  finden. Dazu haben wir  $S \subseteq T$  sowie  $S_y$  in  $O(n)$  konstruiert. Nun können wir mit *Lemma 2* den letzten entscheidenden Schritt machen:

Wir gehen einmal durch die Liste  $S_y$  und bestimmen für jedes  $s \in S_y$  die Distanz zu jedem der folgenden 15 Punkte in  $S_y$ . (Durch die konstante Zahl 15 gelingt uns dies in  $O(n)$ .)

*Lemma 2* sagt dann, dass wir auf diese Weise die Distanzen *aller* Punktepaare in  $S$  bestimmt haben, deren Distanz kleiner  $\delta$  sein *könnten*! Bezeichnen wir mit  $\mu$  das Minimum eben dieser Distanzen. Es gibt nun zwei Möglichkeiten:

1.  $\mu < \delta$  : Das zu  $\mu$  gehörige Paar ist das nächste Punktepaar in  $T$ .
2.  $\mu \geq \delta$  : Das nächste Punktepaar welches durch die Rekursion gefunden wurde, ist das nächste Punktepaar in  $T$ .

Wir haben das closest pair in  $T$  gefunden!

Auf diese Weise bestimmt der Algorithmus somit nun das closest pair in ganz  $P$ .

## 4.2 Der Algorithmus in Pseudocode

Wir behalten wieder obige Notationen bei und stellen fest:

---

### Algorithm 1 Closest-Pair-Algorithmus

---

```
1: procedure Closest-Pair( $P$ )
2:   Erstelle  $P_x$  und  $P_y$  //  $O(n \cdot \log(n))$ 
3:    $(p_0^*, p_1^*) = \text{Closest-Pair-Rec}(P_x, P_y)$ 
4: end procedure

5: procedure Closest-Pair-Rec( $P_x, P_y$ )
6:   if  $|P| \leq 3$  then
7:     finde das closest pair durch paarweises Messen aller Distanzen
8:   else
9:     konstruiere  $Q_x, Q_y, R_x, R_y$  //  $O(n)$ 
10:     $(q_0^*, q_1^*) = \text{Closest-Pair-Rec}(Q_x, Q_y)$ 
11:     $(r_0^*, r_1^*) = \text{Closest-Pair-Rec}(R_x, R_y)$ 

12:     $\delta \leftarrow \min(d(q_0^*, q_1^*), d(r_0^*, r_1^*))$ 
13:     $x^* \leftarrow$  maximale x-Koordinate eines Punktes der Menge  $Q$ 
14:     $L \leftarrow \{(x, y) : x = x^*\}$ 
15:     $S \leftarrow$  Punkte aus  $P$  mit maximal Abstand  $\delta$  zu  $L$ 

16:    konstruiere  $S_y$  //  $O(n)$ 
17:    for all  $s \in S_y$  do //  $O(n)$ 
18:      bestimme Distanz von  $s$  zu jedem der folgenden 15 Punkte in  $S_y$ 
19:    end for
20:     $(s, s') \leftarrow$  Paar mit minimaler Distanz unter all diesen untersuchten

21:    if  $d(s, s') < \delta$  then
22:      return  $(s, s')$ 
23:    else if  $d(q_0^*, q_1^*) < d(r_0^*, r_1^*)$  then
24:      return  $(q_0^*, q_1^*)$ 
25:    else
26:      return  $(r_0^*, r_1^*)$ 
27:    end if
28:  end if
29: end procedure
```

---

## 5 Algorithmenanalyse

Wegen des Algorithmenentwurfes aus *Abschnitt 3* folgt mehr oder weniger sofort:

**Satz 1:** *Der Algorithmus 1 bestimmt korrekterweise ein closest pair von Punkten aus  $P$ .*

**Beweis:** Wie schon bemerkt, wurden alle Komponenten des Beweises bereits beim Entwurf des Algorithmus' erarbeitet. Hier eine Zusammenfassung, wie alle Komponenten zusammenpassen:

Die Korrektheit wird mit Induktion über  $|P|$  bewiesen. Der Fall  $|P| \leq 3$  ist klar. Bei gegebenem  $P$ , wird das closest pair in den rekursiven Aufrufen offensichtlich korrekt per Induktion bestimmt. Der restliche Teil des Algorithmus', die Kombination der Teillösungen (3.2), entscheidet ob Punkte in  $S$  existieren mit kleinerem Abstand als  $\delta$ . Wenn diese existieren, werden sie korrekt ausgegeben. Jetzt liegen entweder beide Punkte des nächsten Punktepaares von  $P$  in  $Q$  oder  $R$ , oder sie liegen verteilt; ein Punkt in  $Q$  und ein Punkt in  $R$ . Im ersten Fall folgt die Korrektheit durch die Rekursion - sie lieferte bereits ein closest pair. Im zweiten Fall hat das gefundene Paar eine Distanz kleiner  $\delta$  und wurde korrekterweise durch (3.2) gefunden.  $\square$

Es folgt außerdem sofort:

**Satz 2:** *Die Laufzeit von Algorithmus 1 beträgt  $O(n \cdot \log(n))$ .*

**Beweis:** Zu Beginn wurde  $P$  sortiert und es entstanden  $P_x$  und  $P_y$ . Das benötigt  $O(n \cdot \log(n))$ . Die Laufzeit der Rekursion genügt (typischerweise bei Divide-&-Conquer-Verfahren) der Rekurrenz

$$T(n) = 2 T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + O(n)$$

und ist folglich  $O(n \cdot \log(n))$ .

Die geschickte Idee für (3.2) braucht wie oben gesehen pro Schritt nur  $O(n)$  und somit steht die Gesamtlaufzeit von  $O(n \cdot \log(n))$ .  $\square$